A PROTOCOL VALIDATOR FOR THE SCM AND CFSM MODELS

by

Zeki Bulent Bulbul

June 1993

Thesis Advisor: G. M. Lundy

Approved for public release; distribution is unlimited.
A PROTOCOL VALIDATOR FOR THE SCM AND CFSM MODELS

This thesis introduces and describes a software tool called *Mushroom* which automates the analysis of network protocols specified by the Systems of Communicating Machines (SCM) and the Communicating Finite State Machines (CFSM) models. SCM is a formal model for the specification, verification, and testing of communication protocols. This model was originally developed to improve the CFSM model which is a simpler and earlier Formal Description Technique (FDT).

The program is developed as two separate programs in the Ada programming language. The first program automates either the system state analysis (*Smart Mushroom*), or the full global analysis (*Big Mushroom*) for a protocol specified by the SCM model. The second program called *Simple Mushroom*, automates the global reachability analysis for the CFSM model.

*Mushroom* greatly facilitates the use of these models for protocol design and analysis. The run time and memory efficiency of a previous program was improved to allow the analysis of larger and more complex protocols. The program was also extended to accept up to eight machines (processes) in the protocol specification. The user interface of the program has also been improved.

*Mushroom* has been used to verify some well known protocols specified by the SCM and CFSM models such as the token bus protocol, Go Back N and Lap-B data link control protocol.

This thesis introduces and describes a software tool called *Mushroom* which automates the analysis of network protocols specified by the Systems of Communicating Machines (SCM) and the Communicating Finite State Machines (CFSM) models. SCM is a formal model for the specification, verification, and testing of communication protocols. This model was originally developed to improve the CFSM model which is a simpler and earlier Formal Description Technique (FDT).

The program is developed as two separate programs in the Ada programming language. The first program automates either the system state analysis (*Smart Mushroom*), or the full global analysis (*Big Mushroom*) for a protocol specified by the SCM model. The second program called *Simple Mushroom*, automates the global reachability analysis for the CFSM model.

*Mushroom* greatly facilitates the use of these models for protocol design and analysis. The run time and memory efficiency of a previous program was improved to allow the analysis of larger and more complex protocols. The program was also extended to accept up to eight machines (processes) in the protocol specification. The user interface of the program has also been improved.

*Mushroom* has been used to verify some well known protocols specified by the SCM and CFSM models such as the token bus protocol, Go Back N and Lap-B data link control protocol.
Approved for public release; distribution is unlimited

A Protocol Validator for the SCM and CFSM Models

by
Zeki Bulent Bulbul
LTJG, Turkish Navy
B.S., Turkish Naval Academy, 1987

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL
June 1993

Author:
Zeki Bulent Bulbul

Approved By:
G. M. Lundy, Thesis Advisor

Man-Tak Shing, Second Reader

Gary Hughes, Chairman,
Department of Computer Science
ABSTRACT

This thesis introduces and describes a software tool called Mushroom which automates the analysis of network protocols specified by the Systems of Communicating Machines (SCM) and the Communicating Finite State Machines (CFSM) models. SCM is a formal model for the specification, verification, and testing of communication protocols. This model was originally developed to improve the CFSM model which is a simpler and earlier Formal Description Technique (FDT).

The program is developed as two separate programs in the Ada programming language. The first program automates either the system state analysis (Smart Mushroom), or the full global analysis (Big Mushroom) for a protocol specified by the SCM model. The second program called Simple Mushroom, automates the global reachability analysis for the CFSM model.

Mushroom greatly facilitates the use of these models for protocol design and analysis. The run-time and memory efficiency of a previous program was improved to allow the analysis of larger and more complex protocols. The program was also extended to accept up to eight machines (processes) in the protocol specification. The user interface of the program has also been improved.

Mushroom has been used to verify some well-known protocols specified by the SCM and CFSM models such as the token bus protocol, Go Back N and Lap-B data link control protocol.
# TABLE OF CONTENTS

I. INTRODUCTION .......................................................... 1  
   A. MOTIVATION ..................................................... 1  
   B. SCOPE OF THE THESIS ........................................... 2  
   C. ORGANIZATION .................................................. 2  

II. BACKGROUND OF MODELS ............................................. 4  
   A. COMMUNICATING FINITE STATE MACHINES ....................... 4  
      1. Model Definition ............................................. 4  
      2. An Example of Protocol Specification and Analysis Using CFSM .... 7  
      3. Summary ..................................................... 9  
   B. SYSTEMS OF COMMUNICATING MACHINES ......................... 10  
      1. Model Definition ........................................... 10  
      2. Algorithm: System State Analysis ........................... 12  
      3. An Example of Protocol Specification and Analysis Using SCM .... 13  
      4. Summary .................................................... 16  

III. SIMPLE MUSHROOM: A PROGRAM FOR AUTOMATING  
     CFSM REACHABILITY ANALYSIS ................................ 17  
   A. PROGRAM STRUCTURE ........................................... 18  
   B. INPUT .......................................................... 20  
   C. REACHABILITY ANALYSIS ....................................... 22  
   D. OUTPUT ........................................................ 25  

IV. SMART AND BIG MUSHROOM: A PROGRAM FOR  
     AUTOMATING SCM REACHABILITY ANALYSIS ..................... 28  
   A. PROGRAM STRUCTURE ........................................... 28
B. INPUT ...................................................... 31
   1. Finite State Machines .................................. 31
   2. Variable Definitions ................................... 33
   3. Predicate-Action Table ................................ 34
C. REACHABILITY ANALYSIS .................................. 39
   1. Global Reachability Analysis ............................ 39
   2. System state analysis ................................... 41
D. OUTPUT .................................................... 42
V. EXAMPLES FOR USING THE MUSHROOM PROGRAM ........... 48
   A. CFSM MODEL ............................................. 48
      1. A Simple Four Machine Protocol ....................... 48
      2. Analysis of Information Transfer Phase of the LAP-B Protocol .... 52
   B. SCM MODEL ............................................. 60
      1. Go Back N ........................................... 60
      2. Token Bus ........................................... 64
VI. CONCLUSIONS AND FURTHER RESEARCH POSSIBILITIES ....... 70
APPENDIX A (LAP-B Protocol Information Transfer Phase) ........... 74
   FSM Text File ............................................. 74
   Program Output ............................................ 77
APPENDIX B (Go back N Window Size of 10) ......................... 80
   FSM Text File ............................................. 80
   Variable Definitions ....................................... 82
   Predicate Action Table .................................... 83
   Output format ............................................. 88
   Program Output(System State Analysis) ....................... 89
APPENDIX C (Token Bus Protocol) ................................ 101
FSM Text File .................................................... 101
Variable Definitions ............................................. 103
Predicate Action Table ............................................ 109
Output Format ..................................................... 117
Program Output (System State Analysis) ......................... 118
Program Output (Global Reachability Analysis) ................... 127
REFERENCES .................................................... 133
INITIAL DISTRIBUTION LIST ................................. 135
I. INTRODUCTION

A. MOTIVATION

In the last decade increasing complexity in computer communication systems have created a growing demand for formal techniques to specify, design, verify and test protocols. In order to have a clear understanding of the protocols, both for the protocol designer and implementor, it is essential to have a formal protocol specification.

There are a large number of formal techniques available for modeling protocols. Most of these methods can be placed into one of the following general classifications [Ref. 1]: communicating finite state machines, Petri nets, programming languages and hybrids. Some models that have found most interest and chosen for standardization are ESTELLE, LOTOS and SDL. Each of these has its own pros and cons.

Systems of communicating machines (SCM) is also a formally defined model for specification, analysis and testing of protocols that is defined in [Ref. 2]. This model uses a combination of finite state machines and variables, which may be local to a single machine or shared by two or more machines, so it can be classified in the models known as "extended finite-state machines." The main goal of the SCM model was to improve the well-known simpler Communicating Finite-State Machines (CFSM) model. The SCM model has been used to specify and analyze several protocols [Ref. 3], [Ref. 4], [Ref. 5], [Ref. 6], [Ref. 7]. Analysis of protocols specified with this model can be executed using a method called system state analysis. This analysis is similar to global reachability analysis, but generates a subset of all reachable states. Sometimes this subset is sufficient to verify the protocol. In some cases system state analysis is not sufficient for protocol analysis, and
global analysis is needed. However, it is possible to automate the system state analysis and global analysis based on the SCM model.

Several tools exist for the design and verification of protocols. These tools are very important for increasing the usefulness of the formal description techniques (FDT).

While there is no "perfect" formal specification technique, there is still room for more work to understand the advantages of different formal models and develop better tools to increase the utilization of these models.

B. SCOPE OF THE THESIS

The goal of the thesis is to present a software tool, called mushroom that automates the reachability analysis of protocols formally specified using CFSM and SCM models. The name mushroom was chosen as a symbol of something that starts out relatively small (specification) and gets much bigger quickly (analysis). An earlier version of the program [Ref. 8] was capable of generating reachability analysis for the protocols consisting of only two machines. This thesis expands on this earlier work and is capable of analyzing protocols that has any number of machines from two to eight. In addition, the user interface for the program has also been improved. The program was tested against results of several previous works and has confirmed their results. It is also believed that this program will help to solve some problems concerning the SCM model.

C. ORGANIZATION

The thesis has six chapters. Chapter II reviews the Communicating Finite State Machines (CFSM) and Systems of Communicating Machines (SCM) models. In Chapter III, a program called simple mushroom, which automates the global reachability analysis based on CFSM model, is described. Chapter IV describes a program that automates the system state analysis (smart mushroom), or the full global analysis (big mushroom) for
a protocol specified formally using the SCM model. In Chapter V, some examples of the
use of the program are given. Chapter VI concludes the thesis with a research review and
suggestions for future work.
II. BACKGROUND OF MODELS

A. COMMUNICATING FINITE STATE MACHINES

Communicating finite state machine (CFSM) model is a simple model and perhaps the earliest FDT. In this model, each machine in the network is modeled as a finite automaton or finite state machine (FSM), with communication channels between pairs of machines modeled as one-way, infinite length FIFO queues. There is a great deal of literature on this model [Ref. 9] [Ref. 10] [Ref. 11]. The model is defined for an arbitrary number of machines; however, for simplicity, a two machine model (shown in Figure 1) will be presented here.

![Machine 1](machine1.png) ![Machine 2](machine2.png)

Figure 1: CFSM, 2 machine model representation

1. Model Definition

This section defines the CFSM model [Ref. 12] and provides a simple protocol specification and analysis to clarify the definition.

A communicating machine $M$ is a finite, directed labeled graph with two types of edges, sending and receiving. A sending (receiving) edge is labeled `$-g$' (`$+g$') for some message $g$, taken from a finite set $G$ of messages. One of the nodes in $M$ is identified as the initial node, and each node is reachable from the initial node by some directed path. A node in $M$ whose outgoing edges are all sending (receiving) edges is a sending (receiving) node; otherwise the node is a mixed node. If the outgoing edges of each node in $M$ have distinct
labels, then \( M \) is deterministic; otherwise \( M \) is nondeterministic. The nodes of \( M \) are often referred to as states; these two terms will be used interchangeably throughout this thesis.

Let \( M \) and \( N \) be two communicating machines having the same set \( G \) of messages; the pair \((M,N)\) is a network. A global state of this network is a four tuple \([m,c_m,n,c_n]\), where \( m \) and \( n \) are nodes (states) from \( M \) and \( N \), and \( c_m \) and \( c_n \) are strings from the set \( G \) of messages. Intuitively, the global state \([m,c_m,n,c_n]\) means that the machines \( M \) and \( N \) have reached states \( m \) and \( n \), and the communication channels contain the strings \( c_m \) and \( c_n \) of messages, where \( c_m \) denotes the messages sent from \( M \) to \( N \) in channel \( C_M \), and \( c_n \) denotes the messages sent from \( N \) to \( M \) in channel \( C_N \). In the case of say \( k \) number of machines where \( k > 2 \) the global state can be represented as \([m_1,q_{12},q_{13},...m_2,q_{21},q_{23},...m_3,q_{31},q_{32},...m_k,q_{k1},q_{k2},...]\) where \( m_i \)'s are the nodes of machines \( M_i \) and \( q_{ij} \) contains the messages sent from \( M_i \) to \( M_j \). Subscripts \( i \) and \( j \) ranges from \( 1..k \) and \( i \neq j \).

The initial global state of \((M,N)\) is \([m_0,E,n_0,E]\), where \( m_0 \) and \( n_0 \) are the initial states of \( M \) and \( N \), and \( E \) is the empty string.

The network progresses as transitions are taken in either \( M \) or \( N \). Each transition consists of a state change in one of the machines, and either the addition of a message to the end of one channel (sending transition) or the deletion of a message from the front of one channel (receiving transition).

A sending transition in \( M \) (\( N \)) adds a message to the end of channel \( C_M \) (\( C_N \)); a receiving transition in \( M \) (\( N \)) removes a message from the front of channel \( C_M \) (\( C_N \)).

Suppose \(+g\) is a receiving transition from state \( i \) to \( j \) in machine \( M \) (\( N \)). The transition can be executed if and only if \( M \) (\( N \)) is in state \( i \) and the message \( g \) is at the front
of the channel $C_N$ ($C_M$). The execution takes zero time. After its execution, machine $M$ ($N$) is in state $j$, and the message $g$ has been removed from the channel $C_N$ ($C_M$).

Similarly, suppose $-g$ is a sending transition from state $i$ to $j$ in $M$ ($N$). The transition can be executed if and only if $M$ ($N$) is in state $i$. Afterwards, $g$ appears on the end of the outgoing channel, and the machine has transitioned to state $j$.

Suppose $s_1 = [m,c_i,n,c_j]$ is a global state of ($M,N$). State $s_2$ follows $s_1$ if there is a transition (in $M$ or $N$) which can be executed in $s_1$ if there is a sequence of states $s_i, s_{i+1}, \ldots, s_{i+p}$ such that $s_i$ follows $s_1, s_{i+1}$ follows $s_i$, and so on, and $s_2$ follows $s_{i+p}$. A state $s$ is reachable if it is reachable from the initial state.

The communication of a network ($M,N$) is bounded if, for every reachable state $[m,c_m,n,c_n]$ there is a nonnegative integer $k$ such that $|c_m| \leq k$ and $|c_n| \leq k$, where $|c|$ denotes the number of messages in channel $C$.

A reachability graph of a network ($M,N$) is a directed graph in which the nodes correspond to the reachable global states of ($M,N$), and the edges represent the follows function. That is, there is an edge from state $s_i$ to state $s_j$ if and only if $s_j$ follows $s_i$. The edges are labeled with the transitions which they represent. This reachability graph can be generated by starting with the initial state, and adding the states which follow it, connecting them to it with edges; and repeating for each new state generated.

The next two definitions are of errors that may occur in a communication protocol, which are detectable by analysis.

A global state $[m,c_m,n,c_n]$ is a deadlock state if both $m$ and $n$ are receiving nodes, and $c_m = c_n = E$, where $E$ denotes the empty string.

A global state $[m,c_m,n,c_n]$ is an unspecified reception state if one of the following two conditions is true:
(1) $m$ is a receiving state, the message at the head of channel $c_m$ is $g$, and none of $m$'s outgoing transitions is labeled `+g.'

(2) $n$ is a receiving state, the message at the head of channel $c_n$ is $g$, and none of $n$'s outgoing transitions is labeled `+g.'

These error conditions can be identified by generating the reachability graph for a network, and inspecting all states as they are generated.

In the next section, an example protocol is specified and analyzed using the CFSM model.

2. An Example of Protocol Specification and Analysis Using CFSM

CFSM specification of an imaginary ring-like network consisting of three communicating machines is shown in Figure 2.

![CFSM specification for the example protocol](image)

Figure 2: CFSM specification for the example protocol

It is assumed that the protocol is used at the data link layer, making use of the services provided by the physical layer.
Edges are labeled such that the characters following the ‘-/+’ shows the messages and the numbers represent the destination machine. Each machine sends one message to the next machine and receives a message from the previous machine in clockwise direction forming a ring. Ignore the dashed edges and nodes for the time being. The initial state of each machine is 1; thus the initial global state is [1,E,E,1,E,E,1,E,E].

The reachability analysis can be done by a simple procedure. Starting with the initial global state only one transition is possible, the ‘-DO’ of the machine 1 from state 1. This leads to global state [2,D0,E,1,E,E,1,E,E]. We can continue the analysis in the same manner detecting the possible transitions from this new global state. The complete reachability analysis is given in Figure 3 consisting of a total of six states.

![Figure 3: Reachability analysis of the example protocol](image)

In this sample protocol, there are no deadlocks or unspecified receptions. If the dashed edges and states in Figure 2 are added to the specification, the reachability analysis
shown in Figure 4 would be achieved. In this analysis there is one deadlock condition and one unspecified reception. In global state [3,E,E,3,E,E,1,E,E], all the channels are empty and all the nodes are receiving nodes satisfying the deadlock condition. In global state [2,E,E,1,E,E,3,D4,E], machine 1 and machine 2 are in receiving states but none of the outgoing transitions are labeled '+D4', satisfying an unspecified reception condition.

3. Summary

The CFSM model is simple and easy to understand. However, as the protocols become more complex, this model becomes difficult to use due to a combinatorial explosion of states. The analysis might not terminate if the queue length is unbounded. The number of states in the reachability graph will be unmanageably large for such complex
proteks even if the queue length is bounded. A computer analysis might eventually terminate, but still the CPU time would be days even months, obviously impractical.

Another disadvantage is that as the protocols become more complex, the specification of the protocol can be so large, consisting of many states and transitions, that it makes it very hard to understand if it is the intended specification. Several examples are given in Chapter V that show the largeness of analysis for some protocols.

B. SYSTEMS OF COMMUNICATING MACHINES

In this section the SCM model is described. First the model definition is given, then the algorithm for generating the system state analysis is described. Finally the model is used for specification and analysis of an example protocol to illustrate the important aspects of the model.

1. Model Definition

A system of communicating machines is an ordered pair $C = (M,V)$, where

$$M = \{m_1, m_2, \ldots, m_n\}$$

is a finite set of machines, and

$$V = \{v_1, v_2, \ldots, v_k\}$$

is a finite set of shared variables, with two designated subsets $R_i$ and $W_i$ specified for each machine $m_i$. The subset $R_i$ of $V$ is called the set of read access variables for machine $m_i$, and the subset $W_i$ the set of write access variables for $m_i$.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s_i, L_i, N_i, \tau_i)$, where

1. $S_i$ is a finite set of states;
2. $s_i \in S_i$ is a designated state called the initial state of $m_i$;
3. $L_i$ is a finite set of local variables;
(4) $N_i$ is a finite set of names, each of which is associated with a unique pair $(p,a)$, where $p$ is a predicate on the variables $L_i \cup R_i$, and $a$ is an action on the variables of $L_i \cup R_i \cup W_i$. Specifically, an action is a partial function

$$a: L_i \times R_i \rightarrow L_i \times W_i$$

from the values of the local variables and read access variables to the values of the local variables and write access variables.

(5) $\tau_i: S_i \times N_i \rightarrow S_i$ is a transition function, which is a partial function from the states and names of $m_i$ to the states of $m_i$.

Machines model the entities, which in a protocol system are processes and channels. The shared variables are the means of communication between the machines. Intuitively, $R_i$ and $W_i$ are the subsets of $V$ to which $m_i$ has read and write access, respectively. A machine is allowed to make a transition from one state to another when the predicate associated with the name for that transition is true. Upon taking the transition, the action associated with that name is executed. The action changes the values of local and/or shared variables, thus allowing other predicates to become true.

The sets of local and shared variables specify a name and range for each. In most cases, the range will be a finite or countable set of values. For proper operation, the initial values of some or all of the variables should be specified.

A **system state tuple** is a tuple of all machine states. That is, if $(M,V)$ is a system of $n$ communicating machines, and $s_i$, for $1 \leq i \leq n$, is the state of machine $m_i$, then the $n$-tuple $(s_1,s_2,...,s_n)$ is the system state tuple of $(M,V)$. A **system state** is a system state tuple, plus the outgoing transitions which are enabled. Thus two system states are equal if every machine is in the same state, and the same outgoing transitions are enabled.

The **global state** of a system consists of the system state tuple, plus the values of all variables, both local and shared. It may be written as a larger tuple, containing the
system state tuple with the values of the variables. The initial global state is the initial system state tuple, with the additional requirement that all variables have their initial values. The initial system state is the system state such that every machine is in its initial state, and the outgoing transitions are the same as in the initial global state.

A global state corresponds to a system state if every machine is in the same state, and the same outgoing transitions are enabled. Clearly, more than one global state may correspond to the same system state.

Let $\tau(s_1, n) = s_2$ be a transition which is defined on machine $m_i$. Transition $\tau$ is enabled if the enabling predicate $p$, associated with name $n$, is true. Transition $\tau$ may be enabled whenever $m_i$ is in state $s_i$ and the predicate $p$ is true (enabled). The execution of $\tau$ is an atomic action, in which both the state change and the action $a$ associated with $n$ occur simultaneously.

It is assumed that if a transition is enabled indefinitely, then it will eventually occur. This is an assumption of fairness, and is needed for the proofs of certain properties.

2. Algorithm: System State Analysis

The process of generating the set of all system states reachable from the initial state is called system state analysis. This analysis constructs a graph, whose nodes are the reachable system states, and whose arcs indicate the transitions leading from each system state to another. This graph may be generated by a mechanical procedure which consists of the following three steps [Ref. 1]:

1. Set each machine to its initial state, and all variables to their initial values. The initial set of reachable system states consists of only the initial system state; the initial graph is a single node representing this state.

