

Interactive Monitoring of Distributed Systems

Final Report

Page

Section 1	INTRODUCTION.....	1
Section 2	RADAR DESIGN.....	7
Section 3	COLLECTING INFORMATION.....	9
Section 4	REPLAYING PROGRAM EVENTS.....	20
Section 5	PRONET IMPLEMENTATION.....	28
Section 6	RESULTS AND CONCLUSIONS.....	34
BIBLIOGRAPHY.....		45
APPENDIX A	The LL(1) Grammar.....	47
APPENDIX B	The LL(1) Grammar of A.....	66
APPENDIX C	An Example NETSL.....	80
APPENDIX D	A Network Specification.....	87
APPENDIX E	The Event Handling Module.....	88
APPENDIX F	A Script for Sender Processes.....	90
APPENDIX G	The Preprocessor-generated Link.....	91
APPENDIX H	A Script for the Receiver.....	93
APPENDIX I	The Preprocessor-generated Link.....	94
APPENDIX J	Event Replay Example.....	96

Richard J. LeBlanc

July 16, 1986

U.S. Army Institute For Research in
Management Information and Computer Science
Atlanta, Georgia 30332

Contract No. DAAK70-79-D-0087-0015

GIT Project No. G36-605

TABLE OF CONTENTS

	Page
Section 1 INTRODUCTION.....	1
Section 2 RADAR DESIGN.....	7
Section 3 COLLECTING INFORMATION.....	9
Section 4 REPLAYING PROGRAM EXECUTION.....	20
Section 5 PRONET IMPLEMENTATION.....	28
Section 6 RESULTS AND CONCLUSIONS.....	34
BIBLIOGRAPHY.....	45
APPENDIX A The LL(1) Grammar of NETSLA.....	47
APPENDIX B The LL(1) Grammar of ALSTEN.....	66
APPENDIX C An Example NETSLA program - Broadcasting.....	86
APPENDIX D A Network Specification Module.....	87
APPENDIX E The Event Handling Module.....	88
APPENDIX F A Script for Sender Processes.....	90
APPENDIX G The Preprocessor-generated Code for Sender Processes.....	91
APPENDIX H A Script for the Receiver Processes.....	93
APPENDIX I The Preprocessor-generated Code for Receiver Processes.....	94
APPENDIX J Event Replay Example.....	96

Section 1

INTRODUCTION

1.1 Problems with Monitoring Distributed Programs

In a conventional programming environment, there are two principal purposes for monitoring the run-time behavior of a program: performance measurement and debugging. (By "monitoring" we refer to some mechanism for obtaining information about the performance of a program, *external* to the program itself.) Performance measurement is a relatively mundane application of monitoring in such an environment, being principally concerned with the processor time requirements of various parts of a program and requiring little or no interactive intervention by a programmer. Debugging is considerably more interesting, requiring extensive programmer interaction by its very nature.

When we generalize our thinking to a distributed system from a traditional single-processor environment, the uses of monitoring become somewhat different and we must develop a new conceptual view of a major part of the monitoring task. We are, of course, still interested in performance measurement and debugging, but these tasks become quite different in this new environment. The reason for this difference is that we are now concerned with distributed programs - programs which cannot be monitored by considering a single address space on a single machine. Rather, we must now be concerned with the communication between the various parts of a program, for these interactions will play a crucial part in the monitoring task.

Performance measurement in a distributed system is made more complex by a number of new considerations. Communication costs and the overall time it takes to execute a program, which is affected by the potential for parallel execution of subtasks and by time spent waiting for messages, are equally important considerations in many situations. Further, it is much more difficult for a measurement program to monitor an entire program, since the monitored program may be distributed arbitrarily across a network of machines. It will be necessary for

any monitoring program to obtain information about the distribution of a program and about its communication linkage and behavior.

This need to obtain information from distributed execution sites naturally applies to debuggers as well as to performance monitors. In fact, it is a more complex problem in the case of a debugger since the debugger must somehow assist a programmer in comprehending the "state" of a program which consists of a number of processes running *asynchronously* on several machines. Conventional debugging tools are certainly of little use in this situation, since they are typically oriented toward monitoring the operation of what would only be a single process of a distributed program. Once again, tools which provide information about the status of process interactions will be required. (Such tools should also have the capability to interface with more traditional monitoring tools which can be used on the individual processes.)

