THESIS

A Tool for Automated Validation of Network Protocols

by

Matthew James Rothlisberger

24 September 1992

Thesis Advisor: Dr. G. M. Lundy

Approved for public release
distribution unlimited.

92-28850
This thesis introduces a program that analyzes network protocols using the Communicating Finite State Machines (CFSM) model and the System of Communicating Machines (SCM) model. A simple two machine implementation of CFSM model is initially explored. A number of simple protocols are demonstrated as a means to validate the automated tool (program). The second model implemented is that of SCM model. The SCM tool uses many of the same data structures designed in the CFSM program. The SCM program is validated with an analysis of widely used data link protocols. Both programs were done in the Ada language environment.
A Tool for Automated Validation of Network Protocols

by
Matthew James Rothlisberger
Captain, United States Army
B. S. Business Administration, Clarion State College, 1982

Submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the
NAVAL POSTGRADUATE SCHOOL

6 August 1992

Author:

Matthew James Rothlisberger

Approved By:

G. M. Lundy, Thesis Advisor

Amr Zaky, Second Reader

Robert B. McGhee, Chairman,
Department of Computer Science
ABSTRACT

This thesis introduces a program that analyzes network protocols using the Communicating Finite State Machines (CFSM) model and the System of Communicating Machines (SCM) model. A simple, two machine implementation of CFSM model is initially explored. A number of simple protocols are demonstrated as a means to validate the automated tool (program).

The second model implemented is that of the SCM model. The SCM tool uses many of the same data structures designed in the CFSM program. The SCM program is validated with an analysis of widely used data link protocols.

Both programs were done in the Ada language environment.
# TABLE OF CONTENTS

I. **INTRODUCTION**.......................................................................................................................... 1

II. **BACKGROUND OF MODELS**......................................................................................................... 5
   A. **GENERAL**................................................................................................................................. 5
   B. **COMMUNICATING FINITE STATE MACHINES**................................................................. 7
   C. **SYSTEM OF COMMUNICATING MACHINES**............................................................... 12
   D. **LANGUAGE CONSIDERATIONS**...................................................................................... 18

III. **A PROGRAM FOR GENERATING A CFSM REACHABILITY ANALYSIS**................................... 21
    A. **PROGRAM STRUCTURE**..................................................................................................... 21
    B. **INPUT**............................................................................................................................... 24
    C. **REACHABILITY ANALYSIS**.............................................................................................. 27
    D. **OUTPUT**............................................................................................................................. 31

IV. **AN AUTOMATED TOOL FOR SCM REACHABILITY ANALYSIS**.............................................. 34
   A. **PROGRAM STRUCTURE**..................................................................................................... 35
   B. **INPUT**............................................................................................................................... 39
      1. Protocol Variable Definitions ............................................................................................ 40
      2. Predicate-Action Table Representation .......................................................................... 42
      3. Finite State Machines ....................................................................................................... 45
   C. **REACHABILITY ANALYSIS**.............................................................................................. 47
      1. Global State Analysis ........................................................................................................... 47
      2. System State Analysis ......................................................................................................... 52
   D. **OUTPUT**............................................................................................................................. 56

V. **AN AUTOMATED ANALYSIS OF SELECTED DATA LINK PROTOCOLS**................................. 59
   A. **CFSM MODEL**.................................................................................................................... 59
      1. Alternating Bit Protocol ...................................................................................................... 59
   B. **SCM MODEL**.................................................................................................................... 64
      1. Go Back N.............................................................................................................................. 64
      2. Selective Repeat .................................................................................................................. 74

VI. **CONCLUSIONS AND RECOMMENDATIONS**.............................................................................. 82

APPENDIX A **CFSM CODE**............................................................................................................ 84
APPENDIX B **SCM CODE**............................................................................................................... 129
APPENDIX C (CFSM) **ALTERNATING BIT**.................................................................................. 172
   INPUT (FSM) ............................................................................................................................... 172
   OUTPUT ....................................................................................................................................... 173
APPENDIX D (CFSM) **SLIDING WINDOW** .................................................................................... 174
   INPUT(FSM) ............................................................................................................................... 174
   OUTPUT ....................................................................................................................................... 175
APPENDIX E (SCM) **GO_BACK_N, W=1**.................................................................................... 176
   INPUT (FSM) ............................................................................................................................... 176
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>177</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>178</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>179</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>180</td>
</tr>
<tr>
<td>APPENDIX F (SCM) GO_BACK_N, W=2</td>
<td>181</td>
</tr>
<tr>
<td>INPUT (FSM)</td>
<td>181</td>
</tr>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>182</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>183</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>185</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>186</td>
</tr>
<tr>
<td>APPENDIX G (SCM) GO_BACK_N, W=3</td>
<td>188</td>
</tr>
<tr>
<td>INPUT (FSM)</td>
<td>188</td>
</tr>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>189</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>190</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>192</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>194</td>
</tr>
<tr>
<td>APPENDIX H (SCM) GO_BACK_N, W=4</td>
<td>201</td>
</tr>
<tr>
<td>INPUT (FSM)</td>
<td>201</td>
</tr>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>202</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>203</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>205</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>207</td>
</tr>
<tr>
<td>APPENDIX I (SCM) SELECTIVE REPEAT, W=1</td>
<td>228</td>
</tr>
<tr>
<td>INPUT</td>
<td>228</td>
</tr>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>229</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>230</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>231</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>232</td>
</tr>
<tr>
<td>APPENDIX J (SCM) SELECTIVE REPEAT, W=2</td>
<td>233</td>
</tr>
<tr>
<td>INPUT (FSM)</td>
<td>233</td>
</tr>
<tr>
<td>VARIABLE DEFINITIONS</td>
<td>234</td>
</tr>
<tr>
<td>PREDICATE-ACTION</td>
<td>235</td>
</tr>
<tr>
<td>OUTPUT FORMAT</td>
<td>238</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>240</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>241</td>
</tr>
<tr>
<td>INITIAL DISTRIBUTION LIST</td>
<td>243</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1, CFSM, two machine behavior representation</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>Figure 2, CFSM model representation</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Figure 3, CFSM specification for stop-and-wait</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Figure 4, CFSM, global reachability analysis, stop-and-wait</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>Figure 5, SCM, two machine behavior representation</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>Figure 6, SCM, general model representation</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>Figure 7, SCM specification for stop-and-wait, finite state machines and variable definitions</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Figure 8, SCM, global reachability analysis, stop-and-wait</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Figure 9, SCM, system reachability analysis, stop-and-wait</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>Figure 10, CFSM run time behavior</td>
<td>22</td>
<td></td>
</tr>
<tr>
<td>Figure 11, CFSM compilation units</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Figure 12, Finite State Machine representation, stop_and_wait</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>Figure 13, CFSM internal reachability graph, stop_and_wait</td>
<td>29</td>
<td></td>
</tr>
<tr>
<td>Figure 14, CFSM, analysis output, stop_and_wait</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>Figure 15, CFSM, analysis output, deadlock/unspecified reception/unexecuted transition example</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>Figure 16, SCM, specification for stop_and_wait</td>
<td>35</td>
<td></td>
</tr>
<tr>
<td>Figure 17, SCM run time behavior</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>Figure 18, SCM compilation units</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>Figure 19, SCM, definitions package template</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>Figure 20, SCM, definitions package, stop_and_wait</td>
<td>42</td>
<td></td>
</tr>
<tr>
<td>Figure 21, SCM, Analyze_Predicate function template</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>Figure 22, SCM, Action procedure template</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Figure 23, SCM, analyze_predicate.a, stop_and_wait</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Figure 24, Input File definitions</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>Figure 25, SCM, Input File, stop_and_wait</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>Figure 26, SCM, global definitions</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>Figure 27, SCM internal global reachability graph, stop_and_wait</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>Figure 28, SCM, system definitions</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>Figure 29, SCM internal system reachability graph, stop_and_wait</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>Figure 30, SCM, output_Gtuple procedure template</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>Figure 31, SCM, output format, stop_and_wait</td>
<td>57</td>
<td></td>
</tr>
<tr>
<td>Figure 32, SCM, analysis output, stop_and_wait</td>
<td>58</td>
<td></td>
</tr>
<tr>
<td>Figure 33, CFSM specification, Alternating Bit</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>Figure 34, CFSM specification, sliding window (w=3)</td>
<td>62</td>
<td></td>
</tr>
<tr>
<td>Figure 35, SCM specification, Go Back N, window size of l..w</td>
<td>66</td>
<td></td>
</tr>
<tr>
<td>Figure 36, SCM, system state analysis, Go Back N, w=1</td>
<td>71</td>
<td></td>
</tr>
</tbody>
</table>
Figure 37, SCM, system state analysis, Go Back N, w=2........................................... 72
Figure 38, SCM specification, Selective Repeat, window size of 1..w. ..................... 75
Figure 39, SCM, system state analysis, Selective Repeat, w=1............................. 80
Figure 40, SCM, system state analysis, Selective Repeat, w=2............................. 80
LIST OF TABLES

Table 1, SCM, two machine predicate-action table format .......................... 14
Table 2, SCM specification for stop_and_wait, predicate action table .................. 16
Table 3, CFSM compilation units .................................................................. 21
Table 4, SCM compilation units ...................................................................... 36
Table 5, SCM, predicate action table, selective repeat ................................. 66
Table 6, SCM, predicate action table, go_back_n ......................................... 75
I. INTRODUCTION

"simulations are set up to answer the question, what if..."

-Hamming, R. W., Future Engineering Practice Course Notes, 11 May 92.

A. BACKGROUND

The past ten years have seen a substantial increase in the need to communicate quickly and reliably over long distances using a wide range of architectures. We, the users, live in a ‘come as you are’ digital society. Access to some sort of network is needed to move information from user to user. This movement of information takes place on many networks (voice, message, data) and at many levels (physical, data link, network, and higher). The networks in use today tend to be a heterogeneous mix of equipment and protocols. We, the network designers/engineers, must allow the user access to the available resources at the lowest cost. To do this a firm understanding of how machines ‘handshake’ and talk to one another must be realized. This is accomplish by applying protocol design principles. These principles can be applied to protocols and studied using a wide range of Formal Description Techniques (FDT’s). Examples of existing FDT’s will be reviewed and the need for an automated set of tools will also be explored.

The need for machines to be able to communicate is inherent in any heterogenous environment. Machines do this through the use of standardized protocols. A protocol is a set of rules that govern the interaction of concurrent processes in distributed systems. Another widely used definition of a protocol is, a set of rules used for communication between two or more processes connected by a communication network. Hand in hand, protocol design and analysis is an important consideration in operating systems, computer networks, and data communications. For a protocol designer/architect to build an appropriate specification, he must use one of a number of modeling techniques.
Models or FDT's of protocols are used for many purposes. They are used to describe the protocol unambiguously so that the exact operation is understood by both the protocol designer, implementor and user. A model is also used to provide a formal framework for a rigorous analysis of the protocol specification. With the advent of internetworking and the birth of Integrated Services Digital Network (ISDN), computer protocols have become more and more complex. The designer must now develop large sets of rules for information exchange that is logically consistent and efficiently implemented. To design a new protocol or to implement an existing one into a computing environment gives rise to a need for such a tool.

There are many formal models available to protocol architects. Some of the more common models include Petri Nets, Communicating Processes, Communicating Finite State Machines (CFSM), System of Communicating Machines (SCM), the Language of Temporal Ordering Specifications (LOTOS), Specification and Description Language (SDL), and Extended State Transition Model Language (ESTELLE). LOTOS and ESTELLE are formal description techniques developed by the International Organization for Standardization (ISO) working laterally with the International Telephone and Telegraph Consultative Committee (CCITT).

The ISO is a standards publishing body including the American National Standards Institute (ANSI). In 1980, the ISO saw the advantages of standardizing a hierarchy of protocol services as a reference model for protocol designers. The model includes seven layers: physical, data link, network, transport, session, presentation, and application. The layer/class of protocols that will be analyzed in this thesis are the data link layer protocols.

Each of the models that will be discussed have a means to amplify design principles of communications systems [BART 87]. The first principle is for a model to reflect the behavior of the protocol. Behavior is modeled using conformance models, an example found in [RAND 92]. Secondly, the model must allow refinement by the user. Safety and lively properties should be proven true. And the last principles that must be supported are
those of concurrency and nondeterminism. A good overview is found in King’s article [KING 91].

The need for a variety of models is apparent when it is considered that the interaction between machines occurs at different levels in the OSI structure and that behavior can be quite different among levels and machines. The models listed above will be reviewed in the following section. Most of the models have a means to informally follow the design principles. Upon close inspection of each FDT it is apparent that no one is perfect for all applications. As such, an automated tool will be presented that will make the use of two models (CFSM and SCM). The intuitive feel of each protocols coupled with the power of automating such a such models will enable the user to fully enjoy the important design principle of refinement.

B. OBJECTIVES

The objective of this thesis is to present a means of automating two powerful models of protocol validation and analysis. The first that was to be automated is the CFSM model. The data structures and program entities were developed and verified. The second tool automated is the SCM model. Although the SCM used many of the underlying data structures and logic of the CFSM, the SCM model is much more elegant and much more complex. The output information is provided to the user in an intuitive format. Once the two models were fully functional, test cases were input to the models to verify the functionality. Finally, a select number of existing protocols were input and the analysis compared to previous research using a manual method.

C. SCOPE

This thesis presents automated implementation of both the CFSM and SCM models. The implementation of each model was limited to two machine protocols. The specifications covered in this thesis lend themselves very well to the simulation or automated analysis.
The unboundedness properties of CFSM channels are obviously limited to the hardware that the tool is run on. A channel can have a bound the size of the largest machine register, in the case of the test runs, a SUN SPARC station, the upper bound was that of the largest integer($4.294967 \times 10^9$ items).

An analysis of select data link protocols are included to illustrate the use of the CFSM and SCM automated models. The specifications will only address procedural rules, not formatting of messages.

D. ORGANIZATION

This thesis is organized into three sections. The first section includes Chapters II and III. Chapters II and III give background information of pertinent models and language considerations. The next section, Chapters IV and V, give a detailed description of how the code was implemented to reflect the behavior of the two models. The final section, Chapter VI, describes the specifications of Alternating Bit, Go_Back_N, and Selective Repeat network protocols. It also describes how the user inputs the information into and receives output from the tool. A means for validating each automated model is discussed in this chapter. Finally, Chapter VII includes conclusions made based on the thesis work and recommendations for future work in the area.
II. BACKGROUND OF MODELS

A. GENERAL

This section contains an overview of some existing FDT's. Each model is a different way to represent a protocol design or reflect network behavior. Each has its own inherent advantages and disadvantages, of which must be considered before application. Two models listed (ESTELLE and LOTOS) are automated.

The first method of description is Petri Nets. Petri Nets are a graphical representation of a systems's states and state changes. The possible states are captured using places which can hold tokens. A particular state is represented by a movement of tokens to states. State changes are described using transitions. This can be visualized as being similar to a directed graph. The input and output arcs associated with each transition determine how token placement changes. The behavior of a system can be determined by examining token movement within the net. Deadlock and freedom of livelock are examined in this model. The complexity of Petri Net representation increases with the size of the protocol being modeled. A major consideration for using this model is the intuitive feel of a protocol is lost on the complex cases.

Another class of FDT are models is called “communicating processes.” The following description is more closely examined in Lundy’s article[LUND 92b]. The elimination of a set of global states is done through the use of invariants. Rather than generating the set of all possible states, and inspecting them to be sure no undesirable state exists, an assertion is made. The assertion states the desired property. It is proven that the protocol always satisfies the assertion, this must be proven without having to compute all the possible states which might be reached. The communication between processes takes place between unbounded FIFO queues. Processes are emulated by use of variables and statements. The execution of an action is an atomic event and no two actions may occur simultaneously. Since communication between processes can only occur using FIFO queues, actions may
only follow a given sequence. The SCM model demonstrates how this is characteristic is overcome.

Extended State Transition Model Language (ESTELLE) can describe a system in terms of a set of communicating extended finite state machines through use of FIFO channels, similar the to definition of CFSM. This model describes the protocols as a collection of modules, each module is an extended FSM having memory-the difference between an CFSM and Estelle model is that the CFSM model has no memory. Modules of an entity can communicate through FIFO channels[SARI 91]. Messages are exchanged between entities as parameters to the modules. Estelle is based on Pascal and the extension of the language is a feature available to the programmer/user. The models automated in this thesis are similar to Estelle, however, data representation is implemented differently and the power of the language implemented (ADA) is utilized. ESTELLE also allows dynamic module creation/destruction and transition priorities. A model implementation consideration is ESTELLE cannot adequately represent broadcast channels, a shortcoming that the SCM model has shown very well suited for, such as CMSA/CD analysis [LUND 91a].

Specification and Description Language was designed and implemented by groups SGXI and SGX of the CCITT. It was meant as a tool for the design and specification of telephone switches and their underlying protocols. Currently there are two versions of SDL; a graphical tool and a text program tool. Processes are represented by flowcharts, which could be concurrent to other processes. The eight traditional flowchart symbols represent atomic actions such as internal events, input and output, boolean expressions, wait conditions, statements, transitions, and connectors. Each flowchart has an associated channel (queue) used to process messages. The Holzman text [HOLZ 91] includes a more specific definition of SDL with some examples. One advantage to this approach is the user gets a feel, graphically, of the behavior of the protocol. The process execution is somewhat restricted to the properties that a FIFO queue has.
The ISO language of Temporal Ordering Specifications (LOTOS) is a means of representation using hierarchically structured processes. As with ESTELLE, LOTOS was also developed by the ISO. A hierarchy of processes can correspond to one entity [SIST 91]; a concept that is reflected in object-oriented design environments. Systems represented using this model are organized using a set of interacting processes which exchange information with each other and with the external systems environment through gates. LOTOS is a superset language consisting of an abstract data type language and an algebraic notation language, both of which uphold good design principles covered earlier. Interaction is synchronous through gates that have a one-to-one mapping to interaction points.

The tool implemented in this thesis uses the technique of representing portions of machine behavior as abstract data types, as demonstrated using a technique of interpreting machine behavior through use of finite state machine representation, as with ESTELLE, will also be integrated into the design. Plans for future upgrade of this tool include a graphical interface similar to that of SDL.

B. COMMUNICATING FINITE STATE MACHINES

One of the first manual tools used for analyzing communication protocol behavior was the communicating finite state machine (CFSM) model. This modeled each machine in the network as a finite automaton, or finite state machine, with communication channels between pairs of machines modeled as one-way, infinite length FIFO queues. There has been a great deal of work in this area, a few include [PENG 91], [VUON 83] and [RUDI 83]. The model is defined for an arbitrary number of machines; however for simplicity sake will be presented as a two machine model as shown in Figure 1.

![Figure 1: CFSM, two machine behavior representation](image-url)
In this section the CFSM model will be defined [GOUD 83] followed by a simple protocol analysis to illustrate the model.

A communicating machine $M$ is a finite, directed labeled graph with two types of edges, sending edges and receiving edges. 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 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; the two terms are used interchangeably.

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 $[m,c_m,c_n,n]$ 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 $[m,c_m,c_n,n]$ 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. Channel $c_m$ contains the messages sent from $M$ and $N$, and channel $c_n$ the messages sent from $N$ to $M$. The string $c_j$ will be referred to as channel $c_j$.

The initial global state of $(M,N)$ is $[m_o,E,E,n_o]$, where $m_o$ an $n_o$ 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 $c_n(c_m)$.

If $s_1=[m,c_i,c_j,n]$ is a global state of $(M,N)$, and state $s_2$ follows $s_1$ if there is a transition (in $M$ or $N$) which can be executed in $s_1$, such that the resulting state is $s_2$. A state $s_2$ is
reachable from state \( s_i \) 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 \([m,c_m,c_n,n]\) 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, such that 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 transition which they represent. The 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. An overview of the functional units of the CFSM model is shown in Figure 1.

![Figure 2: CFSM model representation.](image)

A global state \([m,c_m,c_n,n]\) 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 \([m,c_m,c_n,n]\) is an unspecified reception state if one of the following two conditions are 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 \( c_m \) is \( g \); and none of \( n \)'s outgoing transitions is labeled '+\( g \).

A simplified version of the \textit{stop-and-wait} data link protocol will be analyzed as an example of analysis with the CFSM model. The interfaces between layer 6 (user) and layer 2 (data link) of the Open Systems Interconnection (OSI) model is transparent in all the examples addressed in this thesis. An assumption is made that the higher layer has passed the information/frames without error. The frames at each layer have accomplished the appropriate concatenation of header and address information. So, at layer 2, this protocol consists of two distinct entities, a sender and a receiver. Machine one serves as the sender and machine 2 serves as the receiver as shown in Figure 3. The sender places a frame on the channel to the receiver. The receiver senses a frame on the incoming channel and accepts the message from the channel, removing the message from the incoming channel. The receiver then sends an acknowledgment packet to the sender. The sender senses the acknowledgment packet and is clear to send another frame of information to the receiver.

![Figure 3: CFSM specification for stop-and-wait.](image)

The finite state machines in Figure 3 represent the behavior of the definition of the \textit{stop-and-wait} protocol. The \(-D\) represents sending data, \(+D\), receiving data, \(-A\), send
acknowledgment, and $+A$, receive acknowledge. As per the definition of the CFSM model, there is two channels, one from machine 1 to machine 2 and one from machine 2 to machine 1. The notch on state 1 of both machines represents the initial/starting state.

The global reachability analysis graph shown in Figure 4 is free from deadlock, unspecified receptions, and unexecuted transitions.

$$
\begin{array}{c}
[0, E, E, 0] \\
\downarrow -D \\
[1, D, E, 0] \\
\downarrow +D \\
[1, E, E, 1] \\
\downarrow -A \\
[1, E, A, 0] \\
\downarrow +A \\
\end{array}
$$

Figure 4: CFSM, global reachability analysis, stop-and-wait.

This model has many desirable features as well as some disadvantages that are improved in the SCM model. The one glaring disadvantage is that the analysis might not terminate if the queue length is unbounded. The number of global states in Figure 4 is trivial, but for complex specifications the number of states will lead to a combinatorial state explosion. This is even true when the queue length is very restrictive. As pointed out in [LUND 91b], the specification of a practical protocol can be so complex, containing hundreds of states and transitions, that the user can not be sure of the intended specification or grasp the intuitive feel for what the protocol is intended to do. This model has the advantage of simplicity and a method of analysis that can be easily automated.
C. SYSTEM OF COMMUNICATING MACHINES

In this section the model used to specify and analyze protocols is briefly described. A more detailed description appears in [LUND 91a]. Following the definition of the model will be an analysis of a simple protocol to illustrate the model.

A system of communicating machines is an ordered pair \( C = (M, V) \), where
\[
M = \{m_1, m_2, ..., m_n\}
\]
is a finite set of machines, and
\[
V = \{v_1, v_2, ..., v_n\}
\]
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 \((S_i, s_0, L_i, N_i, \tau_i)\), where

1. \( S_i \) is a finite set of states;
2. \( s_0 \in S_i \) 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 of \( L_i \cup R_i \) and \( a \) is an action on the variables of \( L_i \cup R_i \cup W_i \);
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 set $L_i$ of local variables specifies a name and a range for each. The range must be a finite or countable set of values.

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 together with its enabled outgoing transitions. Two system states are equivalent if every machine is in the same state, and the same outgoing transitions are enabled.

The initial system state is the system state such that every machine is in its initial state, and the enabled outgoing transitions are the same as in the initial global state.

The global state of a system consists of the system state, plus the values of all variables, both local and shared. The initial global state is the initial system state, with the additional requirement that all variables have their initial values. A global state corresponds to a system state if every machine is in the same state and the same outgoing transitions are enabled.

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 executed whenever $m_i$ is in state $s_1$ 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. The format for the associated predicate-action table is shown in Table 1.
Figure 5: SCM, two machine behavior representation.

TABLE 1: SCM, two machine predicate-action table format.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transitions</td>
<td>Values of variables that must hold true for the transition to be enabled.</td>
<td>The local and GLOBAL variable behavior when the transition is taken.</td>
</tr>
<tr>
<td>for Machine 1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transitions</td>
<td>Same as above.</td>
<td>Same as above.</td>
</tr>
<tr>
<td>for Machine 2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that if the values of all variables are restricted to some finite range, then the model can be reduced to a simple finite state machine. Otherwise an infinite number of global states are possible. However, even if the number of global states is infinite, the number of system states is finite, because of the finiteness of each machine. This may allow a reachability analysis on the system states, when a reachability analysis on the global states
is infinite. General behavior of the SCM model is shown in Figure 1 and the general SCM model representation is found in Figure 1.

The stop-and-wait protocol will also be used to demonstrate the analysis using the SCM model. The stop-and-wait protocol specification is the same as defined in the previous section. The specification as represented by the SCM model is shown as a set of finite state machines and a predicate-action table.

The finite state machine representation for the SCM model is similar to the CFSM example. Again this protocol is only demonstrated with two machines. The FSM's are shown in Figure 3. Also shown are the local and global variables. The local variables in Machine 1 and 2 can have the values of D(data), A (acknowledgment), and E(empty). The

![SCM Model Diagram]

Figure 6: SCM, general model representation
initial value for \textit{out\_buff} is D and the initial values for all other variables is E. The system global variable, \textit{CHAN} can have the same values as the local variables.

\textit{Machine 1}

\textit{Machine 2}

![StateMachine Diagram]

Figure 7: SCM specification for \textit{stop-and-wait}, finite state machines and variable definitions.

The predicate-action table is shown in Table 2. For this example the assumption is made that data is always made available to the \textit{CHAN} from \textit{out\_buff}.

TABLE 2: SCM specification for \textit{stop\_and\_wait}, predicate action table.

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D</td>
<td>\textit{CHAN} = E \land \textit{out_buff} \neq E</td>
<td>\textit{CHAN} := \textit{out_buff}</td>
</tr>
<tr>
<td>+A</td>
<td>\textit{RET} = E</td>
<td>\textit{RET} := E \textit{CHAN} := E</td>
</tr>
<tr>
<td>+D</td>
<td>\textit{CHAN} = E</td>
<td>\textit{in_buff} := \textit{CHAN}</td>
</tr>
<tr>
<td>-A</td>
<td>\text{true}</td>
<td>\textit{RET} := E \textit{in_buff} := e</td>
</tr>
</tbody>
</table>
The global state reachability and system state reachability graphs are found in Figure 4 and Figure 4. The format for the global state tuple of the *stop-and-wait* protocol is:

\[ \text{Machine1\_state, out\_buff, CHAN, RET, in\_buff, Machine2\_state} \]

![Diagram of global reachability analysis, stop-and-wait.](image1)

The format for a system state tuple for all cases of analysis is:

\[ \text{Machine1\_state, Machine2\_state} \]

![Diagram of system reachability analysis, stop-and-wait.](image2)
The SCM model has desirable properties found in the CFSM model as well as overcoming some of the inherent disadvantages of the CFSM model. In the SCM model the behavior of the protocol can be clearly and quite adequately represented, maintaining an intuitive feel of the specification. The SCM model can ameliorate the combinatorial state explosion through the use of system state analysis, greatly reducing the generated states. Instead of implicit queues, shared variables are used for communications between processes. This allows communications between machines in non sequential manner, unlike a FIFO queue representation in the CFSM model.

The final advantage is the nature of the SCM’s representation of a protocol gives it the feel of a programming language. Although more complex to program than a CFSM model, the actions associated with the FSM and the predicate-action table lend themselves to automated implementation.

D. LANGUAGE CONSIDERATIONS

Which language should the CFSM and SCM models be implemented in? Before all the available languages were researched, a list of desirable properties that the language must have (specific to the models), was developed. After a close inspection of the definition and nuances of the CFSM model, SCM model, and the reachability analysis generated, there were a number of language properties that were found desirable to this project.

The language properties should support hardware and software design issues. The code must be portable from one architecture to another. The language should have a means to create different class instances from a base class. An intuitive means to provide meaningful output of the analysis and programming error messages to the user enhances the program’s utility. Since the program must simulate network specifications there is an inherent need to be able to do multiprocessing or multitasking in the programming environment. The language of choice should enforce the rules of strong typing, that is not
allowing mixing of types and subtypes. The final property of the language should be its ease of use and understandability.

The language of choice should be portable between different machines. It cannot be assumed that the user has access to a mainframe computer or workstation. The language should be compilable on a machine as small as a personal computer.

Implementation of the models should help the user to avoid and detect mistakes. The environment should prompt the user when a syntactical or semantic error is made. The error messages should be meaningful. Inherent to this requirement, the language should enforce strict definitions of atomic structures, such as data structures.

Dynamic list creation /deletion are necessary in reachability graph construction. This allows flexible and ultimately limitless (hardware specific) analyses to be done. Linked list creation and traversal should make use of reusable programming units. The logic for creating new nodes should allow the program to ‘remember’ where the last node was built. This should be done automatically, without user intervention after compile time.

An important property, although subject to varying opinion, is ease of use. The project is developed in one language, but the human interface to the underlying code must be understandable and intuitive. Hand in hand with ease of use, is ease of maintainability. There should be enough on-line and off-line help to allow the user to navigate the the user interface. An understandable debugger was also a factor in the choice.

Ada was chosen because it supports the above mentioned properties. It is a language that is portable between different architectures. It supports generic class creation and instantiation. Through the use of predefined input and output packages, the user is allowed to build a suitable interface environment. With the use of exception handling meaningful error messages can be created and employed. Ada also has the ability to multitask, simulating parallel processing. Finally, Ada is easy to use. The code can be read by a novice and understand what is meant to happen.

The language of C/C++ was not chosen due to a few limitations. At the time of this writing it could not support multitasking needed to simulate concurrency or broadcast
networks. It was also difficult to do generic-like coding. The object orientation of the design lent itself very nicely to the structures used in the CFSM and SCM model as covered in [RUMB 91]. It became apparent that there was reused code that would have been more efficiently implemented with generic data structures. Although it could have been done with the use of macro-like instructions, generic packages made the project more compact and efficient. Generic package creation and instantiation was not supported by the current version of the C/C++ compiler. The C programming environment does not support exception handling; programming error detection messages were vague and could not be developed by the user. A good means of automatic implementation of error messages in the C++ environment was not available at the time of this publishing. Ada could do this through use of exception handling.
III. A PROGRAM FOR GENERATING A CFSM REACHABILITY ANALYSIS

In this chapter the organization of the CFSM program will be described. The means for input, output, and reachability analysis will be highlighted. Excerpts of the underlying code will be accompanied by a brief explanation. The formal definition of the CFSM model found in Chapter II is the basis for constructing the program.

A. PROGRAM STRUCTURE

The structure of the CFSM program is based on functional units (objects) of the general CFSM model. The data structures of the basic objects must represent communication channels, machine states, transitions, and a means for capturing global tuple (state) values. In addition to constructing the fundamental data structures, there must be an intuitive input mechanism for the FSM's and an understandable display of the analysis.

Implementation details should be hidden from the user. Operations such as loading the CFSM into memory, performing a reachability analysis, constructing the global reachability graph, and traversing the graph during searches/output are independent of the specific protocol to be analyzed.

The program consists of input related procedures, a reachability analysis, and output procedures. To help manage such a complex and large programming project, separate compilation units were used. The compilation units were physically grouped by file according to the function it performed as shown below:

TABLE 3: CFSM compilation units.

<table>
<thead>
<tr>
<th>Compilation Unit</th>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_in_file</td>
<td>parse text input file</td>
<td>input.a</td>
</tr>
<tr>
<td>load_machine_array</td>
<td>builds machine adjacency lists from parsed file</td>
<td>input.a</td>
</tr>
<tr>
<td>Compilation Unit</td>
<td>Description</td>
<td>File</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>build_Gstate_graph</td>
<td>builds global reachability analysis graph</td>
<td>reachability.a</td>
</tr>
<tr>
<td>clear_pointers</td>
<td>clears values for another input file</td>
<td>reachability.a</td>
</tr>
<tr>
<td>search_for_tuple</td>
<td>performs BFS search of graph</td>
<td>search.a</td>
</tr>
<tr>
<td>IsEqual</td>
<td>compares global records for equality (similar to =)</td>
<td>search.a</td>
</tr>
<tr>
<td>output_Gstate_node</td>
<td>format for node output</td>
<td>output.a</td>
</tr>
<tr>
<td>output_Gstate_transition</td>
<td>format for transition output</td>
<td>output.a</td>
</tr>
<tr>
<td>output_Gstates</td>
<td>traverses graph and outputs nodes and transitions</td>
<td>output.a</td>
</tr>
<tr>
<td>output_machine_arrays</td>
<td>format output of contents of adjacency lists</td>
<td>output.a</td>
</tr>
<tr>
<td>create_output_file</td>
<td>creates file for analysis output</td>
<td>output.a</td>
</tr>
</tbody>
</table>

This use of separate subprograms (compilation units) facilitated the development of the SCM program from existing CFSM code.

The behavior at run time is shown in Figure 10 and associated files of the CFSM program are shown in Figure 11.

Figure 10: CFSM run time behavior
During the design phase of the program it became apparent that some software components and structures were used more than once. For instance, when doing a reachability analysis many types of stacks and queues were used. Although the underlying data types were different, the algorithm for each structure was exactly the same. To increase efficiency, generic packages were used. Generic units are defined as a reusable software module or a program unit template [GONZ 91].

The implementation of stacks and queues is accomplished using generics. For instance, within the program there is a need for a queue of characters representing the flow of information on the channels between two machines. To assist in the construction of the reachability graph there needs to be a queue of pointers to graph nodes (see Section C). Each type of queue has some common procedures and functions. Each needs procedures to clear the queue, enqueue, and dequeue. Each must also have functions that return the value
of the first item of the queue, determine if the queue is empty and determine if the contents of two queues are equal. If these common functions and procedures had to be rewritten with a different underlying data type, the number of compilation units as well as the object code would increase; thus the user would be saddled with more ‘waiting’ time.

The protocol environment can be modified by using generic parameters. The generic package, *queues*, has two parameters to the object- the item type and the maximum size of the queue. This allows the user to define what type of items are contained in the channel(queue) and how big the channel(queue) can be. Two instantiations of *queues* in the program are:

```plaintext
package queue_pack is new queues(character, MAX=>3);
package Gpointer_queue_pack is new queues(Glink_type, MAX=>10);
```

The *queue_pack* package defines a queue of characters. A ceiling or bound can be placed on the amount of messages on a channel. If an unbounded channel is to be simulated the maximum allowable integer can be given. The pointer queue *Gpointer_queue_pack* gives the user a means to determine the maximum size of a reachability graph. Although, in the general case, a large number is preferred to allow all tuples (states) to be generated in a protocol reachability graph. The generic package *stacks* was implemented in a similar fashion.

### B. INPUT

An important step in designing the CFSM and SCM programs is developing a meaningful method of inputting the finite state machines. The graphical representation of a simple FSM conveys a behavior associated with a protocol specification. A means to transfer this graph into a data structure that can be used in the reachability analysis was developed.

The FSM’s were input as a text file. This file is built by the user with a set of language rules similar to Backus-Naur Form (BNF). The input file is parsed one line at a time. Each line is read into a line buffer and tokens formed according to the rules defined below. From
the tokens, an internal data structure is generated to represent the set of finite state machines. The list of valid instructions for finite state machine input is:

```
start
machine <natural>
state <natural>
initial_state <natural> <natural>
trans <-|-> <a|b|...|z|A|B|...|Z> <natural>
finish
```

The tokens are cast into either enumerated types (instructions) or integers (integer variables). The integer variables have been formally defined within the main procedure in Appendix A.

The meaning of the instructions are:

```
start Serves as a beginning flag for the file.
machine Defines the current machine.
state Defines the current state.
initial_state The initial/start state for machines one and two.
trans Transition type, transition message, and next state.
finish This token serves as an ending flag for the file.
```

Representation of a finite state machine using the above convention has some inherent constraints. Since an input token, such as the transition -D, cannot be directly caste into an enumeration token (no special characters at the beginning of a token), the (-,+) must be converted separately to (snd,rcv) tokens. The use of alphabetic characters to represent messages in a channel, limits messages to 52 distinct types (a..z,A..Z). The input file for stop_and_wait is:

```
start
machine 1
state 0
trans -D 1
state 1
trans +A 0
machine 2
state 0
trans +D 1
state 1
trans -A 0
initial_state 0 0
finish
```
The data structure representing the CFSM is then used to construct a reachability analysis graph. The two data structures that support directed graphs (or finite state machines) are adjacency lists and adjacency matrices. Since the use of adjacency matrices to construct directed graphs can lead to wasted hardware memory, adjacency lists (one dimensional array of linked lists) were implemented.

The data structure to build the adjacency list and the constraints are:

```adala
type machine_array_record_type is record
    transition : cfsm_transition_type;
    message : character;
    next_state : natural;
    executed : executed_type;
    Mlink : Mlink_type
end record;

type machine_array_type is array(positive range<>)
of Mlink_type;

type system_array_type is array(1..2) of machine_array_type;
```

Some data structures shown above are peculiar to Ada. Access types are data types that provide an access (“pointer”) to an object of another type or subtype. It reserves storage locations during the execution of a program dynamically by use of a memory allocator. A record type is simply a collection of elements where each element is referred to by its name.
The array of linked lists is defined as an unconstrained array; whereas, at compile time, the number of machines is set at two. To illustrate the finite state machine data structure the CFSM stop_and_wait protocol is shown in Figure 12.