2. From the current system state vector and variable values, determine which transitions are enabled. For each of these transitions, determine the system state which results from its execution. If this state (with the same enabled transitions)
has already been generated, then draw an arc from the current state to it, labelling the arc with the transition name. Otherwise, add the new system state to the graph, draw an arc from the current state to it, and label the arc with the name of the transition.

3. For each new state generated in step 2, repeat step 2. Continue until step 2 has been repeated for each system state thus generated, and no more new states are generated.

3. An Example of Protocol Specification and Analysis Using SCM

The specification of an imaginary ring-like network consisting of three machines similar to the CFSM example in the previous section is given in Figure 5. The specification consists of the finite state machines, the local and shared variables, and the predicate action table, shown in Table 1. The local variables are: \textit{in\_buff1}, \textit{in\_buff2}, \textit{in\_buff3}, \textit{out\_buff1}, \textit{out\_buff2}, and \textit{out\_buff3} and shown under the corresponding FSMs with their initial values. The shared variables are: \textit{CHAN1}, \textit{CHAN2}, and \textit{CHAN3} and shown between the two machines. The initial state of each machine is 0, with the shared variables and local variables are empty except the local variable \textit{out\_buff1}, which has data in it. \textit{E} in the predicate-action table shows the empty string. A character \textit{D} will be used to represent the data in the \textit{out\_buff1} local variable. Other notations in the predicate-action table are intuitive.

Each machine sends one message to the next machine and receives a message from the previous machine in clockwise direction forming a ring. The global reachability analysis, shown in Figure 6, has 12 states. The system state analysis, shown in Figure 7, has only 6 states. The subscripts in Figure 7 are used so that distinct system states having the same tuple (but not the outgoing transitions) may easily distinguished.
Figure 5: FSMs and variables for the example protocol

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>snd_data1</td>
<td>$\text{CHAN1} = E \land \text{out_buff1} \neq E$</td>
<td>$\text{CHAN1} \leftarrow \text{out_buff1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff1} \leftarrow E$</td>
</tr>
<tr>
<td>rcv_data3</td>
<td>$\text{CHAN3} \neq E$</td>
<td>$\text{in_buff1} \leftarrow \text{CHAN3}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff1} \leftarrow \text{in_buff1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{CHAN3} \leftarrow E$</td>
</tr>
<tr>
<td>snd_data2</td>
<td>$\text{CHAN2} = E \land \text{out_buff2} \neq E$</td>
<td>$\text{CHAN2} \leftarrow \text{out_buff2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff2} \leftarrow E$</td>
</tr>
<tr>
<td>rcv_data1</td>
<td>$\text{CHAN1} \neq E$</td>
<td>$\text{in_buff2} \leftarrow \text{CHAN1}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff2} \leftarrow \text{in_buff2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{CHAN1} \leftarrow E$</td>
</tr>
<tr>
<td>snd_data3</td>
<td>$\text{CHAN3} = E \land \text{out_buff3} \neq E$</td>
<td>$\text{CHAN3} \leftarrow \text{out_buff3}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff3} \leftarrow E$</td>
</tr>
<tr>
<td>rcv_data2</td>
<td>$\text{CHAN2} \neq E$</td>
<td>$\text{in_buff3} \leftarrow \text{CHAN2}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{out_buff3} \leftarrow \text{in_buff3}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$\text{CHAN2} \leftarrow E$</td>
</tr>
</tbody>
</table>
Thus, for this protocol we have 6 system states, and 12 global states. For more complex protocols, the difference between these numbers can be much more. For example, a sliding window protocol with a window size of 8 the system state analysis was shown to generate 165 states, while the full global analysis generated 11880 states [Ref. 1].
4. Summary

The SCM model has desirable properties which overcome some of the disadvantages of the CFSM model. One of the advantages of the SCM model is that it greatly reduces the number of state explosion through the use of system state analysis. In some cases, however, the system state analysis is not sufficient for protocol analysis, and some other method - such as global analysis - must be done. A problem with the system state analysis is the loops in the state machines which may cause an insufficient analysis. This problem is illustrated with an example in Chapter V.

Another advantage of SCM model is that it allows communication between machines in nonsequential manner, unlike a FIFO queue representation in the CFSM model. The SCM model specification is also easier to understand than the CFSM model for more complex protocols.
III. SIMPLE MUSHROOM: A PROGRAM FOR AUTOMATING CFSM REACHABILITY ANALYSIS

This Chapter and the next Chapter will describe a program called mushroom, which was written in the Ada programming language. Mushroom automates the reachability analysis of protocols specified by the CFSM and the SCM models. The Mushroom program was first developed as two separate programs. The first program called simple mushroom, automates the CFSM analysis. The second program automates either system state analysis (smart mushroom), or the full global analysis (big mushroom) for a protocol specified formally by the SCM model. The General structure of the Mushroom program is shown in Figure 8.

![Diagram of Mushroom program structure]

Figure 8: General structure of Mushroom program
The Simple Mushroom program, is described in this chapter in four sections: program structure, inputs to the program, generating the reachability analysis, and outputs of the program.

A. PROGRAM STRUCTURE

The Simple Mushroom program consists of Ada subprograms (procedures and functions), which are separate compilation units and subunits of compilation units. Related subprograms are also gathered in the same files. The compilation units of the program are shown in Table 2. Procedure main is the parent unit. All of the subprograms are the subunits of procedure main. [Ref. 13]

<table>
<thead>
<tr>
<th>Compilation Unit</th>
<th>Description</th>
<th>File name</th>
</tr>
</thead>
<tbody>
<tr>
<td>main (procedure)</td>
<td>This is the parent unit. Contains the main data structures, global variables, and the driver.</td>
<td>tmain.a</td>
</tr>
<tr>
<td>load_machine_array (procedure)</td>
<td>Builds the adjacency lists from FSMs.</td>
<td>tinput.a</td>
</tr>
<tr>
<td>read_in_file (procedure)</td>
<td>Parses the input FSM text file.</td>
<td>tinput.a</td>
</tr>
<tr>
<td>build_Gstate_graph (procedure)</td>
<td>Generates the reachability graph.</td>
<td>treachability.a</td>
</tr>
<tr>
<td>IsEqual (function)</td>
<td>Compares two global states for equality.</td>
<td>treachability.a</td>
</tr>
<tr>
<td>hash (function)</td>
<td>Generates an index number according to the hashing function.</td>
<td>treachability.a</td>
</tr>
<tr>
<td>clear_pointers (procedure)</td>
<td>Deallocates the dynamic memory space for another analysis.</td>
<td>treachability.a</td>
</tr>
<tr>
<td>find_tuple (function)</td>
<td>Searches the reachability graph for the equivalent tuples using external (open) hashing.</td>
<td>tsearch.a</td>
</tr>
<tr>
<td>Compilation Unit</td>
<td>Description</td>
<td>File Name</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------</td>
<td>-----------</td>
</tr>
<tr>
<td>clear_hash_array (procedure)</td>
<td>Clears the hash array and deallocates the memory.</td>
<td>tsearch.a</td>
</tr>
<tr>
<td>Print Queue (procedure)</td>
<td>Prints the FIFO queues.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>output_Gstate_transition (procedure)</td>
<td>Outputs the transition name.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>output_Gstate_node (procedure)</td>
<td>Outputs the machine states, unspecified receptions, and the states with deadlocks.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>output_machine_arrays (procedure)</td>
<td>Outputs the FSM description in a tabular format.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>output_unexecuted_transitions (procedure)</td>
<td>Outputs the unexecuted transitions.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>create_output_file (procedure)</td>
<td>Creates an output file for storing the analysis results.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>output_analysis (procedure)</td>
<td>Driver for the output subprograms.</td>
<td>toutput.a</td>
</tr>
<tr>
<td>system_call (procedure)</td>
<td>Interface procedure for Unix system calls via C.</td>
<td>tsystem.a</td>
</tr>
<tr>
<td>message_queues (package)</td>
<td>Implements the queue operations for the FIFO communication channels.</td>
<td>tqueues.a</td>
</tr>
<tr>
<td>pointer_queues (generic package)</td>
<td>Implements the queue operations for the pointer queue that stores the globals tuples temporarily.</td>
<td>queues_2.a</td>
</tr>
</tbody>
</table>
The method of splitting the program into separate compilation units has permitted a hierarchical program development.

B. INPUT

The CFSM specification of a protocol consists of only FSMs of the communicating machines. In the program, FSMs are represented with a text file. The user enters the directed graphs as a text file using some reserved words, numbers, and characters representing the machines, states and the transitions. The list of reserved words and the syntax for the FSM text description are shown in Figure 9 in Backus-Naur Form (BNF).

```
reserved_word ::= start
                | number_of_machines
                | machine
                | state
                | trans
                | initial_state
                | finish

number_of_machines <machine_number>
machine 1 | <machine_number>
state <state_number>
trans { | <message> | <next_state> | <next_machine>
initial_state <state_number> | <state_number> | [ <state_number> ] | [ <state_number> ]
                | [ <state_number> ] | [ <state_number> ] | [ <state_number> ] | [ <state_number> ]
<machine_number> ::= 21345678
<state_number> ::= 0123456789
<message ::= { <letter> } | [ { <letter> } ] | [ { <digit> } ] | [ { <digit> } ]
<next_state> ::= <state_number>
<next_machine> ::= 1 | <machine_number>
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

Figure 9: Syntax for the text description of FSM
As can be seen from Figure 9, the maximum number of machines allowed is eight, and the number of states for each machine can be from 0 to 50. Transition names must be at most three characters long and may be any combination of letters or digits. These constraints can be relaxed with slight modifications to the program, if necessary.

The input file for the example protocol in Chapter II for the CFSM model is shown in Figure 10. For example, “trans -D3 3 2” represents a transition from state 1 to state 3 (first number) in machine 1 sending ("-" sign) the message “D3” to machine 2. “Initial_state 1 1 1” means that the initial states of machine 1, machine 2, and machine 3 are state 1.

```
start
number_of_machines 3
machine 1
state 1
trans -D3 3 2
trans -D0 2 2
state 2
trans +D2 1 3
machine 2
state 1
trans +D3 3 1
trans +D0 2 1
state 2
trans -D1 1 3
machine 3
state 1
trans +D2 2 2
state 2
trans -D4 3 1
trans -D2 1 1
initial_state 1 1 1
finish
```

Figure 10: Text file description of the FSM

First, this file is parsed by read_in_file procedure and tokens are generated. Then, Load_machine_array procedure constructs an adjacency list which represents the FSMs.
The data structure for the adjacency list is shown below:

```lp-code
type cfsm_transition_type is (s,r,u);
type visit_type is (yes,no);
type state_type is range 0..50;
type next_machine_type is range 1..8;
type machine_array_record_type;
type Slink_type is access machine_array_record_type;
type machine_array_record_type is
  record
    transition : cfsm_transition_type := u;
    message : message_queue.message_queue_type;
    next_Mstate : state_type := 0;
    other_machine : next_machine_type := 1;
    visited : visit_type := no;
    Slink : Slink_type := null;
  end record;
type machine_array_type is array(state_type range 0..50) of Slink_type;
type system_array_type is array(next_machine_type range 1..8) of machine_array_type;
```

The adjacency list for the example protocol is depicted in Figure 12. This adjacency list is used for constructing the global reachability graph. The adjacency list contains all the necessary information for generating the global reachability graph.

The user also provides the name of the text input file and a file name for storing the analysis results. Input file name must end with ".fsm" extension to prevent confusion. The output file name must be no more than 20 characters long.

C. REACHABILITY ANALYSIS

After reading the input file the program starts generating the global reachability graph. The program uses the adjacency list and the initial state to construct the global reachability graph. Starting with the initial state, the new states are added and linked to the graph dynamically. The algorithm to construct the global reachability graph is given in Figure 13.

During the graph construction, the program also detects the global states with deadlocks and unspecified receptions. The program also finds the maximum message queue size and channel overflows. Analysis results are stored in the output file in parallel.
Figure 12: Adjacency list for the example ring protocol in Chapter II
with the graph construction. This prevents the traversal of the entire graph one more time at the end of the program and decreases the run time.

\[
\text{loop (main loop)}
\]
\[
\text{for index1 in 1 .. total_number_of_machines loop}
\]
\[
\text{place_holder(index1) := machine_array(index1)(M_state(index1))}
\]
\[
\text{while (place_holder(index) /= null) loop}
\]
\[
\text{loop}
\]
\[
\text{if (place_holder(index1).transition = s) then}
\]
\[
\text{Enqueue the message into the corresponding message queue}
\]
\[
\text{search the graph for this new global state tuple}
\]
\[
\text{if not found then create a new node and link to the graph}
\]
\[
\text{Enqueue this new node to the pointer_queue}
\]
\[
\text{else link the transition to found global state tuple}
\]
\[
\text{else}
\]
\[
\text{if (place_holder(index1).transition) = r and at least one of the message queues for}
\]
\[
\text{this machine is not empty then}
\]
\[
\text{find this message queue and Dequeue}
\]
\[
\text{search the graph for this new global state tuple}
\]
\[
\text{if not found then create a new node and link to the graph}
\]
\[
\text{Enqueue this new node to the pointer_queue}
\]
\[
\text{else link the transition to found global state tuple}
\]
\[
\text{end if;}
\]
\[
\text{place_holder(index1) := place_holder(index1).Slink}
\]
\[
\text{exit}
\]
\[
\text{end loop}
\]
\[
\text{end loop}
\]
\[
\text{if pointer_queue empty then}
\]
\[
\text{exit}
\]
\[
\text{else}
\]
\[
\text{Dequeue pointer_queue and update M_state for this new node}
\]
\[
\text{end if}
\]
\[
\text{end loop (main loop)}
\]

**Figure 13: Algorithm for generating global reachability graph for CFSM**

One of the most time consuming procedures is the search algorithm for detecting if a node was previously created. The previous version of the program [Ref. 8] used a depth first search / breadth first search in a recursive manner. In this program, the search is made
more efficient using a hashing algorithm. The hash function is obtained from the machine states of the global tuple which has provided an efficient mapping. Therefore, the complexity of the search algorithm is \(O(1)\) when the hash function generates a distinct index (no collision) and \(O(n)\) when the same index is generated, where \(n\) is the number of hash collisions for that state. In many sample runs of the program, the complexity was \(O(1)\) for about 30% of the global states, and 3 nodes had to be traversed on the average for 70% of the global states. The reachability analysis is limited by the storage capacity of the computer. The run time is also another factor that must be considered. The largest analysis carried out by the program thus far has generated about 160,000 states in 12 hours for a six machine protocol specification. Some alternative methods for improving the efficiency of the program and analysis size using other search techniques are discussed in Chapter VI.

The structure of a global node is shown in Figure 14. The maximum number of outgoing transitions is limited to 7, which can be increased if needed. Also, a maximum channel capacity of 6 messages is introduced to ensure that the analysis eventually stops.

**D. OUTPUT**

The program stores the analysis results in a file named by the user during the reachability graph construction. This file contains the specification in a tabular format, reachability graph and the results of the analysis consisting of the number of states generated, number of states analyzed, number of deadlocks, number of unspecified receptions, maximum message queue size and number of channel overflows. Global states with deadlocks and unspecified receptions are also marked in the reachability graph. The output file also lists the unexecuted transitions. A menu is displayed at the end of the analysis. From this menu the user has the option of displaying or printing the results or continuing the program for another analysis.
If the analysis generates more than 2000 states, the program gives an interim summary of the analysis and asks the user if they would like to continue. If the user wishes to continue, analysis proceeds in steps of 1000 states until the analysis ends or the user terminates the analysis (as long as memory is available). For analyzing large protocols, the number of states between these "stops" can be made larger (for example, increments of 5000 or 10000). The program output for the example protocol in Chapter II is given in Figure 15.

![Global state structure with outgoing transitions](image)

**Figure 14: Global state structure with outgoing transitions**

26
REACHABILITY ANALYSIS of : ring.fsm

SPECIFICATION

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

REACHABILITY GRAPH

1 [ 1,E,E, 1,E,E, 1,E,E]
   -d0  2 [ 2,E,E, 1,E,E, 1,E,E] 2
   -d3  2 [ 3,E,E, 1,E,E, 1,E,E] 3
2 [ 2,d0 ,E, 1,E,E, 1,E,E]
   +d0  3 [ 2,d3 ,E, 1,E,E, 1,E,E] 4
3 [ 3,d3,E, 1,E,E, 1,E,E]
   +d3  1 [ 3,d3,E, 3,E,E, 1,E,E] 5
4 [ 2,E,E, 2,E,E, 1,E,E]
   -d1  3 [ 2,E,E, 1,E,E, d1,E,E] 6
5 [ 3,E,E, 3,E,E, 1,E,E]********DEADLOCK condition************
6 [ 2,E,E, 1,E,E, d1,E,E]
   +d1  2 [ 3,E,E, 1,E,E, 2,E,E] 7
7 [ 2,E,E, 2,E,E, 2,E,E]
   -d2  1 [ 2,E,E, 1,E,E, 1,d2,E] 8
   -d4  1 [ 2,E,E, 1,E,E, 3,d4,E] 9
8 [ 2,E,E, 1,E,E, 1,d2,E]
   +d2  3 [ 1,E,E, 1,E,E, 1,d2,E] 1
9[2,E,E, 1,E,E, 3,d4,E]********Unspecified Reception***********

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)

Total number of states generated : 9
Number of states analysed : 9
Number of deadlocks : 1
Number of unspecified receptions : 1
Maximum message queue size : 1
Channel overflow :NONE

UNEXECUTED TRANSITIONS

********NONE*******

Figure 15: Program output for the example ring protocol
IV. SMART AND BIG MUSHROOM: A PROGRAM FOR AUTOMATING SCM REACHABILITY ANALYSIS

In this Chapter, programs that automate either system state analysis (smart mushroom), or the full global analysis (big mushroom) for a protocol specified by SCM are described. The program is described in four sections: general program structure, inputs to the program, generating the reachability graph, and outputs of the program.

A. PROGRAM STRUCTURE

Program structure of Smart Mushroom and Big Mushroom are similar to the structure of Simple Mushroom. The SCM model specification is more complicated than the CFSM specification, but this complexity in the specification brings some advantages to the analysis as mentioned in Chapter II. A protocol specified by the SCM model consists of FSMs, variable definitions, and predicate-action table, rather than just the FSMs as in CFSM model.

FSMs are entered into the program in the same manner as in Simple Mushroom program using a text file. The variable definitions and predicate-action table must also be entered into the program. The user enters these parts by completing Ada packages and subprograms using the templates provided.

The compilation units for the program are shown in Table 3. The user has access to the last four packages/subprograms. Once the user completes these subprograms using the templates and compiles them with the other compilation units, the analysis of the specified

1. Ada packages are one of the four forms of program unit, of which programs can be composed. The other forms are subprograms, task units, and generic units. Packages allow the specification of groups of logically related entities. In their simplest form packages specify pools of common object and type declarations. [Ref. 13]
protocol can be performed. Construction of the specification in the form of Ada packages and subprograms is explained in the next section.

**TABLE 3: SMART AND BIG MUSHROOM PROGRAM COMPILATION UNITS**

<table>
<thead>
<tr>
<th>Compilation Unit</th>
<th>Description</th>
<th>File name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Main (procedure)</td>
<td>This is the parent unit. Contains the main data structures, global variables, and the driver.</td>
<td>smain.a</td>
</tr>
<tr>
<td>load_machine_array (procedure)</td>
<td>Builds the adjacency lists from FSMs.</td>
<td>sinput.a</td>
</tr>
<tr>
<td>read_in_file (procedure)</td>
<td>Parses the input FSM text file.</td>
<td>sinput.a</td>
</tr>
<tr>
<td>build_Gstate_graph (procedure)</td>
<td>Generates the global reachability graph.</td>
<td>sg_reachability.a</td>
</tr>
<tr>
<td>build_system_state_graph (procedure)</td>
<td>Generates the system reachability graph.</td>
<td>sg_reachability.a</td>
</tr>
<tr>
<td>hash (function)</td>
<td>Generates an index number according to the hashing function.</td>
<td>sg_reachability.a</td>
</tr>
<tr>
<td>clear_pointers (procedure)</td>
<td>Deallocates the dynamic memory space for another analysis.</td>
<td>sg_reachability.a</td>
</tr>
<tr>
<td>search_for_Gtuple (function)</td>
<td>Searches the reachability graph for the equivalent global tuples using hashing.</td>
<td>sg_search.a</td>
</tr>
<tr>
<td>clear_hash_array (procedure)</td>
<td>Clears the hash array and deallocates the memory for global reachability analysis.</td>
<td>sg_search.a</td>
</tr>
<tr>
<td>search_for_Stuple (function)</td>
<td>Searches the reachability graph for the equivalent system tuples using hashing.</td>
<td>sg_search.a</td>
</tr>
<tr>
<td>clear_hs_hash_array (procedure)</td>
<td>Clears the hash array and deallocates the memory for system state analysis.</td>
<td>sg_search.a</td>
</tr>
<tr>
<td>output_Gstate_node (procedure)</td>
<td>Outputs the machine states, and states with deadlock for global reachability analysis.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>Compilation Unit</td>
<td>Description</td>
<td>File Name</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>output_sys_node (procedure)</td>
<td>Outputs machine states, and states with deadlock for system state analysis.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>output_Gstate_transition (procedure)</td>
<td>Outputs the transition name for global reachability analysis.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>output_sys_transition (procedure)</td>
<td>Outputs the transition name for system state analysis.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>output_unexecuted_transitions (procedure)</td>
<td>Outputs the unexecuted transitions.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>output_machine_arrays (procedure)</td>
<td>Outputs the FSM description in a tabular format.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>output_analysis (procedure)</td>
<td>Driver for the output subprograms.</td>
<td>sg_output.a</td>
</tr>
<tr>
<td>system_call (procedure)</td>
<td>Interface program for Unix system calls via C.</td>
<td>ssystem.a</td>
</tr>
<tr>
<td>queues (generic package)</td>
<td>Implements the queue operations for the pointer queue that stores the nodes temporarily.</td>
<td>queues.a</td>
</tr>
<tr>
<td>stacks (generic package)</td>
<td>Implements the stack operations for storing enabled transitions.</td>
<td>sstacks.a</td>
</tr>
<tr>
<td>definitions (package)</td>
<td>Includes user defined local and shared variables.</td>
<td>named by the user</td>
</tr>
<tr>
<td>Analyze_Predicates (procedure)</td>
<td>Determines the enabled transitions from the predicates.</td>
<td>named by the user</td>
</tr>
<tr>
<td>Action (procedure)</td>
<td>Executes the actions for the enabled transitions.</td>
<td>named by the user</td>
</tr>
<tr>
<td>output_gtuple (procedure)</td>
<td>Outputs the global state tuples in a format defined by the user.</td>
<td>named by the user</td>
</tr>
</tbody>
</table>
B. INPUT

The inputs to the program consists of three parts, as mentioned earlier. FSMs are entered using a text file representation as in *Simple Mushroom* program. Variables and predicate-action table are entered as Ada packages/subprograms. The user needs to complete these packages and subprograms by filling in templates provided.

The Ada package template for the variable declarations is called "definitions." The predicate-action table is entered using an Ada subprogram template which consists of one procedure named "Action" and two to eight procedures called "Analyze_Predicates_Machine*" according to the number of machines in the protocol. The "*" at the end of the procedure name is replaced by the corresponding machine number for each machine in the protocol.

After completing the templates described above, the user must compile these units with the other compilation units listed in Table 3. The program units can be compiled by entering a "make" command. The "make" command executes a list of shell commands in the "Makefile" file which contains the commands for compiling the program units according to their dependencies. After issuing the "make" command, the executable file is stored in a file named "scm." The "Makefile" is provided to the user with the mushroom program.

Each of these program units will be explained in the following subsections. The example ring protocol described in Chapter II is also used to illustrate how to complete the templates.

1. Finite State Machines

There are a few differences in the FSM description of *Smart* and *Big Mushroom* programs from *Simple Mushroom* program. The same reserved words are used to write the
FSM text file. These are listed in Figure 9. The syntax changes that must be made to this form are shown in Figure 16.

In the SCM model, explicit machine numbers to show which machine the message sent to or received from are not needed for the transition names. Since shared variables are used for communication between machines, this information is included in the predicate-action table. The FSM text file for the example ring protocol is shown in Figure 17.