Machine 1

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Message</th>
<th>Next State</th>
<th>Executed</th>
<th>Mlink</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>transition</td>
<td>snd</td>
<td>1</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>transition</td>
<td>rcv</td>
<td>A</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>transition</td>
<td>snd</td>
<td>0</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Machine 2

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Message</th>
<th>Next State</th>
<th>Executed</th>
<th>Mlink</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>transition</td>
<td>rcv</td>
<td>1</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>transition</td>
<td>snd</td>
<td>A</td>
<td>no</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>transition</td>
<td>snd</td>
<td>0</td>
<td>no</td>
<td></td>
</tr>
</tbody>
</table>

Figure 12: Finite State Machine representation, stop_and_wait

C. REACHABILITY ANALYSIS

In order to determine if all states in a network are reachable a graph is constructed. After the textual representation of the CFSM is input, the adjacency lists are constructed as described in the previous section. The initial states for each machine indicate the starting...
position of each list. From the lists a directed graph is constructed. It is from this directed graph that deadlocks and unspecified receptions are sensed and the appropriate output message is displayed.

The algorithm to construct the global reachability graph is:

```
loop
    for machine1 array_index in 1..rowsize loop
        if machine1 (array_index).transition=snd or
           machine1(array_index).message=top_of.queue1 then
            make temp Gstate record
            search list for Gstate record
            ifound then link current to found state
            else make new node and link into Gstate_graph
                   and push pointer onto pointer stack
        else none_found
    end loop

    for machine2 array_index in 1..rowsize loop
        if machine2 (array_index).transition=snd or
           machine2(array_index).message=top_of.queue2 then
            make temp Gstate record
            search list for Gstate record
            ifound then link current to found state
            else make new node and link into Gstate_graph
                   and push pointer onto pointer stack
        else none_found
    end loop

    if stack is empty then
        raise STACK_EMPTY
    else
        pop last Gstate
    end loop
```

The initial global state tuple (node) is created from the starting state of each machine's adjacency list. From the top node, tuples (global states) are added to the graph using the reachability algorithm. The algorithm shows the graph being constructed with stack based implementation, allowing a breadth first construct. The option is given to the user to construct the graph depth first. A case statement is used to toggle between stack or queue procedures/functions (this is not shown in the algorithm above.) Figure 13 shows the internal representation of the global reachability graph for the *stop_and_wait* protocol.
Figure 13: CFSM internal reachability graph, *stop_and_wait*
Note that when a tuple is generated a data structure representing a transition and a node are separately added. The field \textit{new\_node} was included in the transition structure to allow proper traversal of the graph. The current version of the program allows for four transitions from each tuple. This can be expanded upward if needed.

Exception handlers were used to maintain control in the reachability graph construction. Whenever a queue or stack is empty the control is handed to the exception handler to continue program execution. The exception handlers allow definition of specific error conditions to be sensed and appropriate action taken.

During graph construction, global state tuples are identified that satisfy the deadlock and unspecified reception properties. If a global state node has only receiving transitions from it and both the queues are empty, a deadlock message is displayed to output. If the global state node has outgoing receive transitions and the head of the respective queue does not match the receive transition (assuming the queue(s) are not empty) then an unspecified reception message is displayed to output. When the construction of the graph is complete the adjacency lists are checked for any unexecuted transitions. The contents of the lists are displayed after the output of the graph is done. Unexecuted transitions are identified by the \textit{execution} field, with a \textit{no} entry. For an example see Figure 15

Upon completion of the reachability graph construction, a pointer to the top global state node is passed to the output procedure.

When constructing a reachability graph there are two factors that need to be considered—run time and the size of the graph generated. As noted earlier, a ceiling can be placed on the size of the graph by the user prior to compilation. Ideally, a specification can be input to the program and an analysis could run for as long as needed (perhaps days); however, most computer systems are limited by storage. The question of storage capacity is left to the user of the program. A determination must be made as to how large a graph to anticipate (worst case is the largest integer represented on the register) and how much storage space can the underlying system provide.
The design of the program addresses the issue of running time. The running time, or complexity, of the reachability analysis is dominated by the algorithm that governs the directed graph traversal. All traversals are done in both models (CFSM and SCM) in a recursive, depth first manner. The complexity, or big $O$ notation, for traversals of a directed graph can easily be defined. Consider a reachability graph $G=(V,E)$ consisting of a set $V$ of vertices (nodes), and a set $E$ of edges (transitions). Each edge corresponds to a pair of distinct vertices in the directed graph. The running time or complexity of such a graph traversal is proven by induction to be $O(|V| + |E|)$. A rigorous proof of the complexity appears in [MANB 89].

D. OUTPUT

The output procedure for the CFSM tool displays the reachability graph and associated messages to both a text file and the default output device. The output procedure has as a parameter a pointer to the top global state (node). From the top node the graph is traversed in a depth first manner and saved to an output medium. The contents of the adjacency list are also displayed to output providing a means to cross check the CFSM construction and identify unexecuted transitions as shown for stop_and_wait in Figure 14.
REACHABILITY ANALYSIS of: stop_and_wait

1 [0, E, E, 0] -D [1, D, E, 0] 2
2 [1, D, E, 0] +D [1, E, E, 1] 3
3 [1, E, E, 1] -A [1, E, A, 0] 4
4 [1, E, A, 0] +A [0, E, E, 0] 1

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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 Array Contents</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>

* The nodes generated by the analysis were done in a breadth first manner

Figure 14: CFSM, analysis output, stop_and_wait

To illustrate the formatting of the model’s output an example is presented. Assume that a CFSM exists reflected by the following specification:

Machine 1

The input file is:
The output file reflecting deadlock, unspecified receptions, and unexecuted transitions is shown in Figure 15.

Figure 15: CFSM, analysis output, deadlock/unspecified reception/unexecuted transition example.
IV. AN AUTOMATED TOOL FOR SCM REACHABILITY ANALYSIS

In this chapter, a program is introduced that automates the SCM model. It provides an intuitive environment to input a protocol specification and receive the analysis in an understandable format. Since the model only uses variables and finite state machines to describe a protocol's behavior, it is considered an approximate model. There are certain details of protocol design, such as message and header format, that are abstract from the analysis. The succinctness of a protocol representation helps analyze the logic and structure without getting lost in a myriad of detail.

The organization of this program is similar to that of the CFSM program. A means for input, output, global reachability analysis, and system reachability analysis are highlighted. Excerpts of the code are accompanied by a brief explanation of structure and application. The formal definition of the SCM model found in Chapter II is the basis for constructing the program.
The protocol specification and analysis of the stop_and_wait data link protocol will be used throughout this chapter to demonstrate the interface of the program to the user. The specification for the sample protocol is shown in Figure 16.

**Figure 16: SCM, specification for stop_and_wait.**

### A. PROGRAM STRUCTURE

The structure of the SCM program is similar to the CFSM implementation. There must be a means for input, output, and reachability analysis. The input is more complex because not only must the FSM's must be entered, but also variable definitions and the associated predicate-action table as shown in Figure 16. The input can be viewed as hierarchical. The global and system reachability analysis are performed using different algorithms and are
described in later sections. The code for producing output is identical to the CFSM program with some addition to allow the user to tailor variable output.

![Diagram](image)

**Figure 17: SCM run time behavior.**

The program, written in Ada, consists of packages, procedures, and functions that make up the basic structure mentioned above. A package specifies a group of logically related entities, such as types, and objects of those types as defined in [GONZ 91] and [SKAN 88]. The procedures and function that were subject to change/updates were also treated as separate compilation units. To give a 'feel' for the different components of the program, the separate compilation units and the files that contain them are shown in Table 4.

**TABLE 4: SCM compilation units.**

<table>
<thead>
<tr>
<th>Compilation Unit</th>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_in_file</td>
<td>parses text input file</td>
<td>input.a</td>
</tr>
<tr>
<td>load_machine_array</td>
<td>builds machine adjacency lists form parsed file</td>
<td>input.a</td>
</tr>
<tr>
<td>build_Gstate_graph</td>
<td>builds global reachability analysis graph</td>
<td>global_reachability.a</td>
</tr>
</tbody>
</table>

36
<table>
<thead>
<tr>
<th>Compilation Unit</th>
<th>Description</th>
<th>File</th>
</tr>
</thead>
<tbody>
<tr>
<td>clear_pointers</td>
<td>clears the values for another input file</td>
<td>global_eachability.a</td>
</tr>
<tr>
<td>search_for_Gtuple</td>
<td>performs BFS search of global reachability graph</td>
<td>global_search.a</td>
</tr>
<tr>
<td>IsEqual</td>
<td>compares global records and associated transitions for equality</td>
<td>global_search.a</td>
</tr>
<tr>
<td>build_Sstate_graph</td>
<td>builds system reachability analysis graph</td>
<td>system_reachability.a</td>
</tr>
<tr>
<td>search_for_Stuple</td>
<td>performs BFS search of graph</td>
<td>system_search.a</td>
</tr>
<tr>
<td>IsSysStateEqual</td>
<td>compares system records and associated transitions for equality</td>
<td>system_search.a</td>
</tr>
<tr>
<td>output_Gstate_node</td>
<td>format for node output</td>
<td>global_output.a</td>
</tr>
<tr>
<td>output_Gstate_transition</td>
<td>format for transition output</td>
<td>global_output.a</td>
</tr>
<tr>
<td>output_Gstates</td>
<td>traverses graph and outputs nodes and transitions</td>
<td>global_output.a</td>
</tr>
<tr>
<td>output_machine_arrays</td>
<td>format output of contents of adjacency lists</td>
<td>global_output.a</td>
</tr>
<tr>
<td>output_Sstate_node</td>
<td>format for node output</td>
<td>system_output.a</td>
</tr>
<tr>
<td>output_Sstate_transition</td>
<td>format for transition output</td>
<td>system_output.a</td>
</tr>
<tr>
<td>output_Sstates</td>
<td>traverses graph and outputs nodes and transitions</td>
<td>system_output.a</td>
</tr>
<tr>
<td>output_Gtuple</td>
<td>format global record for output</td>
<td>user_output.a</td>
</tr>
<tr>
<td>variable_definitions</td>
<td>user defined protocol variables</td>
<td>user_definitions.a</td>
</tr>
<tr>
<td>Analyze_Predicates</td>
<td>performs analysis of predicates and determines which transitions are enabled</td>
<td>predicate_action.a</td>
</tr>
<tr>
<td>Compilation Unit</td>
<td>Description</td>
<td>File</td>
</tr>
<tr>
<td>------------------</td>
<td>------------------------------------------------------------------------------</td>
<td>---------------------</td>
</tr>
<tr>
<td>Action</td>
<td>changes the global and local variables based on the transition executed</td>
<td>predicate_action.a</td>
</tr>
</tbody>
</table>

The user has access to the last three files shown in Table 4. The variable definition package, Predicate_Analysis function, and Action procedure contained in these files are modified by the user to reflect the specific protocol to be analyzed. Formats for each unit will be outlined in Sections B, C, and D. The other files and procedures will remain hidden from the user because they are independent of any protocol to be analyzed. Figure 18 shows the files and generic units used at compilation time.

![Diagram of SCM compilation units](image-url)

Figure 18: SCM compilation units.
B. INPUT

An intuitive and understandable means to input a specification is helpful in any protocol analysis program. A protocol specification is divided into shared and local variable definitions, predicate-action table representation, and finite state machine storage structure.

The different parts to each specification to be analyzed must be input in a certain order. The definition package, Analyze_Predicate function, and Action procedure must be constructed and compiled before the program is executed. When the program is executed the user then inputs the FSM text file and obtains the reachability analysis. Since the compilation of the program depends on the variables in the definition package, this package is written and compiled first. This is a technique to verify variable definition correctness in the Ada environment. Once the definitions package is compiled the Analyze_Predicate function and Action procedure can then be compiled. This step-wise refinement facilitates error free specification representation.

At any point in execution of the program the status of all variables is kept in the global state record. Each node in the global reachability graph has a copy of this record. Within the main procedure a global state record is declared as

```ada
type Gstate_record_type is
record
    machine1_state : machine1_state_type;
    machine2_state : machine2_state_type;
    global_variables : global_variable_type;
end record;
```

Having the machine and global types defined in a separate package ensures that only select pieces of code can be modified by the user. The definitions package contains the machine1_state_type, machine2_state_type, and global_variable_type declarations; thus this package must be compiled first.

The order that each input category is covered in this chapter reflects order that the protocol specification should be constructed and compiled.
1. Protocol Variable Definitions

The user defines the protocol environment variables in the *definitions* package. Variables can either be local to a specific machine or global to the system. The global variables are considered *shared* and allow communication between the machines in the system. The local variables are only visible to the machine that they are defined for. A discrete variable can be one of the many Ada defined types such as:

```
integer    natural    character
array      digit      boolean
record     access    
```

These types variables, or their subtypes, can be used to define protocol environment.

A template for the *definitions* package is illustrated in Figure 19. The shaded areas of the figure are where the variables of the protocol are inserted. All other code should remain unchanged. Additional type declarations should be placed before the machine type
declarations. The state_number of each machine is initialized to one even though this maybe different based on the FSM text file is input (the initial state is explicitly given.)

package definitions is
  type scm_transition_type is ( );

  type machine1_state_type is record
    state_number : natural := 1;
    end record;

  type machine2_state_type is record
    state_number : natural := 1;
    end record;

  type global_variable_type is record
    k_ltiXglobal (shared) variables
    end record;

end definitions;

Figure 19: SCM, definitions package template.

The variable declarations for the stop_and_wait protocol are: shown in Figure 20.

package definitions is
  type scm_transition_type is (snd_data, rcv_data, snd_ack, rcv_ack);

  type buffer_type is (d, e, a);

  type machine1_state_type is record
    state_number : natural := 1;
    out_buff : buffer_type := d;
    end record;

  type machine2_state_type is record
    state_number : natural := 1;
    in_buff : buffer_type := e;
    end record;

41
type global_variable_type is
  record
    CHAN : buffer_type := E;
    RET  : buffer_type := E;
  end record;
end definitions;

end definitions;

Figure 20: SCM, definitions package, \textit{stop\_and\_wait}.

The transitions are represented as \texttt{xmt\_data}, \texttt{xmt\_ack}, \texttt{rcv\_data}, and \texttt{rcv\_ack} instead of $-D$, $-A$, $+D$, and $+A$. Machine one has a local variable that serves as an out-bound buffer (\texttt{out\_buff}). It is initialized with data present in the buffer, represented by 'd'. The only machine that sees the variable contents is Machine one. Machine two is similar in that it has a buffer for receiving (\texttt{in\_buff}) data from the channel. The global variables are the shared variables channel (\texttt{CHAN}) and a return link (\texttt{RET}). Both variables are initialized empty and can be accessed by each machine. The values that \texttt{CHAN}, \texttt{RET}, \texttt{in\_buff}, and \texttt{out\_buff} can have are defined as a \texttt{buffer\_type}. The \texttt{buffer\_type} variables can have the values \texttt{e} (empty), \texttt{d} (data), or \texttt{a} (acknowledgement). The \textit{stop\_and\_wait} protocol example shows how easily variables can be represented. All the text in bold lettering are user defined variables and types.

2. Predicate-Action Table Representation

The predicate-action table serves as the engine to the analysis. The enabling predicate defines the logic that must hold true for the transition to be taken (refer to Table 1). Local and global variables must meet these conditions. A number of transitions could be enabled, but, for a transition to be executed the state of the machine must be considered. The action column of the predicate-action table identifies the variable changes that must take place when the transition is executed. The program captures the essence of the predicate-action table by breaking the components of the table into subprograms. A subprogram in the Ada environment is a function, procedure, or package. Since the user must have access to a number of the subprograms they are represented as separate compilation units.
The first subprogram is the Analyze_Predicate function. A function is a subprogram that returns a value to the location in which it was invoked. It can only have input parameters. The function is handed the machine local variables and the system global variables as input parameters. Since more than one transition could be enabled, a stack is used to place all transitions that are enabled. A transition is pushed onto the stack if it is enabled and the function returns the entire transition_stack. From the transition_stack values and a pointer to the current state in machine adjacency matrix, a determination is made on which transition can actually be executed. There are a number of Analyze_Predicate functions, one for each machine. The template for the Analyze_Predicate function is shown in Figure 21.

```
separate (main)
function Analyze_Predictes_Machine1(local : machine1_state_type;
    GLOBAL: global_variable_type)
return transition_stack_package.stack is
begin
    MakeEmpty (transition_stack);  
    enabling condition
    if ( ) then
        Push (transition_stack, );  
        enabled transition
    end if;
    return transition_stack;
end Analyze_Predictes_Machine1;
```

Figure 21: SCM, Analyze_Predicate function template.

Once a transition is executed, changes must be made to some or all the variables. A procedure using a case statement was the simplest way to make the changes to the global state record. The Action procedure has three parameters: the transition that is executed and the current global state record are in parameters, and the updated global state record is the out parameter. The transition is passed into the procedure and a case statement determines which series of instructions are to be executed. These instructions make the appropriate changes to the protocol environment variables. The out_system_state is handed out of the
procedure containing the changed protocol variables. The template for the Action procedure is shown in Figure 22.

```haskell
separate (main)
procedure Action( in system state : in Gstate_record_type;
in transition : in scm_transition_type;
out system state : out Gstate_record_type) is
begin
    case (in-transition) is
        when (po enabled transition) => action taken
        when others =>
            put_line("Error in the Action procedure");
        end case;
end Action;
```

Figure 22: SCM, Action procedure template

The three subprograms that reflect the logic of the predicate-action table are grouped together in one file (predicate_action.a). The file for the stop_and_wait protocol is:

```haskell
separate (main)
function Analyze_Predicates_Machine1(local : machinel_state_type;
GLOBAL: global_variable_type)
return transition stack_package.stack is
begin
    MakeEmpty(transition_stack);
    if ((local.out_buff /= e) and (GLOBAL.CHAN = E)) then
        Push(transition_stack, xmt_data);
    end if;
    if (GLOBAL.RET = A) then
        Push(transition_stack, rcv_ack);
    end if;
    return transition_stack;
end Analyze_Predicates_Machine1;

separate (main)
function Analyze_Predicates_Machine2(local : machine2_state_type;
GLOBAL: global_variable_type)
return transition stack_package.stack is
begin
    MakeEmpty(transition_stack);
    if ((GLOBAL.CHAN /= E) and (local.in_buff = e)) then
        Push(transition_stack, xmt_data);
    end if;
    Push(transition_stack, xmt_ack);
    return transition_stack;
end Analyze_Predicates_Machine2;
```

44
procedure Action(in_system_state : in Gstate_record_type;
in_transition : in scm_transition_type;
out_system_state : out Gstate_record_type) is
begin
    case (in_transition) is
        when (zmt_data) =>
            out_system_state.GLOBAL_VARIABLES.CHAN :=
            in_system_state.machine1_state.out_buff;
        when (rcv_ack) =>
            out_system_state.GLOBAL_VARIABLES.RET :=
            out_system_state.GLOBAL_VARIABLES.CHAN :=
            out_system_state.machine2_state.in_buff :=
        when (xmt_ack) =>
            out_system_state.GLOBAL_VARIABLES.RET :=
            out_system_state.machine2_state.in_buff :=
        when others =>
            put_line("Error in the Action procedure");
    end case;
end Action;

Figure 23: SCM, analyze_predicate.a, stop_and_wait.

The bold text in the code indicates what the user provided as input to define the
specification shown in the stop_and_wait predicate-action table (See Figure 16.)

3. Finite State Machines

The FSM’s are input as a text file during program execution. This file is built by
the user with a set of language rules similar to the Backus-Naur Form (BNF) shown in
Chapter III. The only change to the format of the input is the transition (trans) lines. In the
CFSM model only send and receive transitions were allowed; whereas in the SCM model
a transition can have any label that follows the enumeration rules. The lines of the text file
are buffered and parsed. From the parsed line groups of strings called tokens are
manipulated as described in Chapter III. The list of valid instructions for finite state
machine input is:

start
machine <natural>
state <natural>
initial_state <natural> <natural>
trans <enumeration_literal> <natural>
finish
The tokens are cast into either enumerated types (instructions) or integers (integer variables). The integer variables have been formally defined within the main procedure in Appendix B. The meaning of the instructions are found in Figure 24.

<table>
<thead>
<tr>
<th>Token</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>Serves as a beginning flag for the file.</td>
</tr>
<tr>
<td>machine</td>
<td>Defines the current machine.</td>
</tr>
<tr>
<td>state</td>
<td>Defines the current state.</td>
</tr>
<tr>
<td>initial_state</td>
<td>The initial/start state for machines one and two.</td>
</tr>
<tr>
<td>trans</td>
<td>Transition type and next state.</td>
</tr>
<tr>
<td>finish</td>
<td>This token serves as an ending flag for the file.</td>
</tr>
</tbody>
</table>

Figure 24: Input File definitions.

Representation of a finite state machine using the above convention has some inherent constraints. Rules for constructing enumeration literals must be followed. For instance, the list of values in an enumeration literal can only be character literals and cannot contain a digit in the first position. The input file for stop_and_wait is shown in Figure 25:

```
start
machine 1
state 0
trans xmt_data 1
state 1
trans rcv_ack 0
machine 2
state 0
trans rcv_data 1
state 1
trans xmt_ack 0
initial_state 0 0
finish
```

Figure 25: SCM, Input File, stop_and_wait.

A data structure that contributes to the reachability analysis is the FSM adjacency list. The adjacency list was chosen as the structure to represent the directed graph of the finite state machines.

The actual data structure to build the adjacency list and the defined constraints are:

```plaintext
type machine_array_record_type;
type Mlink_type is access machine_array_record_type;
type executed_type is (yes,no);
type machine_array_record_type is record
    transition : scm_transition_type;
```
next_state : natural;
excecuted : executed_type;
Mlink : Mlink_type
end record;

type machine_array_type is array(positive range<>)
of Mlink_type;
type system_array_type is array(1..2)
of machine_array_type;

The internal representation of the FSM adjacency lists are the same as Figure 12 except the SCM adjacency lists do not have a message field.

C. REACHABILITY ANALYSIS

The process of generating the set of all states reachable from the initial state is called state reachability analysis. During the reachability analysis a check for deadlock, unspecified reception, and unexecuted transitions are done. The reachability analysis of a specific protocol is done in two phases.

The first is to generate a global state reachability graph. This analysis constructs a graph, whose nodes are the reachable global states, and whose arcs indicate the transitions leading from each global state to another. The global state (node) contains the state of each machine and the values of all the variables.

The second phase of the analysis is to generate an separate system state reachability graph from the global state reachability graph. The system reachability graph contains nodes with just the state information of each machine. The rules for the generation of new states will be discussed in Section 2.

1. Global State Analysis

The process of generating the set of all global states reachable from the initial global state is called global state analysis. This analysis produces a graph, whose nodes are the reachable global states, and whose arcs indicate the transitions leading from each global state to another. The global state of a system consists of the system state tuple, plus the values of all variables, both local and shared. The algorithm as it appears in [LUND 91a] is:
(1) Set each machine to its initial state, and all variables to their initial values. The initial set of reachable global states consists of the initial system state and the value of all variables; the initial graph is a single node representing this state.

(2) From the current global state vector and variable values, determine which transitions are enabled. For each of these transitions, determine the global 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, labeling the arc with the transition name. Otherwise, add the new global 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 global state thus generated, and no more new states are generated.

The algorithm above was modified to make use of the existing data structures introduced in the CFSM program. A pseudocode algorithm to construct the global reachability graph is:

```plaintext
create top_Gstate pointer and initial node in Gstate_graph
main loop
  for machine_index in 1..number_of_machines loop
    transition_stack := Analyze_predicate(machine_index, current_Gstate)
    while transition_stack is not empty loop
      while current row of machine(machine_index) is not null loop
        if current Gstate.transition = top of transition_stack then
          perform Action procedure on current_Gstate and place
          results in a temp_Gstate
          search Gstate_graph for temp_Gstate
          if temp_Gstate found then
            insert temp_Gstate in Gstate_graph
            Enqueue pointer to location in Gstate_pointer_queue
          else
            link current_Gstate to found_Gstate
            set new_node flag to false
        else
          traverse current row of machine(machine_index)
        end loop -- machine row traversal loop
      end loop -- transition loop
      Pop a transition from the transition_stack
    end loop --transitions loop
  end loop -- machine loop
exception
  when Gstate_pointer_queue is empty then
    exit main loop
```

48
when Gstate_pointer_queue is full then
    print error message
    exit main loop
end main_loop

The data structures that are used in the program are defined in the specification of the main procedure and the definitions package. The machine states and variables along with the global variables are defined in the definitions package. The remaining data structures which are hidden from the user are shown in Figure 26

--data structures for the global state tuple(node)
type global_state_type;
type Glink_Type is access global_state_type;

--transition structure
type Gstate_transition_type is
    record
    Gtransition : scm_transition_type;
    new_node : boolean := true;
    Glink : Glink_Type;
    executed : boolean := false;
    end record;

--global tuple structure
type Gstaterecord_type is
    record
    machine1_state : machine1_state_type;
    machine2_state : machine2_state_type;
    global_variables : global_variable_type;
    end record;

--Global state node, contains transition, state, and link information
-- needed for building the global state graph
type global_state_type is
    record
    node_number : natural := 0;
    Gtuple Gstaterecord_type;
    link1 : Gstate_transition_type;
    link2 : Gstate_transition_type;
    link3 : Gstate_transition_type;
    link4 : Gstate_transition_type;
    end record;

Figure 26: SCM, global definitions.

The data structure of the global node (global_state_type) encapsulates the information contained in the global state record. The global transition record has the type of transition and information about the node it is pointing to. If the node it is pointing to is a newly created node the new field is set to false, otherwise it maintains its initialized value of true. It also has a visited field, used during the construction of the system state.
reachability graph. The internal representation of the graph generated by the algorithm for
stop_and_wait protocol highlights all the data structure used (see Figure 27).
Figure 27: SCM internal global reachability graph, stop_and_wait.
2. System State Analysis

System state analysis is similar to global state analysis. The number of states generated by the system state analysis is less than or equal to the number of states in the CFSM model or the global analysis of the SCM model. Only the states of the machines and the transitions from those states are considered when generating a new state. The formal steps in constructing a system state graph as it appears in [LUND 91a] is:

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, labeling 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.

The portion of the program that builds the system state graph makes use of the information already available in the global state graph. In the current version of the program the global reachability graph is constructed followed by the system reachability graph. Future versions would allow the user to select which analysis to perform but currently both are constructed and output. The pointer to the initial global state is provided as an input parameter to the `build_State_graph` procedure. The global state graph is traversed in a breadth first manner, as the nodes are visited the system state graph is constructed. If the system state graph were being constructed independently of the global state graph the algorithm would be very similar to the one in the previous section. Since the system state graph is being constructed based on only the global reachability graph the machine matrices are not used. The pseudo-code algorithm for this approach is:
create top_Sstate pointer and build initial_Sstate node;
main loop
while( current_Gstate.node not null and not visited) loop
visit a Gstate_graph node using BFS traversal
mark the link taken as visited
create a temp_Sstate with values of current_Gstate
search Sstate_graph for temp_Sstate
if temp_Sstate found then
insert temp_Sstate in Sstate_graph
else
link current_Sstate to found_Sstate
set new_node flag to false
end loop
end main loop

The data structures for the system state graph construction, except for the transition labels, are completely hidden from the user. The system related data structures as they appear in the main procedure are shown in Figure 28.

type system_state_type;
type Syslink_type is access system_state_type;

--transition structure for system state
type Sstate_transition_type is
record
  Stransition : scm_transition_type;
  new_node : boolean := true;
  Syslink   : Syslink_type;
end record;

type Sstate_record_type is
record
  machine1_state : natural := 0;
  machine2_state : natural := 0;
end record;

--system state structure
type system_state_type is
record
  node_number : natural := 0;
  Stuple      : Sstate_record_type;
  link1       : Sstate_transition_type;
  link2       : Sstate_transition_type;
  link3       : Sstate_transition_type;
  link4       : Sstate_transition_type;
end record;

Figure 28: SCM, system definitions.

To follow through with the stop_and_wait analysis example, an internal representation of the system state graph is shown in Figure 29. Although this example does give a graphical picture of how the data structures are used it does not show the advantages
of a system state analysis over global state analysis. Examples covered in Chapter V illustrate how much smaller system state graphs can be when compared to the global analysis graphs.
Figure 29: SCM internal system reachability graph, stop_and_wait.
D. OUTPUT

Output of the analysis is provided to a text file and to a default device (workstation display). Figure 32 shows a captured image of default output to a workstation screen for stop_and_wait. There are features available to allow the user to step through the output one screen at a time. Output messages are provided to the user when a deadlock, unexecuted transition, or an unspecified reception occur. A message is also displayed when the length of the graph exceeds the bounds defined by the user (capacity of the channel is exceeded.) The contents of the machine adjacency lists are also output.

The user may format output for the global state graph. This is done through the file user_output.a. The procedure, output_Gtuple, contained in the file allows the user to format the variables for default output. The template for output procedures are found in Figure 30.

```plaintext
separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
    put( " [ & integer'image(tuple.machine1_state.state_number) & " , " ];" );
    put( " , & integer'image(tuple.machine2_state.state_number) & " ] " );
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tuple : in out Gstate_record_type) is
begin
    put(reach, " [ & integer'image(tuple.machine1_state.state_number)];" );
    put(reach, " , & integer'image(tuple.machine2_state.state_number) & " ] " );
end output_Gtuple_to_file;
```

Figure 30: SCM, output_Gtuple procedure template.

An example of how a user could format output is given for the stop_and_wait protocol is given in Figure 31.
Figure 31: SCM, output format, *stop_and_wait*.

Consistent with previous examples the boldface code is that which the user provides. The user does not provide any parameters for system state output. The output shown in Figure 32 was formatted according to the procedures used above.
REACHABILITY ANALYSIS of: stop_and_wait

Global State GRAPH
0 [0, d, E, E, e, 0] xmt_data 1
1 [1, d, d, E, E, e, 0] rcv_data 2
2 [1, d, d, D, E, d, 1] xmt_ack 3
3 [1, d, D, A, d, 0] rcv_ack 0

System State GRAPH
0 [0, 0] xmt_data [1, 0] 1
1 [1, 0] rcv_data [1, 1] 2
2 [1, 1] xmt_ack [1, 0] 3
3 [1, 0] rcv_ack [0, 0] 0

-----------------------------------------------
<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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 Array Contents</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>

Figure 32: SCM, analysis output, stop_and_wait.
V. AN AUTOMATED ANALYSIS OF SELECTED DATA LINK PROTOCOLS

In this section the programs that were developed in the last two chapters will be demonstrated. Some well known data link protocols will be analyzed using the CFSM and SCM programs. The CFSM program will be used to analyze the alternating bit and the sliding window protocols. The SCM program will be used to analyze go_back_n and selective repeat protocols. In each analysis the specification will be described; the program input and results are in the Appendices of this publication.

A. CFSM MODEL

The examples used as input show the advantages of the CFSM program. The alternating bit protocol analysis was chosen as a simple class of protocols. The sliding window with a window size of three shows how a graphically complex protocol can be analyzed quite easily.

1. Alternating Bit Protocol

The specification of the alternating bit protocol will be used as the first example for the CFSM program. The protocol consists of two machines. Machine one serves as a sender and Machine two as the receiver. The sender sends a message (-X) to the receiver. The receiver then accepts the message (+X) and sends an acknowledgment (-A). The acknowledgment at the machine level is done with the toggling of a bit, wherein the name
alternating bit is derived. The sender is clear to send another message when the
acknowledgment is received.

\[ \text{Machine 1} \]

\[ \text{Machine 2} \]

Figure 33: CFSM specification, Alternating Bit.

The input file for the specification is

```
start
machine 1
state 1
trans -X 2
state 2
trans +A 3
state 3
trans -Y 4
state 4
trans +B 1
machine 2
state 1
trans +X 2
state 2
trans -A 3
state 3
trans +Y 4
state 4
trans -B 1
initial_state 1
finish
```

The analysis of the alternating bit specification is:
REACHABILITY ANALYSIS of: output.alt_bit

1 [1, E, E, 1] -X [2, X, E, 1] 2
2 [2, X, E, 1] +X [2, E, E, 2] 3
7 [4, E, E, 4] -B [4, E, B, 1] 8
8 [4, E, B, 1] +B [1, E, E, 1] 1

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

* The nodes generated by the analysis were done in a breadth first manner


The analysis of a sliding window protocol is a more complex example. To represent the protocol as a set of graphical finite state machines can be quite tedious. The essence of the protocol must be captured with the use of transitions, oftentimes this can lead to an intricate diagram as in this example.

The sliding window protocol can also be represented as a two machine CFSM. As in the previous example, Machine one is the sender and Machine two is the receiver. At any instant of time the sender maintains a list of consecutive sequence numbers corresponding to frames it is permitted to send."TANE 81] These frames are said to fall
within the sending window. The receiver also maintains a receiving window corresponding to frames it is permitted to accept. The sending window and the receiving window need not have the same lower and upper limits, or even have the same size.

A window size of three is used in the specification given in Figure 34. The messages or packets are shown as transitions labeled X, Y, and Z and the acknowledgments are A, B, and C.

Figure 34: CFSM specification, sliding window (w=3).
The text file of the specification is:

```
start
machine 1  machine 2
state 1  state 1
trans -X 2  trans -X 2
state 2  state 2
trans +Y 3  trans +Y 3
state 3  state 3
trans +Z 4  trans +Z 4
state 4  state 4
state 5  state 5
state 6  state 6
trans +A 7  trans +A 7
state 7  state 7
state 8  state 8
trans +C 9  trans +C 9
state 9  state 9
state 10 finish
```

The analysis of the specification as contained in the output text file:

```
REACHABILITY ANALYSIS of output sliding

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>+Y</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>+Z</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>+Z</td>
<td>yes</td>
</tr>
</tbody>
</table>
```
The unexecuted transitions account for error control. Error control refers to mechanisms to detect and/or correct errors that occur in the transmission of information. So the unexecuted transitions identified in Machine 2's array are transitions that would have been executed if a loss of a message had occurred. Error and time-out transitions are not shown in the CFSM.

B. SCM MODEL

The examples used as input validate the use of the SCM program as a tool to verify protocols. The go back n protocol was analyzed first due to the availability of prior modelling done using SCM. The output of the program were compared to the manual SCM modelling results of this protocol. The selective repeat specification provided additional evidence as to the programs validity as well as demonstrating how it can be used to improve a specification. In both examples it must be shown that the use of the analysis should help the designer or reviewer to gain a greater understanding of the protocol, as well as in detecting errors.

The analysis of any protocol using this program contains varying information. A global reachability graph and system reachability graph are provided. Following the graph is a description of the contents of each machine array upon termination of the graph construction. Error messages (deadlock, unspecified reception, and unexecuted transitions) are placed at the point in the analysis where they occur. The system state graph will be used in this section to provide a means to validate output results. The system state graph can be viewed as a three dimensional object whose tuple values provide a vector to 3 dimensional space.

1. Go Back N

The first protocol which was chosen to model is a go_back_n protocol with a variable window size, which is a subset of the High-level Data Link Control (HDLC) class of protocols. 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 \( w \). 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 block expected (which is stored in local variable \( \text{exp} \)). 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.

**machine 1**

![Transition diagram for machine 1]

**machine 2**

![Transition diagram for machine 2]

<table>
<thead>
<tr>
<th>Transition</th>
<th>Enabling Predicate</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>-D</td>
<td>DATA(i)=E ∧ SEQ(i)=E</td>
<td>DATA(i) := Sdata(i)\nSEQ(i) := seq\ninc(i,seq)</td>
</tr>
<tr>
<td>+A_k (0 ≤ k &lt; w)</td>
<td>ACK ⊗ k = seq ∧ ACK ≠ E (next state : k)</td>
<td>ACK := E</td>
</tr>
<tr>
<td>+D</td>
<td>DATA(j) ≠ E ∧ SEQ(j)=exp</td>
<td>Rdata := DATA(j)\nDATA(j), SEQ(j) := E\ninc(j,exp)</td>
</tr>
<tr>
<td>-A</td>
<td>DATA(j)=E</td>
<td>ACK := exp</td>
</tr>
</tbody>
</table>

Figure 35: SCM specification, Go Back N, window size of 1..w.
The general specification of the protocol is given in Figure 35. Included in this figure are the state machine diagrams, variables and the predicate action table. 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.

There are four basic types of transitions described in the predicate action table. In the sender 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 receive transition in the receiver, m2, is enabled whenever a data block of the appropriate sequence number is in the jth element of DATA and SEQ. An acknowledgment may be sent by m2 in any state except 0, in which case no unacknowledged data blocks have been received. The +A is a receive transition. If m1 is in state u, 1 ≤ u ≤ w, and there is a nonempty value in shared variable ACK, then exactly one of the transitions +A0, +A1,..., +Aw-1 will be enabled; it will be that Ak such that the predicate \(ACK \oplus k = seq\) is true, and the next state is k. In the state diagram, all of the transition +Ak are shown using the same vertical line.

a. Input of variable definitions.

A sample interaction using the program for the analysis of go_back_n, w=1, consists of input files and an output file. The variable definitions contained in the user specification file are:

```plaintext
class definitions is
  type scm_transition_type is (snd_data, rcv_data,
    snd_ack, rcv_ack0, unused);
```

67
type buffer_type is (d,e,a);
type buffer_array_type is array(1..1) of buffer_type;
type seq_array_type is array(1..1) of integer range -1..1;

type machine1_state_type is
record
  state_number : natural := 1;
  Sdata : buffer_array_type := (others=>d);
  seq : integer range 0..1 := 0;
  i : integer range 1..1 := 1;
end record;

type machine2_state_type is
record
  state_number : natural := 1;
  Rdata : buffer_type := e;
  exp : integer range 0..1 := 0;
  j : integer range 1..1 := 1;
end record;

type global_variable_type is
record
  DATA : buffer_array_type := (others=>e);
  SEQ : seq_array_type := (others=>-1);
  ACK : integer range -1..1 := -1;
end record;
end definitions;

b. Input of predicate analysis.

The analyze predicate functions e contained in the predicate_action.a file are:

function Analyze_Predicates_Machine1(local : machine1_state_type;
  GLOBAL: global_variable_type) return transition_stack_package.stack is
begin
  templ : integer := GLOBAL.ACK + 0;
  MakeEmpty(transition_stack);
  if ((GLOBAL.DATA(local.i) = E)
      and (GLOBAL.CONTENT(local.i) = -1)) then
    Push(transition_stack,snd_data);
  end if;
  if ((templ = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(transition_stack,rcv_ack0);
  end if;
  return transition_stack;
end Analyze_Predicates_Machine1;

function Analyze_Predicates_Machine2(local : machine2_state_type;
  GLOBAL: global_variable_type) return transition_stack_package.stack is
begin
  MakeEmpty(transition_stack);
  if ((GLOBAL.DATA(local.j) /= E)
      and (GLOBAL.CONTENT(local.j) = local.exp)) then
    Push(transition_stack,rcv_data);
  end if;
  if (GLOBAL.DATA(local.j)=E) then
    Push(transition_stack,snd_ack);
  end if;
end Analyze_Predicates_Machine2;
end if;
return transition_stack;
end Analyze_Predicates_Machine2;

c. Input of action table.

The action procedure is also a separate compilation unit contained in the predicate_action.a file:

procedure Action(in_system_state : in out Gstate_record_type;
In_transition : in out scm_transition_type;
out_system_state : in out Gstate_record_type) is

temp : integer := 0;
begin
  case (In_transition) is
    when (snd_data) =>
      out_system_state.GLOBAL_VARIABLES.
      DATA(in_system_state.machinel_state.i) :=
      in_system_state.machinel_state.
      Sdata(in_system_state.machinel_state.i);
      out_system_state.GLOBAL_VARIABLES.
      SEQ(in_system_state.machinel_state.i) :=
      in_system_state.machinel_state.seq;
      out_system_state.machinel_state.i :=
      (((in_system_state.machinel_state.i) +
      1)mod 1) + 1;
      out_system_state.machinel_state.seq :=
      (((in_system_state.machinel_state.seq) +
      1)mod 2);
    when (rcv_ack0) =>
      out_system_state.GLOBAL_VARIABLES.ACK := -1;
    when (snd_ack) =>
      out_system_state.GLOBAL_VARIABLES.ACK :=
      in_system_state.machine2_state.exp;
      out_system_state.machine2_state.Rdata := e;
    when (rcv_data) =>
      out_system_state.machine2_state.Rdata :=
      in_system_state.GLOBAL_VARIABLES.DATA
      (in_system_state.machine2_state.j);
      out_system_state.GLOBAL_VARIABLES.DATA
      (in_system_state.machine2_state.j) := E;
      out_system_state.GLOBAL_VARIABLES.SEQ
      (in_system_state.machine2_state.j) := -1;
      out_system_state.machine2_state.j :=
      (((in_system_state.machine2_state.j) +
      1)mod 1) + 1;
      out_system_state.machine2_state.exp :=
      (((in_system_state.machine2_state.exp) +
      1)mod 2);
    when others =>
      put_line("There is an error in the
      Action procedure");
  end case;
end Action;

69
d. Input of finite state machines.

And, finally, the input file for the finite state machines is:

```
start
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack 0
machine 2
state 0
trans rcv_data 1
state 1
trans snd_ack 0
initial_state 0 0
finish
```

e. Output of analysis.

The output of the analysis is:

```
REACHABILITY ANALYSIS of: go_back_n_k1

Global State GRAPH

0 [ 0, 0, 0, 1, 0, 1, E, -1, -1]  snd_data 1
1 [ 1, 0, 1, 0, 1, 0, 0, -1]  rcv_data 2
2 [ 1, 1, 1, 0, 0, 0, 0, -1]  snd_ack 3
3 [ 1, 0, 1, 1, 0, 0, 0, -1]  rcv_ack 0 4
4 [ 1, 0, 0, 0, 1, 0, 0, -1]  snd_data 5
5 [ 1, 0, 0, 0, 0, 1, 0, -1]  rcv_data 6
6 [ 1, 1, 0, 0, 0, 0, 0, -1]  snd_ack 7
7 [ 1, 0, 0, 0, 1, 1, 0, -1, 0]  rcv_ack 0

System State GRAPH

0 [ 0, 0 ]  snd_data [ 1, 0 ]  1
1 [ 1, 0 ]  rcv_data [ 1, 1 ]  2
2 [ 1, 1 ]  snd_ack [ 1, 0 ]  3
3 [ 1, 0 ]  rcv_ack 0 [ 0, 0 ]  0

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>snd_data</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>rcv_ack</td>
<td>yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>rcv_data</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>snd_ack</td>
<td>yes</td>
</tr>
</tbody>
</table>
```

70
The output indicates that no errors were encountered. The format for the variables are user dependent and, for brevity sake, the code for formatting the output was not included.

**f. System state analysis.**

System state analysis is similar to the reachability analysis used with the pure finite state machine model, but the total number of states which must be generated with system state analysis is significantly smaller.

The system state analysis for a window size of 1 is shown in Figure 36. The subscripts are used so that distinct system states having the same tuple may easily be distinguished. The convention is that the subscript is initially 0, and is increased whenever a "-A" transition is taken, by the number of messages which are being acknowledged.

![Figure 36: SCM, system state analysis, Go Back N, w=1.](image)

The analysis for w=2 is shown in Figure 37. The initial states and variable values are the same as for the w=1, however there are clearly more states in the analysis.

In a comparison between window sizes of 1 and 2, it is noted that the smaller graph is a subgraph of the larger; either can be obtained from the other. If the subscripts are taken as the third coordinate
in a 3-dimensional cartesian coordinate system, with the states of each machine as the first two coordinates. The graph then is the shape of a tetrahedron, with edges which are directed and labeled.

![Graph](image)

**Figure 37: SCM, system state analysis, Go Back N, w=2.**

The graphs contained in Figure 36 and Figure 37 are defined with respect to a window size $w$. The graphs, $DTI(w)$ for a nonnegative integer $w$ is a labeled, directed graph, defined by the tuple $(N, E, L, \Phi)$, where $N = \{ (x, y, z) | 0 \leq z \leq w, z \leq x \leq w, 0 \leq y \leq x-z \}$ is a finite set of nodes, where each node is specified by an ordered triple; $L = \{-D, +D, -A, +A_0, +A_1, ..., +A_w\}$ is a finite set of label; the set $E$ of edges is a set of ordered pairs $((x_1, y_1, z_1), (x_2, y_2, z_2))$ of nodes from $N$, and is the union of the following four sets:

- $E_1 = \{ ((x, y, z), (x+1, y, z)) | (x, y, z) \in N, x < w \}$
- $E_2 = \{ ((x, y, z), (x, y+1, z)) | (x, y, z) \in N, y < x-z \}$
- $E_3 = \{ ((x, y, z), (x, 0, y+z)) | (x, y, z) \in N, y = x-z, z > 0 \}$
- $E_4 = \{ ((x, y, z), (x-z, y, 0)) | (x, y, z) \in N, z > 0 \}$

and the mapping $\Phi(L \leftarrow E)$ is defined as follows:
\[ \forall (x, y, z) \in E_1, \Phi(x, y, z) = -D \]
\[ \forall (x, y, z) \in E_2, \Phi(x, y, z) = +D \]
\[ \forall (x, y, z) \in E_3, \Phi(x, y, z) = -A \]
\[ \forall (x, y, z) \in E_4, \Phi(x, y, z) = +A_k \text{, where } k = x - z \]

Each node of the graph can be thought of as a point in 3-dimensional space, with nonnegative, integral coordinates \((x, y, z)\). The structure of the graph is a sequence of \(w + 1\) triangles, one on top of the other, with the largest triangle at the bottom and the smallest is a single point at the top level.

One of the nice features of the geometric structure of this graph is that the state of the system can be easily inferred from the \(x, y, z\) coordinates. For example, in Figure 37, point \((2, 1, 0)\), or system state \([2, 1, 0]\), the sender has transmitted 2 data blocks for which no acknowledgment has yet been received, the receiver has received 3 of these, but acknowledgment non.

Let \(f(w)\) be the amount of nodes in a system state graph and \(g(w)\) be the amount of nodes in the global reachability graph. The equations for \(g(w)\) and \(f(w)\), and the lemmas that support them, for the go_back_n protocol are found in [lund 91a]. For instance, the graph DT1(w) has

\[ f(w) = \frac{1}{6} w^3 + \frac{11}{6} w + 1. \]

The size of the graphs according to window size are:

<table>
<thead>
<tr>
<th>(w)</th>
<th>(f(w))</th>
<th>(g(w))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>240</td>
</tr>
<tr>
<td>4</td>
<td>35</td>
<td>700</td>
</tr>
<tr>
<td>5</td>
<td>56</td>
<td>1680</td>
</tr>
<tr>
<td>6</td>
<td>84</td>
<td>3528</td>
</tr>
<tr>
<td>7</td>
<td>120</td>
<td>6720</td>
</tr>
</tbody>
</table>

The output of the program for this protocol was compared to values for \(f(w)\) and \(g(w)\). Test runs were done for window sizes of 0 thru 5 and the amount of nodes in each graph were consistent with the table above. The specification input files and the output is in Appendices E through H.
2. Selective Repeat

The next analysis is the selective repeat protocol. The specification defined in [BENV 91] and [STAL 91] has been modified as follows. There are two machines in the system, a sender ($m_1$) and a receiver ($m_2$). The sender's initial state is 0. Two assumptions were made for the analysis. First, all the packets transmitted were received without error and second, no packets were lost or reordered during the transmission.

The specification for the sender is found in Figure 38. As the buffer manager places data in the next available sequence number, the sender places the packet on the channel and increments the index for the next packet to be transmitted. As long as the next packet is not empty, the sender will continue this process until the bottom state on the finite state machine is reached, indicating the transmission of a full window. Acknowledgments (ACK) are passed to the transmitter as they are received. If an ACK is received then the transmitter must determine if the window may be opened and if so, how far. If the ACK is not for the first packet in the window then the flag ack_rec is set, indicating that the packet was received correctly. The window is not advanced because packets that were transmitted earlier are still outstanding. The sequence number within each ACK represents the actual sequence number of the packet received and not the sequence number of the next expected packet, as is common in many protocols.

When an ACKs for the first packet in the window is received the machine clears its buffer, advances the window, and looks at the next sequence number. If the packet has not been received, then that becomes the beginning of the window. If it has been received then the next sequence number is examined until the earliest outstanding packet is found or
the window is fully opened. ACKs that do not correspond to any of the sequence numbers within the current window are ignored.

\[ \text{machine 1} \]

\[ \text{machine 2} \]

\[
\begin{align*}
\text{transition} & \quad \text{enabling predicate} & \quad \text{action} \\
-D & \quad \text{out_buffer}(i) \neq E \land \text{hold} = f & \quad \text{DATA}(i) := \text{out_buffer}(i) \\
& \quad \text{inc}(i) & \quad \text{if}(i = w) \text{hold} := t \\
+A & \quad \text{ack_rec}(i) = f \land \text{CONTROL} = \text{A}(i) & \quad \text{ack_rec}(i) := t \\
& \quad \text{CONTROL}(i) := E \\
R_k & \quad \text{true} & \quad \text{ack_rec}(i) := f \\
(0 \leq k < w) & \quad \text{hold} := f & \\
-A & \quad \text{CONTROL}(j) = E \land \text{pkt_rec}(j) = f & \quad \text{CONTROL}(j) := \text{A}(j) \\
& \quad \text{pkt_rec}(j) := f & \quad \text{pkt_rec}(j) := \text{f} \\
& \quad \text{in_buff}(j) := e, \text{inc}(j) & \\
+D & \quad \text{pkt_rec}(j) = f & \quad \text{in_buff}(j) := \text{DATA}(j) \\
& \quad \text{DATA}(j) := E & \quad \text{pkt_rec}(j) := t \\
& \quad \text{pkt_rec}(j) := \text{f} \\
\end{align*}
\]

Figure 38: SCM specification, Selective Repeat, window size of 1..w.
The receiver as shown in Figure 38 follows the specification given in the predicate action table. The initial state of the receiving machine is 0. Any packets that are received with sequence numbers outside of the window are dropped. If a valid data packet is received then the \( +D \) is taken, based upon whether the sequence number of the received packet is equal to \( i_s \). If the sequence number is not \( i_s \) then the flag \( pkt_{rec} \) is set to \( t \) and the packet is stored. If it is equal to \( i_s \), then the \( pkt_{rec} \) is set, the packet is released to buffer, and \( i_s \) is incremented until a sequence number with \( pkt_{rec}=f \) is found.

A sample interaction using the program for the analysis of selective repeat, \( w=1 \), consists of input files and an output file. The variable definitions contained in the user specification file are:

```plaintext
definitions
  package definitions is
    type scm_transition_type is (snd_data, rcv_data,
      snd_ack, rcv_ack,
      adv_win, unused);
    type buffer_type is (dl, e, al);
    type boolean_type is (t, f);
    subtype ack_buffer_type is buffer_type range e..al;
    subtype data_buffer_type is buffer_type range dl..e;
    type ack_array_type is array(1..1) of ack_buffer_type;
    type data_array_type is array(1..1) of data_buffer_type;
    type boolean_array_type is array(1..1) of boolean_type;
    type machine1_state_type is record
      state_number : natural := 0;
      out_buffer   : data_array_type := (dl);
      ack_rec      : boolean_array_type := (others=>f);
      current      : integer range 1..1 := 1;
      hold         : boolean_type := f;
    end record;
    type machine2_state_type is record
      state_number : natural := 0;
      in_buffer    : data_array_type := (others=>e);
      pkt_rec      : boolean_array_type := (others=>f);
    end record;
    type global_variable_type is record
      DATA         : data_array_type := (others=>e);
      CONTROL     : ack_array_type := (others=>e);
    end record;
end definitions;
```
a. Input of predicate analysis.

The analyze predicate functions contained in the predicate_action.a file are:

```haskell
function Analyze_Predicates_Machine1(local : machine1_state_type;
        GLOBAL: global_variable_type) return transition_stack_package.stack is
begin
    MakeEmpty(transition_stack);
    if (local.out_buffer(1) /= E) then
        Push(transition_stack,snd_datal);
    end if;
    if ((local.ack_rec(1)-f) and GLOBAL.DATA=A1) then
        Push(transition_stack,rcv_ack1);
    end if;
    Push(transition_stack,adv_win1);
    return transition_stack;
end Analyze_Predicates_Machine1;

function Analyze_Predicates_Machine2(local : machine2_state_type;
        GLOBAL: global_variable_type) return
        transition_stack~package.stack is
begin
    MakeEmpty(transition_stack);
    if (GLOBAL.DATA = D1) and (local.pkt_rec(1)=f)) then
        Push(transition_stack,rcv_datal);
    end if;
    if (local.pkt_rec(1)=t) then
        Push(transition_stack,snd_ack1);
    end if;
    return transition_stack;
end Analyze_Predicates_Machine2;
```

b. Input of action table.

The action procedure is also a separate compilation unit contained in the predicate_action.a file:

```haskell
procedure Action(in_system_state : in out Gstate_record_type;
        in_transition : in out scm_transition_type;
        out_system_state : in out Gstate_record_type) is
begin
    temp : integer := 0;
    begin
        case (in_transition) is
            when (snd_datal) =>
                out_system_state.GLOBAL_VARIABLES.DATA :=
                        in_system_state.machine1_state.out_buffer(1);
            when (rcv_ack1) =>
                out_system_state.machine1_state.ack_rec(1) := t;
                out_system_state.GLOBAL_VARIABLES.DATA := e;
                out_system_state.machine1_state.current := 1;
            when (rcv_datal) =>
                out_system_state.machine2_state.in_buffer(1) :=
                        in_system_state.GLOBAL_VARIABLES.DATA;
                out_system_state.GLOBAL_VARIABLES.DATA := e;
```
out_system_state.machine2_state.pkt_rec(1) := t;
when (snd_ack1) =>
  out_system_state.GLOBAL_VARIABLES.DATA := a1;
out_system_state.machine2_state.pkt_rec(1) := f;
out_system_state.machine2_state.in_buffer(1) := e;
when (adv_win1) =>
  out_system_state.machine1_state.ack_rec
    (in_system_state.machine1_state.current) := f;
when others =>
  put_line("There is an error in the Action procedure");
end case;
end Action;

c. Input of finite state machines.

And, finally, the input file for the finite state machines is:

```
start
machine 1
state 0
trans snd_data1 1
state 1
trans rcv_ack1 2
state 2
trans adv_win1.0
machine 2
state 0
trans rcv_data1 1
state 1
trans snd_ack1 0
initial_state 0 0
finish
```

d. Output of analysis.

The output of the analysis is:

78
The system state analysis for a window size of 1 is shown in Figure 39. The subscripts are used so that distinct system states having the same tuple may easily be distinguished. The convention is that the subscript is initially 0, and is increased whenever a -A transition is taken, by the number of messages which are being acknowledged. The analysis for w=2 is shown in Figure 40. As previously pointed out initial states and variable values are the same as for the w=1, however there are clearly more states in the analysis.
Figure 39: SCM, system state analysis, Selective Repeat, w=1.

Figure 40: SCM, system state analysis, Selective Repeat, w=2.

The graphs contained in Figure 39 and Figure 40 are the basis to the definition of a window size of w. The graphs, $SRI(w)$ for a nonnegative integer w is a labeled, directed graph, defined by the tuple $(N,E,L,\Phi)$, where

$N = \{(x,y,z)| 0 \leq z \leq w, z \leq x \leq 2w, 0 \leq y \leq x-z\}$ is a finite set of nodes, where each node is specified by an ordered triple; $L = \{-D, +D, -A, +A, R_0, \ldots, R_w\}$ is a finite set of label; the set $E$ of edges is a set of ordered pairs $((x_1, y_1, z_1), (x_2, y_2, z_2))$ of nodes from N, and is the union of the following four sets:
\[ E_1 = \{(x, y, z), (x + w, y, z) \mid (x, y, z) \in \mathbb{N}, x < w\} \]
\[ E_2 = \{(x, y, z), (x, 1, 0) \mid (x, y, z) \in \mathbb{N}, y + z < x\} \]
\[ E_3 = \{(x, y, z), (x, y - 1, z + 1) \mid (x, y, z) \in \mathbb{N}, x > z\} \]
\[ E_4 = \{(x, y, z), (x + 1, y, z) \mid (x, y, z) \in \mathbb{N}, y + z < x\} \]
\[ E_5 = \{(x, y, z), (x - (w + k), y, z) \mid (x, y, z) \in \mathbb{N}, x - (y + z) \leq k\} \]

and the mapping \( \Phi (L \leftarrow E) \) is defined as follows:

\[ \forall (x, y, z) \in E_1, \Phi (x, y, z) = -D \]
\[ \forall (x, y, z) \in E_2, \Phi (x, y, z) = +D \]
\[ \forall (x, y, z) \in E_3, \Phi (x, y, z) = -A \]
\[ \forall (x, y, z) \in E_4, \Phi (x, y, z) = +A \]
\[ \forall (x, y, z) \in E_5, \Phi (x, y, z) = R_k, k = |w - x| \]

As with the go_back_n analysis, each node of the graph can be thought of as a point in 3-dimensional space, with nonnegative, integral coordinates \((x, y, z)\). The structure of the graph is a sequence of \(w+1\) triangles, one on top of the other, with the largest triangle at the bottom and the smallest is a single point at the top level.

Let \( f(w) \) be the amount of nodes in a system state graph. The lemmas that support it, for the selective repeat protocol are found in [JENS 92]. The graph \( \text{SR}(w) \) has \( f(w) = w^2 + (w + 1)^2 \). The size of the graphs according to window size are:

<table>
<thead>
<tr>
<th>w</th>
<th>( f(w) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>25</td>
</tr>
<tr>
<td>4</td>
<td>41</td>
</tr>
</tbody>
</table>

Test runs were done for and the amount of nodes in each graph were consistent with the table above. The specification input files and the output is in Appendices I and J.
VI. CONCLUSIONS AND RECOMMENDATIONS

In this thesis, a program was introduced that analyzed network protocols using the CFSM and SCM models. The program was successfully developed in an Ada environment. The Ada tools that made implementing the models easier were encapsulation, information hiding, generic programming units.

In Ada, network protocol specifications can be represented in an intuitive manner. The finite state machines and the associated predicate action tables were converted to Ada language parameters for the analysis. The language environment enforces the rules of the protocol as well as the allowable behavior of all the variables. Dynamic construction of reachability graphs allowed the user to determine how large or small an analysis should be. The protocol designer or engineer that uses this program can quickly become familiar with the behavior of the protocol by simply constructing the Ada specification. It is interesting to note that due to the automated specification analysis some previous work using the SCM model has since been modified. The analysis provides information on occurrences of deadlock, unspecified reception, unexecuted transition, and message flow exceeding channel capacity.

The programming environment provided an adequate platform to develop the program. The ability to use encapsulation, information hiding, tasking, and generic code allowed the program to be developed in a step-wise, compartmented fashion. The availability of a powerful debugger (DBX) enhanced the transition from developing to testing the program. To allow for a more transportable product this program might be converted to C or C++, so that others may benefit from its use.

The program was validated with previous work done on widely used protocols. In Lundy’s papers [LUND 88], [LUND 91a], [LUND 91b], [LUND 92a] and [LUND 92b] a number of protocols were manually analyzed using the SCM model. A subset of those analysis were performed using the program, achieving identical results. It is interesting to note that due to the automated analysis of previous work using the SCM model, some
specifications have had to be modified. In must also be noted that example analysis were only done on two machine specifications.

There are several questions and areas open for further work which remain. An important step would be to expand the program to allow for more than two machines. Although most protocols can be modelled with two machines, it is a realistic requirement to model three or more. The program could be made more interactive with the user, allowing the user to change the specification real time when an error occurs. The program was developed for use on a workstation, future work could concentrate on PC versions of Ada or C/C++. A picture is worth a thousand words; what is done textually can sometimes be represented better graphically. A graphical user interface would enhance the user's ability to specify a protocol and understand the analysis.

It is important to say that developing this program was FUN. One of the features of this automated tool is its understandability. By developing an automated tool it became apparent that the user needs to feel comfortable with the language used as well as how a specification looks and feels when input for analysis. The results provided by the many runs of this program focused attention to where it needed to be- the protocol behavior, not the programming language anomalies.
APPENDIX A CFSM CODE

The program listing begins on the following page.
with Text 10, queues, stacks;
use Text 10;

procedure main is
max length : natural := 40;  --length of global state array
type cfsm_instruction_type is (start, number_of_machines, machine, number_of_states, state, initial_state, trans, finish);
type cfsm_transition_type is (snd, rcv, unused);
type visit_type is (yes, no);
type search_type is (dfs, bfs);
package trans_enum_io is new Text 10.Enumeration_IO(cfsm_transition_type);
package instr_enum_io is new Text 10.Enumeration_IO(cfsm_instruction_type);
package vst_enum_io is new Text 10.Enumeration_IO(visit_type);
package src_enum_io is new Text 10.Enumeration_IO(search_type);
package cfsm_integer_io is new Integer_IO(integer);
use trans_enum_io, instr_enum_io, vst_enum_io, cfsm_integer_io;
  -- data structures for the machine arrays
package queue_pack is new queues(character, MAX => 3);
use queue_pack;
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 := unused;
  message : character := "";
  next Mstate : natural := 0;
  visited : visit_type := no;
  Slink : Slink_type := null;
end record;
type machine_array_type is array(positive range 1 .. 10) of Slink_type;
type system_array_type : s array(1 .. 2) of machine_array_type;
  -- data structures for the global state tuple(node)
type global_state_type:
  type Glink_type is access global_state_type;

--transition structure

type Gstate_transition_type is
record
  Gtransition : cfsm_transition_type := unused;
  Gmessage : character := ';';
  new_node : boolean := true;
  Glink : Glink_type;
end record;

--global tuple structure

type Gstate_record_type is
record
  machine1_state : natural := 0;
  machine2_state : natural := 0;
  queue 1 : queue_pack.Queue; --array of characters
  queue 21 : queue_pack.Queue;
end record;

--Global state node, contains transition, state, and link information
--needed for building the global state graph

type global_state_type is
record
  system_state_number : natural := 0;
  Gtuple : Gstate_record_type;
  link1 : Gstate_transition_type;
  link2 : Gstate_transition_type;
  link3 : Gstate_transition_type;
  link4 : Gstate_transition_type;
end record;

--declaration of stacks of pointers and queues of pointers
--along with queues of messages to simualate two way communications
--between machines

package Gpointer_stack_pack is new stacks(Glink_type, MAX => 10);
package Gstate_stack_pack is new stacks(global_state_type);
package Gpointer_queue_pack is new queues(Glink_type, MAX => 10);
use Gpointer_stack_pack, Gstate_stack_pack, Gpointer_queue_pack;
--- global variables are initialized

```plaintext
initial_global_state : Glink_type; -- top tuple in global graph
cfsm_instruction : cfsm_instruction_type; -- used in case statements
total_machines : natural := 0; -- number of machines in the system
total_states : natural := 0; -- number of states per machine
current_state : natural := 0; -- current state of the current machine
next_state : natural := 0;
current_machine : natural := 0;
current_transition : cfsm_transition_type;
search : search_type; -- values dfs(depth_first) and bfs(breadth_first)
current_message : character; -- message off the top of the queue
current_queue1 : queue pack.Queue;
current_queue2 : queue pack.Queue;
start_state1 : natural := 0;
start_state2 : natural := 0;
start_state3 : natural := 0;
temp_Gstate : Gstate_record_type; -- a temporary holding place for tuples
machine_array : system_array_type; -- an array of arrays so that the machines can be subscripted to be
identified
GPointer_stack : G_inter_stack.pack.stack;
GPointer_queue : G_inter_queue.pack.queue;
number_of_nodes : natural := 0;
output_line_count : natural := 1; -- stops output from scrolling if longer than 25 lines
file_name : string(1 .. 20) := (others => ' '); -- string that holds type of analysis(dfs,bfs)
out_file_name : string(1 .. 20) := (others => ' '); -- variables that holds length of a string
in_name_len : integer;
out_name_len : integer;
analysis_len : integer;
cfsm : file_type; -- input file pointer
reach : file_type; -- output file pointer
```
Transition : in out cfsm_transition_type;
Message : in out character) is separate;
procedure read_in_file(cfsm : in out file_type) is separate;

----------------------------------reachability.a----------------------------------
function IsEqual(tuple1 : in Gstate_record_type; tuple2 : in Gstate_record_type) return boolean is separate;

----------------------------------search.a-------------------------------------
function search_for_tuple(top : in Glink_type; tuple : in Gstate_record_type) return Glink_type is separate;

----------------------------------reachability.a----------------------------------
procedure build_Gstate_graph(machine_array : in out system_array_type;
start_state_1 : in out natural;
start_state_2 : in out natural;
Gopt_Gstate : in out Glink_type;
number_of_nodes : in out natural) is separate;
procedure output_analysis is separate;
procedure clear_pointers is separate;

begin
main_cfsm_loop :
    loop
        put(ASCII.ESC); --call to operating system to clear the screen
        put_line("[2J");
        new_line(10);
        --query the user for the input file
        loop
            begin
                set_col(10);
                put("Filename of cfsm specification('Q' to quit): ");
                get_line(file_name, in name_len);
                if [file_name[1] = 'Q'] then
                    exit main_cfsm_loop;
                else
                    open(cfsm, mode => in_file, name => file_name(1 .. in_name_len));
                    end if;
                    exit;
                    exception
                        when NAME_ERROR =>
                            put_line("That file does not exist. Reenter!!");
                        end;
                    end loop;
        --query the user for the way that the global analysis graph is to
        --be built. A stack based construction will lead to a breadth first
--construction and a queue based construction will lead to a depth
--first construction.

loop
  begin
    set col(10);
    put("Breadth First (B) or Depth First (D) Analysis:");
    get_line分析, 分析_len);
    if (分析(1) = 'B' or 分析(1) = 'b') then
      search := bfs;
    elsif (分析(1) = 'D' or 分析(1) = 'd') then
      search := dfs;
    else
      raise NAME_ERROR;
    end if;
    exit;
  exception
    when NAME_ERROR =>
      put_line("Not a correct choice. Enter a 'B' or a 'D'!!");
  end;
end loop;
create_output_file;
read_in_file(cfsm);
output_analysis;
close(cfsm);
close(reach);
clear_pointers;
end loop main_cfsm
end main;
procedure load_machine_array(number : in out natural;
   state : in out natural;
   Next_state : in out natural;
   Transition : in out cfsm_transition_type;
   Message : in out character) is

   end_of_list : Slink_type := machine_array(number) (state);
   temp_link : Slink_type := new machine_array_record_type;

begin

   temp_link.transition := Transition;
   temp_link.message := Message;
   temp_link.next_Mstate := Next_state;

   if (end_of_list = null) then
      machine_array(number) (state) := temp_link;
   else
      while (end_of_list.Slink /= null) loop
         end_of_list := end_of_list.Slink;
      end loop;
      end_of_list.Slink := temp_link;
   end if;

end load_machine_array;

procedure read_in_file(cfsms : in out file_type) is

    line_buffer : string(1 .. 80) := (others => ' ');
    token      : string(1 .. 80) := (others => ' ');
    token_index : integer := 1;  -- clears the token used in parsing
    line_index  : integer := 1;  -- variable used for halting output every
    line_length : integer;      -- 25 lines.
    enum_length : integer;
    conv_length : integer;
    line_count  : integer := 1;

begin

    -- reads input file one line at a time. Each line is read into a line buffer and tokens
    -- are extracted from the line. A token may be an understandable machine instruction:
    --
    --    start: placed as a flag at the beginning of the file
    --    number_of States: a variable used to tell the dimensions of
    --    machine array.
    --    number_of_Machines: same as above, this variable is not used
    --    but built in to use for more dynamic versions
    --    machine: a flag to identify the array that the machine
    --    instructions are to be loaded.
    --    state: identifies the row of the machine array that
    --    the instruction and message are to be loaded
    --    trans: identify a transition <trans - X 2> means
    --    transition from the current machine and
    --    state that is a send with message X to
    --    state 2.
    --    initial_state: identifies the starting state of machines.
    --    <initial_state 1 > means that the starting
    --    state of machine one is state one and the
    --    starting state of machine two is state one.
    --    finish: a end of text file delimiter.
    --
    -- As the file is scanned and the text is converted to usable tokens the tokens
    -- are output to default output(CRT) for the user to view. The tokens determine
    -- how the machine adjacency lists are built.
    --

    -- procedure clear_token(in_token : in out string) is

    for i in 1 .. 80 loop
        in_token(i) := ' ';
    end loop;

end read_in_file;
end clear_token;

begin
new_line(4);

while not End_of_File(cfsm) loop
    line_count := line_count + 1;
    if (line_count mod 20) = 0 then
        pause_output;
    end if;
    get_line(cfsm, line_buffer, line_length);

    ----begin parsing line get rid of leading blanks
    while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
        line_index := line_index + 1;
    end loop;

    ---read in instruction token and cast string to enumerated type
    while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
        token(token_index) := line_buffer(line_index);
        token_index := token_index + 1;
        line_index := line_index + 1;
    end loop;

    --cast a string into an enumerated type
    instr enum io.get(token(1..token_index - 1), cfsm_instruction, enum_length);
    token_index := 1;
    clear_token(token);
    case cfsm_instruction is
        when start =>
            put_line("PROGRAM IS LOADING.............");
            new_line(2);
            set_col(20);
            put(cfsm_instruction, set => lower_case);
            line_index := 1;

    when number_of_machines =>
        set_col(20);
        put(cfsm_instruction, set => lower_case);
        while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
            line_index := line_index + 1;
        end loop;
        while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
            token(token_index) := line_buffer(line_index);
            token_index := token_index + 1;
            line_index := line_index + 1;
        end loop;
        get(token, total_machines, conv_length);
put(total_machines, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when number of states =>
set_col(20);
put(cfsm_instruction, set => lower_case);
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, total_states, conv_length);
put(total_states, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when machine =>
set_col(20);
put(cfsm_instruction, set => lower_case);
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, current_machine, conv_length);
put(current_machine, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when state =>
set_col(20);
put(cfsm_instruction, set => lower_case);
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, current_state, conv_length);
put(current_state, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when trans =>
  set col(20);
  put(cfsm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
  if token(1) = '-' then
    current_transition := snd;
  else
    current_transition := rcv;
  end if;
  current_message := token(2);
  clear_token(token);
  token_index := 1;
  while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
  get(token, next_state, conv_length);
  clear_token(token);
  token_index := 1;
  line_index := 1;
  if current_transition = snd then
    put(" #");
  else
    put(" +");
  end if;
  put(current_message);
  put(next_state, width => 2);
  load_machine_array(current_machine, current_state, next_state, current_transition, current_message);
when initial state =>
  set_col(20);
  put(cfsm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
    line_index := line_index + 1;
  end loop;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, start_state_1, conv_length);
put(start_state_1, width => 2);
token_index := 1;
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, start_state_2, conv_length);
put(start_state_2, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when finish =>
  set_col(20);
  put(cfsm instruction, set => lower_case);
  new_line(2);
  put_line(" cfsm's are loaded!!");
  exit;

when others =>
  put_line(" error---error---error");
  for x in 1 .. line_length loop
    put(line_buffer(x));
  end loop;
end case;
new_line;
end loop;
end read_in_file;
-- Title : queues
-- File  : queues.a
-- Author: Matthew J. Rothlisberger, Captain, U. S. Army
-- Date  : 10 February 1992
-- Revised: Sun 4 Workstation
-- Compiler: Verdiix ADA, Version 6.0
-- Description: This generic package contains procedures and functions to implement a queue of
-- type Item. In the model there are queues of characters (to simulate communication
-- channels) and queues of global tuple pointers to aid in the construction
-- of the global reachability graph. The variables chosen for the procedures and functions
-- are self-descriptive. The default size of a queue is 10, the user has the option of
-- making the queue bigger (up to the maximum size of the supporting hardware.) A number
-- of the functions were taken from the text, "Data Structures in Ada" by Michael Feldman.