```
trans <transition name> <next_state>
<transition name> ::= <identifier>
<identifier> ::= ([underline] l letter_or_digit)
<letter_or_digit> ::= <letter> | <digit>
```

Figure 16: Syntax changes for FSM description of SCM model

```
start
number_of_machines 3
machine 1
state 0
trans snd_data1 1
state 1
trans rcv_data3 0
machine 2
state 0
trans rcv_data1 1
state 1
trans snd_data2 0
machine 3
state 0
trans rcv_data2 1
state 1
trans snd_data3 0
initial_state 0 0 0
finish
```

Figure 17: Text file description of the example ring protocol
The FSM text file is read by the input procedures and the adjacency list, which is used during the construction of system and global reachability graphs is generated. The data structure for the adjacency list is shown in Figure 18.

```ada
visit_type is (yes, no);
type machine_array_record_type;
type Slink_type is access machine_array_record_type;
type machine_array_record_type is
  record
    transition : scm_transition_type := unused;
    next_Mstate : natural := 0;
    visited     : visit_type := no;
    Slink       : Slink_type := null;
  end record;
type machine_array_type is array(integer range 0 .. 50) of Slink_type;
type system_array_type is array (1 .. num_of_machine) of machine_array_type;
```

Figure 18: Data structure for the adjacency list.

2. Variable Definitions

The user defines the protocol variables in an Ada package named *definitions*. This package includes the local variables for each machine and the global variables, which are considered shared and allow communication between machines. A variable can be one of the Ada defined types such as: integer, array, string, record, character, boolean, etc. These types and their subtypes are used to define the protocol variables.

The template for the *definitions* package is given in Figure 19. The shaded areas show where the variables of the protocol are inserted by the user. Additional type declarations should be placed before the machine type declarations.

The variable declarations for the example ring protocol is also shown in Figure 20. The local variables of the protocol are: *in_buff1*, *in_buff2*, *in_buff3*, *out_buff1*, *outBuff2*, and *out_buff3*. The shared variables are: *CHAN1*, *CHAN2* and *CHAN3*. The type definition, *Dummy_type* is placed in each of the local variable declarations of
machines in case the protocol has less than eight machines. When declaring the local variables for each machine, this dummy variable can be deleted from the corresponding machine. The initial values of the variables are also assigned with the variable declarations.

```plaintext
with TEXT_IO;
use TEXT_IO;
package definitions is
  num_of_machines: constant := #8;  
  type scm_transition_type is (
    transition names of FSMs
  );
  type dummy_type is range 1..255;
  type machine1_state_type is
    record
      dummy: dummy_type;
    end record;
  type machine8_state_type is
    record
      dummy: dummy_type;
    end record;
  type global_variable_type is
    record
    end record;
end definitions;
```

Figure 19: Template for definitions package

3. Predicate-Action Table

The predicate-action table is represented by a number of subprograms as separate compilation units. These subprograms are named Analyze_PREDICATES and are used to determine the enabled transitions for each machine. The procedure named Action executes the actions to be taken for the corresponding enabled predicates. There is one
*Analyze_Predicates* procedure for each machine and one *Action* procedure for the protocol.

The template for the *Analyze_Predicates* procedure is shown in Figure 21.

```plaintext
with TEXT_IO;
use TEXT_IO;
package definitions is
  num_of_machines : constant := 3;
  type scm_transition_type is (snd_data1, rcv_data3, snd_data2,
                             rcv_data1, snd_data3, rcv_data2, unused);
  type buffer_type is (D, E);
  package buf_enum_io is new enumeration_io (buffer_type);
  use buf_enum_io;
  type dummy_type is range 1..255;
  type machine1_state_type is record
    out_buff1 : buffer_type := D;
    in_buff1 : buffer_type := E;
  end record;
  type machine2_state_type is record
    out_buff2,
    in_buff2 : buffer_type := E;
  end record;
  type machine3_state_type is record
    out_buff3,
    in_buff3 : buffer_type := E;
  end record;
  type machine4_state_type is record
    dummy : dummy_type;
  end record;
  type machine5_state_type is record
    dummy : dummy_type;
  end record;
  type machine6_state_type is record
    dummy : dummy_type;
  end record;
  type machine7_state_type is record
    dummy : dummy_type;
  end record;
  type global_variable_type is record
    CHAN1,
    CHAN2,
    CHAN3 : buffer_type := E;
  end record;
end definitions;

Figure 20: Completed *Definitions* package for the example ring protocol
```
The user completes the template for each state of the machines. For each machine state there is one “when” statement. “If” statements specify the predicates for possible transitions from the current state. The “Push” statement stores these transitions in the stack. Since more than one transition can be enabled in some states, a stack is used to store all possible transitions. The “s” parameter, in the formal parameter list of the procedure, passes the machine state; and the “w” parameter passes the stack name to the procedure. The file for the example ring protocol is given in Figure 22.

The template for the Action procedure is shown in Figure 23. The enabled transitions are passed into this procedure through the “in_transition” formal parameter and the necessary changes are made to the local and shared variables by the Action procedure. The “out_system_state” parameter passes the changed protocol variables to the calling procedure. The completed Action procedure is shown in Figure 24. Text in boldface shows the user defined parts.
separate (main)
procedure Analyze_Predicates_Machine1(local : machine1_state_type; GLOBAL: global_variable_type;
   s : natural; w : in out transition_stack_package.stack) is
begin
   case s is
      when 0 =>
         if (GLOBAL.CHAN1 /= E) then
            Push(w.rev_data1);
         end if;
      when 1 =>
         if ((GLOBAL.CHAN2 = E) and (local.out_buff2 /= E)) then
            Push(w.rev_data2);
         end if;
      when others =>
         null;
   end case;
end Analyze_Predicates_Machine1;
separate (main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type; GLOBAL: global_variable_type;
   s : natural; w : in out transition_stack_package.stack) is
begin
   case s is
      when 0 =>
         if (GLOBAL.CHAN1 /= E) then
            Push(w.rev_data1);
         end if;
      when 1 =>
         if ((GLOBAL.CHAN3 = E) and (local.out_buff2 /= E)) then
            Push(w.rev_data2);
         end if;
      when others =>
         null;
   end case;
end Analyze_Predicates_Machine2;
separate (main)
procedure Analyze_Predicates_Machine3(local : machine3_state_type; GLOBAL: global_variable_type;
   s : natural; w : in out transition_stack_package.stack) is
begin
   case s is
      when 0 =>
         if (GLOBAL.CHAN2 /= E) then
            Push(w.rev_data2);
         end if;
      when 1 =>
         if ((GLOBAL.CHAN3 = E) and (local.out_buff3 /= E)) then
            Push(w.rev_data3);
         end if;
      when others =>
         null;
   end case;
end Analyze_Predicates_Machine3;
separate (main)
procedure Analyze_Predicates_Machine4(local : machine4_state_type; GLOBAL: global_variable_type;
   s : natural; w : in out transition_stack_package.stack) is
begin
   null;
end Analyze_Predicates_Machine4;
begin
   null;
end Analyze_Predicates_Machine5;
begin
   null;
end Analyze_Predicates_Machine6;
begin
   null;
end Analyze_Predicates_Machine7;
begin
   null;
end Analyze_Predicates_Machine8;

Figure 22: Completed Analyze_Predicates procedures for the example ring protocol
separate(main)

procedure Action (in_system_state : in out GameState_record_type;
in_transition : in out scm_transition_type;
out_system_state : in out GameState_record_type) is
begin

case in_transition is
  when ENABLED_TRANSITION =>
    action_taken;

  when others =>
    put("Error in the action procedure");
end case;
end Action;

Figure 23: Template for Action procedure

separate(main)

procedure Action (in_system_state : in out GameState_record_type;
in_transition : in out scm_transition_type;
out_system_state : in out GameState_record_type) is
begin

case in_transition is
  when (md_data1) =>
    out_system_state.GLOBAL_VARIABLES.CHAN1 :=
in_system_state.machine1_state.out_buff1;
    out_system_state.machine1_state.out_buff1 := E;

  when (rcv_data3) =>
    out_system_state.machine1_state.in_buff1 :=
in_system_state.GLOBAL_VARIABLES.CHAN3;
    out_system_state.machine1_state.out_buff1 := out_system_state.machine1_state.in_buff1;
    out_system_state.GLOBAL_VARIABLES.CHAN3 := E;

  when (md_data2) =>
    out_system_state.GLOBAL_VARIABLES.CHAN2 :=
in_system_state.machine2_state.out_buff2;
    out_system_state.machine2_state.out_buff2 := E;

  when (rcv_data1) =>
    out_system_state.machine2_state.in_buff2 :=
in_system_state.GLOBAL_VARIABLES.CHAN1;
    out_system_state.machine2_state.out_buff2 := out_system_state.machine2_state.in_buff2;
    out_system_state.GLOBAL_VARIABLES.CHAN1 := E;

  when (md_data3) =>
    out_system_state.GLOBAL_VARIABLES.CHAN3 :=
in_system_state.machine3_state.out_buff3;
    out_system_state.machine3_state.out_buff3 := E;

  when (rcv_data2) =>
    out_system_state.machine3_state.in_buff3 :=
in_system_state.GLOBAL_VARIABLES.CHAN2;
    out_system_state.machine3_state.out_buff3 := out_system_state.machine3_state.in_buff3;
    out_system_state.GLOBAL_VARIABLES.CHAN2 := E;

  when others =>
    put_line("There is an error in the Action procedure");
end case;
end Action;

Figure 24: Completed Action procedure for the example protocol
C. REACHABILITY ANALYSIS

The process of generating the set of all states reachable from the initial state is called reachability analysis. The program is capable of generating both the global and system reachability analyses separately for a protocol specified formally by the SCM model.

The user selects either global reachability analysis or system state analysis from a menu. During the graph construction, the program also detects the states with deadlock condition. Analysis results are stored in the output file named “rgraph.dat” in parallel with the graph construction.

Generating the global reachability analysis and system state analysis will be described in the following subsections.

1. Global Reachability Analysis

The structure of the global node representation used for the program is shown in Figure 25. This node structure also includes the outgoing transitions. The maximum number of outgoing transitions is limited to 7, which can be increased if necessary. The shared variables are stored in the global variables variable and local variables are stored separately for each machine in the machine_state* variables.

The initial global state is constructed from both the FSM text file and the initial values of the variables assigned in the definitions package. All the outgoing transitions are set to null initially. Starting with the initial global state, new nodes are added and linked to the graph. The algorithm for generating the global reachability graph is the same as the algorithm given for the system state analysis in Chapter II except that the “system states” must be replaced by “global states.” Figure 26 shows a pseudo-code algorithm to construct the global reachability graph.
The program uses hashing for searching the reachability graph which increases the run time efficiency of the program. The reachability analysis is limited by the storage capacity of the computer and by the run time as in *Simple Mushroom* program. For example, the program generated 31,460 global states for a sliding window protocol of two machines defined in [Ref. 1] for a window size of 10. The run time for this example was about 10 minutes. The number of states and the run time increases greatly as the number of machines in the protocol increases and the protocol specifications become larger.

---

**Figure 25: Global state structure with outgoing transitions**

<table>
<thead>
<tr>
<th>GTUPLE</th>
<th>LINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>machine_state</td>
<td>1 2 3 4 5 6 7 8</td>
</tr>
<tr>
<td>global variables</td>
<td></td>
</tr>
<tr>
<td>machine1_state</td>
<td></td>
</tr>
<tr>
<td>machine2_state</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>machine5_state</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>.</td>
<td></td>
</tr>
<tr>
<td>machine8_state</td>
<td></td>
</tr>
</tbody>
</table>

| 1 | Gtransition |
| 2 | new node |
| 7 | Glink visited |
loop (main loop)
  for index1 in total_number_of_machines loop
    position_holder(index1) := machine_array(index1)(M_state(index1))
    Determine the enabled transitions for the machine(index1) and push into transition_stack
    While not Empty(transition_stack) loop
      while (position_holder(index1) /= null) loop
        Traverse the machine arrays for each enabled transition in the stack
        if a transition found in the machine arrays create a temporary node resulting from this transition
          call Action procedure to make the necessary changes to the variables of this node
        Search the graph for this node
          if a node not found then
            insert and link the node to the graph
          Enqueue the node into the Gpointer_queue
        else
          link the node to the graph
        end if
      else
        position_holder(index1) := position_holder(index1).Slink
      end if
    end loop
    if not Empty(transition_stack) and a transition not found in the machine arrays
      pop the stack
    end if;
  end loop
  if Gpointer_queue Empty then
    exit
  else
    Dequeue Gpointer_queue
    Update M_state for this new node
  end if
end loop (main loop)

Figure 26: Algorithm for generating global reachability graph for Big Mushroom

2. System State Analysis

The steps in constructing the system state graph are detailed in Chapter II. The structure of a system state is shown in Figure 27. Since the variables are not part of the system state, system state nodes are much smaller than the global state nodes. However, in order to determine the enabled transitions, variables are still needed for each node in the graph. The program stores the variables in secondary storage, instead of keeping them as a
part of the node, which decreases the amount of primary memory used and allows the
analysis of larger and more complex protocols.

The pseudo-code algorithm for constructing the system reachability graph is
shown in Figure 28.

<table>
<thead>
<tr>
<th>system_state_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>STUPLE</td>
</tr>
<tr>
<td>machine_state</td>
</tr>
<tr>
<td>subscript</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Transition</td>
</tr>
<tr>
<td>Syslink</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>LINK</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

Figure 27: System state structure for Smart Mushroom program

D. OUTPUT

The program stores the results of the analysis in a file named “rgraph.dat.” This file
contains FSMs in a tabular format, system/global reachability graph, and the results of the
analysis consisting of number of states generated, number of states analyzed, and number
of deadlocks. Unexecuted transitions are also listed at the end of the analysis.

Since each protocol specification has different variables, the user also has the
flexibility to output the desired variables. This is done in a similar manner to the predicate-
action table and variable definitions representation explained earlier using an Ada
procedure template. The template for the Output_Gtuple procedure is shown in Figure 29.
The user completes the template with Ada "put" statements for outputting the global states. Since the system state tuples do not include the variables, there is no need to define an output format for system reachability graph.

```ada
loop (main loop)
  for index1 in 1 .. num_of_trans loop
    if parent_Sstate.link(index1).transition /= unused then
      for index2 in 1 .. total_num_of_machines loop
        position_holder := machine_array(index2) (M_state(index2))
        while position_holder /= null loop
          if position_holder.transition = parent_Sstate.link(index1).transition then
            create a temporary system state and store the corresponding variables
            determine the enabled outgoing transitions
            search the system state graph for this node
            if node not found then
              insert the node and link to the graph
            Enqueue the node into sys_pointer_queue
            else
              link the node to the graph
              exit
            end if
        end loop
        if an enabled transition found in the machine arrays then
          exit
        end if
      end loop
      if sys_pointer_queue empty then
        exit
      else
        Dequeue the sys_pointer_queue
        update M_state
      end if
    end if
  end loop
end loop (main loop)
```

Figure 28: Algorithm for generating system state graph for *Smart Mushroom* program

The completed template for the *output_Gtuple* procedure is also given in Figure 30. As in *Simple Mushroom* program, if the analysis generates more than 2000 states, the program gives an interim summary and continues in steps as described in Chapter III. At the end of the program, the user can display/print the results or continue with another
system/global state analysis selecting the desired options from the menu. The output of the program for the example ring protocol is given in Figures 31 and 32.

```verbatim
separate (main)
procedure output_Gtuple (tuple : in out Gstate_record_type) is begin
  if print_header then
    new_line(2);
    set_col(5);
    print_header := false;
  else
    put("[" & integer'image (tuple.machine_state (1)));
    put(" , ");
    put("[" & integer'image (tuple.machine_state (2)));
    put(" , ");
    put("[" & integer'image (tuple.machine_state (8)));
    put(" , ");
  end if;
end output_Gtuple;
```

Figure 29: Template for `output_Gtuple` procedure
separate (main)
procedure output_Gtuple(tuple : in out Ostate_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(5);
    put_line("m1(in_buff1,out_buff1), m2(in_buff2,out_buff2), m3(in_buff3,out_buff3),
             (CHAN1,CHAN2,CHAN3)"");
    print_header := false;
  else
    put("[" & integer'image(tuple.machine.state(1)));
    put("",");
    buff_enum_io.put(tuple.machine.state.in_buff1);
    put("",");
    buff_enum_io.put(tuple.machine.state.out_buff1);
    put("",");
    buff_enum_io.put(tuple.machine.state.in_buff2);
    put("",");
    buff_enum_io.put(tuple.machine.state.out_buff2);
    put("",");
    buff_enum_io.put(tuple.machine.state.in_buff3);
    put("",");
    buff_enum_io.put(tuple.machine.state.out_buff3);
    put("",");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN1);
    put("",");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN2);
    put("",");
    buff_enum_io.put(tuple.GLOBAL_VARIABLES.CHAN3);
    put("]");
  end if;
end output_Gtuple;

Figure 30: Completed output_Gtuple procedure for the example protocol
REACHABILITY ANALYSIS of :ring.com
SPECIFICATION

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

GLOBAL REACHABILITY GRAPH

m1(in_buff1,out_buff1),m2(in_buff2,out_buff2),m3(in_buff3,out_buff3),(CHAN1,CHAN2,CHAN3)

0 [0, E, D, 0, E, E, 0, E, E, E, E] snd_data1 1
1 [1, E, E, 0, E, E, 0, E, E, E, E] rcv_data1 2
2 [1, E, E, 1, D, D, 0, E, E, E, E] and_data2 3
3 [1, E, E, 0, D, E, 0, E, E, D, E] rcv_data2 4
4 [1, E, E, 0, D, E, 1, D, D, E, E] and_data3 5
5 [1, E, E, 0, D, E, 0, D, E, E, E] rcv_data3 6
6 [0, D, D, 0, D, E, 0, D, E, E, E] and_data4 7
7 [1, D, E, 0, D, E, 0, D, E, E, E] rcv_data4 8
8 [1, D, E, 1, D, D, 0, D, E, E, E] and_data5 9
9 [1, D, E, 0, D, E, 0, D, E, E, E] rcv_data6 10
10 [1, D, E, 0, D, E, 1, D, D, E, E] and_data7 11
11 [1, D, E, 0, D, E, 0, D, E, E, E, D] rcv_data8 6

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)

Number of states generated : 12
Number of states analysed : 12
Number of deadlocks : 0

UNEXECUTED TRANSITIONS
*****WORST*****

Figure 31: Program output for global reachability analysis

46
REACHABILITY ANALYSIS of :ring.scm

SPECIFICATION

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-----------------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

SYSTEM REACHABILITY GRAPH
0 [ 0, 0, 0 ] 0 snd_data1 1
1 [ 1, 0, 0 ] 0 rcv_data1 2
2 [ 1, 1, 0 ] 0 snd_data2 3
3 [ 1, 0, 0 ] 1 rcv_data2 4
4 [ 1, 0, 1 ] 0 snd_data3 5
5 [ 1, 0, 0 ] 2 rcv_data3 0

SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)
Number of states generated :6
Number of states analysed :6
Number of deadlocks : 0

UNEXECUTED TRANSITIONS
*****NONE*****

Figure 32: Program output for system state analysis²

2. The number next to "|" sign shows the subscripts that is explained in Chapter II.
V. EXAMPLES FOR USING THE MUSHROOM PROGRAM

In this Chapter, the programs Simple Mushroom, Big Mushroom, and Smart Mushroom are demonstrated with several examples.

The Simple Mushroom program will be used to analyze a simple example four machine protocol which illustrates some important aspects of the program, such as detecting unspecified receptions, unexecuted transitions etc. Also, the information transfer phase of a full duplex LAP-B protocol specified by the CFSM model will be analyzed. This protocol illustrates a larger and more complex analysis.

The Big Mushroom and Smart Mushroom programs will be used to analyze the GO BACK N protocol with a window size of 10, and the Token Bus protocol, which illustrates some important aspects of the system state analysis.

A. CFSM MODEL

1. A Simple Four Machine Protocol

The specification of the protocol using the CFSM model is shown in Figure 33. Each of the machines sends/receives a message/acknowledgment from another machine. Machines 2 and 3 also have another send transition from state 1 to state 3. The FSM description of the protocol is shown in Figure 34, and analysis results obtained by the Simple Mushroom program are shown in Figure 35. The analysis generated 36 global states. There are three unspecified receptions and one unexecuted transition. No deadlocks or channel overflows are recorded. The maximum channel size is 2. These results are obtained by simply entering the FSM text file into the program. This analysis would be very cumbersome to do manually, even for a simple specification like this one.
Figure 33: Specification for the example four machine protocol

Figure 34: FSM text file for the example protocol
### REACHABILITY ANALYSIS of: four_machine.fsm

#### SPECIFICATION

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Other machine</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td># D</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td># A</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 4 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

#### REACHABILITY GRAPH