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

generic

    type Item is private;
    MAX : in integer := 10;

package queues is

    type ArrayList is array(1 .. MAX) of Item;
    type Queue is
        record
            Store : ArrayList;
            tail : integer range 0 .. MAX := 0;
        end record;

    procedure MakeEmpty(Q : in out Queue);
    procedure Enqueue(Q : in out Queue; E : in Item);
    procedure Dequeue(Q : in out Queue);
    function First(Q : Queue) return Item;
    function IsEmpty(Q : Queue) return boolean;
    function IsEqual(Q1 : Queue; Q2 : Queue) return boolean;
    QueueFull : exception;
    QueueEmpty : exception;

end queues;

package body queues is

    procedure MakeEmpty(Q : in out queue) is
        blank_line : string(1 .. 80) := (others => ' ');
    begin

procedure Enqueue(Q : in out Queue; E : in Item) is
begin
  if Q.tail = MAX then
    raise QueueFull;
  else
    Q.tail := Q.tail + 1;
    Q.Store(Q.tail) := E;
  end if;
end Enqueue;

procedure Dequeue(Q : in out Queue) is
begin
  if Q.tail = 0 then
    raise QueueEmpty;
  else
    for i in 2 .. Q.tail loop
      Q.Store(i - 1) := Q.Store(i);
    end loop;
    Q.tail := Q.tail - 1;
  end if;
end Dequeue;

function First(Q : Queue) return Item is
begin
  if Q.tail = 0 then
    raise QueueEmpty;
  else
    return Q.Store(1);
  end if;
end First;

function IsEmpty(Q : Queue) return boolean is
begin
  return Q.tail = 0;
end IsEmpty;

function IsEqual(Q1 : Queue; Q2 : Queue) return boolean is
begin
  for i in 1 .. Q1.tail loop
    if (Q1.Store(i) /= Q2.Store(i)) then
      return false;
    end if;
  end loop;
  return true;
end IsEqual;

end queues;
-- Title : stacks
-- File : stacks.a
-- Author : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date : 10 February 1992
-- Revised :
-- System : Sun 4 Workstation
-- Compiler : VerdiX ADA, Version 6.0
-- Description : This generic package contains procedures and functions to implement a stack of
-- type item. In the model there are only stacks of pointers to aid in the construction
-- of the global reachability graph. The variables chosen for the procedures and functions
-- are self descriptive. The default size of a queue is 10, the user has the option of
-- making the queue bigger (up to the maximum size of the supporting hardware.) A number
-- of the functions were taken from the text, "Data Structures in Ada" by Michael Feldman.

generic

   type Item is private;
   MAX : in integer := 10;

package stacks is

   type ArrayList is array(0 .. MAX) of Item;
   type Stack is
      record
         Store : ArrayList;
         latest : integer range 0 .. MAX := 0;
      end record;

      procedure MakeEmpty(S : in out Stack);
      function IsEmpty(S : Stack) return boolean;
      procedure Push(S : in out Stack; E : in Item);
      procedure Pop(S : in out Stack; E : out Item);
      function Top(S : Stack) return Item;

   StackFull : exception;
   StackEmpty : exception;

end stacks;

package body stacks is

   procedure MakeEmpty(S : in out Stack) is
      begin
         S.latest := 0;
      end MakeEmpty;
procedure Push(S : in out Stack; E : in Item) is
begin
  if S.latest = MAX then
    raise StackFull;
  else
    S.latest := S.latest + 1;
    S.Store(S.latest) := E;
  end if;
end Push;

procedure Pop(S : in out Stack; E : out Item) is
begin
  if S.lat st = 0 then
    raise StackEmpty;
  else
    E := S.Store(S.latest);
    S.latest := S.latest - 1;
  end if;
end Pop;

function Top(S : Stack) return Item is
begin
  if S.lat test = 0 then
    raise StackEmpty;
  else
    return S.Store(S.latest);
  end if;
end Top;

function IsEmpty(S : Stack) return boolean is
begin
  return S.lat est = 0;
end IsEmpty;

end stacks;
function IsEqual (tup1 : in Gstate_record_type; tup2 : in Gstate_record_type) return boolean is
begin
  if ((tup1.machine1.state = tup2.machine1.state) and then (tup1.machine2.state = tup2.machine2.state) and then
    (tup1.queue_12.tail = tup2.queue_12.tail) and then (tup1.queue_21.tail = tup2.queue_21.tail)) then
    if (tup1.queue_12.tail = 0) then
      if (tup1.queue_21.tail = 0) then
        return true;
      else
        if (IsEqual(tup1.queue_21, tup2.queue_21)) then
          return true;
        else
          return false;
        end if;
      end if;
    else
      if (tup1.queue_1.tail = 0) then
        if (IsEqual(tup1.queue_12, tup2.queue_12)) then
          return true;
        else
          return false;
        end if;
      else
        if ((IsEqual(tup1.queue_12, tup2.queue_12)) and then (IsEqual(tup1.queue_21, tup2.queue_21))) then
          return true;
        else
          return false;
        end if;
      end if;
    end if;
  end if;
  else
    return false;
  end if;
end IsEqual;
procedure build_Gstate_graph(machine_array : in out system_array_type;
                        start_state_1 : in out natural;
                        start_state_2 : in out natural;
                        top_Gstate : in out Glink_type;
                        number_of_nodes : in out natural is

parent_Gstate, current_Gstate, find_Gstate, temp_tuple_pointer : Glink_type := null;
Gstate_counter
machine_1
machine_2
M1_state, M2_state
next_state
place_holder1
place_holder2
initial_Gstate
is_new :

-- Title : build_Gstate_graph;
-- File : reachability.a
-- Author : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date : 10 February 1992
-- Revised :
-- System : Sun 4 Workstation
-- Compiler : Verdict ADA, Version 6.0
-- Description : This procedure builds the system reachability graph for two machines. Each system
-- state node can have up to 4 transitions. The algorithm for this procedure is
-- as follows. Each machine state is checked for valid transitions to another state.
-- All send transitions are checked and all receive transitions are checked. A temporary
-- tuple is constructed to hold the values of each machines current state and the queue
-- contents. A BFS/DFS search is done of the graph, checking all nodes for the same values
-- in temp tuple to see if it already exists. If it does, a link is made from the current
-- node to the found node. If a match for the temp is not found a node is created and
-- inserted into the graph. A pointer to the newly added node is added to the pointer
-- queue-stack. A pointer is retrieved from the stack/queue after a scan is done from each
-- machine array. This procedure terminates whenever there are no transitions out of any
-- machine array states, or the initial state for each machine is reached, or the pointer
-- queue-stack is empty. As can be demonstrated, CFSM reachability graphs can grow without
-- bounds. A max pointer queue/stack size is introduced to eliminate unboundedness. Variables
-- accepted by this procedure are machine states, machine queue contents, number of nodes,
-- currently in the graph, and the pointer to the head of the graph.

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

procedure Link_Gstate(temp
begin
if (current_Gstate.link1.Glink = null) then
  current_Gstate.link1.Gmessage := mess;
  current_Gstate.link1.Gtransition := trans;
  current_Gstate.link1.Glink := next_Gstate;
  if (not new one) then
    current_Gstate.link1.new_node := false;
  end if;
else
  if (current_Gstate.link2.Glink = null) then
    current_Gstate.link2.Gmessage := mess;
    current_Gstate.link2.Gtransition := trans;
    current_Gstate.link2.Glink := next_Gstate;
    if (not new one) then
      current_Gstate.link2.new_node := false;
    end if;
  else
    if (current_Gstate.link3.Glink = null) then
      current_Gstate.link3.Gmessage := mess;
      current_Gstate.link3.Gtransition := trans;
      current_Gstate.link3.Glink := next_Gstate;
      if (not new one) then
        current_Gstate.link3.new_node := false;
      end if;
    else
      if (current_Gstate.link4.Glink = null) then
        current_Gstate.link4.Gmessage := mess;
        current_Gstate.link4.Gtransition := trans;
current_Gstate.link4.Glink := next_Gstate;
if (not new one) then
  current_Gstate.link4.new_node := false;
end if;
else
  put_line("All transition pointers used!!!");
end if;
end if;
end if;
end Link_Gstate;

--This procedure simply links the newly created global state node to the previous
--node in the graph.

procedure Insert_Gstate(temp : in out Gstate_record_type;
trans : in out cfsm_transition_type;
mess : in out character;
counter : in out natural) is
  temp_node : Glink_type := new global_state_type;
  temp_ptr : Glink_type;
  is_new : boolean;
  Error_flag : boolean;
begin
  temp_node.system_state_number := counter;
  temp_node.Gtuple := temp;
  is_new := true;
  Link_Gstate(temp, trans, mess, current_Gstate, temp_node, is_new);
  current_Gstate := temp_node;
end Insert_Gstate;

begin
  temp_tuple_pointer := new global_state_type;
  temp_tuple_pointer.system_state_number := Gstate_counter; --initialize the tuple counter
  temp_tuple_pointer.Gtuple.machine1.state := start_state1; --build initial system tuple
  temp_tuple_pointer.Gtuple.machine2.state := start_state2;
  top_Gstate := temp_tuple_pointer;
  parent_Gstate := top_Gstate;
  current_Gstate := parent_Gstate;
  M1.state := start_state1;
  M2.state := start_state2;
  initial_Gstate := temp_tuple_pointer.Gtuple;
new_line;--main loop:
loop
  place_holder1 := machine_array(1) (M1.state);
  while (place_holder1 /= null) loop
    --if the current state in the machine does not have a null value check for valid transitions.
    loop
begin
  current_Gstate := parent_Gstate;
  current_queue2 := current_Gstate.QUEUE.queue_12;
  current_queue21 := current_Gstate.QUEUE.queue_21;
  if (place_holder1.transition = snd) then
    place_holder1.visited := yes;
    Enqueue(current_queue2, place_holder1.message);
    temp_Gstate.machinel_state := place_holder1.next_Mstate;
    temp_Gstate.machinel2.state := M2_state;
    temp_Gstate.queue_12 := current_queue2;
    temp_Gstate.queue_21 := current_queue21;
    find Gstate := search for tuple(top_Gstate, temp_Gstate);
    if (find Gstate = null) then
      Gstate_counter := Gstate_counter + 1;
      Insert_Gstate(temp_Gstate, place_holder1.transition, place_holder1.message, current_Gstate,
        Gstate_counter);
      case search is
        when bfs =>
          Enqueue(Gpointer_queue, current_Gstate);
          when dfs =>
            Push(Gpointer_stack, current_Gstate);
            when others =>
              put_line("error");
          end case;
        else
          is new := (false;
          Link_Gstate(temp_Gstate, place_holder1.transition, place_holder1.message, current_Gstate,
            find_Gstate, is_new);
        end if;
      else
        if ((not IsEmpty(current_queue21)) and then (place_holder1.transition = rcv and then
          (place_holder1.message = First(current_queue21))) then
          place_holder1.visited := yes;
          Dequeue(current_queue21);
          temp_Gstate.machinel_state := place_holder1.next_Mstate;
          temp_Gstate.machinel2.state := M2_state;
          temp_Gstate.queue_12 := current_queue2;
          temp_Gstate.queue_21 := current_queue21;
          find Gstate := search for tuple(top_Gstate, temp_Gstate);
          if (find Gstate = null) then
            Gstate_counter := Gstate_counter + 1;
            Insert_Gstate(temp_Gstate, place_holder1.transition, place_holder1.message, current_Gstate,
              Gstate_counter);
            case search is
              when bfs =>
                Enqueue(Gpointer_queue, current_Gstate);
                when dfs =>
                  Push(Gpointer_stack, current_Gstate);
                when others =>
                  put_line("error");
              end case;
            else
              is new := (false;
            end if;
        end if;
      end if;
    end if;
end if;
    put_line("error");
end case;
else
  is_new := false;
  Link_Gstate(temp_Gstate, place_holder1.transition, place_holder1.message, current_Gstate, find_Gstate, is_new);
end if;
end if;
place_holder1 := place_holder1.Slink;
exit;
exception
when Gpointer_queue_pack.QueueFull =>
  put("The pointer queue is full at state: ");
  put(current_Gstate.system_state_number, width => 1);
  new_line;
  put(reach, "The pointer queue is full at state: ");
  put(reach, current_Gstate.system_state_number, width => 1);
  new_line(reach);
  Gpointer_queue_pack.MakeEmpty(Gpointer_queue);
  exit main_loop;
when Gpointer_stack_pack.StackFull =>
  put("The pointer stack is full at state: ");
  put(current_Gstate.system_state_number, width => 1);
  new_line;
  put(reach, "The pointer stack is full at state: ");
  put(reach, current_Gstate.system_state_number, width => 1);
  new_line(reach);
  Gpointer_stack_pack.MakeEmpty(Gpointer_stack);
  exit main_loop;
when queue_pack.QueueFull =>
  put("Exceeded the channel capacity in queue_12 at state: ");
  put(parent_Gstate.system_state_number, width => 1);
  new_line;
  put(reach, "Exceeded the channel capacity in queue_12 at state: ");
  put(reach, parent_Gstate.system_state_number, width => 1);
  new_line(reach);
  place_holder1 := place_holder1.Slink;
  exit;
end;
end loop;
end loop;
place_holder2 := machine_array(2) (M2_state);
while(place_holder2 /= null) loop
begin
  current_Gstate := parent_Gstate;
  current_queue12 := current_Gstate.Gtuple.queue_12;
  current_queue21 := current_Gstate.Gtuple.queue_21;
  if (place_holder2.transition = snd) then
    place_holder2.visited := yes;
    Enqueue(current_queue21, place_holder2.message);
    temp_Gstate.machine1_state := M1_state;
    temp.Gstate.machine2_state := place_holder2.next_Mstate;
temp.Gstate.queue_12 := current_queue12;
temp.Gstate.queue_21 := current_queue21;
find.Gstate := search for tuple(top.Gstate, temp.Gstate);
if (find.Gstate = null) then
    Gstate_counter := Gstate_counter + 1;
    Insert(Gstate, temp.Gstate, place_holder2.transition, place_holder2.message, current_Gstate, Gstate_counter);
    case search is
        when bfs =>
            Enqueue(Gpointer_queue, current_Gstate);
        when dfs =>
            Push(Gpointer_stack, current_Gstate);
        when others =>
            put_line("error");
    end case;
else
    is_new := false;
    Link(Gstate, temp.Gstate, place_holder2.transition, place_holder2.message, current_Gstate, find.Gstate, is_new);
end if;
else
if ((not isEmpty(current_queue12)) and then (place_holder2.transition = rcv and then
    (place_holder2.message = First(current_queue12))))) then
    place_holder2.visited := yes;
    Dequeue(current_queue12);
    temp.Gstate.machine2.state := M1.state;
    temp.Gstate.machine2.state := place_holder2.next.Mstate;
    temp.Gstate.queue_12 := current_queue12;
    temp.Gstate.queue_21 := current_queue21;
    find.Gstate := search for tuple(top.Gstate, temp.Gstate);
    if (find.Gstate = null) then
        Gstate_counter := Gstate_counter + 1;
        Insert(Gstate, temp.Gstate, place_holder2.transition, place_holder2.message, current_Gstate, Gstate_counter);
        case search is
            when bfs =>
                Enqueue(Gpointer_queue, current_Gstate);
            when dfs =>
                Push(Gpointer_stack, current_Gstate);
            when others =>
                put_line("error");
        end case;
    else
        is_new := false;
        Link(Gstate, temp.Gstate, place_holder2.transition, place_holder2.message, current_Gstate, find.Gstate, is_new);
    end if;
end if;
end if;
place_holder2 := place_holder2.Slink;
exit;
exception
when Gpointer queue pack.QueueFull =>
put(" The pointer queue is full at state: ");
put(current_Gstate.system_state_number, width => 1);
new_line;
put(reach, " The pointer queue is full at state: ");
put(reach, current_Gstate.system_state_number, width => 1);
new_line(reach);
case search is
when bfs =>
    Gpointer_queue_pack.MakeEmpty(Gpointer_queue);
when dfs =>
    Gpointer_stack_pack.MakeEmpty(Gpointer_stack);
when others =>
    put_line("error");
end case;
exit main_loop;
when queue pack.QueueFull =>
put(" Exceeded the channel capacity in queue_21 at state: ");
put(parent_Gstate.system_state_number, width => 1);
new_line;
put(reach, " Exceeded the channel capacity in queue_21 at state: ");
put(reach, parent_Gstate.system_state_number, width => 1);
new_line(reach);
place_holder2 := place_holder2.Slink;
exit;
end;
end loop;
case search is
when bfs =>
    if (IsEmpty(Gpointer_queue)) then
        exit;
    else
        parent_Gstate := First(Gpointer_queue);
        Dequeue(Gpointer_queue);
        M1_state := parent_Gstate.Gtuple.machine1_state;
        M2_state := parent_Gstate.Gtuple.machine2_state;
        end if;
when dfs =>
    if (IsEmpty(Gpointer_stack)) then
        exit;
    else
        Pop(Gpointer_stack, parent_Gstate);
        M1_state := parent_Gstate.Gtuple.machine1_state;
        M2_state := parent_Gstate.Gtuple.machine2_state;
        end if;
when others =>
    put_line("error error error");
end case;
end loop main_loop;
new_line;
number of nodes := Gstate_counter;
put(" Number of nodes in the Reachability Graph: ");
put(Gstate_counter, width => 1);
new line;
end build_Gstate_graph;
procedure clear_pointers is
begin
    initial_global_state := null;
    for i in 1 .. 2 loop
        for j in 1 .. 10 loop
            machine_array(i)(j) := null;
        end loop;
    end loop;
end clear_pointers;
function search_for_tuple(top : in Glink_type; tuple : in Gstate_record_type) return Glink_type is
begin
  if (top = null) then
    return null;
  elsif (isEqual(tuple, top.Gtuple)) then
    return top;
  else
    if ((temp = null) and then (top.link1.new_node = true)) then
      temp := search_for_tuple(top.link1.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link2.new_node = true)) then
      temp := search_for_tuple(top.link2.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link3.new_node = true)) then
      temp := search_for_tuple(top.link3.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link4.new_node = true)) then
      temp := search_for_tuple(top.link4.Glink, tuple);
    end if;
    return temp;
  end if;
end search_for_tuple;
separate(main)
----------------------------------------------
-- Title     : pause_output
-- File      : output.a
-- Author    : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date      : 10 February 1992
-- Revised   :  
-- System    : Sun 4 Workstation
-- Compiler  : Verdict ADA, Version 6.0
-- Description: Procedure called when default output screen reaches max. Default max is 25 lines.
----------------------------------------------

procedure pause_output is
  keyboard : string(1 .. 1);
  len      : natural;
begin
  new_line;
  put_line("ENTER to continue");
  get_line(keyboard, len);
  put(ASCII.ESC);
  put_line("[2J");
end pause_output;

separate(main)
----------------------------------------------
-- Title     : unprompted pause
-- File      : output.a
-- Author    : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date      : 10 February 1992
-- Revised   :  
-- System    : Sun 4 Workstation
-- Compiler  : Verdict ADA, Version 6.0
-- Description: Procedure called when default output screen reaches max. Default max is 25 lines. 
User is not prompted for ENTER.
----------------------------------------------

procedure unprompted_pause is
  keyboard : string(1 .. 1);
  len      : natural;
begin
  new_line;
  get_line(keyboard, len);
end unprompted_pause;
procedure PrintQueue(Q : in queue_pack.Queue) is
begin
  if Q.tail = 0 then
    put('E');
  else
    for i in 1 .. Q.tail loop
      put(Q.Store(i));
    end loop;
  end if;
end PrintQueue;
procedure output Gstate_node(Gstate_pointer : in out Glink_type; Error_flag : in out boolean) is
begin
  output line count := output line count + 1;
  if ((output_line_count mod 10) = 0) then
    unprompted_pause;
  end if;
  set col(15);
  put(Gstate_pointer.system_state_number, width => 3);
  put("" & integer'image(Gstate_pointer.Gtuple.machine1_state) & ", ");
  PrintQueue(Gstate_pointer.Gtuple.queue_12);
  put("", ");
  PrintQueue(Gstate_pointer.Gtuple.queue_21);
  put("" & integer'image(Gstate_pointer.Gtuple.machine2_state) & "");
  if ((Gstate_pointer.Gtuple.link1.Glink = null) and then (Gstate_pointer.link2.Glink = null)) and then
    (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null)) then
    set col(42);
    put("" & integer'image(Gstate_pointer.Gtuple.queue_12.tail /= 0) or (Gstate_pointer.Gtuple.queue_21.tail /= 0)) then
      set col(42);
      Error_flag := true;
    else
      set col(42);
      put("" & integer'image(Gstate_pointer.Gtuple.queue_12.tail /= 0) or (Gstate_pointer.Gtuple.queue_21.tail /= 0)) then
        set col(42);
        Error_flag := true;
      end if;
  else
    Error_flag := false;
  end if;
end output Gstate_node;
procedure output_Gstate_transition(Gstate_ptr : in out Glink_type) is
begin
   if (Gstate_ptr.link1.Glink /= null) then
      set_col(42);
      if (Gstate_ptr.link1.Gtransition = snd) then
         put("- ");
      else
         put("*");
      end if;
      put(Gstate_ptr.link1.Gmessage);
      put("- ");
      put("\text{integer}\text{image}(\text{Gstate_ptr.link1.Glink.Gtuple.machinel_state}) \& \text{" "}");
      PrintQueue(Gstate_ptr.link1.Glink.Gtuple.queue,12);
      put("- ");
      PrintQueue(Gstate_ptr.link1.Glink.Gtuple.queue,21);
      put("\text{integer}\text{image}(\text{Gstate_ptr.link1.Glink.Gtuple.machine2_state}) \& \text{" "})
      put(Gstate_ptr.link1.Glink.Gtuple.system.state_number, width => 1);
      new_line;
   end if;
   if (Gstate_ptr.link2.Glink /= null) then
      set_col(42);
      if (Gstate_ptr.link2.Gtransition = snd) then
         put("- ");
      else
         put("*");
      end if;
      put(Gstate_ptr.link2.Gmessage);
      put("- ");
      put("\text{integer}\text{image}(\text{Gstate_ptr.link2.Glink.Gtuple.machinel_state}) \& \text{" "}");
      PrintQueue(Gstate_ptr.link2.Glink.Gtuple.queue,12);
      put("- ");
      PrintQueue(Gstate_ptr.link2.Glink.Gtuple.queue,21);
      put("\text{integer}\text{image}(\text{Gstate_ptr.link2.Glink.Gtuple.machine2_state}) \& \text{" "})
      put(Gstate_ptr.link2.Glink.Gtuple.system.state_number, width => 4);
   end if;
end output_Gstate_transition;
new_line;
end if;
if (Gstate_pointer.link3.Glink /= null) then
set col(42);
    if (Gstate_pointer.link3.Gtransition = snd) then
        put("=");
    else
        put("+");
    end if;
    put(Gstate_pointer.link3.Gmessage);
    put(" ");
    put(" & integer\'image(Gstate_pointer.link3.Glink.Gtiple.machinel_state) & " , ");
    PrintQueue(Gstate_pointer.link3.Glink.Gtiple.queue_12);
    put(" ");
    PrintQueue(Gstate_pointer.link3.Glink.Gtiple.queue_21);
    put(" ");
    put(" & integer\'image(Gstate_pointer.link3.Glink.Gtiple.machinel2_state) & " ] "");
    put(Gstate_pointer.link3.Glink.system_state_number, width => 4);
    new_line;
end if;
if (Gstate_pointer.link4.Glink /= null) then
set col(42);
    if (Gstate_pointer.link4.Gtransition = snd) then
        put("-");
    else
        put("=");
    end if;
    put(Gstate_pointer.link4.Gmessage);
    put(" ");
    put(" & integer\'image(Gstate_pointer.link4.Glink.Gtiple.machinel_state) & " , ");
    PrintQueue(Gstate_pointer.link4.Glink.Gtiple.queue_12);
    put(" ");
    PrintQueue(Gstate_pointer.link4.Glink.Gtiple.queue_21);
    put(" ");
    put(" & integer\'image(Gstate_pointer.link4.Glink.Gtiple.machinel2_state) & " ] "");
    put(Gstate_pointer.link4.Glink.system_state_number, width => 4);
    new_line;
end if;
end output.Gstate_transition;
procedure output_Gstates(top_Gstate : in out Glink_type; number_of_nodes : in out natural) is
  Error_flag : boolean := true;
  current : Glink_type := top_Gstate;
begin
  if (top_Gstate = null) then
    return;
  else
    output_Gstate_node(top_Gstate, Error_flag);
    if (not Error_flag) then
      output_Gstate_transition(top_Gstate);
      end if;
      if (top_Gstate.link1.new_node = true) then
        output_Gstates(top_Gstate.link1.Glink, number_of_nodes);
      end if;
      if (top_Gstate.link2.new_node = true) then
        output_Gstates(top_Gstate.link2.Glink, number_of_nodes);
      end if;
      if (top_Gstate.link3.new_node = true) then
        output_Gstates(top_Gstate.link3.Glink, number_of_nodes);
      end if;
      if (top_Gstate.link4.new_node = true) then
        output_Gstates(top_Gstate.link4.Glink, number_of_nodes);
      end if;
    end if;
  end output_Gstates;
procedure output_machine_arrays(machine_array : in out system_array_type) is
    line_count : integer := 7;
begin
    new line(2);
    set_col(20);
    put("-----------------------------------");
    set_col(20);
    put(" Machine 1 Array Contents ");
    set_col(20);
    put("-----------------------------------");
    set_col(20);
    put(" From i To j Transition i Executed ");
    set_col(20);
    put("-----------------------------------");
    set_col(20);
    new line;
    for curr_state in 1 .. 10 loop
        place_holder := machine_array(curr_state);
        while (place_holder /= null) loop
            line_count := line_count + 1;
            if (line_count mod 20) = 0 then
                pause_output;
            end if;
            set_col(20);
            put("[" & integer'image(curr_state));
            put("[" & integer'image(place_holder.next_Mstate) & "]");
            put(place_holder.transition, set => lower_case);
            put("[" & place_holder.message & "]");
            put("[");
            put(place_holder.visited, set => lower_case, width => 3);
            put("]");
            new line;
            place_holder := place_holder.Slink;
        end loop;
    end loop;
    set_col(20);
    put("-----------------------------------");
    new line(2);
    set_col(20);
put("---------------------------------------");
set_col(20);
put("Machine 2 Array Contents");
set_col(20);
put("---------------------------------------");
set_col(20);
put("From | To | Transition | Executed");
set_col(20);
put("---------------------------------------");
for curr_state2 in 1 .. 10 loop
  place_holder := machine_array[2](curr_state2);
  while (place_holder /= null) loop
    line_count := line_count + 1;
    if ((line_count mod 20) = 0) then
      pause_output;
    end if;
    set_col(20);
    put(" | " & integer'image(curr_state2));
    put(" | " & integer'image(place_holder.next_Mstate) & ";
    put(" | place_holder.transition, set -> lower_case);
    put(" | place_holder.message & ");
    put(" |");
    put(place_holder.visited, set -> lower_case, width -> 3);
    put(" |");
    new_line;
    place_holder := place_holder.Slink;
  end loop;
end loop;
set_col(20);
put("---------------------------------------");
set_col(20);
put("|
new line:
end output machine_arrays;
procedure PrintQueue_to_file(Q : in queue_pack.Queue) is
begin
    if (Q.tail = 0) then
        put(reach, 'E');
    else
        for i in 1 .. Q.tail loop
            put(reach, Q.Store(i));
        end loop;
    end if;
end PrintQueue_to_file;
procedure output_Gstate_node_to_file(Gstate Pointer : in out Glink_type; Error_flag : in out boolean) is
begin
  set col(reach, 15); put(reach, Gstate Pointer.system.state_number, width => 3); put(reach, " ");
  put(reach, " "); PrintQueue to file(Gstate Pointer.Gtuple.queue_12); put(reach, " ");
  PrintQueue to file(Gstate Pointer.Gtuple.queue_21); put(reach, " ");
  if ((Gstate Pointer.Gtuple.machine1.state & 0) = 0) then
    Error_flag := true;
  else
    Error_flag := false;
  end if;
end output_Gstate_node_to_file;
```plaintext
procedure output_Gstate_transition_to_file(Gstate_pointer : in out Glink_type) is
begin
    if (Gstate_pointer.link1.Glink /= null) then
        set col(reach, 42);
        if (Gstate_pointer.link1.Gtransition = snd) then
            put(reach, "-");
        else
            put(reach, "+");
        end if;
        put(reach, Gstate_pointer.link1.Gmessage);
        put(reach, ",");
        put(reach, "[" & integer'image(Gstate_pointer.link1.Glink.Gtuple.machine1_state) & "," & ",");
        PrintQueue_to_file(Gstate_pointer.link1.Glink.Gtuple.queue_12);
        put(reach, ",");
        PrintQueue_to_file(Gstate_pointer.link1.Glink.Gtuple.queue_21);
        put(reach, "[" & integer'image(Gstate_pointer.link1.Glink.Gtuple.machine2_state) & "," & ",");
        put(reach, Gstate_pointer.link1.Glink.system_state_number, width => 4);
        new_line(reach);
    end if;
    if (Gstate_pointer.link2.Glink /= null) then
        set col(reach, 42);
        if (Gstate_pointer.link2.Gtransition = snd) then
            put(reach, "-");
        else
            put(reach, "+");
        end if;
        put(reach, Gstate_pointer.link2.Gmessage);
        put(reach, ",");
        put(reach, "[" & integer'image(Gstate_pointer.link2.Glink.Gtuple.machine1_state) & "," & ",");
        PrintQueue_to_file(Gstate_pointer.link2.Glink.Gtuple.queue_12);
        put(reach, ",");
        PrintQueue_to_file(Gstate_pointer.link2.Glink.Gtuple.queue_21);
        put(reach, "[" & integer'image(Gstate_pointer.link2.Glink.Gtuple.machine2_state) & "," & ",");
        put(reach, Gstate_pointer.link2.Glink.system_state_number, width => 4);
        new_line(reach);
    end if;
    if (Gstate_pointer.link3.Glink /= null) then
        set col(reach, 42);
        if (Gstate_pointer.link3.Gtransition = snd) then
```
put(reach, "-");
else
put(reach, "*");
end if;
put(reach, Gstate_pointer.link3.Gmessage);
put(reach, " ");
put(reach, "[" & integer\"image\"(Gstate_pointer.link3.Glink.Gtuple.machine1_state) & ", "]");
PrintQueue to file(Gstate_pointer.link3.Glink.Gtuple.queue_12);
put(reach, " ");
PrintQueue to file(Gstate_pointer.link3.Glink.Gtuple.queue_21);
put(reach, " ");
put(reach, "[" & integer\"image\"(Gstate_pointer.link3.Glink.Gtuple.machine2_state) & "] ");
put(reach, Gstate_pointer.link3.Glink.system_state_number, width => 4);
new line(reach);
end if;
if (Gstate_pointer.link4.Glink /= null) then
set col(reach, 42);
if (Gstate_pointer.link4.Gtransition = snd) then
put(reach, "-");
else
put(reach, "*");
end if;
put(reach, Gstate_pointer.link4.Gmessage);
put(reach, " ");
put(reach, "[" & integer\"image\"(Gstate_pointer.link4.Glink.Gtuple.machine1_state) & ", "]");
PrintQueue to file(Gstate_pointer.link4.Glink.Gtuple.queue_12);
put(reach, " ");
PrintQueue to file(Gstate_pointer.link4.Glink.Gtuple.queue_21);
put(reach, " ");
put(reach, "[" & integer\"image\"(Gstate_pointer.link4.Glink.Gtuple.machine2_state) & "] ");
put(reach, Gstate_pointer.link4.Glink.system_state_number, width => 4);
new line(reach);
end if;
end output_Gstate_transition_to_file;
procedure output_Gstates_to_file(top_Gstate := in out Glink_type; number_of_nodes := in out natural) is
  Error_flag : boolean := true;
  current : Glink_type := top_Gstate;
begin
  if (top_Gstate = null) then
    return;
  else
    output_Gstate_node_to_file(top_Gstate, Error_flag);
    if (not Error_Flag) then
      output_Gstate_transition_to_file(top_Gstate);
    end if;
    if (top_Gstate.link1.new_node = true) then
      output_Gstates_to_file(top_Gstate.link1.Glink, number_of_nodes);
    end if;
    if (top_Gstate.link2.new_node = true) then
      output_Gstates_to_file(top_Gstate.link2.Glink, number_of_nodes);
    end if;
    if (top_Gstate.link3.new_node = true) then
      output_Gstates_to_file(top_Gstate.link3.Glink, number_of_nodes);
    end if;
    if (top_Gstate.link4.new_node = true) then
      output_Gstates_to_file(top_Gstate.link4.Glink, number_of_nodes);
    end if;
  end if;
end output_Gstates_to_file;
procedure output_machine_arrays_to_file(machine_array : in out system_array; pe) is
place_holder := Slink_Type;
begin
new_line(reach, 2);
set_col(reach, 20);
put(reach, "------------------------------------------");
set_col(reach, 20);
put(reach, "  Machine 1 Array Contents ");
set_col(reach, 20);
put(reach, "------------------------------------------");
set_col(reach, 20);
put(reach, "  From | To | Transition | Executed |");
set_col(reach, 20);
put(reach, "------------------------------------------");
for curr_state in 1 .. 10 loop
place_holder := machine_array(1) (curr_state);
while (place_holder /= null) loop
set_col(reach, 20);
put(reach, "  | integer'image(curr_state)");
put(reach, "  | integer'image(place_holder.next_Mstate) & " ");
put(reach, "  | place_holder.transition, set => lower_case
put(reach, "  | place_holder.message & " ");
put(reach, "  ");
put(reach, "  ");
new_line(reach);
place_holder := place_holder.Slink;
end loop;
end loop;
set_col(reach, 20);
put(reach, "------------------------------------------");
new_line(reach, 2);
set_col(reach, 20);
put(reach, "--------------------------------------------------");
set_col(reach, 20);
put(reach, "Machine 2 Array Contents */");
set_col(reach, 20);
put(reach, "--------------------------------------------------");
set col(reach, 20);
put(reach, "! From | To | Transition | Executed */");
set col(reach, 20);
put(reach, "--------------------------------------------------");
for curr_state2 in 1 .. 10 loop
    place_holder := machine_array(2) (curr_state2);
    while (place_holder /= null) loop
        set_col(reach, 20);
        put(reach, "i & integer'image(curr_state2));
        put(reach, "i & integer'image(place_holder.next_Mstate) & " | "");
        put(reach, place_holder.transition, set => lower_case);
        put(reach, "i & place_holder.message & " | "");
        put(reach, " ");
        put(reach, place_holder.visited, set => lower_case, width => 3);
        put(reach, " ");
        new_line(reach);
        place_holder := place_holder.Slink;
    end loop;
end loop;
set col(reach, 20);
put(reach, "--------------------------------------------------");
new_line(reach, 2);
put(reach, "The nodes generated by the analysis");
new_line(reach);
put(reach, "we were done in a ");
case search is
    when dfs =>
        put_line(reach, "depth first manner");
    when bfs =>
        put_line(reach, "breadth first manner");
    when others =>
        put_line("error");
end case;
new_line(reach);
end output_machine_arrays to_file;
procedure create_output_file is
begin
set col(10);
put("Filename of output analysis: ");
get_line(out_file_name, out_name_len);
create(reach, mode => out_file, name => out_file_name(1 .. out_name_len));
put(ASCII.ESC);
push_line("(2J");
new_line(reach, 2);
put(reach, "");
new_line(reach, 2);
end create_output_file;
procedure output_analysis is

begin

pause output;
output_machine_arrays(machine_array);
pause output;
build Gstate_graph(machine_array, start_state_1, start_state_2, initial_global_state, number_of_nodes);
pause output;
new_line(2);
put_line("REACHABILITY GRAPH");
new_line;
output_line_count := 1;
output Gstates(initial_global_state, number_of_nodes);
pause output;
output_machine_arrays(machine_array);
pause output;
new_line(2);
new_line(reach, 2);
put_line(reach, "Reachability Graph");
output Gstates_to_file(initial_global_state, number_of_nodes);
new_line(reach);
output machine_arrays_to_file(machine_array);
end output_analysis;
APPENDIX B SCM CODE

The program listing begins on the following page.
---
Title: main
File: main.a
Author: Matthew J. Rothlisberger, Captain, U. S. Army
Date: 16 March 92
System: Sun 4 Workstation
Compiler: Verdict ADA, Version 6.0
Description: This is the main procedure. The user provides a text file that is the specification of a two
machine model. The text file is parsed and machine adjacency lists are built. A global and
system reachability analysis is done on the machine specification and output to screen and
text file. All procedures that the main uses were created and compiled as separate units.
Descriptions of each function and procedure can be found in the header of the associated
units/files.
---

with Text_10, queues, stacks, definitions;
use Text_10, definitions;

-- User Defined: package definitions

procedure main is
  max_length : natural := 40; -- length of global state array
type scm_instruction_type is (start, number_of_machines, machine, number_of_states, state, initial_state, trans,
  finish);
type visit_type is (yes, no);
package trans_enum_io is new Text_10 Enumeration_10(scm_instruction_type);
package instr_enum_io is new Text_10 Enumeration_10(scm_instruction_type);
package vst_enum_io is new Text_10 Enumeration_10(visit_type);
package buff_enum_io is new Text_10 Enumeration_10(buffer_type);
package scm_integer_io is new Integer_10(integer);
package bool_enum_io is new Text_10 Enumeration_10(boolean_type);
use bool_enum_io;
use trans_enum_io, instr_enum_io, vst_enum_io,
  buff_enum_io, scm_integer_io;

-- data structures for the machine arrays

---
type machine_array_record_type is
record
  transition : scm_instruction_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 .. 10) of Slink_type;

type system_array_type is array(1 .. 2) of machine_array_type;

data structures for the global state tuple(node)

type global_state_type;

type Glink_type is access global_state_type;

transition structure

type Gstate_transition_type is
record
  Gtransition : scm_transition_type := unused;
  new_node   : boolean := true;
  Glink      : Glink_type;
  visited    : boolean := false;
end record;

global tuple structure

type Gstate_record_type is
record
  machine1_state : machine1_state_type;
  machine2_state : machine2_state_type;
  global_variables : global_variable_type;
end record;

Global state node, contains transition, state, and link information
---needed for building the global state graph

type global_state_type is
record
  system_state_number : natural := 0;
  Gtuple   : Gstate_record_type;
  link1    : Gstate_transition_type;
  link2    : Gstate_transition_type;
  link3    : Gstate_transition_type;
  link4    : Gstate_transition_type;
end record;

type system_state_type;

type Syslink_type is access system_state_type;
--transition structure for system state

---

type State_transition_type is
record
  Stransition : scm_transition_type := unused;
  new_node : boolean := true;
end record;

type State_record_type is
record
  machine1_state : natural := 0;
  machine2_state : natural := 0;
end record;

---

type system_state_type is
record
  system_state_number : natural := 0;
  Stuple : State_record_type;
  link1 : State_transition_type;
  link2 : State_transition_type;
  link3 : State_transition_type;
  link4 : State_transition_type;
end record;

---

--declaration of stacks of pointers and queues of pointers
--along with queues of messages to simulate two way communications
--between machines

package transition_stack_package is new stacks(scm_transition_type, MAX => 5);
package Gtrans_stack_package is new stacks(Gstate_transition_type, MAX => 50);
package Gtrans_queue_package is new queues(Gstate_transition_type, MAX => 50);
package Strno_queue_package is new queues(Integer, MAX => 50);
package Gpointer_stack_package is new stacks(Glink_type, MAX => 50);
package Gpointer_queue_package is new queues(Glink_type, MAX => 50);
package Spointer_queue_package is new queues(Syslink_type, MAX => 50);
package Gstate_stack_package is new stacks(global_state_type);
package count_inout
  use Gpointer_stack_package, Gstate_stack_package, Gpointer_queue_package,
    transition_stack_package, count_inout, Gtrans_stack_package, Gtrans_queue_package,
    Strno_queue_package, Spointer_queue_package, Spointer_queue_package;

---

--global variables are initialized

initial global state : Glink_type;   -- top tuple in global graph
initial system state : Syslink_type;
```c
scm_instruction : scm_instruction_type; -- used in case statements
total_machines : natural := 0; -- number of machines in the system
total_states : natural := 0; -- number of states per machine
current_state : natural := 0; -- current state of the current machine
next_state : natural := 0;
current_machine : natural := 0;
current_transition : scm_transition_type;
current_message : character; -- message off the top of the queue
transition_stack : transition_stack_package.stack;
start_state_1 : natural := 0;
start_state_2 : natural := 0;
temp_Gstate : Gstate_record_type; -- a temporary holding place for tuples
gpointer_array : system_array_type; -- an array of arrays so that the machines can be subscripted to be
Gpointer_stack : Gpointer_stack_package.stack;
Gqueue : Gqueue_package.queue;
link_queue : String_queue.package.queue;
Gtrans_stack : Gtrans_stack_package.stack;
Gtrans_queue : Gtrans_queue_package.queue;
Sstate : Squeue_package.queue;
number_of_nodes : natural := 0;
output_line_count : string(1..20) := (other => ' ') ;
out_file_name : integer; -- stops output from scrolling if longer than 25 lines
in_name_len : integer;
out_name_len : integer;
analysis_len : integer;
trans_num : integer := 8;
Scolumn_set : positive_count := 35;
Gcolumn_set : positive_count := 5;
scm : file_type; -- input file pointer
reach : file_type; -- output file pointer

--------------------------------------------------------------------------
procedure scroll_pause
procedure output_Stuple
(tuple : in out Gstate_record_type) is separate; -- global_output.a
--------------------------------------------------------------------------
procedure output_Sstate_node
procedure output_Sstate_transition
(procedure output_Sstates
procedure clear_pause
--------------------------------------------------------------------------
-- User Defined -------------------------------------------------------------------
procedure output_Gtuple
(tuple : in out Gstate_record_type) -- user_output.a
--------------------------------------------------------------------------
procedure output_Gtuple_to_file
(tuple : in out Gstate_record_type) -- user_output.a
```
procedure output_Gstate_node
    (Gstate_pointer : in out Glink_type; --global_output.a
     Error_flag : in out boolean) is separate;

procedure output_Gstate_transition
    (Gstate_pointer : in out Glink_type) is separate; --global_output.a

procedure output_Gstates
    (top_Gstate : in out Glink_type) is separate; --global_output.a

procedure output_machine_arrays
    (machine_array : in out system_array_type) is separate; --global_output.a

procedure load_machine_array
    (number : in out natural; --input.a
     state : in out natural; --
     Next_state : in out natural; --
     Transition : in out scm_transition_type) --

procedure read_in_file
    (scm : in out file_type) is separate; --input.a

function IsSysStateEqual
    (top : in Syslink_type; --system_reachability.a
     current : in system_state_type) --

return boolean is separate;

function search_for_Gtuple
    (top : in Glink_type; --global_search.a
     tuple : in Gstate_record_type) --

return Glink_type is separate; --

function search_for_Stuple
    (top : in Syslink_type; --system_search.a
     current : in system_state_type) --

return Syslink_type is separate; --

function Analyze_Predicates_Machine1
    (local : machine1_state_type; --predicate_action.a
     GLOBAL : global_variable_type) --

return transition_stack_package.stack is separate; --

function Analyze_Predicates_Machine2
    (local : machine2_state_type; --predicate_action.a
     GLOBAL : global_variable_type)

return transition_stack_package.stack is separate; --

procedure Action
    (in_system_state: in out Gstate_record_type; --predicate_action.a
     in_transition : in out scm_transition_type; --
     out_system_state : in out Gstate_record_type) --

is separate;

procedure build_Gstate_graph
    (machine_array : in out system_array_type; --global_reachability.a
     start_state1 : in out natural; --
     start_state2 : in out natural; --
     top_Gstate : in out Glink_type) is separate; --

procedure build_Gstate_graph
    (top_Gstate : in out Glink_type; --system_reachability.a
     top_State : in out Syslink_type) is separate; --

procedure output_analysis
    (top_State : in out Syslink_type) is separate; --

procedure clear_pointers
    (top_State : in out Syslink_type) is separate; --

begin
    main scm_loop ;
loop
  put(ASCII.ESC); -- call to operating system to clear the screen
  put_line("[2J");
  new_line(10);
  -- query the user for the input file
  loop
    begin
      set_col(10);
      put("Filename of scm specification (ENTER to quit): ");
      get_line(filename, in_name_len);
      if (in_name_len = 0) then
        exit main_scm_loop;
      else
        open(scm, mode => in_file, name => file_name(1..in_name_len));
        end if;
        exit;
        exception
          when NAME_ERROR =>
            put_line("That file does not exist. Reenter!!");
        end;
    end loop;
    read_in_file(scm);
    clear_pause;
    output_analysis;
    close(scm);
    clear_pointers;
  end loop main_scm_loop;
end main;
Separate main
--------------------------------
-- Title : load machine array
-- File  : input.a
-- Author: Matthew J. Rothlisberger, Captain, U. S. Army
-- Date  : 20 March 1992
-- Revised:
-- System: Sun 4 Workstation
-- Compiler: Verdict ADA, Version 6.0
-- Description : Builds machine adjacency list. Takes transition and message as input and adds them to
the lists. The possible transitions are determined by the file "user_spec.a".
-- The machine arrays are represented by a single one dimensional array of arrays of
linked lists.
--------------------------------

procedure load machine array (number : in out natural;
state : in out natural;
Next state : in out natural;
Transition : in out scm_transition_type) is
end of list : Slink type := machine_array (number) (state);
temp link : Slink type := new machine_array_record_type;
begi

end load machine array;
procedure read_in_file(scm : in out file_type) is

line_buffer : string(1 .. 80) := (others => ' ');
token : string(1 .. 80) := (others => ' ');
token_index : integer := 1;
line_length : integer := 1; --variable used for halting output every
enum_length : integer := 25 lines; --25 lines.
conv_length : integer := 1;
line_count : integer := 1;

--clears the token used in parsing
procedure clear_token(in_token : in out string) is
begin
for i in 1 .. 80 loop
in_token(i) := ' ';
end loop;
end clear_token;

begin
new_line(4);
while not End of File(scm) loop
line_count := line_count + 1;
if ((line_count mod 20) = 0) then
scroll_pause;
end if;
get_line(scm, line_buffer, line_length);
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
token(token_index) := line_buffer(line_index);
token_index := token_index + 1;
line_index := line_index + 1;
end loop;
instr_enum io.get(token(1 .. token_index - 1), scm_instruction, enum_length);
token_index := 1;
clear_token(token);
case scm_instruction is
when Start =>
put_line("PROGRAM IS LOADING.............");
new_line(2);
set_col(20);
put(scm_instruction, set => lower_case);
line_index := 1;
when number_of_machines =>
set_col(20);
put(scm_instruction, set => lower_case);
while line_index < line_length + 1 and then line_buffer(line_index) = ' ' loop
line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
token(token_index) := line_buffer(line_index);
token_index := token_index + 1;
line_index := line_index + 1;
end loop;
get(token, total_machines, conv_length);
put(total_machines, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);
when number_of_states =>
  set_col(20);
  put(scm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) =~ ' ' loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
  get(token, total_states, conv_length);
  put(total_states, width => 2);
  line_index := 1;
  token_index := 1;
  clear_token(token);

when machine =>
  set_col(20);
  put(scm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) =~ ' ' loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
  get(token, current_machine, conv_length);
  put(current_machine, width => 2);
  line_index := 1;
  token_index := 1;
  clear_token(token);

when state =>
  set_col(20);
  put(scm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) =~ ' ' loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= ' ' loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
get(token, current_state, conv_length);
put(current_state, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when trans =>
  set_col(20);
  put(scm_instruction, set => lower_case);
  put("*");
  while line_index < line_length + 1 and then line_buffer(line_index) = " " loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= " " loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
trans_enumio.get(token(1 .. token_index - 1), current_transition, enum_length);
put(current_transition,set =>lower_case);
token_index := 1;
while line_index < line_length + 1 and then line_buffer(line_index) = " " loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= " " loop
  token(token_index) := line_buffer(line_index);
  token_index := token_index + 1;
  line_index := line_index + 1;
end loop;
get(token, next_state, conv_length);
put(next_state, width => 2);
token_index := 1;
load_machine_array(current_machine, current_state, next_state, current_transition);

when initial state =>
  set_col(20);
  put(scm_instruction, set => lower_case);
  while line_index < line_length + 1 and then line_buffer(line_index) = " " loop
    line_index := line_index + 1;
  end loop;
  while line_index < line_length + 1 and then line_buffer(line_index) /= " " loop
    token(token_index) := line_buffer(line_index);
    token_index := token_index + 1;
    line_index := line_index + 1;
  end loop;
get(token, start_state 1, conv_length);
put(start_state 1, width => 2);
token_index := 1;
while line_index < line_length + 1 and then line_buffer(line_index) = " " loop
  line_index := line_index + 1;
end loop;
while line_index < line_length + 1 and then line_buffer(line_index) /= " " loop
  token(token_index) := line_buffer(line_index);
token_index := token_index + 1;
line_index := line_index + 1;
end loop;
get(token, start_state 2, conv_length);
put(start_state 2, width => 2);
line_index := 1;
token_index := 1;
clear_token(token);

when finish =>
set col(20);
put(scm_instruction, set => lower_case);
new line(2);
put_line("machines are loaded!!!");
exit;

when others =>
put_line("error---error---error");
for x in 1 .. line_length loop
put(line_buffer(x));
end loop;
end case;
new line;
end loop;
end read in_file;
generic

  type item is private;
  MAX : in integer := 10;

package queues is

  type ArrayList is array(1 .. MAX) of item;
  type Queue is
    record
      Store : ArrayList;
      tail : integer range 0 .. MAX := 0;
    end record;

  procedure MakeEmpty(Q : in out Queue);
  procedure Enqueue(Q : in out Queue; E : in Item);
  procedure Dequeue(Q : in out Queue);
  function First(Q : Queue) return Item;
  function IsEmpty(Q : Queue) return boolean;
  function IsEqual(Q1 : Queue; Q2 : Queue) return boolean;

  QueueFull : exception;
  QueueEmpty : exception;
end queues;

package body queues is

  procedure MakeEmpty(Q : in out queue) is
    blank line : string(1 .. 80) := (others => ' ');
  begin
    Q.tail := 0;
    end MakeEmpty;

  procedure Enqueue(Q : in out Queue; E : in Item) is
    begin
      ...
if Q.tail = MAX then
    raise QueueFull;
else
    Q.tail := Q.tail + 1;
    Q.Store(Q.tail) := E;
end if;
end Enqueue;

procedure Dequeue(Q : in out Queue) is
begin
    if Q.tail = 0 then
        raise QueueEmpty;
    else
        for i in Q.tail .. 2 loop
            Q.Store(i - 1) := Q.Store(i);
        end loop;
        Q.tail := Q.tail - 1;
    end if;
end Dequeue;

function First(Q : Queue) return Item is
begin
    if Q.tail = 0 then
        raise QueueEmpty;
    else
        return Q.Store(1);
    end if;
end First;

function IsEmpty(Q : Queue) return boolean is
begin
    return Q.tail = 0;
end IsEmpty;

function IsEqual(Q1 : Queue; Q2 : Queue) return boolean is
begin
    for i in 1 .. Q1.tail loop
        if (Q1.Store(i) /= Q2.Store(i)) then
            return false;
        end if;
    end loop;
    return true;
end IsEqual;

end queues;
generic
  type Item is private;
  MAX : in integer := 10;

package stacks is
  type ArrayList is array(1 .. MAX) of Item;
  type Stack is
  record
    Store : ArrayList;
    latest : integer range 0 .. MAX := 0;
  end record;

  procedure MakeEmpty(S : in out Stack);
  function isEmpty(S : Stack) return boolean;
  procedure Push(S : in out Stack; E : in Item);
  procedure Pop(S : in out Stack; E : out Item);
  function Top(S : Stack) return Item;

  StackFull : exception;
  StackEmpty : exception;
end stacks;

package body stacks is
  procedure MakeEmpty(S : in out Stack) is
    begin
      S.latest := 0;
      end MakeEmpty;

  procedure Push(S : in out Stack; E : in Item) is
    begin
      if S.latest = MAX then
        raise StackFull;
      else

S.lapest := S.lapest + 1;
S.Store(S.lapest) := E;
end if;
end Push;

procedure Pop(S : in out Stack; E : out Item) is
begin
  if S.lapest = 0 then
    raise StackEmpty;
  else
    E := S.Store(S.lapest);
    S.lapest := S.lapest - 1;
  end if;
end Pop;

function Top(S : Stack) return Item is
begin
  if S.lapest = 0 then
    raise StackEmpty;
  else
    return S.Store(S.lapest);
  end if;
end Top;

function IsEmpty(S : Stack) return boolean is
begin
  return S.lapest = 0;
end IsEmpty;

end stacks;
procedure build_Gstate_graph(machine array : in out system_array_type;
   start_state_1 : in out natural;
   start_state_2 : in out natural;
   top_Gstate : in out Glink_type) is

parent_Gstate, current_Gstate, find_Gstate, temp_tuple_pointer : Glink_type := null;

Gstate_counter
M1_state, M2_state
next_state
row_holder1, position_holder1, row_holder2, position_holder2 : integer := 0;

natural; --machine 1 and machine 2 state holder
natural; --returned value from a transition
Gstate_record_type;

boolean;

boolean := false;


procedure Link_Gstate( trans : in out som_transition_type;  
current_Gstate : in out Glink_type;  
next_Gstate : in out Glink_type;  
new_one : in out boolean) is
begin
   if (current_Gstate.link1.Glink = null) then
      current_Gstate.link1.Gtransition := trans;
      current_Gstate.link1.Glink := next_Gstate;
      current_Gstate.link1.new_node := new_one;
   else
      if (current_Gstate.link2.Glink = null) then
         current_Gstate.link2.Gtransition := trans;
         current_Gstate.link2.Glink := next_Gstate;
         current_Gstate.link2.new_node := new_one;
      else
         if (current_Gstate.link3.Glink = null) then
            current_Gstate.link3.Gtransition := trans;
            current_Gstate.link3.Glink := next_Gstate;
            current_Gstate.link3.new_node := new_one;
         else
            if (current_Gstate.link4.Glink = null) then
               current_Gstate.link4.Gtransition := trans;
               current_Gstate.link4.Glink := next_Gstate;
               current_Gstate.link4.new_node := new_one;
            else
               put_line("All transition pointers used!!!");
               scroll_pause;
            end if;
         end if;
      end if;
   end if;
end Link_Gstate;
This procedure simply links the newly created global state node to the previous node in the graph.

procedure Insert_Gstate(temp : in out Gstate_record_type;
trans : in out sgm.transition_type;
current_Gstate : in out Glink_type;
counter : in out natural) is
end Insert_Gstate;

procedure initialize_top_global_state(ptr : in out Glink_type) is
end initialize_top_global_state;

begin
create(reach, mode -> out_file, name -> "file.dat");
temp_tuple_pointer : new global_state_type;
temp_tuple_pointer.system_state_number := Gstate_counter;
initialize_top_global_state(temp_tuple_pointer);
top_Gstate := Temp_tuple_pointer;
parent_Gstate := top_Gstate;
current_Gstate := parent_Gstate;
M1_state := start_state 1;
M2_state := start_state 2;
output Gtuple_to_file(top_Gstate.Gtuple, Gstate_counter);
new_line;
end main_loop: loop
--find all valid transitions in current row of machine array
  position_holder1 := machine_array[I] (M1_state);
  row_holder1 := position holder1;
current_Gstate := parent_Gstate;
--analyze all variables and determine which transitions to put on stack
transition_stack := Analyze_Predicates(Machine1(current_Gstate.Gtuple.machine1_state, current_Gstate.Gtuple.GLOBAL_VARIABLES);
while (not isEmpty(transition_stack)) loop
  --do the loop until the global transition stack is empty
  current_Gstate := parent_Gstate;
  position_holder1 := row_holder1;
  while (position_holder1 /= null) loop
    --analyze the current row of machine array until it reaches the null pointer
    if (position_holder1.transition = Top(transition_stack)) then
      --when a value is found then determine if it is a new node or not
      --if a node exists make a link in graph
      --if the node does not exist make a new node and add to graph
      toggle := true;
      position_holder1.visited := yes;
      top_Gstate := current_Gstate.Gtuple.machine1_state;
      temp_Gstate := current_Gstate.Gtuple.GLOBAL_VARIABLES;
      temp_Gstate.machine1_state := current_Gstate.Gtuple.machine1_state;
      temp_Gstate.GLOBAL_VARIABLES := current_Gstate.Gtuple.GLOBAL_VARIABLES;
      temp_Gstate.machine1_state.state_number := position_holder1.next_Mstate;
      temp_Gstate.machine2.state_number := M2_state;
      Action(current_Gstate.Gtuple, current_transition, temp_Gstate);
      find_Gstate := search for Gtuple(top_Gstate, temp_Gstate);
      if (find_Gstate = null) then
        Gstate_counter := Gstate_counter + 1;
        Insert_Gstate(temp_Gstate, current_transition, current_Gstate, Gstate_counter);
        output_Gtuple_to_file(current_Gstate.Gtuple, Gstate_counter);
        Enqueue(Gpointer_queue, current_Gstate);
      else
        is new := false;
        Link_Gstate(current_transition, current_Gstate, find_Gstate, is_new);
      end if;
      exit;
    else
      position_holder1 := position_holder1.next_Mstate;
    end if;
  end loop;
  if (not isEmpty(transition_stack) and (not toggle)) then
    Pop(transition_stack, current_transition);
  else
    toggle := false;
  end if;
end loop;
--find all valid transitions in current row of machine array 2
position_holder2 := machine_array(2).M2_state;
row_holder2 := position_holder2;
current_Gstate := parent_Gstate;
--analyze all variables and determine which transitions to put on stack
transition_stack := Analyze_Predicates(Machine2(current_Gstate.Gtuple.machine2_state, current_Gstate.Gtuple.GLOBAL_VARIABLES);
while (not isEmpty(transition stack)) loop
  position_holder2 := row_holder2;
  current_Gstate := parent_Gstate;
  -- analyze the current row of machine array until it reaches the null pointer
  while (position_holder2 /= null) loop
    if (position_holder2.transition = Top(transition stack)) then
      -- when a value is found then determine if it is a new node or not
      -- if a node exists make a link in graph
      -- if the node does not exist make a new node and add to graph
      toggle := true;
      position_holder2.visited := yes;
      Pop(transition stack, current_transition);
      temp_Gstate.machine1 state := current_Gstate.Gtuple.machine1 state;
      temp_Gstate.machine2 state := current_Gstate.Gtuple.machine2 state;
      temp_Gstate.GLOBAL VARIABLES := current_Gstate.Gtuple.GLOBAL VARIABLES;
      temp_Gstate.machine1 state.state_number := M1.state;
      temp_Gstate.machine2 state.state_number := position_holder2.next_Mstate;
      Action(current_Gstate.Gtuple, current_transition, temp_Gstate);
      find_Gstate := search for Gtuple(top_Gstate, temp_Gstate);
      if (find_Gstate = null) then
        Gstate_counter := Gstate_counter + 1;
        Insert_Gstate(temp Gstate, current_transition, current_Gstate, Gstate_counter);
        output Gtuple to file(current_Gstate.Gtuple, Gstate_counter);
        Enqueue(Gpointer_queue, current_Gstate);
      else
        is_new := false;
        Link_Gstate(current_transition, current_Gstate, find_Gstate, is_new);
        end if;
      exit;
    else
      position_holder2 := position_holder2.next_node;
    end if;
  end loop;
  if (not isEmpty(transition stack) and (not toggle)) then
    Pop(transition_stack, current_transition);
  else
    toggle := false;
  end if;
end loop;
if (isEmpty(Gpointer_queue)) then
  exit;
else
  parent_Gstate := First(Gpointer_queue);
  Dequeue(Gpointer_queue);
  M1.state := parent_Gstate.Gtuple.machine1.state.state_number;
  M2.state := parent_Gstate.Gtuple.machine2.state.state_number;
end if;
end loop main loop;
-- when the pointer queue is full an exception is raised to exit the
-- main loop exception
when Gpointer_queue_package.QueueFull =>
  put(" The pointer queue is full at state: ");
  put(current_Gstate.system.state_number, width => 1));
procedure clear_pointers is
begin
  initial_system := null;
  initial_global := null;
  for i in 1 .. 2 loop
    for j in 0 .. 10 loop
      machine_array(i) (j) := null;
    end loop;
  end loop;
end clear_pointers;
function search_for_Gtuple(top : in Glink_type; tuple : in Gstate_record_type) return Glink_type is
begin
  temp : Glink_type := null;
  if (top = null) then
    return null;
  elsif (tuple=top.Gtuple) then
    return top;
  else
    if ((temp = null) and then (top.link1.new_node = true)) then
      temp := search_for_Gtuple(top.link1.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link2.new_node = true)) then
      temp := search_for_Gtuple(top.link2.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link3.new_node = true)) then
      temp := search_for_Gtuple(top.link3.Glink, tuple);
    end if;
    if ((temp = null) and then (top.link4.new_node = true)) then
      temp := search_for_Gtuple(top.link4.Glink, tuple);
    end if;
    return temp;
  end if;
end search_for_Gtuple;
procedure clear_pause is
  keyboard : string(1 .. 11);
  len      : natural;
begind
  new line;
  get line(keyboard, len);
  put(ASCII.ESC);
  put_line("\n");
end clear_pause;

procedure scroll_pause is
  keyboard : string(1 .. 11);
  len      : natural;
begind
  new line;
  get_line(keyboard, len);
end scroll_pause;
procedures output_machine_arrays(machine_array : in out system_array_type) is
  place_holder : Slink_type;
  line_count : integer := 7;
begin
  new_line(2);
  set col(20);
  put("--------------------------------------------------")):
  set col (20);
  put("Machine 1 Array Contents */");
  set col(20);
  put("--------------------------------------------------"):
  set col(20);
  put("! From ! To ! Transition ! Executed !"):
  set col(20);
  put("--------------------------------------------------"):
  for curr_statel in 0 .. 10 loop
    place_holder := machine_array(1) (curr_statel);
    while (place_holder /= null) loop
      line_count := line_count + 1;
      if ((line_count mod 20) = 0) then
        clear_pause;
      end if;
      set col(20);
      put("" & integer'image(curr_statel));
      put("" & integer'image(place_holder.next_Mstate) "" & "");
      put(place_holder.transition, set => lower_case);
      set col(50);
      put("" & ";
      put(place_holder.visited, set => lower_case, width => 3);
      set col(61);
      put("" & ";
      new_line;
      place_holder := place_holder.slink;
    end loop;
  end loop;
set_col(20); put("--------------------------------------------------------------------"); new_line(2); set_col(20); put("--------------------------------------------------------------------"); set_col(20); put("*") Machine 2 Array Contents "); set_col(20); put("--------------------------------------------------------------------"); set_col(20); put("*"); From 1 To 1 Transition I Executed "); set_col(20); put("--------------------------------------------------------------------"); for curr_state2 in 0..10 loop
  place_holder := machine_array(2) (curr_state2);
  while (place_holder /= null) loop
    line_count := line_count + 1;
    if (line_count mod 20) = 0 then
      clear_pause;
    end if;
    set_col(20);
    put("""); integer_image(curr_state2));
    put("""); integer_image(place_holder.next.Mstate) & " I ");
    put(place_holder.transition, set => lower_case);
    set_col(50);
    put("*");
    put(place_holder.visited, set => lower_case, width => 3);
    set_col(64));
    put("*");
    new_line;
  place_holder := place_holder.Slink;
  end loop;
end loop;
set_col(20); put("--------------------------------------------------------------------"); new_line;
end output machine_arrays;
procedure output_Gstate_transition(Gstate_pointer : in out Glink_type) is
begin
  if (Gstate_pointer.link1.Glink /= null) then
    set_col(Gcolumn_set + 68);
    put(Gstate_pointer.link1.Gtransition, width => transit_len, set => lower_case);
    put(Gstate_pointer.link1.Glink.system_state_number, width => 4);
    new_line;
  end if;
  if (Gstate_pointer.link2.Glink /= null) then
    set_col(Gcolumn_set + 68);
    put(Gstate_pointer.link2.Gtransition, width => transit_len, set => lower_case);
    put(Gstate_pointer.link2.Glink.system_state_number, width => 4);
    new_line;
  end if;
  if (Gstate_pointer.link3.Glink /= null) then
    set_col(Gcolumn_set + 68);
    put(Gstate_pointer.link3.Gtransition, width => transit_len, set => lower_case);
    put(Gstate_pointer.link3.Glink.system_state_number, width => 4);
    new_line;
  end if;
  if (Gstate_pointer.link4.Glink /= null) then
    set_col(Gcolumn_set + 68);
    put(Gstate_pointer.link4.Gtransition, width => transit_len, set => lower_case);
    put(Gstate_pointer.link4.Glink.system_state_number, width => 4);
    new_line;
  end if;
end output_Gstate_transition;
```
procedure output_Gstates(top_Gstate : in out Glink_type) is
    Error_flag : boolean := true;
    current    : Glink_type := top_Gstate;
begin
    if (top_Gstate = null) then
        return;
    else
        output_Gstate_node(top_Gstate, Error_flag);
        if (not Error_flag) then
            output_Gstate_transition(top_Gstate);
        end if;
        if (top_Gstate.link1.new_node = true) then
            output_Gstates(top_Gstate.link1.Glink);
        end if;
        if (top_Gstate.link2.new_node = true) then
            output_Gstates(top_Gstate.link2.Glink);
        end if;
        if (top_Gstate.link3.new_node = true) then
            output_Gstates(top_Gstate.link3.Glink);
        end if;
        if (top_Gstate.link4.new_node = true) then
            output_Gstates(top_Gstate.link4.Glink);
        end if;
    end if;
end output_Gstates;
```
procedure output_analysis is
begin
output_machine_arrays(machine_array);
clear_pause;
build_Sstate_graph(machine_array, start_state_1, start_state_2, initial_global_state);
clear_pause;
new_line(2);
put_line(" Global State GRAPH ");
new_line;
output_line_count := 1;
output_Gstates(initial_global_state);
clear_pause;
-- new_line(2);
-- put_line(" WE ARE OUT"A HERE!!!!!!!!");
new_line(2);
put_line(" System State GRAPH ");
new_line;
output_line_count := 1;
--test;
build_Sstate_graph(initial_global_state, initial_system_state);
output_Sstates(initial_system_state);
new_line;
clear_pause;
output_machine_arrays(machine_array);
scroll_pause;
-- new_line(2);
-- put_line(reach, *)
Reachability Graph*);
-- output_Gstates_to_file(initial_global_state, number_of_nodes);
-- new_line(reach);
-- output_machine_arrays_to_file(machine_array);
end output_analysis;
procedure build_Sstate_graph(top_Sstate : in out Glink_type; top_Sstate : in out Syslink_type) is
parent_Sstate, current_Sstate, find_Sstate, temp_tuple_pointer : Syslink_type := null;
current_Sstate, Sstate_Counter, temporary_Slink, temp_Sstate, next_state, is_new, current_transition, hold_Srecord, loop_count, linkno, 

-- Title build_Sstate graph
-- File system_reachability.a
-- Author Matthew J. Rothlisberger, Capt., U. S. Army
-- Date 20 March 1992
-- Revised : Sun 4 Workstation
-- Compiler : Verlax ADA, Version 6.0
-- Description : This procedure builds the system reachability graph from the global reachability graph. Each
-- system state node can have up to 4 transitions. The algorithm for this procedure is as follows.
-- Based on the initial state of the global reachability graph an initial state is made in the system
-- reachability graph. Values are assigned to the system tuple and values are placed in the four
-- outgoing transitions. Whenever a temporary system state tuple is filled the machine states are
-- added as well as the outgoing transitions. The global state graph is traversed in a breadth first
-- manner. A temporary system state tuple is made and a check is done in the system state graph is
-- made to determine if the state already exits. If the state already exits a link is made from the
-- current system state to the found state. If the temporary state value is not found the temporary
-- node is linked into the graph. The appropriate link, system state, and global state are 'remembered'
-- with the use of queues to hold the values. A pointer is retrieved from the queue after a scan is
-- done while traversing the global state graph. This procedure terminates whenever there are no values
-- in the global pointer/system state/link queues. In other words, this procedure ends when each node
-- in the global graph has been visited and each transition from each of the global nodes has been
-- analyzed. As can be demonstrated, SCM system reachability graphs can grow without bound.
-- A max pointer queue size is introduced to eliminate unboundedness.

procedure Link_State(current_Strans : in out Sstate_transition_type; next_Sstate : in out Syslink_type) is
begin
    if (current_Strans.Syslink = null) then
        current_Strans.Syslink := next_Sstate;
        current_Strans.new_node := false;
    end if;
end Link_State;

--This procedure simply links the newly created system state node to the previous
--node in the graph.
procedure Insert_State(Shold : in out Syslink_type; current_Strans : in out Sstate_transition_type;
    counter : in out integer) is
begin
    temp_node.system.state_number := counter;
    temp_node.Stuple := Shold.Stuple;
    temp_node.link1 := Shold.link1;
    temp_node.link2 := Shold.link2;
    temp_node.link3 := Shold.link3;
    temp_node.link4 := Shold.link4;
    current_Strans.Syslink := temp_node;
    current_Strans.new_node := true;
end Insert_State;

--build the initial system state
begin
    current_State := new system.state_type;
    current_Sstate := top_Sstate;
    current_State.system.state_number := Sstate_counter;
end
current Staple.machine1_state := current_Gstate.Gtuple.machine1_state.state_number;
current Staple.machine2_state := current_Gstate.Gtuple.machine2_state.state_number;
top Sstate := current_Sstate;
-- save the value of f1 used in the glc.1 reducibility graph
-- for later analysis
if (current_Gstate.link1.Gtransition /= unused) then
  current_Gstate.link1.visited := true;
top_Sstate.link1.Stransition := current_Gstate.link1.Gtransition;
Enqueue[Sstate, current_Sstate];
Enqueue[link_queue, 1];
Enqueue[Gtrans_queue, current_Gstate.link1];
end if;
if (current_Gstate.link2.Gtransition /= unused) then
  current_Gstate.link2.visited := true;
top_Sstate.link2.Stransition := current_Gstate.link2.Gtransition;
Enqueue[Sstate, current_Sstate];
Enqueue[link_queue, 2];
Enqueue[Gtrans_queue, current_Gstate.link2];
end if;
if (current_Gstate.link3.Gtransition /= unused) then
current_Gstate.link3.visited := true;
Enqueue[Sstate, current_Sstate];
Enqueue[link_queue, 3];
Enqueue[Gtrans_queue, current_Gstate.link3];
end if;
if (current_Gstate.link4.Gtransition /= unused) then
current_Gstate.link4.visited := true;
Enqueue[Sstate, current_Sstate];
Enqueue[link_queue, 4];
Enqueue[Gtrans_queue, current_Gstate.link4];
end if;
begin
  -- begin analysis of states held in the event queues
  loop
    loop_count := loop_count + 1;
temp_Glink := First[Gtrans_queue];
current_Sstate := First[Sstate];
linkno := First[link_queue];
current_transition := temp_Glink.Gtransition;
current_Gstate := temp_Glink.Glink;
-- load the temporary record to be used in the search for a matching tuple
hold_Srecord.Stuple.machine1 state := current_Gstate.Gtuple.machine1_state.state_number;
hold_Srecord.Stuple.machine2 state := current_Gstate.Gtuple.machine2_state.state_number;
hold_Srecord.link1.new_node := true;
hold_Srecord.link2.Stransition := current_Gstate.link2.Gtransition;
hold_Srecord.link2.new_node := true;
hold_Srecord.link3.new_node := true;
hold_Srecord.link4.new_node := true;
end loop;
-- search for the tuple in the system reachability graph
find S_state := search for S_tuple(top_S_state, hold_Srecord);
if (find_S_state = null) then
    S_state_counter := S_state_counter + 1;
-- look at the correct link
    case linkno is
        when 1 =>
            Insert_S_state(hold_Srecord, current_S_state.link1, S_state_counter);
            current_S_state := current_S_state.link1.Syslink;
        when 2 =>
            Insert_S_state(hold_Srecord, current_S_state.link2, S_state_counter);
            current_S_state := current_S_state.link2.Syslink;
        when 3 =>
            Insert_S_state(hold_Srecord, current_S_state.link3, S_state_counter);
            current_S_state := current_S_state.link3.Syslink;
        when 4 =>
            Insert_S_state(hold_Srecord, current_S_state.link4, S_state_counter);
            current_S_state := current_S_state.link4.Syslink;
        when others =>
            put_line("what's up doc?");
            scroll_pause;
    end case;

-- enqueue global state transitions that have not been analyzed
if (current_G_state.link1.Glink /= null) and (current_G_state.link1.visited = false) then
    current_G_state.link1.visited := true;
    Enqueue(Gtrans_queue, current_G_state.link1);
    Enqueue(link_queue, 1);
    Enqueue(Sstate, current_S_state);
end if;
if (current_G_state.link2.Glink /= null) and (current_G_state.link2.visited = false) then
    current_G_state.link2.visited := true;
    Enqueue(Gtrans_queue, current_G_state.link2);
    Enqueue(link_queue, 2);
    Enqueue(Sstate, current_S_state);
end if;
if (current_G_state.link3.Glink /= null) and (current_G_state.link3.visited = false) then
    current_G_state.link3.visited := true;
    Enqueue(Gtrans_queue, current_G_state.link3);
    Enqueue(link_queue, 3);
    Enqueue(Sstate, current_S_state);
end if;
if (current_G_state.link4.Glink /= null) and (current_G_state.link4.visited = false) then
    current_G_state.link4.visited := true;
    Enqueue(Gtrans_queue, current_G_state.link4);
    Enqueue(link_queue, 4);
    Enqueue(Sstate, current_S_state);
end if;
else
--link the appropriate transition link

case linkage is
  when 1 =>
    Link_Sstate(current_Sstate.link1, find_Sstate);
    current_Sstate := current_Sstate.link1.Syslink;
  when 2 =>
    Link_Sstate(current_Sstate.link2, find_Sstate);
    current_Sstate := current_Sstate.link2.Syslink;
  when 3 =>
    Link_Sstate(current_Sstate.link3, find_Sstate);
    current_Sstate := current_Sstate.link3.Syslink;
  when 4 =>
    Link_Sstate(current_Sstate.link4, find_Sstate);
    current_Sstate := current_Sstate.link4.Syslink;
  when others =>
    put_line("whats up doc?");
    scroll pause;
end case;

--enqueue global transitions that have not been analyzed
if ((current_Gstate.link1.Glink /= null) and (current_Gstate.link1.visited = false)) then
  current_Gstate.link1.visited := true;
  Enqueue(Gtrans_queue, current_Gstate.link1);
  Enqueue(link queue, 1);
  Enqueue(Sstate, current_Sstate);
end if;

if ((current_Gstate.link2.Glink /= null) and (current_Gstate.link2.visited = false)) then
  current_Gstate.link2.visited := true;
  Enqueue(Gtrans_queue, current_Gstate.link2);
  Enqueue(link queue, 2);
  Enqueue(Sstate, current_Sstate);
end if;

if ((current_Gstate.link3.Glink /= null) and (current_Gstate.link3.visited = false)) then
  current_Gstate.link3.visited := true;
  Enqueue(Gtrans_queue, current_Gstate.link3);
  Enqueue(link queue, 3);
  Enqueue(Sstate, current_Sstate);
end if;

if ((current_Gstate.link4.Glink /= null) and (current_Gstate.link4.visited = false)) then
  current_Gstate.link4.visited := true;
  Enqueue(Gtrans_queue, current_Gstate.link4);
  Enqueue(link queue, 4);
  Enqueue(Sstate, current_Sstate);
end if;
end if;

--delete the top state from each queue, at this point they have been analyzed
Dequeue(Gtrans_queue);
Dequeue(link queue);
Dequeue(Sstate);
end loop;

--when the queues are empty the main loop will raise the following exception handler exception
  when Gtrans_stack_package.StackEmpty =>
    put_line("hi ho hi ho");
scroll_pause;
null;
when Qtrans_queue_package.QueueEmpty =>
null;
when others =>
put_line("error in the system reachability procedure");
gscroll_pause;
end;
put("Number of nodes in the System State Graph: ");
Sstate_counter := Sstate_counter + 1;
put(Sstate_counter);
new line;
scroll_pause;
end build_Sstate_graph;