```
1 [1,E,E, 1,E,E, 1,E,E, 1,E,E]  -D 2 [2,D,E, 1,E,E, 1,E,E, 1,E,E]  3
-D 3 [1,E,E, 1,E,E, 3,E,D, 1,E,E, 1,E,E]  4
-A 1 [1,E,E, 1,E,E, 3,A, 1,E,E, 1,E,E]  5
2 [2,D,E, 1,E,E, 1,E,E, 1,E,E]  -D 3 [2,D,E, 3,E,D, 1,E,E, 1,E,E]  6
-A 1 [2,D,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  7
3 [1,E,E, 3,E,D, 1,E,E, 1,E,E]  -D 2 [2,D,E, 3,E,E, 1,E,E, 1,E,E]  8
-A 1 [2,D,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  9
0D 2 [1,E,E, 3,E,E, 2,E,E, 1,E,E]  10
4 [1,E,E, 1,E,E, 3,A, 1,E,E, 1,E,E]  -D 2 [2,D,E, 1,E,E, 3,A, 1,E,E, 1,E,E]  11
-A 1 [2,D,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  12
5 [2,D,E, 3,E,D, 1,E,E, 1,E,E]  -A 1 [2,D,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  13
0D 2 [2,D,E, 3,E,E, 2,E,E, 1,E,E]  14
6 [2,E,E, 2,E,E, 1,E,E, 1,E,E]  -A 1 [2,E,E, 2,E,E, 3,A, 1,E,E, 1,E,E]  15
7 [2,D,E, 1,E,E, 3,A, 1,E,E, 1,E,E]  16
4A 3 [1,D,E, 1,E,E, 3,A, 1,E,E, 1,E,E]  17
-D 3 [2,D,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  18
-A 1 [2,D,E, 2,E,E, 3,A, 1,E,E, 1,E,E]  19
8 [1,E,E, 3,E,D, 3,A, 1,E,E, 1,E,E]  -D 2 [2,D,E, 3,E,D, 3,A, 1,E,E, 1,E,E]  20
-D 2 [2,D,E, 3,E,D, 3,A, 1,E,E, 1,E,E]  21
9 [1,E,E, 3,E,E, 2,E,E, 1,E,E]  22
-D 2 [2,D,E, 3,E,E, 2,E,E, 1,E,E]  23
-D 4 [1,E,E, 3,E,E, 1,E,E, 1,E,E]  24
-A 3 [1,D,E, 3,E,E, 3,E,E, 1,E,E]  25
10 [2,D,E, 3,E,E, 3,E,E, 1,E,E]  -D 3 [2,D,E, 3,E,E, 1,E,E, 1,E,E]  26
-D 2 [2,D,E, 1,E,E, 1,E,E, 1,E,E]  27
11 [2,D,E, 3,E,D, 1,E,E, 1,E,E]  28
0A 3 [1,E,E, 3,E,E, 3,A, 1,E,E, 1,E,E]  29
12 [2,E,E, 3,E,E, 3,E,E, 1,E,E]  30
4A 3 [1,E,E, 2,E,E, 3,E,E, 1,E,E]  31
13 [1,D,E, 1,E,E, 3,E,E, 1,E,E]  32
-D 2 [2,D,E, 1,E,E, 1,E,E, 1,E,E]  33
-D 3 [1,D,E, 3,E,D, 1,E,E, 1,E,E]  34
0D 1 [1,E,E, 2,E,E, 3,E,E, 1,E,E]  35
50
```
Figure 35: Program output for the example protocol
2. Analysis of Information Transfer Phase of the LAP-B Protocol

In this Section, analysis of a Data Link Control (DLC) protocol is described using the *Simple Mushroom* program. The LAP-B protocol is modeled and analyzed with CFSM model [Ref. 14]. A simplified analysis of the information transfer phase of the protocol, which includes only I-frames with a window size of 2, will be described below.

This analysis is important in two ways. First, it verifies that the program is correct by obtaining the same analysis results as in [Ref. 14]. Secondly, it is a good example to show that the total number of global states can be very large, even for such a limited protocol. The description of the information transfer phase is explained below as it appears in [Ref. 14].

The network nodes, which are connected by the protocol, consist of a Data Terminal Equipment (DTE) and a Data Circuit Terminating Equipment (DCE). In this model, DTE and DCE are considered process 1 and process 2 respectively. Each of these processes are also modeled as three sub-processes: Sender, Receiver and Frame Assembler Disassembler (FAD), which are numbered as 1 or 2 according to their process numbers.

Figure 36 shows the processes and how they are connected. The FAD process combines data blocks from the Sender with acknowledgments from the Receiver, into complete I-frames and sends the I-frames to the FAD of the other process. The FAD also breaks up the I-frames received from the other FAD and sends the acknowledgment to the Sender, and data blocks to the Receiver.

I-frames are expressed by the notation “Inm”, where n is the send sequence number N(S), and m is the receive sequence number N(R). The message “Di” is a data block sent from the Sender to the FAD, or from the FAD to the receiver; it is the data block which is to be placed in, or which is taken out of, the I-frame. The “i” in “Di” is the send sequence number. The message “Ai” is an acknowledgment with a receive sequence number of i.
Figure 36: Processes for the Information Transfer Phase
The finite state machines for the Sender, Receiver and FAD of the DTE are shown in Figures 37, 38 and 39. The FSMs for the DCE are the same except that FAD1, RECEIVER1, and SENDER1 must be replaced with FAD2, RECEIVER2, and SENDER2 respectively. Since no RR-frames are used, I-frames can only be acknowledged by receiving an N(R) from an incoming I-frame.

As an example, suppose the DTE Sender1 has 3 data blocks to send. It can go from state 1 to state 2, sending “D0,” and then to state 3, sending the second block as “D1.” At this point, 2 data blocks are outstanding, so it must wait for an acknowledgment of at least one of them before sending the third.

The DTE FAD1 process, initially in state 1, will receive the D0 from Sender1 and enter state 2. It then sends an “enquiry” to the Receiver1 to get the latest acknowledgment, an N(R), for the data blocks received from the DCE.

Since no data blocks have been received by the DTE yet, Receiver1 will respond with an “A0.” FAD1 will receive the A0, and will transition from state 8 to 11. The FAD1 will then return to state 1 sending the I-frame “I00.” Similarly, the FAD1 will receive the second data block, D1, and transmit it as “I10” after combining with “A0.”

FAD2 will receive the “I00” frame first, entering state 20. It then splits this I-frame and sends the “D0” to Receiver2, and “A0” to Sender2.

Sender2 is in state 1, and simply discards this “A0.” Receiver2 is in state 1, accepts the “D0” data block and transitions to state 2.

Similarly, The DCE FAD2 process receives the “I10” message, and sends the “D1” to Receiver 2, and “A0” to Sender 2. Sender 2 will discard the “A0”, remaining in state 1, and Receiver 2 will receive “D1,” transitioning to state 3.

Suppose at this point a user data block becomes available to send at the DCE. It will send an “I02” frame across the data link to the DTE; and upon receiving the I02, the DTE will now be able to send the third user data block.
Figure 37. Sender 1 (Ref. 14)
For the automated analysis of the protocol, the FSMs in Figures 37, 38, and 39 are converted to a text file and entered into the program as shown in Appendix A. The transition names in this text file are the same as in the FSM diagrams, such as "+I00", "+D0" etc. In order to save memory and generate a larger number of states in the analysis, the transition names can be abbreviated to single characters at the time of the analysis as shown below:

\[
\begin{align*}
D0 & \rightarrow X & I00 & \rightarrow 1 \\
D1 & \rightarrow Y & I01 & \rightarrow 2 \\
D2 & \rightarrow Z & I02 & \rightarrow 3 \\
A0 & \rightarrow A & I10 & \rightarrow 4 \\
A1 & \rightarrow B & I11 & \rightarrow 5 \\
A2 & \rightarrow C & I12 & \rightarrow 6 \\
ENQ & \rightarrow Q & I20 & \rightarrow 7 \\
      &      & I21 & \rightarrow 8 \\
      &      & I22 & \rightarrow 9 \\
\end{align*}
\]

The amount of memory available and the CPU time are always a concern for a full reachability analysis. The program output for the analysis is partially given in Appendix A. Because of the size of the analysis, only a very small portion of the reachable states are included in the output. The total number of global states generated for the information phase was 73391. There were no unspecified receptions, unexecuted transitions, and channel overflows. The maximum channel length was 6. A deadlock condition was found at state 17034 where all the channels were empty and Sender1, Receiver1, FAD1, FAD2, Sender2, Receiver2 were in states 3, 3, 1, 1, 3, 3 respectively. This state deadlock is expected since RR-frames are not included in the analysis. A more detailed explanation including the RR-frames in the protocol is given in [Ref. 14]. The reader may note that the results of the analysis exactly match with the results reported in Reference 14. The deadlock state found in Reference 14 was 67699, which was recorded at state 17034 in this analysis. However, the global states are the same for both analyses. The Simple Mushroom program uses a Breadth-First Search algorithm for choosing the states from the work set.
(i.e., global states that are generated, but have not been analyzed yet). The protocol verifier PROVE, used in Reference 14 might be using a Depth First Search approach, which would result in a different global state number.

The protocol, including the RR-frames, was also entered into the program, but the program could not complete the analysis due to insufficient computer memory. In this analysis, 153565 global states were generated. No unspecified receptions, deadlocks or channel overflows were recorded for the analyzed portion of the protocol. The maximum channel size reached was 4. The program completed the analysis in 11 hours 51 minutes on a Sun SPARC station.

B. SCM MODEL

1. Go Back N

The first protocol selected for analysis using the Big Mushroom and Smart Mushroom programs is a 1-way data transfer protocol with a variable window size, which is essentially a subset of the High-level Data Link Control (HDLC) class of protocols. This protocol is modeled and analyzed with the SCM model in [Ref. 1]. The same specification will be used here and an automated analysis will be described using the programs developed for a window size of 10. The specification is summarized below:

There are two machines in the system, a sender (m_1) and a receiver (m_2). The sender sends data blocks to the receiver, which are numbered sequentially, 0, 1,..., w, 0, 1, ... for a window size of w. As in HDLC, the maximum number of data blocks which can be sent without receiving an acknowledgment is w, the window size. The receiver, m_2, receives the data blocks and acknowledges them by sending the sequence number of the next data block expected (which is stored in local variable exp). The shared variables DATA and SEQ are used to pass messages from sender to receiver, and the shared variable
ACK is used to pass acknowledgments back to the sender. The receiver may acknowledge any number of blocks received up to the window size. Upon receiving the acknowledgment, the sender must be able to deduce how many data blocks are being acknowledged. This is done by observing the difference between the values of the received acknowledgment and the sequence number of the last data block sent.

The general specification of the protocol is given in Figure 40 and in Table 4. Initially, both sender and receiver are in state 0, arrays DATA and SEQ are empty, and ACK is empty. The domains of DATA, Rdata and Sdata are not specified; these are used to hold user data blocks. Sdata and Rdata are the interface or access points of the higher layer (user) protocol. The local variables for the sender are Sdata, used to store data blocks, seq, used to store the sequence number of the next data block to be sent out, and i, used as an index into the DATA and SEQ arrays. Initially seq is set to 0, and i is set to 1. The local variables of the receiver are Rdata, exp, and j. Rdata is used to receive and store incoming data blocks, exp to hold the expected sequence number of the next incoming data block, and j is an index into the shared arrays DATA and SEQ.

The states of both sender and receiver are numbered 0, 1, ..., w, and each state has an easily recognized intuitive meaning. If the sender is in state 0, then all data blocks sent to date have been received by the receiver, so a full window size of w data blocks may be sent without waiting for an acknowledgment. If m₁ is in state w, then a full window of blocks have been sent, so the sender can only wait for the acknowledgment from the receiver.

If the receiver, m₂, is in state 0, then all received data blocks have been acknowledged. If in state w, then a full window of data blocks have been received, but not acknowledged. Whenever the receiver sends an acknowledgment, all data blocks received up to that point are acknowledged.
Figure 40: State machines and variables for Go Back N

**TABLE 4: PREDICATE-ACTION TABLE FOR GO BACK N**

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>(-D)</td>
<td>(\text{DATA}(i) = \varepsilon \land \text{SEQ}(i) = \varepsilon)</td>
<td>(\text{DATA}(i) \leftarrow \text{Sdata}(i))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{SEQ}(i) \leftarrow \text{seq})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{inc}(i, \text{seq}))</td>
</tr>
<tr>
<td>(+A_k)</td>
<td>(\text{ACK} \oplus k = \text{seq} \land \text{ACK} \neq \varepsilon) ((\text{next state} : k))</td>
<td>(\text{ACK} \leftarrow \varepsilon)</td>
</tr>
<tr>
<td>(0 \leq k \leq w)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(+D)</td>
<td>(\text{DATA}(j) \neq \varepsilon \land \text{SEQ}(j) = \text{exp})</td>
<td>(\text{Rdata} \leftarrow \text{DATA}(j))</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{DATA}(j), \text{SEQ}(j) \leftarrow \varepsilon)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{inc} (j, \text{exp}))</td>
</tr>
<tr>
<td>(-A)</td>
<td>(\text{DATA}(j) = \varepsilon)</td>
<td>(\text{ACK} \leftarrow \text{exp})</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{Rdata} \leftarrow \varepsilon)</td>
</tr>
</tbody>
</table>
The enabling predicate and action for each transition are shown in Table 4. The label or transition name is the leftmost column, the enabling predicate in the middle, and the corresponding action on the right. There are four basic types of transitions. In the sender, \( m_1 \), the \(-D\) transition transmits a data block by placing it into the shared variable \( DATA(i) \), and the sequence number into \( SEQ(i) \). The send is enabled whenever those variables are empty. (The interaction between the sender and the user, or higher layer, is implicit, and not specified here). The \( inc \) operation increments its arguments, if less than their maximum value, in which case it resets them to the minimum value. The operator \( \oplus \) represents the \( inc \) operation repeated \( k \) times, if the argument is \( k \) and the symbol \( e \) denotes the empty value. The receive transition in the receiver, \( m_2 \), is enabled whenever a data block of the appropriate sequence number is in the \( j \)th element of \( DATA \) and \( SEQ \). An acknowledgment may be sent by \( m_2 \) in any state except 0, in which case no unacknowledged data blocks have been received.

The remaining transition is the \( +A_k \) receive acknowledgment, in \( m_1 \). If \( m_1 \) is in state \( u, 1 \leq u \leq w \), and there is a nonempty value in shared variable \( ACK \), then exactly one of the transitions \( +A_0, +A_1, ..., +A_{w-1} \) will be enabled; it will be that \( A_k \) such that the predicate \( ACK \oplus k = seq \) is true, and the next state is \( k \). [Ref. 1]

For analyzing this protocol using the \textit{Big Mushroom} and \textit{Smart Mushroom} programs, the inputs to the program must be completed. These consist of a text file description of FSMs, the package, \textit{definitions}, which include the variables of the protocol, and the subprograms \textit{Analyze_Predicates_Machines} and \textit{Action}, which define the predicate-action table. Also an \textit{Output_Gtuple} procedure, which defines the output format for the global tuples, must be entered. Completed packages/procedures for a window size of 10 are given in Appendix B.

The same names are used for local and shared variables in the package \textit{definitions} as in the predicate-action table. Variables \( DATA, ACK \) and \( Sdata \) are declared as one
dimensional arrays of size 10, which is the window size. Local variables $seq$ and $exp$ and
index numbers $i$ and $j$ are declared as integers in the range 0 to 10. Global variable $ACK$ is
declared as integer in the range -1 to 10, where -1 represents $\epsilon$ value in the predicate-action
table. An enumeration type, $buffer\_type$, is declared for storing the data passed by the upper
layer to local variable $Sdata$. Data are declared as $d0, d1, \ldots, d9, e$, where $e$ represents the $\epsilon$
value. Transition names in the specification are defined as $snd\_data, rcv\_data, snd\_ack,$
$rcv\_acki$ for -$D$, +$D$, -$A$, and +$Ai$ in predicate-action table respectively.

Actions and predicates are also translated to Ada statements in the subprograms
$Analyze\_predicates\_Machines$ and $Action$. For each state in both machines there is a
"when" statement. The predicates for the outgoing transitions from that state are translated
to Ada with "if" conditional statements. Actions in the predicate-action table are converted
to Ada statements with "when" statements (see Appendix B).

The program generated 286 system states and 31,460 global states, which are
identical with the results obtained by the formulas given in [Ref. 11]. The protocol is free
from deadlocks and there are no unexecuted transitions. The difference between the
number of system and global states shows the power of the system state analysis which
reduced the number of states in the reachability graph exponentially. However, without the
$Smart\_Mushroom$ program, the system state analysis would be cumbersome to do manually,
and the global reachability analysis would be infeasible.

2. Token Bus

Another example of the program application, the token bus specification in [Ref.
15] will be used. The specification is a simplified one. It assumes that the transmission
medium is error free and all transmitted messages are received undamaged. Both the system
state analysis and global analysis are generated from this token bus specification for a
protocol consisting of 8 machines.
The specification of this simplified protocol is given in Figure 41 and Table 5. The FSM diagram and the local variables are the same for each machine, where the transition names: ready, rcv, pass, get-tk, pass-tk, Xmit, and moreD are appended with the corresponding machine number to the end for each machine in the specification. For example, transitions for machine 7 are named as ready7, rcv7, pass7, etc. This makes it easier to follow the reachability graphs. The remainder of the protocol specification as described in Reference 15 is as follows: The shared variable, MEDIUM, is used to model the bus, which is “shared” by each machine. A transmission onto the bus is modeled by a write into the shared variable. The fields of this variable correspond to the parts of the transmitted message: the first field, MEDIUM.T, takes the values T or D, which indicate whether the frame is a token or a data frame. The second field contains the address of the station to which the message is transmitted (DA for “destination address”); the next field, the originator (SA for “source address”); and finally the data block itself.

The network stations, or machines, are defined by a finite state machine, a set of local variables, and a predicate-action table. The initial state of each machine is state 0, and the shared variable is initially set to contain the token with the address of one of the stations in the “DA” field.

The value of local variable next is the address of the next or downstream neighbor, and these are initialized so that the entire network forms a cycle, or logical ring.

The local variable i is used to store the station’s own address. As implied by the names, the local variables inbuf and outbuf are used for storing data blocks to be transmitted to or retrieved from other machines on the network. The latter of these, outbuf, is an array and thus can store a potentially large number of data blocks. The local variable ctr serves to count the number of blocks sent; it is an upper bound on the number of blocks which can be sent during a single token holding period. The local variable j is an index into the array outbuf.
Figure 41: FSM and variables for the network nodes

The local variables $j$ and $ctr$ are initially set to 1, and $inbuf$ and $outbuf$ are initially set to empty. The shared variable $MEDIUM$ initially contains the token, with the address of the station in the $DA$ field. Thus the initial system state tuple is $(0, 0, ..., 0)$ and the first transition taken will be $get-tk$ by the station which has its local variable $i$ equal to $MEDIUM.DA$.

Each machine has four states. In the initial state, 0, the stations are waiting to either receive a message from another station, or the token. If the token appears in the variable $MEDIUM$ with the station’s own address, the transition to state 2 is taken. When
taking the \textit{get-tk} transition, the machine clears the communication medium and sets the message counter $ctr$ to 1. In state 2, the station transmits any data blocks it has, moving to state 3, or passes the token, returning to state 0. In state 3, the station will return to state 2 if any additional blocks are to be sent, until the maximum count $k$ is reached. When the count is reached, or when all the station's messages have been sent, the station returns to state 0.

The receiving station, as with all stations not in possession of the token, will be in state 0. The message will appear in $MEDIUM$, with the receiving station's address in the \textit{DA} field. The receiving transition to state 1 will then be taken, the data block copied, and $MEDIUM$ cleared. By clearing the medium, the receiving station enables the sending station to return to its initial state (0) or to its sending state (2).

\textbf{TABLE 5: PREDICATE-ACTION TABLE FOR THE NETWORK NODES}

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{rcv}</td>
<td>$MEDIUM. (t, DA) = (D, i)$</td>
<td>$inbuf \leftarrow MEDIUM.(SA, data)$</td>
</tr>
<tr>
<td>\textit{ready}</td>
<td>true</td>
<td>$MEDIUM \leftarrow \emptyset$</td>
</tr>
<tr>
<td>\textit{get-tk}</td>
<td>$MEDIUM. (t, DA) = (T, i)$</td>
<td>$MEDIUM \leftarrow \emptyset; ctr \leftarrow 1$</td>
</tr>
<tr>
<td>\textit{pass}</td>
<td>$outbuf [j] = \emptyset$</td>
<td>$MEDIUM \leftarrow (T, \text{next}, i, \emptyset)$</td>
</tr>
<tr>
<td>\textit{Xmit}</td>
<td>$outbuf [j] \neq \emptyset$</td>
<td>$MEDIUM \leftarrow outbuf [j]$; $ctr \leftarrow ctr \oplus 1$; $j \leftarrow j \oplus 1$; $outbuf [j] \leftarrow \emptyset$</td>
</tr>
<tr>
<td>\textit{moreD}</td>
<td>$MEDIUM = \emptyset \land outbuf [j] \neq \emptyset$</td>
<td>null</td>
</tr>
<tr>
<td>\textit{pass-tk}</td>
<td>$MEDIUM = \emptyset \land (outbuf [j] = \emptyset \lor ctr = k + 1)$</td>
<td>$MEDIUM \leftarrow (T, \text{next}, i, \emptyset)$</td>
</tr>
</tbody>
</table>
The symbol "Θ" indicates that the variable should be incremented unless its maximum value has been reached, in which case it should be reset to the initial value. The notation $MEDIUM.(t, DA)$ is used to denote the first two fields of the variable $MEDIUM$. For example, $MEDIUM.(t, DA) = (T, i)$ is a boolean expression which is true if and only if the first field of $MEDIUM$ contains the value $T$, and the second field contains the value $i$. Other notations in the predicate-action table such as "∧", "∨", "←" etc. are intuitive.

The inputs to the program for the reachability analysis of this protocol are given in Appendix C. The same names as in the specification are used for the local and global variables in the package definitions. Also, the "empty" value is represented by "E" and the data are represented by "I" in this package. The upper bound on the number of data blocks in the outbuf variable is set to 7.

The system state analysis alone did not give a complete analysis due to some loops in the FSMs of the SCM specification. Since the system state analysis assumes that two system states are equivalent if both the machine state tuples and the outgoing transitions are the same, this can cause the system state analysis to give insufficient results in some special cases. For example, incomplete results can arise when the FSMs of the specification include some loops that result with the same states and enabled transitions repeatedly. In such specifications, some of the transitions will stay unexecuted, resulting an incomplete analysis. This situation is observed in this specification when one of the machines had two or more data blocks in its outbuf local variable. For instance, if machine 1 has two data blocks in its outbuf local variable waiting for transmission and it receives the token from $MEDIUM$, it transitions to state 2 with get-tk and then takes the Xmit transition to state 3, sending the first data block. Since it has one more data block to send, the next transition will be moreD, which will take it back to state 2. At this point the system state analysis will stop and the reachability analysis will be incomplete.
The problem can be solved by splitting the system state analysis into three parts. First, the protocol can be analyzed with no messages in the machines and the behavior of the machines including only the transitions of the token can be observed (transitions \textit{get-tk} and \textit{pass}). Then, the analysis can be performed with one message in the \textit{outbuf} local variables of the machines, which allows us to analyze the transitions for receiving/transmitting the messages in addition to the transitions including the token (\textit{get-tk}, \textit{Xmit}, \textit{rcv}, \textit{ready}, \textit{pass-tk}). Finally, the protocol can be analyzed with each machine having more than one message, which includes the last transition in the analysis (\textit{moreD}). Combining the results of these parts shows that the protocol is free from deadlocks and there are no unexecuted transitions.

The definitions packages and the analysis results are given separately for each of the three cases outlined above in Appendix C. The system state analysis generated 16, 40 and 5 system states respectively for the parts explained above. The global analysis has generated 263 global states and there were no deadlocks or unexecuted transitions. The global reachability analysis is also given in Appendix C.

The system state analysis has reduced the number of states from 263 (global) to 61 (for all three parts). This is another example showing the advantage of the system state analysis.
VI. CONCLUSIONS AND FURTHER RESEARCH POSSIBILITIES

In this thesis, a software tool has been described which automates the analysis of protocols specified by the SCM and CFSM models. The program generates either the system state analysis or global reachability analysis for the SCM model. The program also generates the full reachability graph for a protocol specified by the CFSM model.

The major achievement of the thesis was the increase in the number of machines in the protocol specification. The previous work in [Ref. 8] was extended to allow two to eight machines in the specification. The run time and memory efficiency of the program were improved to allow the analysis of larger and more complex protocols. The user interface of the program has also been improved.

The system state analysis reduces the size of the state space greatly, but in some cases, when the system state analysis is not sufficient for the protocol analysis, the global reachability analysis is required. The Smart Mushroom program generates the system state graph. The Simple and Big Mushroom programs are based on exhaustive analysis, and generate the full global reachability graph. The main problem in these programs is the “state space explosion.” As stated in [Ref. 16], an estimate for the maximum size of the state space that can be reached for a full reachability analysis is about $10^5$ states. This is in agreement with the maximum number of states generated so far using the Big Mushroom program (153565 ≅ 1.53 $\times 10^5$ states were generated for the example protocol described in Chapter V).

The size of the state space which can be generated is directly proportional with the memory available on the computer. For a full reachability graph, an equation can be derived for determining the maximum number of states: where,
M: Memory available on the computer (bytes).

S: Amount of memory for storing one system state (bytes).

O: Overhead (memory for storing the program and other data structures etc.).

Then, the number of states that can be analyzed is: \( N = \frac{M-O}{S} \). Usually \( O \ll M \), and \( O \) can be ignored. For instance, for the LAP-B protocol analysis described in Chapter V, \( M=80 \) MBytes, \( S = 516 \) bytes, and \( N = 162596 \). In this analysis, only 153565 states were generated by the Simple Mushroom program. The difference between these numbers is due to the exclusion of the overhead in the calculation. Unfortunately memory was not enough for a 100% coverage in this analysis.

In spite of the state space explosion, the programs developed in this thesis are still very helpful for analyzing protocols. A full reachability analysis may be feasible by keeping the protocol specifications as simple as possible, and using certain assumptions about the behavior of the protocol to reduce the size of the state space. For example, the size of the message queue is very important for the CFSM model. A smaller message queue decreases \( S \) and allows to analyze larger protocols. A specification with less number of processes increases the number of states that can be analyzed. Modeling the machines with less number of states is also helpful. For the SCM model, \( N \) can be increased by keeping the size of global and local variables as small as possible. A simpler protocol specification also reduces the run time.

But, in some cases, even after some simplifications, a full reachability analysis is impossible. Fortunately, still some solutions exist for the automated protocol analysis. One method which is described in [Ref. 16] is using the supertrace algorithm. In the Mushroom program, hashing is used to increase the search efficiency. In the supertrace algorithm a very large hash size (almost the whole available memory) is used, and system states are not stored. This method is explained in [Ref. 16]. For example, with a 10 MB of memory, 80 million states can be generated using this method as described in [Ref. 16]. Of course this
efficiency does not come free. Due to hash conflicts, this method cannot guarantee 100% coverage, but as a partial search technique, this algorithm is very powerful.

This thesis opens several areas for further work. One improvement would be to increase the size of the system space that can be analyzed. Adding the supertrace option to the *Mushroom* program can be a good area for further work.

The number of reachable states is usually very large and it would be awkward to print out or browse through the listing. Another improvement would be to store the reachability analysis results in the form of a database, and provide a query language that allows the user to easily analyze the results of the analysis as suggested in [Ref. 17] (for instance, querying the error sequences and certain paths between any two states etc.).

Finally, another research possibility would be to add a simulator module to the *Mushroom*. For protocols with a large size of state space, where full reachability analysis is infeasible, simulation would be useful.

The Ada programming language was used to develop *Mushroom*. Also, specification of the SCM model must be entered to the program using Ada subprograms and packages. Ada is a well-structured programming language, and supports the modular development of programs. Also, exception handling, generic units, and tasking are important features of Ada. These features were helpful in developing the program. The well-structured property of the programming language makes the input of the specification easier. The tasking mechanism of Ada would be very helpful to develop a simulator module for the program.

The *Simple Mushroom* program is used as a teaching aid in an introductory communications network course at Naval Postgraduate School. This can be another area where student can use the tool as an aid in learning the protocol design and analysis.

The *mushroom* program is a tool which it is hoped that it will greatly improve the design and analysis of protocols specified by the SCM and CFSM models. Especially, this
program may help to solve some questions concerning the SCM model which have not been completely answered.
APPENDIX A (LAP-B Protocol Information Transfer Phase)

FSM Text File