----------------------------------------------
-- Title : IsSysStateEqual
-- File : system_reachability.a
-- Author : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date : 20 March 1992
-- Revised :
-- System : Sun 4 Workstation
-- Compiler : Verdix ADA, Version 6.0
-- Description : Compares two system state tuples for equality. Returns a boolean(true/false).
--- Equality of transitions is determined by the combined alphanumeric (ASCII)
--- value of all four transitions.
----------------------------------------------

function IsSysStateEqual(top : in Syslink_type; current : in system_state_type) return boolean is

top_temp : integer := 0;
current_temp : integer := 0;
str_len : integer;
str : string(1 .. 10);

function enum_value(InString : in string) return integer is
tot : integer := 0;
begin
for i in InString'range loop
  tot := tot + (character'pos(InString(i)));
end loop;
return tot;
end enum_value;

begin
if (top.Stuple = current.Stuple) then

trans_enum_io.put(str, top_link1.transition, set => lower_case);
top_temp := top_temp + enum_value(str);
trans_enum_io.put(str, top_link2.transition, set => lower_case);
top_temp := top_temp + enum_value(str);
trans_enum_io.put(str, top_link3.transition, set => lower_case);
top_temp := top_temp + enum_value(str);
trans_enum_io.put(str, top_link4.transition, set => lower_case);
top_temp := top_temp + enum_value(str);
trans_enum_io.put(str, current_link1.transition, set => lower_case);
current_temp := current_temp + enum_value(str);
trans_enum_io.put(str, current_link2.transition, set => lower_case);
current_temp := current_temp + enum_value(str);
trans_enum_io.put(str, current_link3.transition, set => lower_case);
current_temp := current_temp + enum_value(str);
trans_enum_io.put(str, current_link4.transition, set => lower_case);
current_temp := current_temp + enum_value(str);
if (top_temp = current_temp) then
    return true;
else
    return false;
end if;
else
    return false;
end if;
end isSysStateEqual;
separate(main)

-- Title : search for Stupe
-- File : system_search.a
-- Author : Matthew J. Rothlisberger, Captain, U. S. Army
-- Date : 20 March 1992
-- System : Sun 4 Workstation
-- Compiler : Verdiq ADA, Version 6.0
-- Description : A function that searches the reachability graph for a duplicated tuple. Returns a pointer
-- value of the found tuple location, else returns null.

function search_for_Stupe(top : in Syslink_type; current : in system_state_type) return Syslink_type is
  temp : Syslink_type := null;
  begin
  if (top = null) then
    return null;
  else
    if (IsSysStateEqual(top, current)) then
      return top;
    else
      if ((temp = null) and then (top.link1.new_node = true)) then
        temp := search_for_Stupe(top.link1.Syslink, current);
      end if;
      if ((temp = null) and then (top.link2.new_node = true)) then
        temp := search_for_Stupe(top.link2.Syslink, current);
      end if;
      if ((temp = null) and then (top.link3.new_node = true)) then
        temp := search_for_Stupe(top.link3.Syslink, current);
      end if;
      if ((temp = null) and then (top.link4.new_node = true)) then
        temp := search_for_Stupe(top.link4.Syslink, current);
      end if;
      return temp;
    end if;
  end if;
end search_for_Stupe;
procedure output_Stuple(tuple : in out Sstate_record_type) is
begin
   put(" | " & integer'image(tuple.machine1_state) & " | ");
   put(integer'image(tuple.machine2_state) & " | ");
end output_Stuple;

procedure output_Sstate_node(Sstate pointer : in out Syslink_type; Error_flag : in out boolean) is
begin
   output_line.count := output_line.count + 1;
   if (output_line.count mod 10) = 0 then
      scroll_pause;
   end if;
   Scolunm_set := 10;
   set_col(Scolumn_set);
   put(Sstate pointer.system state number, width => 3);
   output_Stuple(Sstate pointer.Stuple);
   if ((Sstate pointer.link1.Syslink = null) and then (Sstate pointer.link2.Syslink = null)) and then
      (Sstate pointer.link3.Syslink = null) and then (Sstate pointer.link4.Syslink = null) then
      Error_flag := true;
      set_col(Scolumn set + 19);
      put("***** DEADLOCK STATE ****");
   else
      Error_flag := false;
   end if;
end output_Sstate_node;
procedure output_State_transition (State_pointer: in out SysLink_type) is
begin
  if (State_pointer.link.1.Transition /= unused) then
    put(State_pointer.link.1.Transition, width => transition_len, set => lower_case);
  end if;

  if (State_pointer.link.2.Transition /= unused) then
    put(State_pointer.link.2.Transition, width => transition_len, set => lower_case);
  end if;

  if (State_pointer.link.3.Transition /= unused) then
    put(State_pointer.link.3.Transition, width => transition_len, set => lower_case);
  end if;

  if (State_pointer.link.4.Transition /= unused) then
    put(State_pointer.link.4.Transition, width => transition_len, set => lower_case);
  end if;

  end if;  
end output_State_transition;
procedure output_Sstates(top_Sstate : in out Syslink_type) is
  Error_flag : boolean := true;
  current : Syslink_type := top_Sstate;
begin
  if (top_Sstate = null) then
    return;
  else
    output_Sstate_node(top_Sstate, Error_flag);
    if (not Error_flag) then
      output_Sstate_transition(top_Sstate);
      if ((top_Sstate.link1.Syslink /= null) and then (top_Sstate.link1.new_node = true)) then
        output_Sstates(top_Sstate.link1.Syslink);
      end if;
      if ((top_Sstate.link2.Syslink /= null) and then (top_Sstate.link2.new_node = true)) then
        output_Sstates(top_Sstate.link2.Syslink);
      end if;
      if ((top_Sstate.link3.Syslink /= null) and then (top_Sstate.link3.new_node = true)) then
        output_Sstates(top_Sstate.link3.Syslink);
      end if;
      if ((top_Sstate.link4.Syslink /= null) and then (top_Sstate.link4.new_node = true)) then
        output_Sstates(top_Sstate.link4.Syslink);
      end if;
    end if;
  end output_Sstates;
APPENDIX C (CFSM) ALTERNATING BIT

INPUT (FSM)

```
start
machine 1 --------------MACHINE 1------------
state 1
trans -X 2
state 2
trans +A 3
state 3
trans -Y 4
state 4
trans +B 1
machine 2 --------------MACHINE 2------------
state 1
trans +X 2
state 2
trans -A 3
state 3
trans +Y 4
state 4
trans -B 1
initial state 1 1
finish
```
OUTPUT

REACHABILITY ANALYSIS of: output.alt

Reachability Graph

| 1 [1, E, E, 1] | -X [2, X, E, 1] | 2 |
| 2 [2, X, E, 1] | +X [2, E, E, 2] | 3 |
| 7 [4, E, E, 4] | -B [4, E, B, 1] | 8 |
| 8 [4, E, B, 1] | +B [1, E, E, 1] | 1 |

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

* The nodes generated by the analysis were done in a breadth first manner.
APPENDIX D (CFSM) SLIDING WINDOW

INPUT(FSM)

start
machine 1
state 1
trans -X 2
state 2
trans +B 7
trans -Y 3
state 3
trans +B 8
trans +C 4
state 4
trans -Z 5
state 5
trans -X 6
trans +A 1
state 6
trans +A 2
trans +B 7
state 7
trans -Y 8
state 8
trans -Z 9
state 9
trans +C 5
trans +A 1
machine 2
state 1
trans +X 2
state 2
trans +Y 3
state 3
trans -C 4
state 4
trans +Z 5
state 5
trans +X 6
state 6
trans +B 7
state 7
trans +Y 8
state 8
trans +Z 9
state 9
trans -A 1
initial_state 1
finish
OUTPUT

REACHABILITY ANALYSIS of: sliding_window

---

Reachability Graph

<table>
<thead>
<tr>
<th>Node</th>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, E, E, 1</td>
<td>2, X, E, 1</td>
<td>-X</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>2, X, E, 1</td>
<td>3, Y, E, 1</td>
<td>-X</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>3, X, E, 1</td>
<td>4, E, E, 1</td>
<td>+X</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>4, X, E, 1</td>
<td>5, Y, E, 1</td>
<td>+Y</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>5, Y, E, 1</td>
<td>6, E, E, 2</td>
<td>+C</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>6, E, E, 2</td>
<td>7, E, E, 4</td>
<td>-Z</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>7, E, E, 4</td>
<td>8, X, E, 4</td>
<td>+X</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>8, X, E, 4</td>
<td>9, Y, E, 4</td>
<td>+Y</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>9, Y, E, 4</td>
<td>10, E, E, 4</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>10, E, E, 4</td>
<td>11, E, E, 4</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>11</td>
<td>11, E, E, 4</td>
<td>12, E, E, 4</td>
<td>+Z</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>12, E, E, 4</td>
<td>13, E, E, 4</td>
<td>+B</td>
<td>yes</td>
</tr>
<tr>
<td>13</td>
<td>13, E, E, 4</td>
<td>14, E, E, 4</td>
<td>+B</td>
<td>yes</td>
</tr>
<tr>
<td>14</td>
<td>14, E, E, 4</td>
<td>15, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>15</td>
<td>15, E, E, 4</td>
<td>16, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>16</td>
<td>16, E, E, 4</td>
<td>17, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>17</td>
<td>17, E, E, 4</td>
<td>18, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>18</td>
<td>18, E, E, 4</td>
<td>19, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>19</td>
<td>19, E, E, 4</td>
<td>20, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>20</td>
<td>20, E, E, 4</td>
<td>21, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>21</td>
<td>21, E, E, 4</td>
<td>22, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>22</td>
<td>22, E, E, 4</td>
<td>23, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>23</td>
<td>23, E, E, 4</td>
<td>24, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>24</td>
<td>24, E, E, 4</td>
<td>25, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>25</td>
<td>25, E, E, 4</td>
<td>26, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
<tr>
<td>26</td>
<td>26, E, E, 4</td>
<td>27, E, E, 4</td>
<td>-Y</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

Machine 1 Array Contents

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>snd X</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>rcv B</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>snd Y</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>rcv B</td>
<td>no</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>snd Z</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>rcv A</td>
<td>no</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>rcv A</td>
<td>no</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>snd X</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>rcv A</td>
<td>no</td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>12</td>
<td>7</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>14</td>
<td>9</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>15</td>
<td>10</td>
<td>rcv A</td>
<td>yes</td>
</tr>
</tbody>
</table>

---

Machine 2 Array Contents

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>rcv X</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>rcv Y</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>snd C</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>rcv Z</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>rcv X</td>
<td>yes</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>snd B</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>rcv Y</td>
<td>yes</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>rcv C</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>snd A</td>
<td>yes</td>
</tr>
</tbody>
</table>

* The nodes generated by the analysis were done in a breadth-first manner
APPENDIX E (SCM) GO_BACK_N, W=1

INPUT (FSM)

start
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack 0 0
machine 2
state 0
trans rcv_data 1
state 1
trans snd_ack 0
initial_state 0 0
finish
package definitions is

type scm_transition_type is (snd_datal, rcv_datal, snd_ackl, rcv_ackl, adv_win1, unused);
type buffer_type is (dl, el, al);
type boolean_type is (t, f);
type buffer_array_type is array(1..1) of buffer_type;
type boolean_array_type is array(1..1) of boolean_type;
type machine1_state_type is
    record
        state_number : natural := 0;
        out_buffer : buffer_array_type := (others=>dl);
        ack_rec : boolean_array_type := (others=>f);
        current : integer range 1..1 := 1;
    end record;
type machine2_state_type is
    record
        state_number : natural := 0;
        in_buffer : buffer_array_type := (others=>el);
        pkt_rec : boolean_array_type := (others=>f);
        current : integer range 1..1 := 1;
    end record;
type global_variable_type is
    record
        DATA : buffer_type := e;
    end record;
end definitions;
PREDICATE-ACTION

separate (main)
function Analyze_Predicates_Machine1(local : machine1_state_type;
GLOBAL: global_variable_type) return transition_stack is
begin
MakeEmpty(transition_stack);
if (local.out_buffer(l) /= E) then
Push(transition_stack, snd_datal);
end if;
if ((local.ack_rec(l)=f) and GLOBAL.DATA=Al) then
Push(transition_stack, rcc_ack1);
end if;
Push(transition_stack, adv_winl);
return transition_stack;
end Analyze_Predicates_Machine1;

separate (main)
function Analyze_Predicates_Machine2(local : machine2_state_type;
GLOBAL: global_variable_type) return transition_stack is
begin
MakeEmpty(transition_stack);
if ((GLOBAL.DATA = Al) and (local.pkt_rec(l)=t)) then
Push(transition_stack, rcv_datal);
end if;
if (local.pkt_rec(l)=t) then
Push(transition_stack, snd_ack1);
end if;
return transition_stack;
end Analyze_Predicates_Machine2;

separate (main)
procedure Action(in system_state : in out Gstate_record_type;
in transition : in out scm_transition_type;
out system_state : in out Gstate_record_type) is
begin
  case (in transition) is
  when (snd_datal) =>
    out_system_state.GLOBAL_VARIABLES.DATA :=
    in_system_state.machine1_state.out_buffer(l);
  when (rcc_ack1) =>
    out_system_state.machine1_state.ack_rec(l) := t;
  when (rcv_datal) =>
    out_system_state.machine2_state.in_buffer(l) :=
    in_system_state.GLOBAL_VARIABLES.DATA;
  when (snd_ack1) =>
    out_system_state.GLOBAL_VARIABLES.DATA := al;
  when others =>
    put_line("There is an error in the Action procedure");
  end case;
end Action;
OUTPUT FORMAT

separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  put(" integer'image(tuple.machine1_state.state_number)");
  put(" integer'image(tuple.machine2_state.state_number)");
  put(");
  put(tuple.machine1_state.out_buffer(1), width => 1);
  put(");
  put(tuple.machine1_state.ack_rec(1),width=>2);
  put(");
  put(tuple.machine2_state.in_buffer(1), width => 1);
  put(");
  put(tuple.machine2_state.pkt_rec(1),width=>2);
  put(");
  put(tuple.GLOBAL_VARIABLES.DATA, width =>2);
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tuple : in out Gstate_record_type;
counter : in out integer) is
begin
  put(reach,counter);
  put(reach," integer'image(tuple.machine1_state.state_number)");
  put(reach," integer'image(tuple.machine2_state.state_number)");
  put(reach,");
  put(reach,tuple.machine1_state.ack_rec(1),width=>2);
  put(reach,");
  put(reach,tuple.machine2_state.in_buffer(1), width => 1);
  put(reach,");
  put(reach,tuple.machine2_state.pkt_rec(1) ,width=>2);
  put(reach,");
  put(reach,tuple.GLOBAL_VARIABLES.DATA, width =>2);
  put(reach,");
new_line(reach);
end output_Gtuple_to_file;

separate (main)
procedure output_Gstate_node(Gstate_pointer : in out Glink_type;
Error_flag : in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if ((output_line_count mod 10) = 0) then
    scroll_pause;
  end if;
  set_col(Gcolumn_set);
  put(Gstate_pointer.system_state_number, width => 3);
  output_Gtuple(Gstate_pointer.Gtuple);
  if ((Gstate_pointer.link1.Glink = null) and then (Gstate_pointer.link2.Glink = null) and then (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null))
then
  Error_flag := true;
else
  Error_flag := false;
end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of : go_back_n_wl

Global State GRAPH

0  [ 0, 0, 0, 1, 0, 1, E, -1, -1 ]   snd_data  1
1  [ 1, 1, 0, 1, 1, 0, D, 0, -1 ]   rcv_data  2
2  [ 1, 0, 0, 1, 1, 1, E, -1, -1 ]   snd_ack  3
3  [ 1, 0, 0, 1, 1, 1, E, -1, -1 ]   rcv_ack  4
4  [ 0, 0, 1, 1, 1, 1, E, -1, -1 ]   snd_data  5
5  [ 1, 0, 0, 1, 1, 1, D, 1, -1 ]   rcv_data  6
6  [ 1, 1, 0, 0, 1, 1, E, -1, -1 ]   snd_ack  7
7  [ 1, 0, 0, 1, 0, 1, E, -1, 0 ]   rcv_ack  0

System State GRAPH

0  [ 0, 0 ]   snd_data  [ 1, 0 ]   1
1  [ 1, 0 ]   rcv_data  [ 1, 1 ]   2
2  [ 1, 1 ]   snd_ack  [ 1, 0 ]   3
3  [ 1, 0 ]   rcv_ack  [ 0, 0 ]   0

<table>
<thead>
<tr>
<th>Machine</th>
<th>Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
<td>To</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>From</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
</tbody>
</table>
APPENDIX F (SCM) GO_BACK_N, W=2

INPUT (FSM)

```plaintext
start
machine 1
state 0
trans snd_data 1
state 1
trans snd_data 2
trans rcv_ack0 0
state 2
trans rcv_ack0 0
trans rcv_ack1 1
machine 2
state 0
trans rcv_data 1
state 1
trans rcv_data 2
trans snd_ack 0
state 2
trans snd_ack 0
initial_state 0 0
finish
```
package definitions is

  type scm_transition_type is (snd data, rcv data, snd_ack, rcv_ack0, rcv_ack1, unused);
  type buffer_type is (d0, dl, e, a);
  type buffer_array_type is array(1..2) of buffer_type;
  type seq_array_type is array(1..2) of integer range -1..2;

  type machine1_state_type is record
    state_number : natural := 0;
    Sdata : buffer_array_type := (d0, dl);
    seq : integer range 0..2 := 0;
    i : integer range 1..2 := 1;
  end record;

  type machine2_state_type is record
    state_number : natural := 0;
    Rdata : buffer_type := a;
    exp : integer range 0..2 := 0;
    j : integer range 1..2 := 1;
  end record;

  type global_variable_type is record
    DATA : buffer_array_type := (e, e);
    SEQ : seq_array_type := (-1, -1);
    ACK : integer range -1..2 := -1;
  end record;

end definitions;
PREDICATE-ACTION

separate (main)
function Analyze_Predicates_Machinel(local : machine1_state_type;
GLOBAL: global_variable_type) return transition_stack

stack_package.stack is
    temp1 : integer := GLOBAL.ACK + 0;
    temp2 : integer := (GLOBAL.ACK + 1) mod 3;
begin
    MakeEmpty(transition_stack);
    if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
        Push(transition_stack,snd_data);
    end if;
    if ((temp1 = local.seq) and (GLOBAL.ACK = -1)) then
        Push(transition_stack,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK = -1)) then
        Push(transition_stack,rcv_ack1);
    end if;
    return transition_stack;
end Analyze_Predicates_Machinel;

separate (main)
function Analyze_Predicates_Machine2(local : machine2_state_type;
GLOBAL: global_variable_type) return transition_stack

begin
    MakeEmpty(transition_stack);
    if ((GLOBAL.DATA(local.j) = E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(transition_stack,rcv_data);
    end if;
    if (GLOBAL.DATA(local.j) = E) then
        Push(transition_stack,snd_ack);
    end if;
    return transition_stack;
end Analyze_Predicates_Machine2;

separate (main)
procedure Action(in system_state : in out Gstate_record_type;
in transition : in out scm_transition_type;
out system_state : in out Gstate_record_type) is
    temp : integer := 0;
begin
    case (in_transition) is
        when (snd_data) =>
            out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine1_state.i) :=
            in_system_state.machine1_state.sdata(in_system_state.machine1_state.i);
            out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machine1_state.i) :=
            in_system_state.machine1_state.seq;
            if (in_system_state.machine1_state.i = 1) then
                out_system_state.machine1_state.i := 2;
            else
                out_system_state.machine1_state.i := 1;
            end if;
            out_system_state.machine1_state.seq := (((in_system_state.machine1_state.
            seq) + 1) mod 3);
        when (rcv_ack0) =>
            out_system_state.GLOBAL_VARIABLES.ACK := -1;
        when (rcv_ack1) =>
            out_system_state.GLOBAL_VARIABLES.ACK := -1;
        when (snd_ack) =>
            out_system_state.GLOBAL_VARIABLES.ACK := in_system_state.machine2_state.
            exp;
        when (rcv_data) =>
            out_system_state.machine2_state.Rdata := in_system_state.GLOBAL_VARIABLES.
            DATA(in_system_state.machine2_state.j);
            out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j) :=
            E;
            out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machine2_state.j) :=
            -1;
    end if;
    if (in_system_state.machine2_state.j = 1) then
        out_system_state.machine2_state.j := 2;
    end if;
end case;
end Action;
else
    out_system_state.machine2_state.j := 1;
end if;

out_system_state.machine2_state.exp :=
    (((in_system_state.machine2_state.exp) + 1) mod 3);
when others =>
    put_line("There is an error in the Action procedure");
end case;
end Action;
procedure output_GTuple (tuple : in out Gstate_record_type) is
begin
  put("" & integer' image (tuple.machine1_state.state_number) & ",");
  put(integer' image (tuple.machine2_state.state_number));
  put("",");
  put(tuple.machine1_state.seq, width => 1);
  put("",");
  put(tuple.machine1_state.i, width => 1);
  put("",");
  put(tuple.machine2_state.exp, width => 1);
  put("",");
  put(tuple.machine2_state.j, width => 1);
  put("",");
  put (tuple.GLOBAL_VARIABLES.DATA(1), width => 3);
  put (tuple.GLOBAL_VARIABLES.SEQ(1), width => 2);
  put("",");
  put (tuple.GLOBAL_VARIABLES.DATA(2), width => 3);
  put (tuple.GLOBAL_VARIABLES.SEQ(2), width => 2);
  put("",");
  put (tuple.GLOBAL_VARIABLES.ACK, width => 3);
  put("I");
end output_GTuple;

procedure output_GTuple_to_file(tupie : in out Gstate_record_type;
  counter : in out integer) is
begin
  put(reach,counter);
  put(reach,"" & integer' image (tuple.machine1_state.state_number) & ",");
  put(reach,integer' image (tuple.machine2_state.state_number));
  put(reach,"",");
  put(reach,tuple.machine1_state.seq, width => 1);
  put(reach,"",");
  put(reach,tuple.machine1_state.i, width => 1);
  put(reach,"",");
  put(reach,tuple.machine2_state.exp, width => 1);
  put(reach,"",");
  put(reach,tuple.machine2_state.j, width => 1);
  put(reach,"",");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(1), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(1), width => 2);
  put(reach,"",");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(2), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(2), width => 2);
  put(reach,"",");
  put(reach,tuple.GLOBAL_VARIABLES.ACK, width => 3);
  put(reach,"",");
  new_line(reach);
end output_GTuple_to_file;

procedure output_Gstate_node(Gstate_pointer : in out Glink_type;
  Error_flag : in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if ((output_line_count mod 10) = "0") then
    scroll_pause;  
  end if;
  set_col (Gcolumn_set);
  put(Gstate_pointer.system_state_number, width => 3);
  output_GTuple(Gstate_pointer.Gtuple);
  if ((Gstate_pointer.link1.Glink = null) and then (Gstate_pointer.link2.Glink = null) and then (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null)) then
    Error_flag := true;
  else
    Error_flag := false;
  end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of: go_back_n_w2

Global State GRAPH

186
System State GRAPH

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>[0,0]</td>
<td>snd_data</td>
<td>[1,0]</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>[1,0]</td>
<td>snd_data</td>
<td>[2,0]</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>[2,0]</td>
<td>rcv_data</td>
<td>[2,1]</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>[2,1]</td>
<td>rcv_data</td>
<td>[2,2]</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>[2,2]</td>
<td>snd_ack</td>
<td>[2,0]</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>[2,0]</td>
<td>rcv_ack0</td>
<td>[0,0]</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>[1,1]</td>
<td>snd_data</td>
<td>[2,1]</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>[2,0]</td>
<td>rcv_ack1</td>
<td>[1,0]</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>[2,1]</td>
<td>rcv_data</td>
<td>[2,1]</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>[1,0]</td>
<td>rcv_ack1</td>
<td>[0,0]</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>[1,0]</td>
<td>rcv_data</td>
<td>[2,0]</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>[2,1]</td>
<td>rcv_ack1</td>
<td>[1,1]</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>[2,0]</td>
<td>snd_ack</td>
<td>[2,0]</td>
<td>8</td>
</tr>
</tbody>
</table>

Machine 1 Array Contents

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>snd_data</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>snd_data</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>rcv_ack0</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>rcv_ack0</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>rcv_ack1</td>
<td>yes</td>
</tr>
</tbody>
</table>

Machine 2 Array Contents

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>rcv_data</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>rcv_data</td>
<td>yes</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>snd_ack</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>snd_ack</td>
<td>yes</td>
</tr>
</tbody>
</table>
APPENDIX G (SCM) GO_BACK_N, W=3

INPUT (FSM)

start
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack0 0
trans snd_data 2
state 2
trans rcv_ack0 0
trans rcv_ack1 1
trans snd_data 3
state 3
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
machine 2
state 0
trans rcv_data 1
state 1
trans rcv_data 2
trans snd_ack 0
state 2
trans rcv_data 3
trans snd_ack 0
state 3
trans snd_ack 0
initial_state 0 0
finish
package definitions is

  type scm_transition_type is (snd_data, rcv_data,
                              snd_ack, rcv_ack0,
                              rcv_ack1, rcv_ack2, unused);

type buffer_type is (d0, d1, d2, e, a);
type buffer_array_type is array(1..3) of buffer_type;
type seq_array_type is array(1..3) of integer range -1..3;

type machine1_state_type is
  record
    state_number : natural := 0;
    Sdata          : buffer_array_type := (d0, d1, d2);
    seq           : integer range 0..3 := 0;
    l             : integer range 1..3 := 1;
  end record;

type machine2_state_type is
  record
    state_number : natural := 0;
    Rdata        : buffer_type := e;
    exp          : integer range 0..3 := 0;
    j             : integer range 1..3 := 1;
  end record;

type global_variable_type is
  record
    DATA          : buffer_array_type := (e, e, e);
    SEQ           : seq_array_type := (-1, -1, -1);
    ACK           : integer range -1..3 := -1;
  end record;

end definitions;
separate (main)
function Analyze_Predicates_Machine1(local : machine1_state_type;
    GLOBAL: global_Variable_type) return transition-
stack_package.stack is
    temp1 : integer := GLOBAL.ACK + 0;
    temp2 : integer := (GLOBAL.ACK + 1) mod 4;
    temp3 : integer := (GLOBAL.ACK + 2) mod 4;
begin
    MakeEmpty(transition_stack);
    if ((GLOBAL.DATA(local.i) = E) and
        (GLOBAL.SEQ(local.i) = -1)) then
        Push(transition_stack,snd_data);
    end if;
    if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(transition_stack,rcv_ack0);
    end if;
    if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(transition_stack,rcv_ack1);
    end if;
    if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
        Push(transition_stack,rcv_ack2);
    end if;
    return transition_stack;
end Analyze_Predicates_Machine1;

separate (main)
function Analyze_Predicates_Machine2(local : machine2_state_type;
    GLOBAL: global_Variable_type) return transition-
stack_package.stack is
begin
    MakeEmpty(transition_stack);
    if ((GLOBAL.DATA(local.j) = E) and (GLOBAL.SEQ(local.j) = local.exp)) then
        Push(transition_stack,rcv_data);
    end if;
    if (GLOBAL.DATA(local.j) = E) then
        Push(transition_stack,snd_ack);
    end if;
    return transition_stack;
end Analyze_Predicates_Machine2; -- this returned value is then checked against the
machine arrays
    -- to determine if indeed this transition can be
    taken

separate (main)
procedure Action(in system_state : in out Gstate_record_type;
    in transition : in out scm_transition_type;
    out system_state : in out Gstate_record_type) is
    temp : integer := 0;
begin
    case (in_transition) is
        when (snd_data) =>
            out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machinel_state.i) :=
                in_system_state.machinel_state.sdata(in_system_state.machinel_state.i);
            out_system_state.GLOBAL_VARIABLES.SEQ(in_system_state.machinel_state.i) :=
                in_system_state.machinel_state.seq;
            begin
                case (in_system_state.machinel_state.i) is
                    when 1 =>
                        out_system_state.machinel_state.i := 2;
                    when 2 =>
                        out_system_state.machinel_state.i := 3;
                    when 3 =>
                        out_system_state.machinel_state.i := 1;
                    when others =>
                        null;
                end case;
            end;
        end;
        when (rcv_ack0) =>
            out_system_state.GLOBAL_VARIABLES.ACK := -1;
    end case;
end;
when (recv_ack1) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (recv_ack2) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (snd_ack) =>
  out_system_state.GLOBAL_VARIABLES.ACK := in_system_state.machine2_state.exp;
  out_system_state.machine2_state.Rdata := e;
when (recv_data) =>
  out_system_state.machine2_state.Rdata := in_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j);
  out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j) := E;
  out_system_state.GLOBAL_VARIABLES.SEQ (in_system_state.machine2_state.j) := -1;
begin
  case (in_system_state.machine2_state.j) is
    when 1 =>
      out_system_state.machine2_state.j := 2;
    when 2 =>
      out_system_state.machine2_state.j := 3;
    when 3 =>
      out_system_state.machine2_state.j := 1;
    when others =>
      null;
  end case;
end;
out_system_state.machine2_state.exp := ((in_system_state.machine2_state.exp + 1) mod 4);
when others =>
  put_line("There is an error in the Action procedure");
end case;
end Action;
separate (main)
procedure output_Gtuple(tup1e : in out Gstate_record_type) is
begin
  put(" [" & integer' image(tup1e.machi1e1_state.state_number) & ",");
  put (integer' image(tuple.machine2_state.state_number));
  put(" ,");
  put(tuple.machine1_state.seq, width->1);
  put(" ,");
  put(tuple.machinel_state.i, width->1);
  put(" ,");
  put(tuple.machine2_state.exp, width->1);
  put(" ,");
  put(tuple.machinel_state.j, width->1);
  put(" ,");
  put(tuple.GLOBALVARIABLES.DATA(1), width->3);
  put(tuple.GLOBALVARIABLES.SEQ(1),width->2);
  put(" ,");
  put(tuple.GLOBALVARIABLES.DATA(2), width->3);
  put(tuple.GLOBALVARIABLES.SEQ(2),width->2);
  put(" ,");
  put(tuple.GLOBALVARIABLES.DATA(3), width->3);
  put(tuple.GLOBALVARIABLES.SEQ(3),width->2);
  put(" ,");
  put(tuple.GLOBALVARIABLES.ACK, width->3);
  put(" ]");
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tup1e : in out Gstate_record_type;
                                 counter : in out integer) is
begin
  put (reach, counter);
  put(reach," [" & integer' image(tup1e.machi1e1_state.state_number) & ",");
  put (reach, integer' image(tup1e.machine2_state.state_number));
  put(reach," ,");
  put(reach,tuple.machine1_state.seq, width->1);
  put(reach," ,");
  put(reach,tuple.machinel_state.i, width->1);
  put(reach," ,");
  put(reach,tuple.machine2_state.exp, width->1);
  put(reach," ,");
  put(reach,tuple.machinel_state.j, width->1);
  put(reach," ,");
  put(reach,tuple.GLOBALVARIABLES.DATA(1), width->3);
  put(reach,tuple.GLOBALVARIABLES.SEQ(1),width->2);
  put(reach," ,");
  put(reach,tuple.GLOBALVARIABLES.DATA(2), width->3);
  put(reach,tuple.GLOBALVARIABLES.SEQ(2),width->2);
  put(reach," ,");
  put(reach,tuple.GLOBALVARIABLES.DATA(3), width->3);
  put(reach,tuple.GLOBALVARIABLES.SEQ(3),width->2);
  put(reach," ,");
  put(reach,tuple.GLOBALVARIABLES.ACK, width->3);
  put(reach," ]");
  new_line(reach);
end output_Gtuple_to_file;

separate (main)
procedure output_Gstate_node(Gstate_pointer : in out Glink_type;
                             Error_flag : in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if ((output_line_count mod 10) ="0") then
    scroll_pause;
  end if;
  set_col(Gcolumn_set);
  put(Gstate_pointer.system_state.number, width->3);
  output_Gtuple(Gstate_pointer.Gtuple);
  if (Gstate_pointer.link1.Glink = null) and then (Gstate_pointer.link2.Glink = null)
and then (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null)
then
  192
Error_flag := true;
else
  Error_flag := false;
end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of: qo_back_n_w3

Global State GRAPH

<p>| 0 [ 0, 0, 0, 1, 0, 1, E -1, E -1, E -1, -1 ] | snd_data 1 |
| 1 [ 1, 0, 1, 2, 0, 1, DO 0, E -1, E -1, -1 ] | snd_data 4 |
| 2 [ 2, 0, 2, 3, 0, 1, DO 0, D1 1, E -1, -1 ] | rcv_data 3 |
| 4 [ 3, 0, 3, 1, 0, 1, DO 0, D1 1, D2 2, -1 ] | rcv_data 4 |
| 7 [ 3, 1, 3, 1, 2, E -1, D1 1, D2 2, -1 ] | rcv_data 5 |
| 11 [ 3, 2, 3, 1, 2, 3, E -1, E -1, D2 2, -1 ] | rcv_data 6 |
| 16 [ 3, 3, 3, 1, 3, 1, E -1, E -1, E -1, 3 ] | rcv_data 11 |
| 22 [ 3, 0, 3, 1, 1, 1, E -1, E -1, E -1, 3 ] | snd_ack 22 |
| 29 [ 0, 0, 1, 3, 3, 3, E -1, E -1, E -1, -1 ] | snd_data 35 |
| 35 [ 1, 0, 0, 2, 3, 1, DO 3, E -1, E -1, -1 ] | snd_data 43 |
| 43 [ 2, 0, 1, 3, 3, 3, DO 3, D1 0, E -1, -1 ] | rcv_data 44 |
| 51 [ 3, 0, 2, 1, 3, 3, DO 3, D1 0, D2 1, -1 ] | rcv_data 52 |
| 61 [ 3, 1, 2, 1, 0, 2, E -1, D1 0, D2 1, -1 ] | rcv_data 61 |
| 68 [ 3, 2, 2, 1, 1, 3, E -1, E -1, D2 1, -1 ] | snd_data 76 |
| 76 [ 3, 3, 2, 1, 1, 1, E -1, E -1, E -1, -1 ] | snd_ack 82 |
| 82 [ 3, 0, 2, 1, 2, 1, E -1, E -1, E -1, 2 ] | rcv_data 89 |
| 89 [ 0, 0, 2, 1, 2, 1, E -1, E -1, E -1, -1 ] | snd_data 93 |
| 95 [ 1, 0, 3, 2, 2, 1, DO 2, E -1, E -1, -1 ] | snd_data 103 |
| 103 [ 2, 0, 0, 3, 2, 1, DO 2, D1 3, E -1, -1 ] | rcv_data 104 |
| 111 [ 3, 0, 1, 1, 2, 1, DO 2, D1 3, D2 0, -1 ] | rcv_data 111 |
| 121 [ 3, 1, 1, 1, 2, 1, D1 3, D2 0, -1 ] | rcv_data 121 |
| 128 [ 3, 2, 1, 1, 0, 3, E -1, E -1, D2 0, -1 ] | rcv_data 128 |
| 136 [ 3, 3, 1, 1, 1, 1, E -1, E -1, E -1, -1 ] | rcv_data 136 |
| 142 [ 3, 0, 1, 1, 1, 1, E -1, E -1, E -1, -1 ] | snd_data 142 |
| 149 [ 0, 0, 1, 1, 1, 1, E -1, E -1, E -1, -1 ] | rcv_data 149 |
| 155 [ 1, 0, 2, 2, 1, 1, DO 1, E -1, E -1, -1 ] | snd_data 155 |
| 163 [ 2, 0, 3, 3, 1, 1, DO 1, D1 2, E -1, -1 ] | rcv_data 163 |
| 171 [ 3, 0, 0, 1, 1, 1, DO 1, D1 2, D2 3, -1 ] | rcv_data 171 |
| 181 [ 3, 2, 0, 1, 1, 1, E -1, D1 2, D2 3, -1 ] | rcv_data 181 |
| 196 [ 3, 3, 0, 1, 0, 1, E -1, E -1, E -1, -1 ] | snd_ack 196 |
| 202 [ 3, 0, 0, 1, 0, 1, E -1, E -1, E -1, -1 ] | rcv_data 202 |
| 211 [ 2, 2, 3, 3, 3, 3, E -1, E -1, E -1, 3 ] | snd_data 199 |
| 212 [ 2, 3, 3, 3, 3, 3, E -1, E -1, E -1, 3 ] | change_ack 197 |
| 197 [ 0, 0, 3, 3, 3, 3, E -1, E -1, E -1, -1 ] | snd_data 198 |
| 203 [ 1, 0, 0, 1, 3, 3, E -1, E -1, D2 3, -1 ] | snd_data 203 |
| 209 [ 2, 0, 1, 2, 3, 3, DO 0, E -1, D2 3, -1 ] |snd_data 209 |
| 214 [ 3, 0, 2, 3, 3, 3, DO 0, D1 1, D2 3, -1 ] |rcv_data 214 |
| 221 [ 3, 1, 2, 3, 3, 3, DO 0, D1 1, E -1, -1 ] |rcv_data 221 |
| 227 [ 3, 2, 2, 3, 1, 2, E -1, D1 1, E -1, -1 ] |rcv_data 227 |
| 224 [ 3, 3, 2, 3, 1, 2, E -1, E -1, E -1, -1 ] | rcv_data 224 |
| 237 [ 3, 0, 2, 3, 3, 3, E -1, E -1, E -1, -1 ] |snd_ack 237 |
| 245 [ 2, 2, 1, 2, 2, 1, DO 0, E -1, E -1, -1 ] | snd_data 217 |
| 222 [ 2, 2, 1, 2, 1, 2, E -1, E -1, E -1, -1 ] | snd_data 227 |
| 228 [ 2, 0, 1, 2, 1, 2, E -1, E -1, E -1, 1 ] |snd_ack 228 |
| 235 [ 3, 0, 2, 3, 1, 2, E -1, D1 1, E -1, 1 ] | snd_data 235 |
| 238 [ 3, 1, 2, 3, 3, 3, E -1, E -1, E -1, 1 ] | rcv_data 238 |
| 240 [ 1, 1, 0, 1, 0, 1, E -1, E -1, E -1, -1 ] | snd_ack 216 |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>216</td>
<td>1, 0, 0, 1, 0, 1, E -1, E -1, E -1, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>219</td>
<td>3, 0, 1, 2, 3, D0 0, D1 0, D2 0, E -1, E -1, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>222</td>
<td>1, 1, 2, 2, 2, E -1, E -1, E -1, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>225</td>
<td>3, 0, 1, 3, 3, E -1, E -1, E -1, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>228</td>
<td>1, 1, 2, 2, 2, E -1, E -1, E -1, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