```
start
number_of_machines 6

machine 1
state 1
tran +a0 1 3
tran -d0 2 3
state 2
tran +a0 2 3
tran -d1 3 3
tran +a1 4 3
state 3
tran +a0 3 3
tran +a1 5 3
tran +a2 7 3
state 4
tran +a1 4 3
tran -d1 5 3
state 5
tran +a1 5 3
tran +a2 7 3
tran -d2 6 3
state 6
tran +a1 6 3
tran +a0 1 3
tran +a2 8 3
state 7
tran +a2 7 3
tran -d2 8 3
state 8
tran +a2 8 3
tran +a0 1 3
tran -d0 9 3
state 9
tran +a2 9 3
tran +a0 2 3
tran +a1 4 3
machine 2
state 1
tran +eq 4 3
tran +id 2 3
state 2
tran +eq 5 3
tran +id 3 3
state 3
tran +eq 6 3
tran +id 1 3
state 4
tran -a0 1 3
state 5
tran -a1 2 3
state 6
tran -a2 3 3
machine 3
state 1
tran +id 2 1
tran +i0 3 1
tran +id 4 1
tran +id 20 4
tran +i10 21 4
tran +i20 22 4
tran +i101 23 4
tran +i11 24 4
tran +i12 25 4
tran +i102 26 4
tran +i112 27 4
tran +i22 28 4
state 2
tran -eq 8 2
state 3
tran -eq 9 2
state 4
tran -eq 10 2
```
state 0
trans +AO 11 2
trans +AL 12 2
trans +A2 13 2
state 9
trans +AO 14 2
trans +AL 15 2
trans +A2 16 2
state 10
trans +AO 17 2
trans +AL 18 2
trans +A2 19 2
state 11
trans -100 1 4
state 12
trans -101 1 4
state 13
trans -102 1 4
state 14
trans -110 1 4
state 15
trans -111 1 4
state 16
trans -112 1 4
state 17
trans -120 1 4
state 18
trans -121 1 4
state 19
trans -122 1 4
state 20
trans -D0 29 2
state 21
trans -D1 29 2
state 22
trans -D2 29 2
state 23
trans -D0 30 2
state 24
trans -D1 30 2
state 25
trans -D2 30 2
state 26
trans -D0 31 2
state 27
trans -D1 31 2
state 28
trans -D2 31 2
state 29
trans -AO 1 1
state 30
trans -AL 1 1
state 31
trans -A2 1 1
machine 4
state 1
trans +DO 2 5
trans +D1 2 5
trans +D2 4 5
trans +D0 20 3
trans +D10 21 3
trans +D20 22 3
trans +D10 23 3
trans +D11 24 3
trans +D21 25 3
trans +D10 26 3
trans +D12 27 3
trans +D22 28 3
state 2
trans -ERQ 8 6
state 3
trans -ERQ 9 6
state 4
trans -ERQ 10 6
state 8
trans +AO 11 6
trans +AL 12 6
trans +A2 13 6

75
state 0
trans +A2 8 4
trans +A0 1 4
trans +D0 9 4
state 1
trans +ENQ 4 4
trans +D0 2 4
state 2
trans +ENQ 5 4
trans +D1 3 4
state 3
trans +ENQ 6 4
trans +D2 1 4
state 4
trans -A0 1 4
state 5
trans -A1 2 4
state 6
trans -A2 3 4
initial state 1 1 1 1 1 1
finish

Program Output

REACHABILITY ANALYSIS of : fad.fsm

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>other machine</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>s D0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>s D1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>r A1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>3</td>
<td>r A1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>3</td>
<td>r A1</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>s D1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>r A1</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>3</td>
<td>s D2</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>3</td>
<td>r A1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>3</td>
<td>s D2</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>3</td>
<td>s D0</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>3</td>
<td>r A2</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>3</td>
<td>r A0</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>3</td>
<td>r A1</td>
</tr>
</tbody>
</table>

77
### Machine 2 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Other Machine</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>( r ) D0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>3</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>( r ) D1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>3</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>( r ) D2</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>( s ) A0</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>3</td>
<td>( s ) A1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>3</td>
<td>( s ) A2</td>
</tr>
</tbody>
</table>

### Machine 6 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Other Machine</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td>( r ) D0</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>( r ) D1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>4</td>
<td>( r ) EQ</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>4</td>
<td>( s ) A0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>( s ) A1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>4</td>
<td>( s ) A2</td>
</tr>
</tbody>
</table>

### Reachability Graph

#### Graph Details

1. \( [1, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{DO} \) 3 \( [2, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{DO} \) 4 \( [1, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)

2. \( [2, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{DI} \) 3 \( [3, E, D, D, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{DI} \) 4 \( [2, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)

3. \( [1, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{Di} \) 3 \( [3, E, D, D, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{Di} \) 4 \( [2, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)

4. \( [3, E, D, D, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{Di} \) 4 \( [3, E, D, D, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)

5. \( [2, E, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)
   - \( \text{Di} \) 3 \( [3, E, D, D, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E, 1, E, E, E] \)

\[ \text{Page 78} \]
**SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)**

Total number of states generated: 73391  
Number of states analyzed: 73391  
Number of deadlocks: 1  
Number of unspecified receptions: 0  
Maximum message queue size: 6  
Channel overflow: NONE

**UNEXECUTED TRANSITIONS**

****NONE****
APPENDIX B  (Go back N Window Size of 10)

FSM Text File

```
start
number_of_machines 2
machine 1
  state 0
  trans snd_data 1
  state 1
  trans row_ach0 0
  trans snd_data 2
  state 2
  trans row_ach0 0
  trans row_ach1 1
  trans snd_data 3
  state 3
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans snd_data 4
  state 4
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans row_ach3 3
  trans row_ach4 4
  trans snd_data 5
  state 5
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans row_ach3 3
  trans row_ach4 4
  trans row_ach5 5
  trans snd_data 6
  state 6
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans row_ach3 3
  trans row_ach4 4
  trans row_ach5 5
  trans row_ach6 6
  trans snd_data 7
  state 7
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans row_ach3 3
  trans row_ach4 4
  trans row_ach5 5
  trans row_ach6 6
  trans row_ach7 7
  trans snd_data 8
  state 8
  trans row_ach0 0
  trans row_ach1 1
  trans row_ach2 2
  trans row_ach3 3
  trans row_ach4 4
  trans row_ach5 5
  trans row_ach6 6
  trans row_ach7 7
  trans row_ach8 8
  trans snd_data 9
  state 9
```
state 10
trans rrv_ack0 0
trans rrv_ack1 1
trans rrv_ack2 2
trans rrv_ack3 3
trans rrv_ack4 4
trans rrv_ack5 5
trans rrv_ack6 6
trans rrv_ack7 7
trans rrv_ack8 8
trans rrv_ack9 9
machine 2
state 0
trans rrv_data 1
  state 1
  trans rrv_data 2
  trans rrv_ack0 0
  state 2
  trans rrv_data 3
  trans rnv_ack0 0
  state 3
  trans rrv_data 4
  trans rrv_ack0 0
  state 4
  trans rrv_data 5
  trans rrv_ack0 0
  state 5
  trans rrv_data 6
  trans rrv_ack0 0
  state 6
  trans rrv_data 7
  trans rrv_ack0 0
  state 7
  trans rrv_data 8
  trans rrv_ack0 0
  state 8
  trans rrv_data 9
  trans rrv_ack0 0
  state 9
  trans rrv_data 10
  trans rrv_ack0 0
  state 10
trans rnv_ack0 0
initial_state 0 0
finish
Variable Definitions

with TEXT_IO; use TEXT_IO;
package definitions is
  num_of_machines : constant := 2;
  type scm_transition_type is
    (and_data, rcv_data, rcv_ack1, rcv_ack2, rcv_ack3, rcv_ack4,
     rcv_ack5, rcv_ack6, rcv_ack7, rcv_ack8, rcv_ack9, and_ack, unused);
  type buffer_type is (d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, e);
  package buff_enum_io is new enumeration_io (buffer_type);
  use buff_enum_io;
  type buffer_array_type is array (1..10) of buffer_type;
  type seq_array_type is array (1..10) of integer range -1..10;
  type machine1_state_type is
    record
      ddata : buffer_type := e;
      seq : integer range 0..10 := 0;
      i : integer range 1..10 := 1;
    end record;
  type dummy_type is range 1..255;
  type machine2_state_type is
    record
      ddata : buffer_type := e;
      seq : integer range 0..10 := 0;
      j : integer range 1..10 := 1;
    end record;
  type machine3_state_type is
    record
      dummy : dummy_type;
    end record;
  type machine4_state_type is
    record
      dummy : dummy_type;
    end record;
  type machine5_state_type is
    record
      dummy : dummy_type;
    end record;
  type machine6_state_type is
    record
      dummy : dummy_type;
    end record;
  type machine7_state_type is
    record
      dummy : dummy_type;
    end record;
  type machine8_state_type is
    record
      dummy : dummy_type;
    end record;
  type global_variable_type is
    record
      DATA : buffer_array_type := (e, e, e, e, e, e, e, e, e);
      SEQ : seq_array_type := (-1, -1, -1, -1, -1, -1, -1, -1, -1, -1);
      ACK : integer range -1..10 := -1;
    end record;
end definitions;
Predicate-action Table

```plaintext
separate (main)

procedure Analyse_Predicates_Machinal(local : machinal_state_type;
  GLOBAL : global_variable_type;
  s : natural;
  w: in out transition_stack_package.stack) is

  begin
    case s is
    when 0 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;
    when 1 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;
    when 2 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 3 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;

      if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack2);
      end if;
    when 4 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 5 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 6 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 7 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 8 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when 9 =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    when others =>
      if ((GLOBAL.DATA(local.i) = B) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(w, end_data);
      end if;

      if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack0);
      end if;

      if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(w, row_ack1);
      end if;
    end case;
  end
```

83
if (temp3 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack4);
    end if;
if (temp4 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack3);
    end if;
when 5 =>
if ((GLOBAL_DATA(local.i) = E) and (GLOBAL_SEQ(local.i) = -1)) then
    Push(w, end_data);
    end if;
if (temp1 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack0);
    end if;
if (temp2 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack1);
    end if;
if (temp3 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack2);
    end if;
if (temp4 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack3);
    end if;
if (temp5 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack4);
    end if;
end if;
when 6 =>
if ((GLOBAL_DATA(local.i) = E) and (GLOBAL_SEQ(local.i) = -1)) then
    Push(w, end_data);
    end if;
if (temp1 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack0);
    end if;
if (temp2 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack1);
    end if;
if (temp3 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack2);
    end if;
if (temp4 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack3);
    end if;
if (temp5 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack4);
    end if;
end if;
when 7 =>
if ((GLOBAL_DATA(local.i) = E) and (GLOBAL_SEQ(local.i) = -1)) then
    Push(w, end_data);
    end if;
if (temp1 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack0);
    end if;
if (temp2 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack1);
    end if;
if (temp3 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack2);
    end if;
if (temp4 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack3);
    end if;
if (temp5 = local_seq) and (GLOBAL_ACK /= -1) then
    Push(w, rws_ack4);
    end if;
end if;
when 8 =>
if ((GLOBAL_DATA(local.i) = E) and (GLOBAL_SEQ(local.i) = -1)) then

Push(w, end_data);
and if;
if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack0);
  and if;
if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack1);
  and if;
if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack2);
  and if;
if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack3);
  and if;

when 9 =>
if ((GLOBAL.DATA(local.i) = 9) and (GLOBAL.SEQ(local.i) = -1)) then
  Push(w, end_data);
  and if;
if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack0);
  and if;
if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack1);
  and if;
if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack2);
  and if;
if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack3);
  and if;

when 10 =>
if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack0);
  and if;
if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack1);
  and if;
if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack2);
  and if;
if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack3);
  and if;
if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack4);
  and if;
if ((temp6 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack5);
  and if;
if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack6);
  and if;
if ((temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack7);
  and if;
if ((temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack8);
  and if;
if ((temp10 = local.seq) and (GLOBAL.ACK /= -1)) then
  Push(w, raw_ack9);
  and if;

85
Push(w, row_sock4);
end if;
if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
Push(w, row_sock5);
end if;
if ((temp5 = local.seq) and (GLOBAL.ACK /= -1)) then
Push(w, row_sock6);
end if;
if ((temp7 = local.seq) and (GLOBAL.ACK /= -1)) then
Push(w, row_sock7);
end if;
if ((temp8 = local.seq) and (GLOBAL.ACK /= -1)) then
Push(w, row_sock8);
end if;
if ((temp9 = local.seq) and (GLOBAL.ACK /= -1)) then
Push(w, row_sock9);
end if;
when others =>
null;
end case;
end Analyse_Predicates_Machine1;

---

separate (main)
procedure Analyse_Predicates_Machine2(local : machine2_state_type;
  GLOBAL: global_variable_type;
  s: natural;
  w : in out transition_stack_package.stack) is

begin
  case s is
  when 0 =>
    if ((GLOBAL.DATA(local.j) /=E) and (GLOBAL.SEQ(local.j) = local.exp)) then
      Push(w, row_data);
    end if;
    when 1 2 3 4 5 6 7 8 9 =>
      if (GLOBAL.DATA(local.j) =E) then
        Push(w, end_sock);
      end if;
      if ((GLOBAL.DATA(local.j) /=E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(w, row_data);
      end if;
    when 10 =>
      if (GLOBAL.DATA(local.j) =E) then
        Push(w, end_sock);
      end if;
  when others =>
    null;
  end case;
end Analyse_Predicates_Machine2;

---

separate (main)
procedure Analyse_Predicates_Machine3(local : machine3_state_type;
  GLOBAL: global_variable_type;
  s: natural;
  w : in out transition_stack_package.stack) is

begin
  null;
end Analyse_Predicates_Machine3;

---

separate (main)
procedure Analyse_Predicates_Machine4(local : machine4_state_type;
  GLOBAL: global_variable_type;
  s: natural;
  w : in out transition_stack_package.stack) is

begin
  null;
end Analyse_Predicates_Machine4;

---

separate (main)
procedure Analyse_Predicates_Machine5(local : machine5_state_type;
  GLOBAL: global_variable_type;
  s: natural;
  w : in out transition_stack_package.stack) is

begin
  null;
end Analyse_Predicates_Machine5;

---

86
### separate (main)

**procedure Analyse_Predicates_Machine6** (local : machine6 state_type;  
GLOBAL : global_variable_type;  
s : natural;  
w : in out transition_stack_package.stack) is

begin  
null;  
end Analyse_Predicates_Machine6;

---

### separate (main)

**procedure Analyse_Predicates_Machine7** (local : machine7 state_type;  
GLOBAL : global_variable_type;  
s : natural;  
w : in out transition_stack_package.stack) is

begin  
null;  
end Analyse_Predicates_Machine7;

---

### separate (main)

**procedure Analyse_Predicates_Machine8** (local : machine8 state_type;  
GLOBAL : global_variable_type;  
s : natural;  
w : in out transition_stack_package.stack) is

begin  
null;  
end Analyse_Predicates_Machine8;

---

### separate (main)

**procedure Action** (in system_state : in out state_record_type;  
in transition : in out system_transition_type;  
out system_state : in out state_record_type) is

begin  
case (in transition) is  
  when data =>  
    out system_state.GLOBAL_VARIABLES.DATA(in system_state.machine1 state.1) :=  
    in system_state.machine1 state.1.data(in system_state.machine1 state.1);  
    out system_state.GLOBAL_VARIABLES.SEG(in system_state.machine1 state.1) :=  
    in system_state.machine1 state.1.seq;  
    out system_state.machine1 state.1 := (in system_state.machine1 state.1 mod 10) + 1;  
    out system_state.machine1 state.seq := (((in system_state.machine1 state.seq) + 1)mod 11);  
  when recv_ack0 | recv_ack1 | recv_ack2 | recv_ack3 | recv_ack4  
    | recv_ack5 |recv_ack6 | recv_ack7 |recv_ack8 |recv_ack9 =>  
    out system_state.GLOBAL_VARIABLES.ACK := -1;  
  when ack =>  
    out system_state.GLOBAL_VARIABLES.ACK := in system_state.machine2 state.exp;  
    out system_state.machine2 state.data := e;  
  when rev =>  
    out system_state.machine2 state.Data :=  
    in system_state.GLOBAL_VARIABLES.DATA(in system_state.machine2 state.1);  
    out system_state.GLOBAL_VARIABLES.DATA(in system_state.machine2 state.3) := k;  
    out system_state.GLOBAL_VARIABLES.SEG(in system_state.machine2 state.3) := -1;  
    out system_state.machine2 state.exp := (((in system_state.machine2 state.exp) + 1)mod 11);  
  when others =>  
    put_line("There is an error in the Action procedure");  
end case;  
end Action;
separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  if print_header then
    new_line(3);
    set_col(7);
    put_line(" ml(seq.i.Sdata), m2(exp.j,Rdata), (DATA,SEQ,ACK)");
    print_header := false;
  else
    put(" [* & integer'image(tuple.machine_state(1)) ]");
    put(" , ");
    put(tuple.machine1_state.seq, width => 1);
    put(" , ");
    put(tuple.machine1_state.i, width => 1);
    put(" , ");
    buff_enum_io.put(tuple.machine1_state.Sdata(1), set => upper_case);
    put(" [* & integer'image(tuple.machine_state(2)) ]");
    put(" , ");
    put(tuple.machine2_state.exp, width => 1);
    put(" , ");
    put(tuple.machine2_state.j, width => 1);
    put(" , ");
    buff_enum_io.put(tuple.machine2_state.Rdata, set => upper_case);
    for i in 1..10 loop
      put(" , ");
      buff_enum_io.put(tuple.GLOBAL_VARIABLES.DATA(i), set => upper_case);
      put(" , ");
      put(tuple.GLOBAL_VARIABLES.SEQ(1),width=>1);
      end loop;
      put(" , ");
      put(tuple.GLOBAL_VARIABLES.ACK, width => 1);
      put(" ]")
  end if;
end output_Gtuple;
### Program Output (System State Analysis)

**REACHABILITY ANALYSIS of: gbn_10.com**

**SPECIFICATION**

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>send_data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>send_data</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>send_data</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>send_data</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>send_data</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>send_data</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>recv_ack5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>send_data</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>recv_ack5</td>
</tr>
<tr>
<td>7</td>
<td>6</td>
<td>recv_ack6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>send_data</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>recv_ack5</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>recv_ack6</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>recv_ack7</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>send_data</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>recv_ack5</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>recv_ack6</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>recv_ack7</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>recv_ack8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>send_data</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>recv_ack0</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>recv_ack1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>recv_ack2</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>recv_ack3</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>recv_ack4</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>recv_ack5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>recv_ack6</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>recv_ack7</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>recv_ack8</td>
</tr>
<tr>
<td>10</td>
<td>9</td>
<td>recv_ack9</td>
</tr>
</tbody>
</table>
### Machine 2 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv_data</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>rcv_data</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>rcv_data</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>rcv_data</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>rcv_data</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>rcv_data</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>rcv_data</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>rcv_data</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>rcv_data</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>rcv_data</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>snd_ack</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>snd_ack</td>
</tr>
</tbody>
</table>

---

**Reachability Graph**

```
0 [ 0, 0 ] 0  snd_data  1
1 [ 1, 0 ] 0  snd_data  2
2 [ 2, 0 ] 0  snd_data  4
3 [ 1, 1 ] 0  snd_data  5
4 [ 3, 0 ] 0  snd_data  7
5 [ 2, 1 ] 0  snd_data  8
6 [ 1, 0 ] 1  rcv_ack0  0
7 [ 4, 0 ] 0  snd_data 11
8 [ 3, 1 ] 0  snd_data 12
9 [ 2, 2 ] 0  snd_data 13
10 [ 2, 0 ] 1  rcv_ack1  1
11 [ 5, 0 ] 0  snd_data 17
12 [ 4, 1 ] 0  snd_data 18
13 [ 3, 2 ] 0  snd_data 19
14 [ 2, 0 ] 2  rcv_ack0  0
15 [ 3, 0 ] 1  rcv_ack2  2
16 [ 2, 1 ] 1  rcv_ack1  3
17 [ 6, 0 ] 0  snd_data 23
18 [ 5, 1 ] 0  snd_data 25
19 [ 4, 2 ] 0  rcv_data 26
```
| 20 [ 3, 3 ] 0 | and_data 27 | and_ack 28 |
| 21 [ 3, 0 ] 2 | rov_ack1 1 | and_data 29 | rov_data 30 |
| 22 [ 4, 0 ] 1 | rov_ack3 4 | and_data 31 | rov_data 32 |
| 23 [ 3, 1 ] 1 | rov_ack2 5 | and_data 32 | rov_data 33 |
| 24 [ 7, 0 ] 0 | and_data 34 | rov_data 35 |
| 25 [ 6, 1 ] 0 | and_data 35 | rov_data 36 |
| 26 [ 5, 2 ] 0 | and_data 36 | rov_data 37 |
| 27 [ 4, 3 ] 0 | and_data 37 | rov_data 38 |
| 28 [ 3, 0 ] 3 | rov_ack0 0 | and_data 39 |
| 29 [ 4, 0 ] 2 | rov_ack2 2 | and_data 40 | rov_data 41 |
| 30 [ 3, 1 ] 2 | rov_ack1 3 | and_data 41 | and_ack 28 |
| 31 [ 5, 0 ] 1 | rov_ack4 7 | and_data 42 | rov_data 43 |
| 32 [ 4, 1 ] 1 | rov_ack3 8 | and_data 43 | rov_data 44 |
| 33 [ 3, 2 ] 1 | rov_ack2 9 | and_data 44 | and_ack 28 |
| 34 [ 8, 0 ] 1 | and_data 45 | rov_data 46 |
| 35 [ 7, 1 ] 0 | and_data 46 | rov_data 47 |
| 36 [ 6, 2 ] 0 | and_data 47 | rov_data 48 |
| 37 [ 5, 3 ] 0 | and_data 48 | rov_data 49 |
| 38 [ 4, 4 ] 0 | and_data 49 | and_ack 50 |
| 39 [ 4, 0 ] 3 | rov_ack1 1 | and_data 51 | rov_data 52 |
| 40 [ 5, 0 ] 2 | rov_ack3 4 | and_data 53 | rov_data 54 |
| 41 [ 4, 1 ] 2 | rov_ack2 5 | and_data 54 | rov_data 55 |
| 42 [ 6, 0 ] 1 | rov_ack5 11 | and_data 56 | rov_data 57 |
| 43 [ 5, 1 ] 1 | rov_ack4 12 | and_data 57 | rov_data 58 |
| 44 [ 4, 2 ] 1 | rov_ack3 13 | and_data 58 | rov_data 59 |
| 45 [ 9, 0 ] 2 | rov_data 60 | and_data 60 |
| 46 [ 8, 1 ] 0 | and_data 61 | rov_data 62 |
| 47 [ 7, 2 ] 0 | and_data 62 | rov_data 63 |
| 48 [ 6, 3 ] 0 | and_data 63 | rov_data 64 |
| 49 [ 5, 4 ] 0 | and_data 64 | 91 |
50 [4, 0] 4  
rov_data 65  
rov_ack0 0  
and_data 66  
51 [5, 0] 3  
rov_ack2 2  
and_data 67  
rov_data 68  
52 [4, 1] 3  
rov_ack1 3  
and_data 68  
rov_ack5 50  
53 [6, 0] 2  
rov_ack4 7  
and_data 68  
rov_data 70  
54 [5, 1] 2  
rov_ack3 8  
and_data 70  
rov_data 71  
55 [4, 2] 2  
rov_ack2 9  
and_data 71  
rov_data 72  
56 [7, 0] 1  
rov_ack6 17  
and_data 72  
rov_data 73  
57 [6, 1] 1  
rov_ack5 18  
and_data 73  
rov_data 74  
58 [5, 2] 1  
rov_ack4 19  
and_data 74  
rov_data 75  
59 [4, 3] 1  
rov_ack3 20  
and_data 75  
and_ack 50  
60 [10, 0] 3  
rov_data 76  
61 [9, 1] 1  
rov_data 77  
and_data 77  
62 [8, 2] 0  
rov_data 78  
and_data 78  
63 [7, 3] 0  
rov_data 79  
and_data 79  
64 [6, 4] 0  
rov_data 80  
and_data 80  
65 [5, 5] 0  
and_ack 81  
66 [5, 0] 4  
rov_ack1 82  
and_data 82  
rov_data 83  
67 [6, 0] 3  
rov_ack3 84  
and_data 84  
rov_data 85  
68 [5, 1] 3  
rov_ack2 85  
and_data 85  
rov_data 86  
69 [7, 0] 2  
rov_ack5 11  
and_data 87  
rov_data 88  
70 [6, 1] 2  
rov_ack4 12  
and_data 88  
rov_data 89  
71 [5, 2] 2  
rov_ack3 13  
and_data 89  
rov_data 90  
72 [8, 0] 2  
rov_ack7 24  
and_data 91  
rov_data 92  
73 [7, 1] 1  
rov_ack6 25  
and_data 92  
rov_data 92  
74 [6, 2] 1  
rov_ack5 26  
and_data 93  
rov_data 93  
75 [5, 3] 1  
rov_ack4 27  
and_data 94  
rov_data 94  
76 [10, 1] 2  
rov_data 95  
77 [9, 2] 0  
and_data 96
78 [ 8, 3 ] 0  
79 [ 7, 4 ] 0  
80 [ 6, 5 ] 0  
81 [ 5, 0 ] 5  
82 [ 6, 0 ] 4  
83 [ 5, 1 ] 4  
84 [ 7, 0 ] 3  
85 [ 6, 1 ] 3  
86 [ 5, 2 ] 3  
87 [ 8, 0 ] 3  
88 [ 7, 1 ] 2  
89 [ 6, 2 ] 2  
90 [ 5, 3 ] 2  
91 [ 9, 0 ] 3  
92 [ 8, 1 ] 1  
93 [ 7, 2 ] 1  
94 [ 6, 3 ] 1  
95 [ 5, 4 ] 1  
96 [10, 2 ] 1  
97 [ 9, 3 ] 0  
98 [ 8, 4 ] 0  
99 [ 7, 5 ] 0  
100 [ 6, 6 ] 0  
101 [ 6, 0 ] 5  
102 [ 7, 0 ] 4  
103 [ 6, 1 ] 4  
104 [ 8, 0 ] 4  
105 [ 7, 1 ] 3  

rov_data 97  
end_data 97  
rov_data 98  
end_data 98  
rov_data 99  
rov_ack0 0  
end_data 101  
rov_ack2 2  
end_data 102  
rov_data 103  
rov_ack1 3  
end_data 103  
rov_ack4 7  
end_data 104  
rov_data 105  
rov_ack3 8  
end_data 105  
rov_data 106  
rov_ack2 9  
end_data 106  
end_ack 81  
rov_ack6 17  
end_data 107  
rov_data 108  
rov_ack5 18  
end_data 108  
rov_data 109  
rov_ack4 19  
end_data 109  
rov_data 110  
rov_ack3 20  
end_data 110  
end_ack 81  
rov_ack8 34  
end_data 111  
rov_data 112  
rov_ack7 35  
end_data 112  
rov_data 113  
rov_ack6 36  
end_data 113  
rov_data 114  
rov_ack5 37  
end_data 114  
rov_data 115  
rov_ack4 38  
end_data 115  
end_ack 81  
rov_data 116  
end_data 116  
rov_data 117  
end_data 117  
rov_data 118  
end_data 118  
rov_data 119  
end_data 119  
end_ack 120  
rov_ack1 1  
end_data 121  
rov_data 122  
rov_ack3 4  
end_data 123  
rov_data 124  
rov_ack2 5  
end_data 124  
rov_data 125  
rov_ack5 11  
end_data 126  
rov_data 127  
rov_ack4 12
<table>
<thead>
<tr>
<th>Line Number</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Action 1</th>
<th>Action 2</th>
<th>Action 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>106</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>107</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>109</td>
<td>7</td>
<td>2</td>
<td>2</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>110</td>
<td>6</td>
<td>3</td>
<td>2</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>111</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>112</td>
<td>9</td>
<td>1</td>
<td>2</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>113</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>114</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>115</td>
<td>6</td>
<td>4</td>
<td>1</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>116</td>
<td>10</td>
<td>3</td>
<td>0</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>117</td>
<td>9</td>
<td>4</td>
<td>0</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>118</td>
<td>8</td>
<td>5</td>
<td>0</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>119</td>
<td>7</td>
<td>6</td>
<td>0</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>120</td>
<td>6</td>
<td>0</td>
<td>6</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>121</td>
<td>7</td>
<td>0</td>
<td>5</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>122</td>
<td>6</td>
<td>1</td>
<td>5</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>123</td>
<td>8</td>
<td>0</td>
<td>5</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>124</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>125</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>126</td>
<td>9</td>
<td>0</td>
<td>5</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>127</td>
<td>8</td>
<td>1</td>
<td>3</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>2</td>
<td>3</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>129</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>130</td>
<td>10</td>
<td>0</td>
<td>5</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>131</td>
<td>9</td>
<td>1</td>
<td>3</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
<tr>
<td>132</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>and_data</td>
<td>127</td>
<td>raw_data</td>
</tr>
</tbody>
</table>
160 ( 8, 4 ) 1
rcv_data 184
rcv_ack 79
snd_data 184
rcv_data 185
rcv_ack 80
snd_data 185
rcv_data 186
rcv_data 187
rcv_data 188
rcv_data 189
rcv_ack 0
snd_data 189
rcv_data 190
rcv_ack 2
snd_data 190
rcv_data 191
rcv_ack 3
snd_data 191
rcv_data 192
rcv_ack 7
snd_data 192
rcv_data 193
rcv_ack 7
snd_data 193
rcv_data 194
rcv_ack 8
snd_data 194
rcv_data 195
rcv_ack 9
snd_data 195
rcv_data 196
rcv_ack 10
snd_data 196
rcv_data 197
rcv_ack 11
snd_data 197
rcv_data 198
rcv_ack 20
snd_data 198
rcv_data 199
rcv_ack 30
snd_data 199
rcv_data 200
rcv_ack 37
snd_data 200
rcv_data 201
rcv_ack 38
snd_data 201
rcv_data 202
rcv_ack 62
snd_data 202
rcv_data 203
rcv_ack 66
snd_data 203
rcv_data 204
rcv_ack 65
snd_data 204
rcv_data 205
rcv_ack 97
snd_data 205
rcv_data 206
rcv_ack 98
snd_data 206
rcv_data 207
rcv_ack 99
snd_data 207
rcv_ack 100
snd_data 207
rcv_ack 165
snd_data 208
187 [10, 6] 0  rcv_data 208
188 [ 9, 7] 0  snd_data 208
189 [ 8, 8] 0  snd_data 209
190 [ 8, 0] 8  rcv_ack 210
191 [ 9, 0] 8  rcv_ack 212
192 [ 8, 1] 6  rcv_ack 213
193 [10, 0] 8  rcv_data 215
194 [ 9, 1] 6  rcv_data 217
195 [ 8, 2] 5  rcv_data 218
196 [10, 1] 6  rcv_data 220
197 [ 9, 2] 4  rcv_data 221
198 [ 8, 3] 4  rcv_data 223
199 [10, 2] 4  rcv_data 225
200 [ 9, 3] 3  rcv_data 227
201 [ 8, 4] 3  rcv_data 229
202 [10, 3] 2  rcv_data 231
203 [ 9, 4] 2  rcv_data 233
204 [ 8, 5] 2  rcv_data 235
205 [10, 4] 1  rcv_data 237
206 [ 9, 5] 1  rcv_data 239
207 [ 8, 6] 1  rcv_data 241
208 [10, 7] 0  rcv_data 243
209 [ 9, 8] 0  rcv_data 245
210 [ 8, 0] 9  snd_ack 247
211 [ 9, 0] 9  snd_ack 249
212 [ 8, 1] 7  snd_ack 251
213 [10, 0] 9  snd_ack 253
214 [ 9, 1] 7  snd_ack 255
215 [ 8, 2] 6  snd_ack 257

97
<table>
<thead>
<tr>
<th>Row</th>
<th>Column</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>216</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>217</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>218</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>219</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>220</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>221</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>222</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>223</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>224</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>225</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>226</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>227</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>228</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>229</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>230</td>
<td>8</td>
<td>7</td>
</tr>
<tr>
<td>231</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>232</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>233</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>234</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>235</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>236</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>237</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>238</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>239</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>240</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>241</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>242</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>243</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>244</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>Row</td>
<td>Pedal</td>
<td>Data</td>
</tr>
<tr>
<td>-----</td>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>245</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>246</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>247</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>248</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>249</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>250</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>251</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>252</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>253</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>254</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>255</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>256</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>257</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>258</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>259</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>260</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>261</td>
<td>9</td>
<td>6</td>
</tr>
<tr>
<td>262</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>263</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>264</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>265</td>
<td>9</td>
<td>8</td>
</tr>
<tr>
<td>266</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>267</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>268</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>269</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>270</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>271</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>272</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>273</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>274</td>
<td>10</td>
<td>7</td>
</tr>
<tr>
<td>275</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>276</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>277</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>State</td>
<td>Rcv_Ack</td>
<td>End_Ack</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>276</td>
<td>9</td>
<td>276</td>
</tr>
<tr>
<td>279</td>
<td>20</td>
<td>276</td>
</tr>
<tr>
<td>280</td>
<td>38</td>
<td>276</td>
</tr>
<tr>
<td>283</td>
<td>65</td>
<td>276</td>
</tr>
<tr>
<td>284</td>
<td>143</td>
<td>276</td>
</tr>
<tr>
<td>285</td>
<td>232</td>
<td>276</td>
</tr>
</tbody>
</table>

**Summary of Reachability Analysis (Analysis Completed)**

- Number of states generated: 286
- Number of states analyzed: 286
- Number of deadlocks: 0

**Unexecuted Transitions**

*****NONE*****
APPENDIX C (Token Bus Protocol)

FSM Text File