...
162 [ 2, 1, 1, 1, 1, 1, E -1, E -1, E -1, 0 ]
124 [ 2, 0, 0, 3, 3, 2, E -1, D1 3, E -1, 3 ]
131 [ 3, 0, 1, 1, 3, 2, E -1, D1 3, D2 0, 3 ]
141 [ 3, 1, 1, 1, 0, 3, E -1, E -1, D2 0, 3 ]
140 [ 3, 2, 1, 1, 1, E -1, E -1, E -1, 3 ]
132 [ 2, 1, 0, 3, 0, 3, E -1, E -1, E -1, 3 ]
52 [ 2, 1, 1, 3, 0, 2, E -1, D1 0, E -1, -1 ]
62 [ 2, 2, 1, 3, 1, 3, E -1, E -1, E -1, -1 ]
69 [ 2, 0, 1, 3, 1, 3, E -1, E -1, E -1, 1 ]
77 [ 0, 0, 1, 3, 1, 3, E -1, E -1, E -1, 1 ]
83 [ 1, 0, 2, 1, 1, 3, E -1, E -1, D2 1, -1 ]
90 [ 2, 0, 3, 2, 1, 3, D0 2, E -1, D2 1, -1 ]
96 [ 3, 0, 0, 3, 1, 3, D0 2, D1 3, D2 1, -1 ]
105 [ 3, 1, 0, 3, 2, 1, D0 2, D1 3, E -1, -1 ]
114 [ 3, 3, 0, 3, 3, 3, E -1, D1 3, E -1, -1 ]
125 [ 3, 3, 0, 3, 0, 3, E -1, E -1, E -1, -1 ]
133 [ 3, 0, 0, 3, 0, 3, E -1, E -1, E -1, 0 ]
97 [ 2, 1, 3, 2, 2, 1, D0 2, E -1, E -1, -1 ]
106 [ 2, 2, 3, 2, 3, 2, E -1, E -1, E -1, -1 ]
115 [ 2, 0, 3, 2, 3, 2, E -1, E -1, E -1, 3 ]
126 [ 3, 0, 0, 3, 3, 2, E -1, D1 3, E -1, 3 ]
134 [ 3, 1, 0, 3, 0, 3, E -1, E -1, E -1, 3 ]
91 [ 1, 1, 2, 1, 2, 1, E -1, E -1, E -1, -1 ]
98 [ 1, 0, 2, 1, 2, 1, E -1, E -1, E -1, 2 ]
107 [ 2, 0, 3, 2, 2, 1, D0 2, E -1, E -1, 2 ]
116 [ 3, 0, 0, 3, 2, 1, D0 2, D1 3, E -1, 2 ]
127 [ 3, 1, 0, 3, 3, 2, E -1, D1 3, E -1, 2 ]
135 [ 3, 2, 0, 3, 0, 3, E -1, E -1, E -1, 2 ]
117 [ 2, 1, 3, 2, 3, 2, E -1, E -1, E -1, 2 ]
78 [ 3, 0, 2, 1, 1, 3, E -1, E -1, D2 1, 1 ]
84 [ 3, 1, 2, 1, 2, 1, E -1, E -1, E -1, 1 ]
44 [ 1, 1, 0, 2, 0, 2, E -1, E -1, E -1, -1 ]
53 [ 1, 0, 0, 2, 0, 2, E -1, E -1, E -1, 0 ]
63 [ 0, 0, 0, 3, 0, 2, E -1, E -1, E -1, -1 ]
70 [ 1, 0, 1, 0, 2, 0, E -1, D1 0, E -1, -1 ]
79 [ 2, 0, 2, 1, 0, 2, E -1, D1 0, D2 1, -1 ]
85 [ 3, 0, 3, 2, 0, 2, D0 2, D1 0, D2 1, -1 ]
92 [ 3, 3, 3, 2, 1, 3, D0 2, E -1, D2 1, -1 ]
99 [ 3, 2, 3, 2, 2, 1, D0 2, E -1, E -1, -1 ]
108 [ 3, 3, 3, 2, 3, 2, E -1, E -1, E -1, -1 ]
118 [ 3, 0, 3, 2, 3, 2, E -1, E -1, E -1, 3 ]

snd_ack 178
rcv_ack 151
snd_data 170
snd_ack 160
rcv_ack 130
snd_data 131
rcv_data 132
rcv_ack 139
rcv_data 141
rcv_ack 145
rcv_data 146
rcv_ack 145
snd_ack 142
rcv_ack 140
snd_data 141
snd_ack 129
snd_data 61
rcv_data 62
snd_data 68
snd_ack 69
rcv_ack 77
snd_data 78
snd_data 69
rcv_data 97
rcv_data 105
rcv_data 114
crv_data 125
crv_data 133
crv_data 105
crv_data 106
rcv_data 114
crv_ack 115
crv_ack 123
snd_data 97
crv_ack 130
crv_data 134
crv_ack 140
snd_ack 133
crv_data 97
snd_ack 98
rcv_ack 89
snd_data 107
crv_ack 95
crv_data 116
crv_data 117
crv_ack 103
crv_data 127
crv_ack 112
crv_data 135
crv_ack 122
snd_ack 133
crv_ack 104
snd_data 127
snd_ack 115
crv_ack 83
crv_data 84
crv_ack 91
snd_ack 82
snd_data 52
snd_ack 53
crv_data 63
snd_data 64
snd_data 70
snd_data 79
crv_data 80
snd_data 85
rcv_data 86
rcv_data 92
rcv_data 99
rcv_data 108
snd_ack 118
rcv_ack 123
<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</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>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
APPENDIX H (SCM) GO_BACK_N, W=4

INPUT (FSM)

start
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack0 0
trans snd_data 2
state 2
trans rcv_ack0 0
trans rcv_ack1 1
trans snd_data 3
state 3
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans snd_data 4
state 4
trans rcv_ack0 0
trans rcv_ack1 1
trans rcv_ack2 2
trans rcv_ack3 3
machine 2
state 0
trans rcv_data 1
state 1
trans rcv_data 2
trans snd_ack 0
state 2
trans rcv_data 3
trans snd_ack 0
state 3
trans rcv_data 4
trans snd_ack 0
state 4
trans snd_ack 0
initial_state 0 0
finish
VARIABLE DEFINITIONS

package definitions is

type scm_transition_type is (snd_data, rcv_data, snd_ack, rcv_ack0, rcv_ack1, rcv_ack2, rcv_ack3, unused);

type buffer_type is (d0, dl, d2, d3, e, a);

type buffer_array_type is array(1..4) of buffer_type;

type seq_array_type is array(1..4) of integer range -1..4;

type machine1_state_type is
  record
    state_number : natural := 0;
    Sdata : buffer_array_type := (d0, dl, d2, d3);
    seq : integer range 0..4 := 0;
    i : integer range 1..4 := 1;
  end record;

type machine2_state_type is
  record
    state_number : natural := 0;
    Rdata : buffer_type := e;
    exp : integer range 0..4 := 0;
    j : integer range 1..4 := 1;
  end record;

type global_variable_type is
  record
    DATA : buffer_array_type := (e, e, e, e);
    SEQ : seq_array_type := (-1, -1, -1, -1);
    ACK : integer range -1..4 := -1;
  end record;

end definitions;
separate (main)
function Analyze_Predicates_Machine1(local : machinel_state_type;
GLOBAL: global_variable_type) return transition_stack

package stack is
  temp1 : integer := GLOBAL.ACK + 0;
temp2 : integer := (GLOBAL.ACK + 1)mod 5;
temp3 : integer := (GLOBAL.ACK + 2)mod 5;
temp4 : integer := (GLOBAL.ACK + 3)mod 5;
begin
  MakeEmpty(transition_stack);
  if ((GLOBAL.DATA(local.i) = E) and (GLOBAL.SEQ(local.i) = -1)) then
    Push(transition_stack,snd_data);
  end if;
  if ((temp1 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(transition_stack,rcv_ack0);
  end if;
  if ((temp2 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(transition_stack,rcv_ack1);
  end if;
  if ((temp3 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(transition_stack,rcv_ack2);
  end if;
  if ((temp4 = local.seq) and (GLOBAL.ACK /= -1)) then
    Push(transition_stack,rcv_ack3);
  end if;
  return transition_stack;
end Analyze_Predicates_Machine1;

separate (main)
function Analyze_Predicates_Machine2(local : machine2_state_type;
GLOBAL: global_variable_type) return transition_stack

begin
  MakeEmpty(transition_stack);
  if ((GLOBAL.DATA(local.j) = E) and (GLOBAL.SEQ(local.j) = local.exp)) then
    Push(transition_stack,rcv_data);
  end if;
  if (GLOBAL.DATA(local.j)=E) then
    Push(transition-stack~snd-ack);
  end if.
  return transition stack;
end Analyze_Predicates_Machine2;

separate (main)
procedure Action(in system state : in out Gstate_record_type;
in_transition: in out scm_transition_type;
out_system state : in out Gstate_record_type) is
  temp : integer := 0;
begin
  case (in_transition) is
    when (snd_data) =>
      out system state'.GLOBAL VARIABLES.DATA(in system state.machinel_state.i) :=
      in_system_state.machinel_state.sdata(in system state.machinel_state.i);
      out_system state.GLOBAL VARIABLES.SEQ(in system state.machinel_state.i) :=
      in_system_state.machinel_state.seq;
    begin
      case (in_system_state.machinel_state.i) is
        when 1 =>
          out_system state.machinel_state.i := 2;
        when 2 =>
          out_system state.machinel_state.i := 3;
        when 3 =>
          out_system state.machinel_state.i := 4;
        when 4 =>
          out_system state.machinel_state.i := 1;
        when others =>
          null;
      end case;
    end;
    out_system state.machinel_state.seq :=
      ((in_system_state.machinel_state.seq) + 1)mod 5);
end case;
end;
when (rcv_ack0) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (rcv_ack1) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (rcv_ack2) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (rcv_ack3) =>
  out_system_state.GLOBAL_VARIABLES.ACK := -1;
when (snd_ack) =>
  out_system_state.GLOBAL_VARIABLES.ACK := in_system_state.machine2_state.
  exp;
  out_system_state.machine2_state.Rdata := e;
when (rcv_data) =>
  out_system_state.machine2_state.Rdata :=
    in_system_state.GLOBAL_VARIABLES.
    DATA(in_system_state.machine2_state.j);
  out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machine2_state.j) := E;
  out_system_state.GLOBAL_VARIABLES.SEQ (in_system_state.machine2_state.j) :=
    -1;
begin
  case (in_system_state.machine2_state.j) is
    when 1 =>
      out_system_state.machine2_state.j := 2;
    when 2 =>
      out_system_state.machine2_state.j := 3;
    when 3 =>
      out_system_state.machine2_state.j := 4;
    when 4 =>
      out_system_state.machine2_state.j := 1;
    when others =>
      null;
  end case;
end:
out_system_state.machine2_state.exp := (((in_system_state.machine2_state.
  exp) + 1) mod 5);
when others =>
  put_line("There is an error in the Action procedure");
end case;
end Action;
OUTPUT FORMAT

separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  put("[" & integer'image(tuple.machine1_state.state_number) & ",";
  put(integer'image(tuple.machine2_state.state_number));
  put(".");
  put(tuple.machine1_state.seq, width => 1);
  put(".");
  put(tuple.machine1_state.i, width => 1);
  put(".");
  put(tuple.machine2_state.exp, width => 1);
  put(".");
  put(tuple.machine2_state.j, width => 1);
  put(".");
  put(tuple.GLOBAL_VARIABLES.DATA(1), width => 3);
  put(tuple.GLOBAL_VARIABLES.SEQ(1),width => 2);
  put(".");
  put(tuple.GLOBAL_VARIABLES.DATA(2), width => 3);
  put(tuple.GLOBAL_VARIABLES.SEQ(2),width => 2);
  put(".");
  put(tuple.GLOBAL_VARIABLES.DATA(3), width => 3);
  put(tuple.GLOBAL_VARIABLES.SEQ(3),width => 2);
  put(".");
  put(tuple.GLOBAL_VARIABLES.DATA(4), width => 3);
  put(tuple.GLOBAL_VARIABLES.SEQ(4),width => 2);
  put(".");
  put(tuple.GLOBAL_VARIABLES.ACK, width => 3);
  put("]");
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tuple : in out Gstate_record_type;
  counter : in out integer) is
begin
  put([reach," [" & integer'image(tuple.machine1_state.state_number) & ",";
  put(reach.integer'image(tuple.machine2_state.state_number));
  put(reach,".");
  put(reach,tuple.machine1_state.seq, width => 1);
  put(reach,".");
  put(reach,tuple.machine1_state.i, width => 1);
  put(reach,".");
  put(reach,tuple.machine2_state.exp, width => 1);
  put(reach,".");
  put(reach,tuple.machine2_state.j, width => 1);
  put(reach,".");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(1), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(1),width => 2);
  put(reach,".");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(2), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(2),width => 2);
  put(reach,".");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(3), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(3),width => 2);
  put(reach,".");
  put(reach,tuple.GLOBAL_VARIABLES.DATA(4), width => 3);
  put(reach,tuple.GLOBAL_VARIABLES.SEQ(4),width => 2);
  put(reach,".");
  put(reach,tuple.GLOBAL_VARIABLES.ACK, width => 3);
  put(reach,"]");
new_line(reach);
end output_Gtuple_to_file;

separate (main)
procedure output_Gstate_node(Gstate pointer : in out Glink_type;
  Error_flag : in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if (output_line_count mod 10) = 0 then
    scroll_pause;
  end if;
  set_col(Gstate_pointer.system_state_number, depth => 3);
  put(Gstate_pointer.system_state_number, depth => 3);
  output_Gtuple(Gstate_pointer.Gtuple);
  if ((Gstate_pointer.link1.Glink = null) and then (Gstate_pointer.link2.Glink = null)

205
and then
   (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null))
then
   Error_flag := true;
else
   Error_flag := false;
end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of: go_back_n_w4

Global State GRAPH

0 [ 0, 0, 0, 1, 0, 1, E -1, E -1, E -1, E -1, E -1] snd_data 1
1 [ 1, 0, 1, 2, 0, 1, D0 0, E 0, E -1, E -1, E -1] snd_data 2
2 [ 2, 0, 2, 3, 0, 1, D0 0, D1 1, E -1, E -1, E -1] snd_data 3
4 [ 3, 0, 4, 3, 1, 0, D0 0, D1 1, D2 2, E -1, E -1] snd_data 4
7 [ 4, 0, 4, 1, 0, 1, D0 0, D1 1, D2 2, D3 3] snd_data 5
12 [ 4, 1, 4, 1, 2, E -1, D1 1, D2 2, D3 3, E -1] snd_data 6
18 [ 4, 2, 4, 1, 2, 3, E -1, E -1, D2 2, D3 3, E -1] snd_data 7
26 [ 4, 3, 4, 1, 3, 4, E -1, E -1, D3 3, E -1] snd_data 8
36 [ 4, 4, 4, 1, 4, 4, E -1, E -1, E -1, E -1] snd_data 9
47 [ 4, 0, 4, 1, 4, 1, E -1, E -1, E -1, E -1] snd_data 10
60 [ 0, 0, 4, 1, 4, 1, E -1, E -1, E -1, E -1] snd_data 11
72 [ 2, 0, 0, 2, 1, 4, D0 4, E 4, E -1, E -1, E -1] snd_data 12
87 [ 2, 0, 1, 3, 3, 1, D0 4, D1 0, E 0, E 0, E -1] snd_data 13
102 [ 3, 0, 2, 4, 4, 4, D0 4, D1 0, D2 1, E -1, E -1] snd_data 14
120 [ 4, 0, 3, 1, 4, 1, D0 4, D1 0, D2 1, D3 2, E -1] snd_data 15
134 [ 4, 1, 3, 1, 3, 2, E -1, D1 0, D2 1, D3 2, E -1] snd_data 16
135 [ 4, 0, 3, 1, 3, 2, E -1, E -1, D2 1, D3 3, E -1] snd_data 17
162 [ 4, 3, 3, 2, 1, 4, E -1, E -1, E -1, D3 3, E -1] snd_data 18
176 [ 4, 4, 3, 1, 3, 1, E -1, E -1, E -1, E -1] snd_data 19
187 [ 4, 0, 3, 1, 3, 1, E -1, E -1, E -1, E -1, E -1] snd_data 20
212 [ 4, 0, 4, 2, 3, 1, D0 3, E 3, E -1, E -1] snd_data 21
227 [ 2, 0, 0, 3, 3, 1, D0 3, D1 4, E -1, E -1, E -1] snd_data 22
242 [ 3, 0, 1, 4, 3, 1, D0 3, D1 4, D2 0, E -1, E -1] snd_data 23
260 [ 4, 0, 2, 1, 3, 1, D0 3, D1 4, D2 0, D3 1, E -1] snd_data 24
274 [ 4, 1, 2, 1, 4, 2, E -1, D1 4, D2 0, D3 1, E -1] snd_data 25
290 [ 4, 2, 2, 1, 1, 3, E -1, E -1, D2 0, D3 1, E -1] snd_data 26
302 [ 4, 3, 2, 1, 1, 4, E -1, E -1, E -1, D3 1, E -1] snd_data 27
315 [ 4, 0, 3, 1, 1, 4, E -1, E -1, E -1, E -1] snd_data 28
327 [ 4, 0, 4, 1, 1, 4, E -1, E -1, E -1, E -1] snd_data 29
340 [ 4, 0, 4, 2, 1, 1, E -1, E -1, E -1, E -1] snd_data 30
352 [ 4, 0, 3, 2, 2, 1, D0 2, E 2, E -1, E -1] snd_data 31
367 [ 2, 0, 4, 3, 2, 1, D0 2, D1 3, E 1, E 1, E 1] snd_data 32
382 [ 3, 0, 0, 4, 2, 1, D0 2, D1 3, D2 4, E -1, E -1] snd_data 33
400 [ 4, 0, 1, 1, 2, 1, D0 2, D1 3, D2 4, D3 0, E -1] snd_data 34
414 [ 4, 1, 1, 1, 3, 2, E -1, D1 3, D2 4, D3 0, E -1] snd_data 35
430 [ 4, 2, 1, 1, 4, 3, E -1, E -1, D2 4, D3 0, E -1] snd_data 36
442 [ 4, 3, 1, 1, 0, 4, E -1, E -1, E -1, D3 0] snd_data 37
456 [ 4, 4, 1, 1, 1, 1, E -1, E -1, E -1, E -1] snd_data 38
467 [ 4, 0, 1, 1, 1, 1, E -1, E -1, E -1, E -1] snd_data 39
480 [ 4, 0, 1, 1, 1, 1, E -1, E -1, E -1, E -1] snd_data 40
492 [ 4, 0, 2, 2, 1, D0 1, E 1, E -1, E -1] snd_data 41
507 [ 2, 0, 3, 3, 1, 1, D0 1, D1 2, E -1, E -1] snd_data 42
522 [ 3, 0, 4, 4, 1, 1, D0 1, D1 2, D2 3, E -1, E -1] snd_data 43
540 [ 4, 0, 0, 1, 1, 1, D0 1, D1 2, D2 3, D3 4, E -1] snd_data 44
554 [ 4, 1, 0, 1, 2, 2, E -1, D1 2, D2 3, D3 4] snd_data 45
570 [ 4, 2, 0, 1, 3, 3, E -1, E -1, D2 3, D3 4] snd_data 46
582 [ 4, 3, 0, 1, 4, 4, E -1, E -1, E -1, D3 4] snd_data 47
596 [ 4, 4, 0, 0, 1, 1, E -1, E -1, E -1] snd_data 48
600 [ 4, 4, 0, 0, 1, 1, E -1, E -1, E -1, E -1] snd_data 49
541 [ 3, 1, 4, 4, 2, 2, E -1, D1 2, D2 3, E -1] snd_data 50
555 [ 3, 2, 4, 4, 3, 3, E -1, E -1, D2 3, E -1] snd_data 51
<table>
<thead>
<tr>
<th>Line</th>
<th>String</th>
<th>Start_position</th>
<th>End_position</th>
<th>Lines_contained</th>
</tr>
</thead>
<tbody>
<tr>
<td>573</td>
<td>[ 3, 0, 4, 4, 3, 3, E -1, E -1, D2 3, E -1, 3 ] snd_data 653</td>
<td>573</td>
<td>619</td>
<td>653</td>
</tr>
<tr>
<td>585</td>
<td>[ 4, 0, 0, 1, 3, 3, E -1, E -1, D2 3, D3 4, 3 ] rcv_ack1 584</td>
<td>585</td>
<td>631</td>
<td>584</td>
</tr>
<tr>
<td>601</td>
<td>[ 4, 1, 0, 1, 4, 4, E -1, E -1, E -1, D3 4, 3 ] rcv_data 611</td>
<td>601</td>
<td>647</td>
<td>611</td>
</tr>
<tr>
<td>613</td>
<td>[ 4, 2, 0, 1, 0, 1, E -1, E -1, E -1, E -1, 3 ] rcv_ack2 624</td>
<td>613</td>
<td>659</td>
<td>624</td>
</tr>
<tr>
<td>586</td>
<td>[ 3, 1, 4, 4, 4, 4, E -1, E -1, E -1, E -1, 3 ] rcv_ack1 600</td>
<td>586</td>
<td>632</td>
<td>600</td>
</tr>
<tr>
<td>508</td>
<td>[ 1, 1, 2, 2, 2, 2, E -1, E -1, E -1, E -1, 1 ] snd_data 523</td>
<td>508</td>
<td>554</td>
<td>523</td>
</tr>
<tr>
<td>524</td>
<td>[ 1, 0, 2, 2, 2, 2, E -1, E -1, E -1, E -1, 2 ] snd_data 524</td>
<td>524</td>
<td>570</td>
<td>524</td>
</tr>
<tr>
<td>543</td>
<td>[ 0, 0, 2, 2, 2, 2, E -1, E -1, E -1, E -1, -1, 1 ] snd_data 557</td>
<td>543</td>
<td>589</td>
<td>557</td>
</tr>
<tr>
<td>557</td>
<td>[ 1, 0, 3, 3, 2, 2, E -1, D1 2, E -1, E -1, -1 ] snd_data 574</td>
<td>557</td>
<td>603</td>
<td>574</td>
</tr>
<tr>
<td>574</td>
<td>[ 2, 0, 4, 4, 2, 2, E -1, D1 2, D2 3, E -1, -1 ] snd_data 587</td>
<td>574</td>
<td>620</td>
<td>587</td>
</tr>
<tr>
<td>587</td>
<td>[ 3, 0, 0, 1, 2, 2, E -1, D1 2, D2 3, D3 4, -1 ] snd_data 602</td>
<td>587</td>
<td>633</td>
<td>602</td>
</tr>
<tr>
<td>602</td>
<td>[ 4, 0, 1, 2, 2, 2, D0 0, D1 2, D2 3, D3 4, -1 ] snd_data 614</td>
<td>602</td>
<td>648</td>
<td>614</td>
</tr>
<tr>
<td>614</td>
<td>[ 4, 1, 1, 2, 3, 3, D0 0, E -1, E -1, D2 3, D3 4, -1 ] rcv_data 626</td>
<td>614</td>
<td>660</td>
<td>626</td>
</tr>
<tr>
<td>626</td>
<td>[ 4, 2, 1, 2, 4, 4, D0 0, E -1, E -1, E -1, D3 4, -1 ] rcv_data 639</td>
<td>626</td>
<td>672</td>
<td>639</td>
</tr>
<tr>
<td>639</td>
<td>[ 4, 3, 1, 1, 0, 1, D0 0, E -1, E -1, E -1, 1 ] snd_data 652</td>
<td>639</td>
<td>684</td>
<td>652</td>
</tr>
<tr>
<td>654</td>
<td>[ 4, 4, 1, 1, 1, 2, E -1, E -1, E -1, E -1, -1 ] snd_data 669</td>
<td>654</td>
<td>706</td>
<td>669</td>
</tr>
<tr>
<td>669</td>
<td>[ 4, 0, 1, 2, 1, 2, E -1, E -1, E -1, E -1, -1 ] rcv_data 10</td>
<td>669</td>
<td>718</td>
<td>10</td>
</tr>
<tr>
<td>603</td>
<td>[ 3, 1, 0, 1, 3, 3, E -1, E -1, D2 3, D3 4, -1 ] snd_data 614</td>
<td>603</td>
<td>655</td>
<td>614</td>
</tr>
<tr>
<td>615</td>
<td>[ 3, 2, 0, 1, 4, 4, E -1, E -1, E -1, D3 4, -1 ] snd_data 626</td>
<td>615</td>
<td>667</td>
<td>626</td>
</tr>
<tr>
<td>627</td>
<td>[ 3, 3, 0, 1, 0, 1, E -1, E -1, E -1, E -1, -1 ] snd_data 639</td>
<td>627</td>
<td>679</td>
<td>639</td>
</tr>
<tr>
<td>640</td>
<td>[ 3, 0, 0, 1, 2, 0, 1, E -1, E -1, E -1, E -1, 0 ] snd_ack 640</td>
<td>640</td>
<td>691</td>
<td>640</td>
</tr>
<tr>
<td>655</td>
<td>[ 4, 0, 1, 2, 0, 1, D0 0, E -1, E -1, E -1, 0 ] rcv_data 655</td>
<td>655</td>
<td>703</td>
<td>655</td>
</tr>
<tr>
<td>670</td>
<td>[ 4, 1, 1, 2, 1, 2, E -1, E -1, E -1, E -1, 0 ] rcv_data 670</td>
<td>670</td>
<td>715</td>
<td>670</td>
</tr>
<tr>
<td>588</td>
<td>[ 2, 1, 4, 4, 3, 3, E -1, E -1, D2 3, E -1, -1 ] snd_data 603</td>
<td>588</td>
<td>639</td>
<td>603</td>
</tr>
<tr>
<td>604</td>
<td>[ 2, 2, 4, 4, 4, 4, E -1, E -1, E -1, E -1, -1 ] snd_data 615</td>
<td>604</td>
<td>667</td>
<td>615</td>
</tr>
<tr>
<td>616</td>
<td>[ 2, 0, 4, 4, 4, 4, E -1, E -1, E -1, E -1, -1 ] rcv_data 616</td>
<td>616</td>
<td>679</td>
<td>616</td>
</tr>
<tr>
<td>628</td>
<td>[ 3, 0, 0, 1, 4, 4, E -1, E -1, E -1, D3 4, 4 ] snd_data 628</td>
<td>628</td>
<td>681</td>
<td>628</td>
</tr>
<tr>
<td>641</td>
<td>[ 4, 0, 1, 2, 4, 4, D0 0, E -1, E -1, D3 4, 4 ] snd_data 642</td>
<td>641</td>
<td>693</td>
<td>642</td>
</tr>
<tr>
<td>656</td>
<td>[ 4, 1, 1, 2, 0, 1, D0 0, E -1, E -1, E -1, 4 ] rcv_data 656</td>
<td>656</td>
<td>695</td>
<td>656</td>
</tr>
<tr>
<td>671</td>
<td>[ 4, 2, 1, 2, 1, 2, E -1, E -1, E -1, E -1, 4 ] rcv_data 671</td>
<td>671</td>
<td>699</td>
<td>671</td>
</tr>
<tr>
<td>642</td>
<td>[ 3, 1, 0, 1, 0, 1, E -1, E -1, E -1, E -1, 4 ] snd_data 669</td>
<td>642</td>
<td>695</td>
<td>669</td>
</tr>
<tr>
<td>575</td>
<td>[ 1, 1, 3, 3, 3, 3, E -1, E -1, E -1, E -1, -1 ] snd_data 588</td>
<td>575</td>
<td>617</td>
<td>588</td>
</tr>
<tr>
<td>589</td>
<td>[ 1, 0, 3, 3, 3, 3, E -1, E -1, E -1, E -1, -1 ] snd_data 598</td>
<td>589</td>
<td>621</td>
<td>598</td>
</tr>
<tr>
<td>605</td>
<td>[ 2, 0, 4, 4, 3, 3, E -1, E -1, D2 3, E -1, 3 ] rcv_data 605</td>
<td>605</td>
<td>633</td>
<td>605</td>
</tr>
<tr>
<td>617</td>
<td>[ 3, 0, 0, 1, 3, 3, E -1, E -1, D2 3, D3 4, 3 ] rcv_data 617</td>
<td>617</td>
<td>649</td>
<td>617</td>
</tr>
<tr>
<td>629</td>
<td>[ 4, 0, 1, 2, 3, 3, D0 0, E -1, D2 3, D3 4, 3 ] snd_data 629</td>
<td>629</td>
<td>655</td>
<td>629</td>
</tr>
<tr>
<td>643</td>
<td>[ 4, 1, 1, 2, 4, 4, D0 0, E -1, E -1, D3 4, 3 ] rcv_data 643</td>
<td>643</td>
<td>657</td>
<td>643</td>
</tr>
<tr>
<td>657</td>
<td>[ 4, 2, 1, 2, 0, 1, D0 0, E -1, E -1, E -1, 3 ] rcv_data 655</td>
<td>657</td>
<td>681</td>
<td>655</td>
</tr>
</tbody>
</table>
261

261
275  [ 3 , 2 , 1 , 4 , 0 , 3 , E -1 , E -1 , D2 0 , E -1 , -1 ] snd_data 290
rcv_data 291

291  [ 3 , 3 , 1 , 4 , 1 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_data 302

303  [ 3 , 0 , 1 , 4 , 1 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_ack 303
rcv_ack 317

317  [ 0 , 0 , 1 , 4 , 1 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_data 328

328  [ 1 , 0 , 2 , 1 , 4 , E -1 , E -1 , E -1 , D3 1 , -1 ] snd_data 341
rcv_data 342

341  [ 2 , 0 , 3 , 2 , 1 , 4 , D0 2 , E -1 , E -1 , D3 1 , -1 ] snd_data 353
rcv_data 354

353  [ 3 , 0 , 4 , 3 , 1 , 4 , D0 2 , D1 3 , E -1 , D3 1 , -1 ] snd_data 369
rcv_data 370

369  [ 4 , 0 , 0 , 4 , 1 , 4 , D0 2 , D1 3 , D2 4 , D3 1 , -1 ] rcv_data 385

385  [ 4 , 1 , 0 , 4 , 2 , 1 , D0 2 , D1 3 , D2 4 , E -1 , -1 ] rcv_data 405

405  [ 4 , 2 , 0 , 4 , 3 , 2 , E -1 , D1 3 , D2 4 , E -1 , -1 ] rcv_data 420

420  [ 4 , 3 , 0 , 4 , 4 , 3 , E -1 , E -1 , D2 4 , E -1 , -1 ] rcv_data 448

439  [ 4 , 4 , 0 , 4 , 0 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_ack 452

452  [ 4 , 0 , 0 , 4 , 0 , 4 , E -1 , E -1 , E -1 , E -1 , 0 ] rcv_ack 457

457  [ 3 , 1 , 4 , 3 , 1 , 4 , D0 2 , D1 3 , E -1 , E -1 , -1 ] snd_data 386
rcv_data 386

386  [ 3 , 2 , 4 , 3 , 3 , 2 , E -1 , D1 3 , E -1 , E -1 , -1 ] snd_data 405
rcv_data 406

406  [ 3 , 3 , 4 , 3 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_ack 421

421  [ 3 , 0 , 4 , 3 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , 4 ] rcv_ack 0 432

432  [ 4 , 0 , 0 , 4 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , -1 ] rcv_ack 0 439

439  [ 4 , 0 , 0 , 4 , 3 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] rcv_ack 0 444

444  [ 4 , 1 , 0 , 4 , 0 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] rcv_ack1 460

460  [ 2 , 1 , 3 , 2 , 2 , 1 , D0 2 , E -1 , E -1 , E -1 , -1 ] snd_data 370
rcv_data 371

371  [ 2 , 2 , 3 , 2 , 2 , 1 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_ack 386

386  [ 2 , 0 , 3 , 2 , 1 , 2 , E -1 , E -1 , E -1 , E -1 , 3 ] rcv_data 407

407  [ 3 , 0 , 4 , 3 , 3 , 2 , E -1 , D1 3 , E -1 , E -1 , -1 ] rcv_data 417
snd_data 407

422  [ 4 , 0 , 0 , 4 , 3 , 2 , E -1 , D1 3 , D2 4 , E -1 , 3 ] rcv_data 423

423  [ 4 , 0 , 0 , 4 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , -1 ] rcv_data 440

440  [ 4 , 1 , 0 , 4 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , 3 ] rcv_data 448

448  [ 4 , 2 , 0 , 4 , 4 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] rcv_data 454

454  [ 3 , 1 , 4 , 3 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_data 421

421  [ 1 , 1 , 2 , 1 , 2 , 1 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_data 354

354  [ 1 , 0 , 2 , 1 , 2 , 1 , E -1 , E -1 , E -1 , E -1 , 2 ] snd_data 355
rcv_data 355

355  [ 1 , 0 , 2 , 1 , 2 , 1 , E -1 , E -1 , E -1 , E -1 , 2 ] rcv_data 340
snd_data 372

372  [ 2 , 0 , 3 , 2 , 2 , 1 , D0 2 , E -1 , E -1 , E -1 , 2 ] rcv_data 352
snd_data 355
rcv_data 368

368  [ 3 , 0 , 4 , 3 , 2 , 1 , D0 2 , D1 3 , E -1 , E -1 , -1 ] rcv_data 367
snd_data 368

367  [ 4 , 0 , 0 , 4 , 2 , 1 , D0 2 , D1 3 , D2 4 , E -1 , -1 ] rcv_data 382

382  [ 4 , 1 , 0 , 4 , 3 , 2 , E -1 , D1 3 , D2 4 , E -1 , 2 ] rcv_data 401

401  [ 4 , 2 , 0 , 4 , 4 , 3 , E -1 , E -1 , D2 4 , E -1 , 2 ] rcv_data 415

415  [ 4 , 3 , 0 , 4 , 0 , 4 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_ack 383
rcv_data 415

383  [ 3 , 1 , 4 , 3 , 3 , 2 , E -1 , D1 3 , E -1 , E -1 , -1 ] snd_data 382
rcv_data 425

382  [ 3 , 2 , 4 , 3 , 4 , 3 , E -1 , E -1 , E -1 , E -1 , -1 ] snd_data 415
rcv_data 425

425  [ 2 , 1 , 3 , 2 , 3 , 2 , E -1 , E -1 , E -1 , E -1 , 2 ] rcv_data 438
snd_data 415

389  [ 3 , 0 , 4 , 3 , 2 , 1 , D0 2 , D1 3 , E -1 , E -1 , -1 ] snd_data 409

409  [ 2 , 0 , 3 , 2 , 1 , 2 , E -1 , E -1 , E -1 , E -1 , 2 ] rcv_data 421
snd_data 421

387  [ 2 , 2 , 3 , 2 , 1 , 4 , D0 2 , D1 3 , E -1 , E -1 , -1 ] rcv_data 386
<table>
<thead>
<tr>
<th>Line</th>
<th>Data</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td>[ 2, 1, 2, 1, 2, 1, E -1, E -1, E -1, E -1, E -1, 1 ]</td>
<td>snd ack 391</td>
</tr>
<tr>
<td>293</td>
<td>[ 3, 0, 1, 4, 0, 3, E -1, E -1, D2 0, E -1, 0 ]</td>
<td>rcv_ack 357</td>
</tr>
<tr>
<td>305</td>
<td>[ 4, 0, 2, 1, 0, 3, E -1, E -1, D2 0, D3 1, 0 ]</td>
<td>snd_data 305</td>
</tr>
<tr>
<td>321</td>
<td>[ 4, 1, 2, 1, 1, 4, E -1, E -1, E -1, D3 1, 0 ]</td>
<td>rcv_data 341</td>
</tr>
<tr>
<td>333</td>
<td>[ 4, 2, 2, 1, 2, 1, E -1, E -1, E -1, E -1, 0 ]</td>
<td>rcv_ack 345</td>
</tr>
<tr>
<td>306</td>
<td>[ 3, 1, 1, 4, 1, 4, E -1, E -1, E -1, E -1, 0 ]</td>
<td>rcv_ack 327</td>
</tr>
<tr>
<td>228</td>
<td>[ 1, 1, 4, 2, 4, 2, E -1, E -1, E -1, E -1, 1 ]</td>
<td>snd_data 321</td>
</tr>
<tr>
<td>244</td>
<td>[ 1, 0, 4, 2, 4, 2, E -1, E -1, E -1, E -1, 4 ]</td>
<td>rcv_ack 244</td>
</tr>
<tr>
<td>263</td>
<td>[ 0, 0, 4, 2, 4, 2, E -1, E -1, E -1, E -1, 0 ]</td>
<td>rcv_data 264</td>
</tr>
<tr>
<td>277</td>
<td>[ 1, 0, 0, 3, 4, 2, E -1, E -1, E -1, E -1, 1 ]</td>
<td>rcv_data 277</td>
</tr>
<tr>
<td>294</td>
<td>[ 2, 0, 1, 4, 4, 2, E -1, D1 4, D2 0, E -1, 1 ]</td>
<td>snd_data 307</td>
</tr>
<tr>
<td>307</td>
<td>[ 3, 0, 2, 1, 4, 2, E -1, D1 4, D2 0, D3 1, 0 ]</td>
<td>rcv_data 308</td>
</tr>
<tr>
<td>322</td>
<td>[ 4, 0, 3, 2, 4, 2, D0 2, D1 4, D2 0, D3 1, 0 ]</td>
<td>rcv_data 322</td>
</tr>
<tr>
<td>334</td>
<td>[ 4, 1, 3, 2, 0, 3, D0 2, E -1, D2 0, D3 1, 0 ]</td>
<td>rcv_data 334</td>
</tr>
<tr>
<td>347</td>
<td>[ 4, 2, 3, 2, 1, 4, D0 2, E -1, E -1, D3 1, 0 ]</td>
<td>rcv_data 347</td>
</tr>
<tr>
<td>361</td>
<td>[ 4, 3, 3, 2, 2, 1, D0 2, E -1, E -1, E -1, 1 ]</td>
<td>rcv_data 361</td>
</tr>
<tr>
<td>378</td>
<td>[ 4, 4, 3, 2, 2, 1, E -1, E -1, E -1, E -1, 0 ]</td>
<td>rcv_data 378</td>
</tr>
<tr>
<td>396</td>
<td>[ 4, 0, 3, 2, 1, 1, E -1, E -1, E -1, E -1, 0 ]</td>
<td>snd_ack 396</td>
</tr>
<tr>
<td>323</td>
<td>[ 3, 1, 2, 1, 0, 3, E -1, E -1, D2 0, D3 1, 0 ]</td>
<td>snd_data 323</td>
</tr>
<tr>
<td>335</td>
<td>[ 3, 2, 2, 1, 1, 4, E -1, E -1, E -1, D3 1, 0 ]</td>
<td>snd_data 335</td>
</tr>
<tr>
<td>348</td>
<td>[ 3, 3, 2, 1, 2, 1, E -1, E -1, E -1, E -1, 1 ]</td>
<td>snd_data 348</td>
</tr>
<tr>
<td>362</td>
<td>[ 3, 0, 2, 1, 2, 1, E -1, E -1, E -1, E -1, 1 ]</td>
<td>rcv_ack 362</td>
</tr>
<tr>
<td>379</td>
<td>[ 4, 0, 3, 2, 2, 1, D0 2, E -1, E -1, E -1, 1 ]</td>
<td>snd_data 379</td>
</tr>
<tr>
<td>397</td>
<td>[ 4, 1, 3, 2, 3, 2, E -1, E -1, E -1, E -1, 2 ]</td>
<td>rcv_data 397</td>
</tr>
<tr>
<td>308</td>
<td>[ 2, 1, 1, 4, 0, 3, E -1, E -1, D2 0, E -1, 0 ]</td>
<td>rcv_ack 308</td>
</tr>
<tr>
<td>324</td>
<td>[ 2, 2, 1, 4, 1, 4, E -1, E -1, E -1, E -1, 1 ]</td>
<td>rcv_data 324</td>
</tr>
<tr>
<td>336</td>
<td>[ 2, 0, 1, 4, 1, 4, E -1, E -1, E -1, E -1, 0 ]</td>
<td>snd_data 336</td>
</tr>
<tr>
<td>349</td>
<td>[ 3, 0, 2, 1, 1, 4, E -1, E -1, E -1, D3 1, 0 ]</td>
<td>rcv_ack 349</td>
</tr>
<tr>
<td>363</td>
<td>[ 4, 0, 3, 2, 1, 4, D0 2, E -1, E -1, D3 1, 1 ]</td>
<td>rcv_ack 180</td>
</tr>
<tr>
<td>380</td>
<td>[ 4, 1, 3, 2, 2, 1, D0 2, E -1, E -1, E -1, 1 ]</td>
<td>rcv_ack 380</td>
</tr>
<tr>
<td>398</td>
<td>[ 4, 2, 3, 2, 3, 2, E -1, E -1, E -1, E -1, 1 ]</td>
<td>rcv_ack 398</td>
</tr>
<tr>
<td>364</td>
<td>[ 3, 1, 2, 1, 2, 1, E -1, E -1, E -1, E -1, 1 ]</td>
<td>rcv_ack 364</td>
</tr>
<tr>
<td>295</td>
<td>[ 1, 1, 0, 3, 0, 3, E -1, E -1, E -1, E -1, 1 ]</td>
<td>snd_data 308</td>
</tr>
<tr>
<td>309</td>
<td>[ 1, 0, 0, 3, 0, 3, E -1, E -1, E -1, E -1, 0 ]</td>
<td>rcv_ack 309</td>
</tr>
<tr>
<td>325</td>
<td>[ 2, 0, 1, 4, 0, 3, E -1, E -1, D2 0, E -1, 0 ]</td>
<td>snd_data 325</td>
</tr>
<tr>
<td>337</td>
<td>[ 3, 0, 2, 1, 0, 3, E -1, E -1, D2 0, D3 1, 0 ]</td>
<td>rcv_ack 337</td>
</tr>
<tr>
<td>350</td>
<td>[ 4, 0, 3, 2, 0, 3, D0 2, E -1, D2 0, D3 1, 0 ]</td>
<td>rcv_ack 350</td>
</tr>
<tr>
<td>365</td>
<td>[ 4, 1, 3, 2, 1, 4, D0 2, E -1, E -1, D3 1, 0 ]</td>
<td>rcv_ack 355</td>
</tr>
</tbody>
</table>
381 | 4, 2, 3, 2, 2, 1, DO 2, E -1, E -1, E -1, 0 |
399 | 4, 3, 3, 2, 3, 2, E -1, E -1, E -1, 0 |
251 | 3, 1, 2, 1, 1, 4, E -1, E -1, E -1, OJ 1, 0 |
366 | 3, 2, 2, 1, 2, 1, E -1, E -1, E -1, 0 |
338 | 2, 1, 1, 4, 1, 4, E -1, E -1, E -1, 0 |
264 | 2, 0, 0, 3, 4, 2, E -1, D1 4, E -1, E -1, 4 |
278 | 3, 0, 1, 4, 4, 2, E -1, D1 4, D2 0, E -1, 4 |
296 | 4, 0, 2, 1, 4, 2, E -1, D1 4, D2 0, D3 1, 4 |
310 | 4, 1, 2, 1, 0, 3, E -1, E -1, D2 0, D3 1, 4 |
326 | 4, 2, 2, 1, 1, 4, E -1, E -1, E -1, D3 I, 4 |
339 | 4, 2, 2, 2, 2, 1, E -1, E -1, E -1, E -1, 4 |
297 | 3, 1, 1, 4, 0, 3, E -1, E -1, D2 0, E -1, 4 |
311 | 3, 2, 1, 4, 1, 4, E -1, E -1, E -1, E -1, 4 |
279 | 2, 1, 0, 3, 0, 3, E -1, E -1, E -1, E -1, 4 |
121 | 3, 1, 2, 4, 0, 2, E -1, D1 0, D2 1, E -1, -1 |
135 | 3, 2, 2, 4, 1, 3, E -1, E -1, D2 1, E -1, -1 |
151 | 3, 3, 2, 4, 2, 4, E -1, E -1, E -1, E -1, -1 |
163 | 3, 0, 2, 4, 2, 4, E -1, E -1, E -1, E -1, 2 |
177 | 0, 0, 2, 4, 2, 4, E -1, E -1, E -1, E -1, -1 |
188 | 1, 0, 3, 1, 2, 4, E -1, E -1, E -1, D3 2, -1 |
201 | 2, 0, 4, 2, 2, 4, D0 3, E -1, E -1, D3 2, -1 |
213 | 3, 0, 0, 3, 2, 4, DO 3, D1 4, E -1, D3 2, -1 |
229 | 4, 0, 1, 4, 2, 4, D0 3, D1 4, D2 0, D3 2, -1 |
245 | 4, 1, 1, 4, 3, 1, D0 3, D1 4, D2 0, E -1, -1 |
265 | 4, 2, 1, 4, 4, 2, E -1, E -1, D1 4, D2 0, E -1, -1 |
280 | 4, 3, 1, 4, 0, 3, E -1, E -1, D2 0, E E -1, -1 |
298 | 4, 4, 1, 4, 1, 4, E -1, E -1, E -1, E -1, -1 |
312 | 4, 0, 1, 4, 4, 2, E -1, E -1, E -1, E -1, 1 |
310 | 3, 1, 0, 3, 1, 1, D0 3, D1 4, E E -1, E -1, 1 |
246 | 3, 2, 0, 3, 4, 2, E -1, D3 4, E E -1, E -1, -1 |
266 | 3, 3, 0, 3, 0, 3, E -1, E -1, E -1, E -1, -1 |
281 | 3, 0, 0, 3, 3, 0, E -1, E -1, E -1, E -1, 0 |
299 | 4, 0, 1, 4, 0, 3, E -1, E -1, D2 0, E -1, 0 |
313 | 4, 1, 1, 4, 4, 4, E -1, E -1, E -1, E -1, 0 |
214 | 2, 1, 4, 2, 3, 1, DO 3, E -1, E -1, E -1, -1 |
231 | 2, 2, 4, 2, 4, 2, E -1, E -1, E -1, E -1, -1 |
247 | 2, 0, 4, 2, 4, 2, E -1, E -1, E -1, E -1, 4 |
<table>
<thead>
<tr>
<th>MAC Frame Number</th>
<th>Address</th>
<th>Control</th>
<th>DF Acknowledgment</th>
<th>DF Acknowledgment</th>
<th>DF Acknowledgment</th>
<th>DF Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>108</td>
<td>3, 0, 1, 3, 4, 1, 00 4, 3, 104 0</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
</tr>
<tr>
<td>109</td>
<td>2, 1, 0, 2, 0, 2</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
<td>E -1, 0, -1</td>
</tr>
</tbody>
</table>

**MAC Layer**

- **8**: SND DATA
- **13**: RCV DATA
- **19**: SND DATA
- **27**: RCV ACK0
- **37**: SND DATA
- **48**: SND DATA
- **61**: RCV DATA
- **73**: SND_DATA
- **89**: RCV_DATA
- **93**: SND DATA
- **106**: SND DATA
- **126**: SND DATA
- **141**: RCV ACK0
- **159**: SND ACK
- **173**: SND ACK
- **189**: SND ACK
- **210**: SND ACK
- **215**: SND ACK
- **248**: SND ACK
- **262**: SND ACK
- **285**: SND ACK
- **309**: SND ACK
- **332**: SND ACK
- **356**: SND ACK
- **379**: SND ACK
- **403**: SND ACK
- **426**: SND ACK
- **449**: SND ACK
- **472**: SND ACK
- **495**: SND ACK
- **518**: SND ACK
- **541**: SND ACK
- **564**: SND ACK
- **587**: SND ACK
- **610**: SND ACK
- **633**: SND ACK
- **656**: SND ACK
- **679**: SND ACK
- **702**: SND ACK
- **725**: SND ACK
- **748**: SND ACK
- **771**: SND ACK
- **794**: SND ACK
- **817**: SND ACK
- **840**: SND ACK
- **863**: SND ACK
- **886**: SND ACK

---

**MAC Header**

- **8**: [3, 1, 3, 4, 1, 2, E -1, D1 1, D2 2, E -1, -1]
- **13**: [3, 2, 3, 4, 2, 3, E -1, E -1, D2 2, E -1, -1]
- **19**: [3, 3, 3, 4, 3, 4, E -1, E -1, E -1, E -1, -1]
- **27**: [3, 0, 3, 4, 3, 4, E -1, E -1, E -1, E -1, 3]
- **37**: [0, 0, 3, 4, 3, 4, E -1, E -1, E -1, E -1, -1]
- **48**: [1, 0, 4, 1, 3, 4, E -1, E -1, E -1, D3 3, -1]
- **61**: [2, 0, 0, 2, 3, 4, D0 4, E -1, E -1, D3 3, -1]
- **73**: [3, 0, 1, 3, 3, 4, D0 4, D1 0, E -1, D3 3, -1]
- **89**: RCV DATA
- **105**: RCV DATA
- **125**: RCV DATA
- **140**: RCV DATA
- **158**: RCV DATA
- **172**: RCV DATA
- **188**: RCV DATA
- **204**: RCV DATA
- **217**: RCV DATA
- **233**: RCV DATA
- **248**: RCV DATA
- **262**: RCV DATA
- **275**: RCV DATA
- **289**: RCV DATA
- **303**: RCV DATA
- **317**: RCV DATA
- **331**: RCV DATA
- **345**: RCV DATA
- **358**: RCV DATA
- **372**: RCV DATA
- **386**: RCV DATA
- **400**: RCV DATA
- **414**: RCV DATA
- **428**: RCV DATA
- **442**: RCV DATA
- **456**: RCV DATA
- **470**: RCV DATA
- **484**: RCV DATA
- **498**: RCV DATA
- **512**: RCV DATA
- **526**: RCV DATA
- **540**: RCV DATA
- **554**: RCV DATA
- **568**: RCV DATA
- **582**: RCV DATA
- **596**: RCV DATA
- **610**: RCV DATA
- **624**: RCV DATA
- **638**: RCV DATA
- **652**: RCV DATA
- **666**: RCV DATA
- **680**: RCV DATA
- **694**: RCV DATA
- **708**: RCV DATA
- **722**: RCV DATA
- **736**: RCV DATA
- **750**: RCV DATA
- **764**: RCV DATA
- **778**: RCV DATA
- **792**: RCV DATA
- **806**: RCV DATA
- **820**: RCV DATA
- **834**: RCV DATA
- **848**: RCV DATA
- **862**: RCV DATA
- **876**: RCV DATA
- **890**: RCV DATA
- **904**: RCV DATA
- **918**: RCV DATA
- **932**: RCV DATA
- **946**: RCV DATA
- **960**: RCV DATA
- **974**: RCV DATA
- **988**: RCV DATA
- **1002**: RCV DATA
- **1016**: RCV DATA
<table>
<thead>
<tr>
<th>Number</th>
<th>Data or Ack</th>
<th>Data or Ack</th>
</tr>
</thead>
<tbody>
<tr>
<td>85</td>
<td>[4, 1, 0, 2, 3, 4, 0, 4, E -1, E -1, D3 3, 2]</td>
<td>rcv_data 85</td>
</tr>
<tr>
<td>101</td>
<td>[4, 2, 0, 2, 4, 1, 0, 4, E -1, E -1, E -1, 2]</td>
<td>rcv_data 101</td>
</tr>
<tr>
<td>119</td>
<td>[4, 3, 0, 2, 0, 2, E -1, E -1, E -1, E -1, 2]</td>
<td>rcv_data 119</td>
</tr>
<tr>
<td>71</td>
<td>[3, 1, 4, 1, 3, 4, E -1, E -1, E -1, D3 3, 2]</td>
<td>rcv_data 77</td>
</tr>
<tr>
<td>86</td>
<td>[3, 2, 4, 1, 4, 1, E -1, E -1, E -1, E -1, 2]</td>
<td>rcv_data 94</td>
</tr>
<tr>
<td>58</td>
<td>[2, 1, 3, 4, 3, 4, E -1, E -1, E -1, 2]</td>
<td>rcv_data 86</td>
</tr>
<tr>
<td>11</td>
<td>[2, 0, 2, 3, 1, 2, E -1, D1 1, E -1, E -1, 1]</td>
<td>rcv_data 85</td>
</tr>
<tr>
<td>16</td>
<td>[3, 0, 3, 4, 1, 2, E -1, D1 1, D2 2, E -1, 1]</td>
<td>rcv_data 116</td>
</tr>
<tr>
<td>24</td>
<td>[4, 0, 4, 1, 1, 2, E -1, D1 1, D2 2, D3 3, 1]</td>
<td>rcv_data 71</td>
</tr>
<tr>
<td>34</td>
<td>[4, 1, 4, 1, 2, 3, E -1, E -1, D2 2, D3 3, 1]</td>
<td>rcv_data 65</td>
</tr>
<tr>
<td>46</td>
<td>[4, 2, 4, 1, 3, 4, E -1, E -1, E -1, D3 3, 1]</td>
<td>rcv_data 101</td>
</tr>
<tr>
<td>59</td>
<td>[4, 3, 4, 1, 4, 1, E -1, E -1, E -1, 1]</td>
<td>rcv_data 40</td>
</tr>
<tr>
<td>25</td>
<td>[3, 1, 3, 4, 2, 3, E -1, E -1, D2 2, E -1, 1]</td>
<td>rcv_data 56</td>
</tr>
<tr>
<td>35</td>
<td>[3, 2, 3, 4, 3, 4, E -1, E -1, E -1, E -1, 1]</td>
<td>rcv_data 15</td>
</tr>
<tr>
<td>17</td>
<td>[2, 1, 2, 3, 2, 3, E -1, E -1, E -1, E -1, 1]</td>
<td>rcv_data 14</td>
</tr>
</tbody>
</table>
## System State Graph

<table>
<thead>
<tr>
<th>State</th>
<th>Sending Data</th>
<th>Receiving Data</th>
<th>Sending Acknowledgment</th>
<th>Receiving Acknowledgment</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 0</td>
<td>1, 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1, 0</td>
<td>2, 0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2, 0</td>
<td>1, 1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>3, 0</td>
<td>4, 0</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4, 0</td>
<td>3, 1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>4, 1</td>
<td>2, 1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>4, 2</td>
<td>4, 3</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>21</td>
<td>4, 3</td>
<td>4, 4</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>27</td>
<td>4, 4</td>
<td>0, 0</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>31</td>
<td>4, 0</td>
<td>0, 0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3, 1</td>
<td>4, 1</td>
<td>11</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>3, 2</td>
<td>3, 3</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>3, 3</td>
<td>4, 3</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>22</td>
<td>3, 0</td>
<td>4, 2</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>28</td>
<td>4, 0</td>
<td>1, 0</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>32</td>
<td>4, 1</td>
<td>4, 1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>2, 1</td>
<td>3, 1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>2, 2</td>
<td>3, 2</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>2, 0</td>
<td>0, 0</td>
<td>13</td>
<td>13</td>
</tr>
<tr>
<td>18</td>
<td>3, 0</td>
<td>1, 0</td>
<td>18</td>
<td>18</td>
</tr>
<tr>
<td>23</td>
<td>4, 0</td>
<td>3, 1</td>
<td>23</td>
<td>23</td>
</tr>
<tr>
<td>29</td>
<td>4, 1</td>
<td>2, 1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>33</td>
<td>4, 2</td>
<td>2, 2</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>24</td>
<td>3, 1</td>
<td>1, 1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1, 1</td>
<td>2, 1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>1, 0</td>
<td>0, 0</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>10</td>
<td>2, 0</td>
<td>1, 0</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>3, 0</td>
<td>2, 0</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>19</td>
<td>4, 0</td>
<td>3, 0</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>25</td>
<td>4, 1</td>
<td>3, 1</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>4, 2</td>
<td>3, 2</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>34</td>
<td>4, 3</td>
<td>3, 3</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>20</td>
<td>3, 1</td>
<td>2, 1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>26</td>
<td>3, 2</td>
<td>3, 2</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>2, 1</td>
<td>1, 1</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

226
<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</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>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
APPENDIX I (SCM) SELECTIVE REPEAT, W=1

INPUT

start
machine 1
state 0
tran snd_data 1
state 1
tran rcv_ack 2
state 2
tran adv_win 0
machine 2
state 0
tran rcv_data 1
state 1
tran snd_ack 0
initial_state 0 0
finish
package definitions is

  type scm_transition_type is (snd_data, rcv_data, snd_ack, rcv_ack, adv_win, unused);
  type buffer_type is (dl, e, al);
  type boolean_type is (t, f);
  type buffer_array_type is array(1..1) of buffer_type;
  type boolean_array_type is array(1..1) of boolean_type;

  type machine1_state_type is
    record
      state_number : natural := 0;
      out_buffer : buffer_array_type := (others=>dl);
      ack_rec : boolean_array_type := (others=>f);
      current : integer range 1..1 := 1;
    end record;

  type machine2_state_type is
    record
      state_number : natural := 0;
      in_buffer : buffer_array_type := (others=>e);
      pkt_rec : boolean_array_type := (others=>f);
      current : integer range 1..1 := 1;
    end record;

  type global_variable_type is
    record
      DATA : buffer_type := e;
    end record;

end definitions;
separate (main) function Analyze_Predicates_Machine1(local : machinel_state_type; GLOBAL: global_variable_type) return transition_stack is begin
  MakeEmpty(transition_stack);
  if (local.out_buffer(1) /= E) then
    Push(transition_stack,snd_data1);
  end if;
  if ((local.ack_rec(1)=f) and GLOBAL.DATA=A1) then
    Push(transition_stack,rcv_ack1);
  end if;
  Push(transition_stack,adv_win1);
  return transition_stack;
end Analyze_Predicates_Machine1;

separate (main) function Analyze_Predicates_Machine2(local : machine2_state_type; GLOBAL: global_variable_type) return transition_stack is begin
  MakeEmpty(transition_stack);
  if (GLOBAL.DATA = DI) and (local.pkt_rec(1)=t) then
    Push(transition_stack,rcv_data1);
  end if;
  if (local.pkt_rec(1)=t) then
    Push(transition_stack,snd_ack1);
  end if;
  return transition_stack;
end Analyze_Predicates_Machine2; -- this returned value is then checked against the machine arrays
-- to determine if indeed this transition can be taken

separate (main) procedure Action(in_system_state : in out Gstate_record_type;
  in_transition : in out scm_transition_type;
  out_system_state : in out Gstate_record_type) is
  temp : integer := 0;
begin
  case (in_transition) is
    when (snd_data1) =>
      out_system_state.GLOBAL_VARIABLES.DATA :=
      in_system_state.machinel_state.out_buffer(1);
    when (rcv_ack1) =>
      out_system_state.machinel_state.ack_rec(1) := t;
      out_system_state.GLOBAL_VARIABLES.DATA := e;
    when (rcv_data1) =>
      out_system_state.machinel_state.in_buffer(1) :=
      in_system_state.GLOBAL_VARIABLES.DATA;
      out_system_state.machinel_state.current := e;
    when (snd_ack1) =>
      out_system_state.GLOBAL_VARIABLES.DATA := a1;
      out_system_state.machine2_state.pkt_rec(1) := t;
      out_system_state.machine2_state.ack_rec(1) := t;
      out_system_state.machine2_state.in_buffer(1) :=
      in_system_state.GLOBAL_VARIABLES.DATA;
      out_system_state.machine2_state.current := f;
    when (adv_win1) =>
      out_system_state.machinel_state.ack_rec(1) := f;
      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
  put("","integer'image(tuple.machine1_state.state_number));
  put("","integer'image(tuple.machine2_state.state_number));
  put("","); put(tuple.machine1_state.out_buffer(1), width => 1);
  put("","); put(tuple.machine1_state.ack_rec(1), width => 2);
  put("","); put(tuple.machine2_state.in_buffer(1), width => 1);
  put("","); put(tuple.machine2_state.pkt_rec(1), width => 2);
  put("","); put(tuple.GLOBAL_VARIABLES.DATA, width => 2);
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tuple: in out Gstate_record_type;
  counter: in out integer) is
begin
  put(reach, counter);
  put(reach,"","integer'image(tuple.machine1_state.state_number));
  put(reach,"","integer'image(tuple.machine2_state.state_number));
  put(reach,""); put(reach,tuple.machine1_state.out_buffer(1), width => 1);
  put(reach,""); put(reach,tuple.machine1_state.ack_rec(1), width => 2);
  put(reach,""); put(reach,tuple.machine2_state.in_buffer(1), width => 1);
  put(reach,""); put(reach,tuple.machine2_state.pkt_rec(1), width => 2);
  new_line(reach);
end output_Gtuple_to_file;

separate (main)
procedure output_Gstate_node(Gstate_pointer: in out Glink_type;
  Error_flag: in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if ((output_line_count mod 10) = 0) then scroll_pause;
  end if;
  set_col(Gcolumn_set);
  put(Gstate_pointer.system_state_number, width => 3);
  output_Gtuple(Gstate_pointer.Gtuple);
  if (Gstate_pointer.link1.Glink = null) then (Gstate_pointer.link2.Glink = null) and then (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null)
  then Error_flag := true;
  else Error_flag := false;
  end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of: sel_rep_W1

Global State GRAPH

<table>
<thead>
<tr>
<th>State</th>
<th>From State</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>snd_datal</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rcv_datal</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>snd_ackl</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>rcv_ackl</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>adv_winl</td>
<td>0</td>
</tr>
</tbody>
</table>

System State GRAPH

<table>
<thead>
<tr>
<th>State</th>
<th>From State</th>
<th>Transition</th>
<th>Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>snd_datal</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>rcv_datal</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>snd_ackl</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>rcv_ackl</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>adv_winl</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</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>
APPENDIX J (SCM) SELECTIVE REPEAT, W=2

INPUT (FSM)

```
start
machine 1
state 0
trans snd_data 1
state 1
trans rcv_ack 2
state 2
trans adv_win 0
machine 2
state 0
trans rcv_data 1
state 1
trans snd_ack 0
initial_state 0 0
finish
```
package definitions is

  type scm_transition_type is (snd_data, rcv_data, 
    snd_ack, rcv_ack, 
    adv_win1, unused);

  type buffer_type is (d1,d2,e,a1,a2);

  type boolean_type is (t,f);

  subtype ack_buffer_type is buffer_type range e..a2;

  subtype data_buffer_type is buffer_type range d1..e;

  type ack_array_type is array(1..2) of ack_buffer_type;

  type data_array_type is array(1..2) of data_buffer_type;

  type boolean_array_type is array(1..2) of boolean_type;

  type machine1_state_type is
    record
      state_number : natural := 0;
      out_buffer   : data_array_type := (dl,d2);
      ack_rec      : boolean_array_type := (others=>f);
      current      : integer range 1..2 := 1;
      hold         : boolean_type := f;
    end record;

  type machine2_state_type is
    record
      state_number : natural := 0;
      in_buffer    : data_array_type := (others=>e);
      pkt_rec      : boolean_array_type := (others=>f);
    end record;

  type global_variable_type is
    record
      DATA         : data_array_type := (others=>e);
      CONTROL     : ack_array_type   := (others=>e);
    end record;

end definitions;
**PREDICATE-ACTION**

separate (main)
function AnalyzePredicates_Machinel(local : machinel_state_type;
GLOBAL: global_variable_type) return transition-

stack_package.stack is
begin
MakeEmpty(transition_stack);
if (((local.hold = f) and (local.out_buffer(1) /= E) and (GLOBAL.DATA(1)=E)) or
((local.hold = f) and (local.out_buffer(2) /= E) and (GLOBAL.DATA(2)=E))) then
Push(transition_stack, snd_data);
end if;
if (((local.ack_rec(1)=f) and (GLOBAL.CONTROL(1) = A1)) or
((local.ack_rec(2)=f) and (GLOBAL.CONTROL(2) = A2))) then
Push(transition_stack, rcv_ack);
end if;
Push(transition_stack, adv_win1);
return transition_stack;
end AnalyzePredicates_Machinel;

separate (main)
function AnalyzePredicates_Machine2(local : machine2_state_type;
GLOBAL: global_variable_type) return transition-

stack_package.stack is
begin
MakeEmpty(transition_stack);
if (((GLOBAL.DATA(1)=D1) and (GLOBAL.DATA(2) =E) and (local.pkt_rec(1)=f)) or
((GLOBAL.DATA(1) =E) and (GLOBAL.DATA(2) = D2) and (local.pkt_rec(2)=f)) or
((GLOBAL.DATA(1) = D1) and (GLOBAL.DATA(2) = D2) and (local.pkt_rec(1)=f) and
(local.pkt_rec(2)=f)) or
((GLOBAL.DATA(1) = D1) and (GLOBAL.DATA(2) = D2) and (local.pkt_rec(1)=f) and
(local.pkt_rec(2)=f))) then
Push(transition_stack, rcv_data);
end if;
if (((GLOBAL.CONTROL(1)=E) and (local.pkt_rec(1)=t)) or
((GLOBAL.CONTROL(2)=E) and (local.pkt_rec(2)=t)) then
Push(transition_stack,snd_ack);
end if;
return transition_stack;
end AnalyzePredicates_Machine2;  -- this returned value is then checked against the
machine arrays
-- to determine if indeed this transition can be taken

separate (main)
procedure Action(in_system_state : in out Gstate_record_type;
in transition : in out scm_transition_type;
out_system_state : in out Gstate_record_type) is

temp : integer := 0;
begin
case (in transition) is
when (snd-data) =>
out_system_state.GLOBAL_VARIABLES.DATA(in_system_state.machinel_state.current) :=
in_system_state.machinel_state.out_buffer(in_system_state.machinel_state.current);
out_system_state.machinel_state.current := in_system_state.machinel_state.-
current;
begin
case (in_system_state.machinel_state.current) is
when 1 =>
out_system_state.machinel_state.current := 2;
when 2 =>
out_system_state.machinel_state.current := 1;
out_system_state.machinel_state.out := t;
when others =>
put_line("error in the action procedure");
end case;
end;
end (rcv_ack) =>
if(in_system_state.GLOBAL_VARIABLES.CONTROL(1)=A1) then
out_system_state.machinel_state.ack_rec(1) := t;
out_system_state.GLOBAL_VARIABLES.CONTROL(1) := e;
out_system_state.machinel_state.current := 1;
end if;
end;
else
if (in system state.GLOBAL VARIABLES.CONTROL(2)=A2) then
out system state.machine1 state.ack_rec(2) := t;
out system state.GLOBAL VARIABLES.CONTROL(2) := e;
system state.machine1 state.current := 2;
end if;
end if;
when (rcv data) =>
if (in system state.GLOBAL VARIABLES.DATA(1) = D1) then
out system state.machine2 state.in buffer(1) :=
in system state.GLOBAL VARIABLES.DATA(1);
else
if (in system state.GLOBAL VARIABLES.DATA(2) = D2) then
out system state.machine2 state.in buffer(2) :=
in system state.GLOBAL VARIABLES.DATA(2);
end if;
end if;
when (snd ack) =>
if (in system state.machine2 state.pkt_rec(1)=t) then
out system state.GLOBAL VARIABLES.CONTROL(1) := a1;
out system state.machine2 state.pkt_rec(1) := f;
out system state.machine2 state.in buffer(1) := e;
else
if (in system state.machine2 state.pkt_rec(2)=t) then
out system state.GLOBAL VARIABLES.CONTROL(2) := a2;
out system state.machine2 state.pkt_rec(2) := f;
out system state.machine2 state.in buffer(2) := e;
end if;
end if;
when (adv win) =>
if (in system state.machine1 state.ack_rec(1)=t and
in system state.machine1 state.ack_rec(2)=f and
in system state.machine1 state.current = 1 and
in system state.GLOBAL VARIABLES.DATA(1) = E) and
(in system state.GLOBAL VARIABLES.DATA(2) = D2) then
out system state.machine1 state.ack_rec(1) := f;
out system state.machine1 state.ack_rec(2) := f;
out system state.machine1 state.current := 2;
out system state.GLOBAL VARIABLES.DATA(1) := D1;
out system state.GLOBAL VARIABLES.DATA(2) := E;
elsif
(in system state.machine1 state.ack_rec(1)=t and
in system state.machine1 state.ack_rec(2)=f and
in system state.machine1 state.current = 1 and
in system state.machine1 state.current = 1 and
in system state.GLOBAL VARIABLES.DATA(1) = E) and
(in system state.GLOBAL VARIABLES.DATA(2) = D2) and
(in system state.machine1 state.current = 1) then
out system state.machine1 state.ack_rec(1) := f;
out system state.machine1 state.ack_rec(2) := f;
out system state.machine1 state.current := 2;
elsif
(in system state.machine1 state.ack_rec(1)=t and
in system state.machine1 state.current = 1 and
in system state.machine1 state.current = 1 and
in system.state.GLOBAL VARIABLES.CONTROL(1) = E) and
(in system state.GLOBAL VARIABLES.CONTROL(2) = A2) then
out system state.machine1 state.ack_rec(1) := f;
out system state.machine1 state.ack_rec(2) := f;
out system state.machine1 state.current := 2;
out system state.GLOBAL VARIABLES.CONTROL(1) := A1;
out_system_state.GLOBAL_VARIABLES.CONTROL(2) := E;
else

if ((in_system_state.machinel_state.ack_rec(1)=t) and
(out_system_state.machinel_state.ack_rec(2)=E)) then
out_system_state.machinel_state.hold := t;
elsif ((in_system_state.machinel_state.ack_rec(1)=t) and
(out_system_state.machinel_state.ack_rec(2)=f) and
(out_system_state.GLOBAL_VARIABLES.DATA(2)=E)) then
out_system_state.machinel_state.ack_rec(1) := f;
elsif ((in_system_state.machinel_state.ack_rec(1)=f) and
(in_system_state.machinel_state.ack_rec(2)=t) and
(out_system_state.machinel_state.ack_rec(2)=E)) then
out_system_state.machinel_state.ack_rec(2) := f;
out_system_state.machinel_state.hold := f;
else
if ((in_system_state.machinel_state.ack_rec(1)=t) and
(in_system_state.machinel_state.ack_rec(2)=f) and
(out_system_state.GLOBAL_VARIABLES.DATA(1)=E) and
(out_system_state.GLOBAL_VARIABLES.DATA(2)=E)) then
out_system_state.machinel_state.hold := f;
out_system_state.machinel_state.ack_rec(1) := f;
out_system_state.machinel_state.ack_rec(2) := f;
end if;
end if;
out_system_state.machinel_state.current := 1;
end if;
when others =>
put_line("There is an error in the Action procedure");
end case;
end Action;
OUTPUT FORMAT

separate (main)
procedure output_Gtuple(tuple : in out Gstate_record_type) is
begin
  put(""," & integer'image(tuple.machinel_state.state_number));
  put"," & integer'image(tuple.machine2_state.state_number));
  put(',');
  put(tuple.machinel_state.out_buffer(1), width => 2);
  put(',');
  put(tuple.machinel_state.out_buffer(2), width => 2);
  put(',');
  put(tuple.machinel_state.ack_rec(1),width=>1);
  put(',');
  put(tuple.machinel_state.ack_rec(2),width=>1);
  put(',');
  put(tuple.machinel_state.ack_rec(1),width=>1);
  put(',');
  put(tuple.machinel_state.out_buffer(1), width => 2);
  put(',');
  put(tuple.machinel_state.out_buffer(2), width => 2);
  put(',');
  put(tuple.machinel_state.ack_rec(1),width=>1);
  put(',');
  put(tuple.machinel_state.ack_rec(2),width=>1);
  put(',');
  put(tuple.machinel_state.current, width => 1);
end output_Gtuple;

separate (main)
procedure output_Gtuple_to_file(tuple : in out Gstate_record_type;
  counter : in out integer) is
begin
  put(reach,counter);
  put(reach,""," & integer'image(tuple.machinel_state.state_number));
  put(reach"," & integer'image(tuple.machine2_state.state_number));
  put(reach,"");
  put(reach,tuple.machinel_state.out_buffer(1), width => 1);
  put(reach,"");
  put(reach,tuple.machinel_state.out_buffer(2), width => 1);
  put(reach,"");
  put(reach,tuple.machinel_state.ack_rec(1),width=>1);
  put(reach,"");
  put(reach,tuple.machinel_state.ack_rec(2),width=>1);
  put(reach,"");
  put(reach,tuple.machinel_state.current, width => 1);
end output_Gtuple_to_file;

separate (main)
procedure output_Gstate_node(Gstate_pointer : in out Glink_type;
  Error_flag : in out boolean) is
begin
  output_line_count := output_line_count + 1;
  if (output_line_count mod 10) ="0 then
    scroll_pause;
end output_Gstate_node;
end if;
set_col(Gcolumn_set);
put(Gstate_pointer.system_state_number, width => 3);
output_Gtuple(Gstate_pointer.Gtuple);
if ((Gstate_pointer.link1.Glink = null) and
    then (Gstate_pointer.link2.Glink = null)
    and
    then (Gstate_pointer.link3.Glink = null) and then (Gstate_pointer.link4.Glink = null))
    then Error_flag := true;
else Error_flag := false;
end if;
end output_Gstate_node;
OUTPUT

REACHABILITY ANALYSIS of: sel_rep_w2

Global State GRAPH

System State GRAPH

<table>
<thead>
<tr>
<th>Machine 1 Array Contents</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>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Machine 2 Array Contents</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>
REFERENCES


# INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center  
   Cameron Station  
   Alexandria, VA 22304-6145  

2. Dudley Knox Library  
   Code 52  
   Naval Postgraduate School  
   Monterey, CA 93943-5002  

3. Dr. G. M. Lundy  
   Code CS/Lu  
   Assistant Professor, Computer Science Department  
   Naval Postgraduate School  
   Monterey, CA 93943-5000  

4. Dr. Amr Zaky  
   Code CS/Za  
   Assistant Professor, Computer Science Department  
   Naval Postgraduate School  
   Monterey, CA 93943-5000  

5. Defense Information Systems Agency  
   Code: DO  
   ATTN: COL Bowser  
   701 South Courthouse Road  
   Arlington, VA 22204-2199  

6. Headquarters  
   Total Army Personnel Command  
   TAPC-OPF-SC  
   ATTN: LTC Kehoe  
   200 Stovall Street  
   Alexandria, VA 2332-0415  

7. CPT Matthew J. Rothlisberger  
   7200 Leawood Lane  
   Charlotte, NC 28227