```
start
number_of_machines 8
machine 1
  state 0
  trans rvl 1
  trans gwt_k1 2
  state 1
  trans ready1 0
  state 2
  trans Xmit1 3
  trans pass1 0
  state 3
  trans moreD1 2
  trans pass_tkl 0
  machine 2
  state 0
  trans rvl2 1
  trans gwt_k2 2
  state 1
  trans ready2 0
  state 2
  trans Xmit2 3
  trans pass2 0
  state 3
  trans moreD2 2
  trans pass_tkl 0
  machine 3
  state 0
  trans rvl3 1
  trans gwt_k3 2
  state 1
  trans ready3 0
  state 2
  trans Xmit3 3
  trans pass3 0
  state 3
  trans moreD3 2
  trans pass_tkl 0
  machine 4
  state 0
  trans rvl4 1
  trans gwt_k4 2
  state 1
  trans ready4 0
  state 2
  trans Xmit4 3
  trans pass4 0
  state 3
  trans moreD4 2
  trans pass_tkl 0
  machine 5
  state 0
  trans rvl5 1
  trans gwt_k5 2
  state 1
  trans ready5 0
  state 2
  trans Xmit5 3
  trans pass5 0
  state 3
```
trans moreD5 2
trans pass_tk6 0
machine 6
state 0
trans rcv6 1
trans get_tk6 2
state 1
trans ready6 0
state 2
trans Xmit6 3
trans pass6 0
state 3
trans moreD6 2
trans pass_tk6 0
machine 7
state 0
trans rcv7 1
trans get_tk7 2
state 1
trans ready7 0
state 2
trans Xmit7 3
trans pass7 0
state 3
trans moreD7 2
trans pass_tk7 0
machine 8
state 0
trans rcv8 1
trans get_tk8 2
state 1
trans ready8 0
state 2
trans Xmit8 3
trans pass8 0
state 3
trans moreD8 2
trans pass_tk8 0
initial_state 0 0 0 0 0 0 0 0 0
finish
Variable Definitions (No Message in outbuf Variables)

with TEXT_IO; use TEXT_IO;
package definitions is
  num_of_machines : constant := 3;
  k : constant := 7; -- number of rows (messages) in output buffer
  type scm_transition_type is (pass1, pass2, pass3, pass4, pass5, pass6, pass7, pass8, get_tk1, get_tk2, get_tk3, get_tk4, get_tk5, get_tk6, get_tk7, get_tk8, exit1, exit2, exit3, exit4, exit5, exit6, exit7, exit8, moreD1, moreD2, moreD3, moreD4, moreD5, moreD6, moreD7, moreD8, pass_tk4, pass_tk5, pass_tk6, pass_tk7, pass_tk8, pass1_tk1, pass1_tk2, pass1_tk3, rov1, rov2, rov3, rov4, rov5, rov6, rov7, rov8, rov9, rov10, rov11, rov12, rov13, rov14, rov15, rov16, rov17, rov18, rov19, rov20, ready1, ready2, ready3, ready4, ready5, ready6, ready7, ready8, unused);

  type dummy_type is range 1..255;
  type t_field_type is (D, Y, 2);
  package t_field_enum_io is new enumeration_IO(t_field_type);
  use t_field_enum_io;

  type MEDIUM_TYPE is record
    t : t_field_type;
    DA : integer range 1..8;
    SA : integer range 1..8;
    data : character;
  end record;

  type input_buffer_type is record
    DA : integer range 0..8 := 0;
    SA : integer range 0..8 := 0;
    data : character := 'X';
  end record;

  type output_buffer_type is array (1..k) of MEDIUM_TYPE;

  type machine1_state_type is record
    next : integer := 2; -- address of downstream neighbor
    i : integer := 1; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E, 2, 1, 'I'), (E, 3, 1, 'I'),
                                   (E, 4, 1, 'I'), (E, 5, 1, 'I'),
                                   (E, 6, 1, 'I'), (E, 7, 1, 'I'), (E, 8, 1, 'I'));
  end record;

  type machine2_state_type is record
    next : integer := 3; -- address of downstream neighbor
    i : integer := 2; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E, 1, 2, 'I'), (E, 3, 2, 'I'),
                                   (E, 4, 2, 'I'), (E, 5, 2, 'I'),
                                   (E, 6, 2, 'I'), (E, 7, 2, 'I'), (E, 8, 2, 'I'));
  end record;

  type machine3_state_type is record
    next : integer := 4; -- address of downstream neighbor
    i : integer := 3; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received messages
outbuf : output_buffer_type := ((E,1,3,'I'),(E,2,3,'I'),
                                (E,4,3,'I'),(E,5,3,'I'),
                                (E,6,3,'I'),(E,7,3,'I'),(E,8,3,'I'));

end record;

type machine4_state_type is
  record
    next : integer := 5; -- address of downstream neighbor
    i : integer := 4; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E,1,4,'I'),(E,2,4,'I'),(E,3,4,'I'),
                                     (E,5,4,'I'),(E,6,4,'I'),(E,7,4,'I'),(E,8,4,'I'));
  end record;

type machine5_state_type is
  record
    next : integer := 6; -- address of downstream neighbor
    i : integer := 5; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E,1,5,'I'),(E,2,5,'I'),(E,3,5,'I'),(E,4,5,'I'),
                                     (E,5,5,'I'),(E,6,5,'I'),(E,7,5,'I'),(E,8,5,'I'));
  end record;

type machine6_state_type is
  record
    next : integer := 7; -- address of downstream neighbor
    i : integer := 6; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E,1,6,'I'),(E,2,6,'I'),(E,3,6,'I'),(E,4,6,'I'),
                                     (E,5,6,'I'),(E,6,6,'I'),(E,7,6,'I'),(E,8,6,'I'));
  end record;

type machine7_state_type is
  record
    next : integer := 8; -- address of downstream neighbor
    i : integer := 7; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E,1,7,'I'),(E,2,7,'I'),(E,3,7,'I'),(E,4,7,'I'),
                                     (E,5,7,'I'),(E,6,7,'I'),(E,7,7,'I'));
  end record;

type machine8_state_type is
  record
    next : integer := 1; -- address of downstream neighbor
    i : integer := 8; -- stations own address
    ctr : integer range 1..(k+1) := 1; -- counter for messages sent
    j : integer range 1..k := 1; -- index for output buffer
    inbuf : input_buffer_type; -- stores the received messages
    outbuf : output_buffer_type := ((E,1,8,'I'),(E,2,8,'I'),(E,3,8,'I'),(E,4,8,'I'),
                                     (E,5,8,'I'),(E,6,8,'I'),(E,7,8,'I'));
  end record;

type global_variable_type is
  record
    MEDIUM : MEDIUM_TYPE := (T,1,2,'A');
  end record;

end definitions;
Variable Definitions (One Message in outbuf Variables)

with TEXT_IO; use TEXT_IO;
package definitions is
  nux_of_machines : constant := 8;
k : constant := 7; -- number of rows (messages) in output buffer
type som_transition_type is (pass1, pass2, pass3, pass4, pass5, pass6,
  pass7, pass8, get_tk1, get_tk2,
  get_tk3, get_tk4, get_tk5, get_tk6,
  get_tk7, get_tk8, xmit1, xmit2, xmit3,
  xmit4, xmit5, xmit6, xmit7, xmit8, moreD1,
  moreD2, moreD3, moreD4, moreD5,
  moreD6, moreD7, moreD8, pass_tk4, pass_tk5,
  pass_tk6, pass_tk7, pass_tk8,
  pass_tk1, pass_tk2, pass_tk3, 
  rov1, rov2, rov3, rov4, rov5, rov6, rov7, rov8,
  rov2, rov3, ready1, ready2, ready3,
  ready4, ready5, ready6, ready7, ready8, unused);

  type dummy_type is range 1..255;
  type t_field_type is (D, 2, R);
  package t_field Enum_IO is new enumeration_IO(t_field_type);
  use t_field_enum_io;

  type MEDIUM_TYPE is
    record
      t : t_field_type;
      DA : integer range 1..8;
      SA : integer range 1..8;
      data : character;
      end record;

  type input_buffer_type is
    record
      DA : integer range 0..8 := 0;
      SA : integer range 0..8 := 0;
      data : character := 'F';
      end record;

  type output_buffer_type is array (1..k) of MEDIUM_TYPE;

  type machine1_state_type is
    record
      next : integer := 2; -- address of downstream neighbor
      i : integer := 2; -- stations own address
      ctr : integer range 1..(k+1) := 1; -- counter for messages sent
      j : integer range 1..k := 1; -- index for output buffer
      inbuf : input_buffer_type; -- stores the received messages
      outbuf : output_buffer_type := ((D, 2, 1, 'I'), (R, 3, 1, 'I'),
        (R, 4, 1, 'I'), (R, 5, 1, 'I'),
        (R, 6, 1, 'I'), (R, 7, 1, 'I'), (R, 8, 1, 'I'));
      end record;

  type machine2_state_type is
    record
      next : integer := 3; -- address of downstream neighbor
      i : integer := 2; -- stations own address
      ctr : integer range 1..(k+1) := 1; -- counter for messages sent
      j : integer range 1..k := 1; -- index for output buffer
      inbuf : input_buffer_type; -- stores the received messages
      outbuf : output_buffer_type := ((D, 1, 2, 'I'), (R, 3, 2, 'I'),
        (R, 4, 2, 'I'), (R, 5, 2, 'I'),
        (R, 6, 2, 'I'), (R, 7, 2, 'I'), (R, 8, 2, 'I'));
      end record;

  type machine3_state_type is
    record
      next : integer := 4; -- address of downstream neighbor
      i : integer := 3; -- stations own address
      ctr : integer range 1..(k+1) := 1; -- counter for messages sent
I : integer range 1..k := 1; -- index for output buffer
inbuf : input buffer type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,3,'I'),(E,2,3,'I'),
(1,4,3,'I'),(E,5,3,'I'),
(1,6,3,'I'),(E,7,3,'I'),(E,8,3,'I'));
end record;

type machine4_state_type is
case
next : integer := 5; -- address of downstream neighbor
i : integer := 4; -- stations own address
ctr : integer range 1..(k+1) := 1; -- counter for massages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,4,'I'),(E,2,4,'I'),(E,3,4,'I'),(E,4,4,'I'),
(1,6,4,'I'),(E,7,4,'I'),(E,8,4,'I'));
end record;

type machine5_state_type is
case
next : integer := 6; -- address of downstream neighbor
i : integer := 5; -- stations own address
ctr : integer range 1..(k+1) := 1; -- counter for massages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,5,'I'),(E,2,5,'I'),(E,3,5,'I'),(E,4,5,'I'),
(1,6,5,'I'),(E,7,5,'I'),(E,8,5,'I'));
end record;

type machine6_state_type is
case
next : integer := 7; -- address of downstream neighbor
i : integer := 6; -- stations own address
ctr : integer range 1..(k+1) := 1; -- counter for massages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,6,'I'),(E,2,6,'I'),(E,3,6,'I'),(E,4,6,'I'),
(1,5,6,'I'),(E,7,6,'I'),(E,8,6,'I'));
end record;

type machine7_state_type is
case
next : integer := 8; -- address of downstream neighbor
i : integer := 7; -- stations own address
ctr : integer range 1..(k+1) := 1; -- counter for massages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,7,'I'),(E,2,7,'I'),(E,3,7,'I'),(E,4,7,'I'),
(1,5,7,'I'),(E,6,7,'I'),(E,8,7,'I'));
end record;

type machine8_state_type is
case
next : integer := 1; -- address of downstream neighbor
i : integer := 8; -- stations own address
ctr : integer range 1..(k+1) := 1; -- counter for massages sent
j : integer range 1..k := 1; -- index for output buffer
inbuf : input_buffer_type; -- stores the received massages
outbuf : output_buffer_type := ((D,1,8,'I'),(E,2,8,'I'),(E,3,8,'I'),(E,4,8,'I'),
(1,5,8,'I'),(E,6,8,'I'),(E,7,8,'I'));
end record;

type global_variable_type is
case
MEDIUM : MEDIUM_TYPE := (2,1,2,'I');
end record;
end definitions;
Variable Definitions

There are seven messages in `outbuf` variable of each machine and each machine sends one message to the other machines in the network.

```plaintext
with TEX_HO; use TEX_HO;
package definitions is

num_of_machines : constant := 8;
k : constant := 7; -- number of rows (messages) in output buffer

package scm_transition_type is
(pas1, pas2, pas3, pas4, pas5, pas6, pas7, pas8, get_th1, get_th2, get_th3, get_th4, get_th5, get_th6, get_th7, get_th8, Xmit1, Xmit2, Xmit3, Xmit4, Xmit5, Xmit6, Xmit7, Xmit8, moreD1, moreD2, moreD3, moreD4, moreD5, moreD6, moreD7, moreD8, pass_th4, pass_th5, pass_th6, pass_th7, pass_th8, pass_th1, pass_th2, pass_th3, rov1, rov4, rov5, rov6, rov7, rov8, rov2, rov3, ready1, ready2, ready3, ready4, ready5, ready6, ready7, ready8, unused);

type dummy_type is range 1..255;
type t_field_type is (D,T,E);
package t_field_enum_io is new enumeration_10(t_field_type);
use t_field_enum_io;

package MEDIUM_TYPE is
record
  t : t_field_type;
  DA : integer range 1..8;
  SA : integer range 1..8;
data : character;
end record;

type input_buffer_type is
record
  DA : integer range 0..8 := 0;
  SA : integer range 0..8 := 0;
data : character := 'I';
end record;

type output_buffer_type is array (1..k) of MEDIUM_TYPE;

type machine1_state_type is
record
  next : integer := 2; -- address of downstream neighbor
  i : integer := 1; -- stations own address
  ctr : integer range 1..(k+1) := 1; -- counter for messages sent
  j : integer range 1..k := 1; -- index for output buffer
  inbuf : input_buffer_type; -- stores the received messages
  outbuf : output_buffer_type := ((D,2,1,'I'),(D,3,1,'I'),
(D,4,1,'I'),(D,5,1,'I'),
(D,6,1,'I'),(D,7,1,'I'),(D,8,1,'I') );
end record;

type machine2_state_type is
record
  next : integer := 3; -- address of downstream neighbor
  i : integer := 2; -- stations own address
  ctr : integer range 1..(k+1) := 1; -- counter for messages sent
  j : integer range 1..k := 1; -- index for output buffer
  inbuf : input_buffer_type; -- stores the received messages
  outbuf : output_buffer_type := ((D,1,2,'I'),(D,3,2,'I'),
(D,4,2,'I'),(D,5,2,'I'),
(D,6,2,'I'),(D,7,2,'I'),(D,8,2,'I') );
end record;
```
type machine3_state_type is
  record
    next: integer := 4; -- address of downstream neighbor
    i: integer := 3; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,3,'I'), (D,2,3,'I'),
                                  (D,4,3,'I'), (D,5,3,'I'),
                                  (D,6,3,'I'), (D,7,3,'I'), (D,8,3,'I'));
  end record;

type machine4_state_type is
  record
    next: integer := 5; -- address of downstream neighbor
    i: integer := 4; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,4,'I'), (D,2,4,'I'), (D,3,4,'I'), (D,5,4,'I'),
                                  (D,6,4,'I'), (D,7,4,'I'), (D,8,4,'I'));
  end record;

type machine5_state_type is
  record
    next: integer := 6; -- address of downstream neighbor
    i: integer := 5; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,5,'I'), (D,2,5,'I'), (D,3,5,'I'), (D,4,5,'I'),
                                  (D,5,5,'I'), (D,6,5,'I'), (D,7,5,'I'), (D,8,5,'I'));
  end record;

type machine6_state_type is
  record
    next: integer := 7; -- address of downstream neighbor
    i: integer := 6; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,6,'I'), (D,2,6,'I'), (D,3,6,'I'), (D,4,6,'I'),
                                  (D,5,6,'I'), (D,6,6,'I'), (D,7,6,'I'), (D,8,6,'I'));
  end record;

type machine7_state_type is
  record
    next: integer := 8; -- address of downstream neighbor
    i: integer := 7; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,7,'I'), (D,2,7,'I'), (D,3,7,'I'), (D,4,7,'I'),
                                  (D,5,7,'I'), (D,6,7,'I'), (D,7,7,'I'));
  end record;

type machine8_state_type is
  record
    next: integer := 1; -- address of downstream neighbor
    i: integer := 8; -- stations own address
    ctr: integer range 1..(k+1) := 1; -- counter for messages sent
    j: integer range 1..k := 1; -- index for output buffer
    inbuf: input_buffer_type; -- stores the received messages
    outbuf: output_buffer_type := ((D,1,8,'I'), (D,2,8,'I'), (D,3,8,'I'), (D,4,8,'I'),
                                  (D,5,8,'I'), (D,6,8,'I'), (D,7,8,'I'));
  end record;

type global_variable_type is
  record
    NDZUM: MUIU-DTYIP := (T,1,2,'W');
  end record;

end definitions;
Predicate-Action Table

```
separate(main)
procedure Analyze_Predicates_Machine1(local : machine1_state_type;
  global : global_variable_type;
  s : natural;
  w : in out transition_stack_package.stack) is
begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv1);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) ) then
        push(w,get_tkl);
      end if;
    when 1 =>
      push(w,ready1);
    when 2 =>
      if (local.outbuf(local.j).t /= E) then
        push(w,Xmit1);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
        (local.ctr <= k) ) then
        push(w,moreD1);
      end if;
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t = E) or
        (local.ctr = (k+1)) ) then
        push(w,pass_tkl);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine1;
```

```
separate(main)
procedure Analyze_Predicates_Machine2(local : machine2_state_type;
  global : global_variable_type;
  s : natural;
  w : in out transition_stack_package.stack) is
begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv2);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) ) then
        push(w,get_tkl);
      end if;
    when 1 =>
      push(w,ready2);
    when 2 =>
      if (local.outbuf(local.j).t /= E) then
        push(w,Xmit2);
      end if;
    when 3 =>
      if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
        (local.ctr <= k) ) then
        push(w,moreD2);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machine2;
```
end if;
if (global.MEDIUM.t = D) and (local.outbuf(local.j).t = E) then
    push(w, more_thk3);
end if;
when others =>
    null;
end case;
end Analyse_Predicates_Machine2;

------------------------------------------------------------------

separate(main)
procedure Analyse_Predicates_Machine3(local : machine3_state_type;
    global : global_variable_type;
    s : natural;
    w : in out transition_stack_package.stack) is
begin
    case s is
    when 0 =>
        if (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) then
            push(w, recv3);
        end if;
        if (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) then
            push(w, get_thk3);
        end if;
    when 1 =>
        push(w, ready3);
    when 2 =>
        if (local.outbuf(local.j).t /= E) then
            push(w, Emit3);
        end if;
        if (local.outbuf(local.j).t = E) then
            push(w, pass3);
        end if;
    when 3 =>
        if (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
            (local.ctr <= k) then
            push(w, more_thk3);
        end if;
        if (global.MEDIUM.t = T) and (local.outbuf(local.j).t = E) or
            (local.ctr = (k+1)) then
            push(w, pass_thk3);
        end if;
    when others =>
        null;
end case;
end Analyse_Predicates_Machine3;
------------------------------------------------------------------

separate(main)
procedure Analyse_Predicates_Machine4(local : machine4_state_type;
    global : global_variable_type;
    s : natural;
    w : in out transition_stack_package.stack) is
begin
    case s is
    when 0 =>
        if (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) then
            push(w, recv4);
        end if;
        if (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) then
            push(w, get_thk4);
        end if;
    when 1 =>
        push(w, ready4);
    when 2 =>
        if (local.outbuf(local.j).t /= E) then
end it;
If (local.outbuf(local.j) .t - Z)
then
push (w, pasa4);
end if;
when 3 a'
if ((global.MDIUK .t = Z) and (local.outbuf(local.j) .t = Z) and (local.ctr <= k))
then
push(w, more5);
end if;
if ((global.MDIUK .t = Z) and (local.outbuf(local.j) .t = Z) or (local.ctr = (k+1)))
then
push(w, pass_tk4);
end if;
when others =>
null;
end case;
end Analyze_Predicates_Machine4;

--

separate(main)
procedure Analyze_Predicates_Machine5(local : machine5_state_type;
        global : global_variable_type;
        s : natural;
        w : in out transition_stack_package.stack) is

begin
  case s is
    when 0 =>
      if ((global.MDIUM .t = D) and (global.MDIUM.DE = local.i))
then
        push(w, rov5);
      end if;
    when 1 =>
      push(w, ready5);
    when 2 =>
      if (local.outbuf(local.j) .t /= Z) then
        push(w, Xmit5);
      end if;
    when 3 =>
      if ((global.MDIUM .t = Z) and (local.outbuf(local.j).t /= Z) and
          (local.ctr <= k))
then
        push(w, more5);
      end if;
    when others =>
null;
  end case;
end Analyze_Predicates_Machine5;

--

separate(main)
procedure Analyze_Predicates_Machine6(local : machine6_state_type;
        global : global_variable_type;
        s : natural;
        w : in out transition_stack_package.stack) is

end procedure Analyze_Predicates_Machine6;
begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv7);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) ) then
        push(w,get_tk7);
      end if;
      when 1 =>
        push(w,ready7);
      when 2 =>
        if (local.outbuf(local.j).t /= E) then
          push(w,xmit7);
        end if;
        if (local.outbuf(local.j).t = E) then
          push(w,pass7);
        end if;
      when 3 =>
        if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
            (local.ctr <= k) ) then
          push(w,more7);
        end if;
        if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t = E) or
            (local.ctr = (k+1) ) ) then
          push(w,pass_tk7);
        end if;
      when others =>
        null;
  end case;
end Analyse_Predicates_Machine6;

------------------------------------------------------------------------------------------------------

separate(main)
procedure Analyse_Predicates_Machine7(local : machine7_state_type;
global : global_variable_type;
s : natural;
w : in out transition_stack_package.stack) is

begin
  case s is
    when 0 =>
      if ( (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) ) then
        push(w,rcv7);
      end if;
      if ( (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) ) then
        push(w,get_tk7);
      end if;
      when 1 =>
        push(w,ready7);
      when 2 =>
        if (local.outbuf(local.j).t /= E) then
          push(w,xmit7);
        end if;
        if (local.outbuf(local.j).t = E) then
          push(w,pass7);
        end if;
      when 3 =>
        if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
            (local.ctr <= k) ) then
          push(w,more7);
        end if;
        if ( (global.MEDIUM.t = E) and (local.outbuf(local.j).t = E) or
            (local.ctr = (k+1) ) ) then
          push(w,pass_tk7);
        end if;
      when others =>
        null;
  end case;
end Analyse_Predicates_Machine7;
null;
end case;
end Analyze_Predicates_Machines;

---------------------------------------------

separate(main)
procedure Analyze_Predicates_Machines(local : machine_state_type;
  global : global_variables_type;
  s : natural;
  w : in out transition_stack_package.stack) is

begin
  case s is
    when 0 =>
      if (global.MEDIUM.t = D) and (global.MEDIUM.DA = local.i) then
        push(w, rcv8);
      end if;
      if (global.MEDIUM.t = T) and (global.MEDIUM.DA = local.i) then
        push(w, get_tk8);
      end if;
    when 1 =>
      push(w, ready8);
    when 2 =>
      if (local.outbuf(local.j).t /= E) then
        push(w, wait8);
      end if;
      if (local.outbuf(local.j).t = E) then
        push(w, pass8);
      end if;
    when 3 =>
      if (global.MEDIUM.t = E) and (local.outbuf(local.j).t /= E) and
          (local.ctr <= k) then
        push(w, more8);
      end if;
      if (global.MEDIUM.t = E) and (local.outbuf(local.j).t = E)
          or (local.ctr = (k+1)) then
        push(w, pass_tk8);
      end if;
    when others =>
      null;
  end case;
end Analyze_Predicates_Machines;

---------------------------------------------

separate(main)
procedure Action (in_system_state : in out State_record_type;
  in_transition : in out SCM_transition_type;
  out_system_state : in out State_record_type) is

begin
  case in_transition is
    when rcv1 =>
      out_system_state.machines1_state.inbuf.sa :=
        in_system_state.global_variables.MEDIUM.sa;
      out_system_state.machines1_state.inbuf.data :=
        in_system_state.global_variables.MEDIUM.data;
    when rcv2 =>
      out_system_state.machines2_state.inbuf.sa :=
        in_system_state.global_variables.MEDIUM.sa;
      out_system_state.machines2_state.inbuf.data :=
        in_system_state.global_variables.MEDIUM.data;
    when rcv3 =>
      out_system_state.machines3_state.inbuf.sa :=
        in_system_state.global_variables.MEDIUM.sa;
      out_system_state.machines3_state.inbuf.data :=
        in_system_state.global_variables.MEDIUM.data;
  end case;
end Action;
when rov4 =>
  out_system_state.machine4_state.inbuf.SA
  := in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine4_state.inbuf.data
  := in_system_state.global_variables.MEDIUM.data;
when rov5 =>
  out_system_state.machine5_state.inbuf.SA
  := in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine5_state.inbuf.data
  := in_system_state.global_variables.MEDIUM.data;
when rov6 =>
  out_system_state.machine6_state.inbuf.SA
  := in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine6_state.inbuf.data
  := in_system_state.global_variables.MEDIUM.data;
when rov7 =>
  out_system_state.machine7_state.inbuf.SA
  := in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine7_state.inbuf.data
  := in_system_state.global_variables.MEDIUM.data;
when rov8 =>
  out_system_state.machine8_state.inbuf.SA
  := in_system_state.global_variables.MEDIUM.SA;
  out_system_state.machine8_state.inbuf.data
  := in_system_state.global_variables.MEDIUM.data;

when ready1 | ready2 | ready3 | ready4 | ready5 | ready6 | ready7 | ready8 =>
  out_system_state.global_variables.MEDIUM.t := T;

when get_tk1 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine1_state.ctr := 1;
when get_tk2 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine2_state.ctr := 1;
when get_tk3 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine3_state.ctr := 1;
when get_tk4 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine4_state.ctr := 1;
when get_tk5 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine5_state.ctr := 1;
when get_tk6 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine6_state.ctr := 1;
when get_tk7 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine7_state.ctr := 1;
when get_tk8 =>
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.machine8_state.ctr := 1;

when pass1 | pass_tk1 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.SA
  := in_system_state.machine1_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA
  := in_system_state.machine1_state.1;
when pass2 | pass_tk2 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.SA
  := in_system_state.machine2_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA
  := in_system_state.machine2_state.1;
when pass3 | pass_tk3 =>
  out_system_state.global_variables.MEDIUM.t := T;

114
out_system_state.global_variables.MEDIUM.DA := in_system_state.machine3_state.next;
out_system_state.global_variables.MEDIUM.data := 'E';
out_system_state.global_variables.MEDIUM.SA := in_system_state.machine3_state.i;
when pass4 | pass_th4 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA := in_system_state.machine4_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine4_state.i;
when pass5 | pass_th5 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA := in_system_state.machine5_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine5_state.i;
when pass6 | pass_th6 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA := in_system_state.machine6_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine6_state.i;
when pass7 | pass_th7 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA := in_system_state.machine7_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine7_state.i;
when pass8 | pass_th8 =>
  out_system_state.global_variables.MEDIUM.t := T;
  out_system_state.global_variables.MEDIUM.DA := in_system_state.machine8_state.next;
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine8_state.i;

when Xmit1 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine1_state.outbuf(in_system_state.machine1_state.j);
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine1_state.i;
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.global_variables.MEDIUM.ctr := (in_system_state.machine1_state.ctr mod 8) + 1;
  out_system_state.global_variables.MEDIUM.state := (in_system_state.machine1_state.j mod 7) + 1;
when Xmit2 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine2_state.outbuf(in_system_state.machine2_state.j);
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine2_state.i;
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.global_variables.MEDIUM.ctr := (in_system_state.machine2_state.ctr mod 8) + 1;
  out_system_state.global_variables.MEDIUM.state := (in_system_state.machine2_state.j mod 7) + 1;
when Xmit3 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine3_state.outbuf(in_system_state.machine3_state.j);
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine3_state.i;
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.global_variables.MEDIUM.ctr := (in_system_state.machine3_state.ctr mod 8) + 1;
  out_system_state.global_variables.MEDIUM.state := (in_system_state.machine3_state.j mod 7) + 1;
when Xmit4 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine4_state.outbuf(in_system_state.machine4_state.j);
  out_system_state.global_variables.MEDIUM.data := 'E';
  out_system_state.global_variables.MEDIUM.SA := in_system_state.machine4_state.i;
  out_system_state.global_variables.MEDIUM.t := E;
  out_system_state.global_variables.MEDIUM.ctr := (in_system_state.machine4_state.ctr mod 8) + 1;
out_system_state.machine4 state.j := (in_system_state.machine4_state.j mod 7) + 1;
when Xmit5 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine5 state.outbuf(in_system_state.machine5_state.j);
  out_system_state.machine5_state.outbuf(in_system_state.machine5_state.j).t := E;
  out_system_state.machine5_state.ctr := (in_system_state.machine5_state.ctr mod 8) + 1;
  out_system_state.machine5_state.j := (in_system_state.machine5_state.j mod 7) + 1;
when Xmit6 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine6_state.outbuf(in_system_state.machine6_state.j);
  out_system_state.machine6_state.outbuf(in_system_state.machine6_state.j).t := E;
  out_system_state.machine6_state.ctr := (in_system_state.machine6_state.ctr mod 8) + 1;
  out_system_state.machine6_state.j := (in_system_state.machine6_state.j mod 7) + 1;
when Xmit7 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine7 state.outbuf(in_system_state.machine7_state.j);
  out_system_state.machine7_state.outbuf(in_system_state.machine7_state.j).t := E;
  out_system_state.machine7_state.ctr := (in_system_state.machine7_state.ctr mod 8) + 1;
  out_system_state.machine7_state.j := (in_system_state.machine7_state.j mod 7) + 1;
when Xmit8 =>
  out_system_state.global_variables.MEDIUM := in_system_state.machine8_state.outbuf(in_system_state.machine8_state.j);
  out_system_state.machine8_state.outbuf(in_system_state.machine8_state.j).t := E;
  out_system_state.machine8_state.ctr := (in_system_state.machine8_state.ctr mod 8) + 1;
  out_system_state.machine8_state.j := (in_system_state.machine8_state.j mod 7) + 1;
when moreD1 | moreD2 | moreD3 | moreD4 | moreD5 | moreD6 | moreD7 | moreD8 =>
  null;
when others =>
  put("Error in action procedure");
end case;
end Action;
Output Format

```plaintext
separate(main)
procedure output_Gtuple(tuple : in out statewide_record_type) is
begin
  if print_header then
    new_line(2);
    set_col(7);
    put_line("m1,m2,m3,m4,m5,m6,m7,m8,MEDIUM.t,MEDIUM.DA,MEDIUM.SA,MEDIUM.data");
    print_header := false;
  else
    put("[" integer' image(tuple.machine_state(1)) );
    put(" , ");
    put( integer' image(tuple.machine_state(2)) );
    put(" , ");
    put( integer' image(tuple.machine_state(3)) );
    put(" , ");
    put( integer' image(tuple.machine_state(4)) );
    put(" , ");
    put( integer' image(tuple.machine_state(5)) );
    put(" , ");
    put( integer' image(tuple.machine_state(6)) );
    put(" , ");
    put( integer' image(tuple.machine_state(7)) );
    put(" , ");
    put( integer' image(tuple.machine_state(8)) );
    put(" , ");
    tfield enum io.put(tuple.global_variables.MEDIUM.t, set => upper_case);
    put(" , ");
    put(tuple.global_variables.MEDIUM.DA, width => 1);
    put(" , ");
    put(tuple.global_variables.MEDIUM.SA, width => 1);
    put(" , ");
    put(tuple.global_variables.MEDIUM.data);
    put(" ]");
  end if;
end output_Gtuple;
```

117
Program Output (No Message in `outbuf` Variable)

**Reachability Analysis of tb8.com Specification**

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 4 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 5 State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
### Machine 6 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv6</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_tk6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk6</td>
</tr>
</tbody>
</table>

### Machine 7 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv7</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_tk7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk7</td>
</tr>
</tbody>
</table>

### Machine 8 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_tk8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit8</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk8</td>
</tr>
</tbody>
</table>

**SYSTEM REACHABILITY GRAPH**

```
0 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 0 get_tk1 1
1 [ 2, 0, 0, 0, 0, 0, 0, 0 ] 0 pass1 2
2 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 1 get_tk2 3
3 [ 0, 2, 0, 0, 0, 0, 0, 0 ] 0 pass2 4
4 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 2 get_tk3 5
5 [ 0, 0, 2, 0, 0, 0, 0, 0 ] 0 pass3 6
6 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 3 get_tk4 7
7 [ 0, 0, 0, 2, 0, 0, 0, 0 ] 0 pass4 8
8 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 4 get_tk5 9
9 [ 0, 0, 0, 0, 2, 0, 0, 0 ] 0 pass5 10
10 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 5 get_tk6 11
11 [ 0, 0, 0, 0, 0, 2, 0, 0 ] 0 pass6 12
12 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 6 get_tk7 13
13 [ 0, 0, 0, 0, 0, 0, 2, 0 ] 0 pass7 14
```
14 \[ 0, 0, 0, 0, 0, 0, 0, 0 \] 7 get_th8 15
15 \[ 0, 0, 0, 0, 0, 0, 0, 2 \] 0 pass8 0

**SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)**

- Number of states generated: 16
- Number of states analysed: 16
- Number of deadlocks: 0

### UNEXECUTED TRANSITIONS

#### Machine 1 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th1</td>
</tr>
</tbody>
</table>

#### Machine 2 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th2</td>
</tr>
</tbody>
</table>

#### Machine 3 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th3</td>
</tr>
</tbody>
</table>

#### Machine 4 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv4</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th4</td>
</tr>
</tbody>
</table>

#### Machine 5 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv5</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready5</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit5</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more5</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th5</td>
</tr>
</tbody>
</table>

120
### Machine 6 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk6</td>
</tr>
</tbody>
</table>

### Machine 7 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk7</td>
</tr>
</tbody>
</table>

### Machine 8 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk8</td>
</tr>
</tbody>
</table>
Program Output (One Message in *outbuf* Variable)

**System Reachability Graph**

```
0 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 0 get_tk1 1
1 [ 2, 0, 0, 0, 0, 0, 0, 0 ] 0 xmit1 2
2 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 0 rcv2 3
3 [ 3, 1, 0, 0, 0, 0, 0, 0 ] 0 ready2 4
4 [ 3, 0, 0, 0, 0, 0, 0, 0 ] 1 pass_tk1 5
5 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 1 get_tk2 6
6 [ 0, 2, 0, 0, 0, 0, 0, 0 ] 0 xmit2 7
7 [ 0, 3, 0, 0, 0, 0, 0, 0 ] 0 rcv1 8
8 [ 1, 3, 0, 0, 0, 0, 0, 0 ] 0 ready1 9
9 [ 0, 3, 0, 0, 0, 0, 0, 0 ] 1 pass_tk2 10
10 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 2 get_tk3 11
11 [ 0, 0, 2, 0, 0, 0, 0, 0 ] 0 xmit3 12
12 [ 0, 0, 3, 0, 0, 0, 0, 0 ] 0 rcv1 13
13 [ 1, 0, 3, 0, 0, 0, 0, 0 ] 0 ready1 14
14 [ 0, 0, 3, 0, 0, 0, 0, 0 ] 1 pass_tk3 15
15 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 3 get_tk4 16
16 [ 0, 0, 0, 2, 0, 0, 0, 0 ] 0 xmit4 17
17 [ 0, 0, 0, 3, 0, 0, 0, 0 ] 0 rcv1 18
18 [ 1, 0, 0, 3, 0, 0, 0, 0 ] 0 ready1 19
19 [ 0, 0, 0, 3, 0, 0, 0, 0 ] 1 pass_tk4 20
20 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 4 get_tk5 21
21 [ 0, 0, 0, 0, 2, 0, 0, 0 ] 0 xmit5 22
22 [ 0, 0, 0, 0, 3, 0, 0, 0 ] 0 rcv1 23
23 [ 1, 0, 0, 3, 0, 0, 0, 0 ] 0 ready1 24
24 [ 0, 0, 0, 0, 3, 0, 0, 0 ] 1 pass_tk5 25
25 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 5 get_tk6 26
26 [ 0, 0, 0, 0, 0, 2, 0, 0 ] 0 xmit6 27
27 [ 0, 0, 0, 0, 3, 0, 0, 0 ] 0 rcv1 28
28 [ 1, 0, 0, 0, 0, 3, 0, 0 ] 0 ready1 29
29 [ 0, 0, 0, 0, 0, 3, 0, 0 ] 1 pass_tk6 30
30 [ 0, 0, 0, 0, 0, 0, 0, 0 ] 6 get_tk7 31
31 [ 0, 0, 0, 0, 0, 0, 2, 0 ] 0 xmit7 32
32 [ 0, 0, 0, 0, 0, 0, 3, 0 ] 0 rcv1 33
33 [ 1, 0, 0, 0, 0, 0, 3, 0 ] 0 ready1 34
```
SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)

Number of states generated: 40
Number of states analyzed: 40
Number of deadlocks: 0

UNEXECUTED TRANSITIONS

<table>
<thead>
<tr>
<th>Machine 1 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 4 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 5 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>Machine 6 Unexecuted Transitions</td>
</tr>
<tr>
<td>---------------------------------</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 7 Unexecuted Transitions</th>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>rcv7</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>ready7</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>pass7</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>more7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 8 Unexecuted Transitions</th>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>rcv8</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>ready8</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
<td>pass8</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>2</td>
<td>more8</td>
</tr>
</tbody>
</table>
Program Output (More Than One Message in *outbuf* Variable)

**SYSTEM REACHABILITY GRAPH**

- 0 [0, 0, 0, 0, 0, 0, 0, 0] 0 get_tkl 1
- 1 [2, 0, 0, 0, 0, 0, 0, 0] 0 xmit1 2
- 2 [3, 0, 0, 0, 0, 0, 0, 0] 0 rsv2 3
- 3 [3, 1, 0, 0, 0, 0, 0, 0] 0 ready2 4
- 4 [3, 0, 0, 0, 0, 0, 0, 0] 1 more1 5

**SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)**

- Number of states generated : 5
- Number of states analysed : 5
- Number of deadlocks : 0

**UNEXECUTED TRANSITIONS**

### Machine 1 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rsv1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tkl</td>
</tr>
</tbody>
</table>

### Machine 2 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>get_tk2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk2</td>
</tr>
</tbody>
</table>

### Machine 3 Unexecuted Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Unexecuted Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rsv3</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_tk3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_tk3</td>
</tr>
</tbody>
</table>

125
<table>
<thead>
<tr>
<th>Machine 4 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 5 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 6 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 7 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 8 Unexecuted Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Program Output (Global Reachability Analysis)

There are seven messages in `oubuf` variable of each machine.

<table>
<thead>
<tr>
<th>Machine 1 State Transitions</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_th1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass1</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>pass_th1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 State Transitions</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_th2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit2</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored2</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>pass_th2</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 3 State Transitions</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_th3</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored3</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>pass_th3</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 4 State Transitions</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_th4</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored4</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>pass_th4</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 5 State Transitions</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv5</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get_th5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit5</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>mored5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass_th5</td>
<td></td>
</tr>
</tbody>
</table>
### Machine 6 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rev6</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get tk6</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready6</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit6</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass6</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more6</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass tk6</td>
</tr>
</tbody>
</table>

### Machine 7 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rev7</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get tk7</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit7</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more7</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass tk7</td>
</tr>
</tbody>
</table>

### Machine 8 State Transitions

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rev8</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>get tk8</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>ready8</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>xmit8</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>pass8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>more8</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>pass tk8</td>
</tr>
</tbody>
</table>

### Reachability Graph

```
[ml, m2, m3, m4, m5, m6, m7, m8, MEDIUM.t, MEDIUM.da, MEDIUM.rr, MEDIUM.data]
0 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, T, 1, 2, E] get tk1  1
1 [ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, E, 1, 2, E] xmit1  2
2 [ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, I] rev2  3
3 [ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, I] ready2  4
4 [ 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, I] more2  5
5 [ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, I] xmit1  6
6 [ 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, I] rev3  7
7 [ 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, D, 3, 1, I] ready3  8
8 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, I] more3  9
9 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, I] xmit1 10
10 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, I] rev4 11
11 [ 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, D, 4, 1, I] ready4 12
12 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 1, I] more4 13
13 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 1, I] xmit1 14
14 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 1, I] rev5 15
15 [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 5, 1, I] ready5 16
16 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 1, I] more5 17
17 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 5, 1, I] xmit1 18
18 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] rev6 19
19 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] ready6 20
20 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] more6 21
21 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] xmit1 22
22 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] rev7 23
23 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] ready7 24
24 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 6, 1, I] more7 25
25 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, I] xmit1 26
26 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, I] rev8 27
27 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, I] ready8 28
28 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, I] more8 29
29 [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 7, 1, I] pass tk2 30
```
<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>246</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>247</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>248</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>249</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>250</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>251</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>252</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>253</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>254</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>256</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>257</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>258</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>259</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>260</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>261</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>262</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**SUMMARY OF REACHABILITY ANALYSIS (ANALYSIS COMPLETED)**

---

Number of states generated: 263
Number of states analysed: 263
Number of deadlocks: 0

**UNEXECUTED TRANSITIONS**

*****NONE*****

132
LIST OF REFERENCES


## INITIAL DISTRIBUTION LIST

1. **Defense Technical Information Center**  
   Cameron Station  
   Alexandria, VA 22304-6145  
   
2. **Dudley Knox Library, Code 052**  
   Naval Postgraduate School  
   Monterey, CA 93943  
   
3. **Chairman, Code 37 CS**  
   Computer Science Department  
   Naval Postgraduate School  
   Monterey, CA 93943-5000  
   
4. **Dr. G. M. Lundy, Code CS/Ln**  
   Assistant Professor, Computer Science Department  
   Naval Postgraduate School  
   Monterey, CA 93943-5000  
   
5. **Dr. Man-Tak Shing, Code CS/Sh**  
   Associate Professor, Computer Science Department  
   Naval Postgraduate School  
   Monterey, CA 93943-5000  
   
6. **Dr. Mohamed Gouda**  
   Department of Computer Science  
   University of Texas at Austin  
   Austin, TX 78712  
   
7. **Dr. Raymond E. Miller**  
   Department of Computer Science  
   A. V. Williams Bldg.  
   University of Maryland  
   College Park, MD 20742  
   
8. **Dr. Krishan Sabnani**  
   AT&T Bell Labs  
   Room 2C-218  
   Murray Hill, NJ 07974
9. Deniz Kuvvetleri Komutanlığı
   Personel Daire Baskanligi
   Bakanliklar, Ankara / TURKEY

10. Golyuk Tersanesi Komutanlığı
    Golyuk, Kocaeli / TURKEY

11. Deniz Harp Okulu Komutanlığı
    81704 Tuzla, Istanbul / TURKEY

12. Taskizak Tersanesi Komutanlığı
    Kasimpasa, Istanbul / TURKEY

13. LTJG Zeki Bulent Bulbul
    Merkez Bankası Evleri
    Ozgurler Sok. No. 9
    Kalaba, Ankara / TURKEY