Just as communication should play an important part in distributed performance measurement, it should also have a crucial role in debugging distributed programs. The correctness of such programs will undoubtedly depend on the correctness of the contents and sequencing of messages transmitted between their constituent processes. Thus a distributed debugging tool must deal with communication as a major part of its job. In fact, it is conceivable that a communication monitor may be *the* debugger at the interprocess level, complementing traditional debuggers which operate on individual processes.

As a final difficulty, any kind of monitoring of a distributed program will potentially generate a great deal of information, which must be conveyed to a programmer in a comprehensible manner. It will presumably not be satisfactory to produce all of this information independently for each of the processes. Rather, the information must be aggregated in some manner consistent with the nature of the monitoring task being performed.

1.2 Proposed Solutions Using PRONET

The solution we have explored is based on our programming language PRONET [Macc82].

The network descriptors of PRONET provide an excellent basis for the operation of distributed monitoring tools. The interconnection information these descriptors provide is exactly what is required by a monitor so that it can easily recognize the structure of an entire program.

As was suggested in the previous section, a communication monitor is a crucial part of our tools. The interconnection specifications in PRONET networks provide the minimum amount of information needed by a communication monitor. That is, they provide a listing of the message paths between processes and the types of the messages which may be transmitted. The task of a monitor will be to provide a programmer with information about message transmission between processes, including information about the sequencing of messages and about their contents. The capability to examine the operation of individual processes (accomplished by interfacing with a traditional single process debugger) is an important part of our tool set.

1.3 Overview of Project Organization

The project was originally planned to include the following tasks as described in the original statement of work:

Task 1 - PRONET Interface

PRONET, a language that provides a high level description of interprocess communication, is currently being implemented on a distributed system of Prime computers at Georgia Institute of Technology. The task is to develop an interface between PRONET and a distributed monitor.

Task 2 - Communication Monitor

The contractor shall determine what data should be collected by the monitor to facilitate development, debugging and maintenance of programs. This task is to develop a monitoring program that interfaces with the communication features of the operating system and collects the necessary data.

Task 3 - Interface to the Communication Monitor

The contractor shall develop a convenient user interface to the communications

monitor. The user interface will provide a graphical display of information collected by the monitor. Also, it will do additional automated processing of the data to consolidate into meaningful form the information generated by the monitor.

Task 4 - Interface with a Process-level Debugger

The contractor shall develop an interface with the communications monitor and an existing symbolic debugger. If this approach is infeasible, then symbolic debugger for individual processes must be implemented and interfaced with the single process debugger.

During the course of the project, some changes from the initial plans were found to be necessary. The most prominent change involved the use of different hardware than originally planned. The main reason for this change was that we found the implementation of PRONET on our Primes too inefficient to be practical. The operating system on these machines does not effectively support dynamic process creation. The Accent operating system available on our Perq computers, on the other hand, supports dynamic process creation as well as message passing between processes on different machines. Thus we chose to do the work using our Perq workstations, which meant that more work on the implementation of Pronet than originally had been planned turned out to be necessary. However, this work was minimized by implementing Pronet through use of a pre-processor which generates Perq pascal code.

The Perqs also have high-resolution, bit-mapped displays. This feature gave considerable support to the development of a very effective graphical user interface to our monitoring system. We consider this interface one of the most successful aspects of the project.

The other major change in our approach involved the development of a passive event recording system rather than a monitor which supports interaction with distributed programs during execution. This passive approach was initially seen as a prototype. However, we found that a simulated replay of program execution using the information we record during execution provides an effective visualization of a distributed programs, so it remained the focus of our

work throughout the project.

Only task 4 went just as it was originally planned. Our program replay system interfaces with the Kraut debugger, which is a standard high-level debugger under Accent.

1.4 Summary of Project Results

As discussed in the previous section, we made use of the bit-mapped displays on our Perq computers to develop a graphical user interface to our monitoring systems. In effect, it produces a high-level, animated view of program execution. We say this view is "high-level" because it includes only events visible at the process interconnection level (e.g., process creation and interprocess communication). This graphical display approach has proved to be an excellent technique for managing the large quantity of information collected in monitoring a distributed program.

One of the hardest issues to be dealt with in the design of a distributed program monitor or debugger is how to minimize the impact it has on the execution of a program under examination. Our ultimate decision to concentrate on passive monitoring followed by a replay was heavily influenced by this consideration. We believe we have developed tools which can be effectively used to debug applications level distributed programs, based on this minimally obtrusive passive monitoring approach.

Part of our methodology for making use of passive monitoring involves what we call multi-level debugging. In addition to looking at the high-level animation of execution described above, the user also has the ability to focus on the execution of a single process, once the source of a failure has been isolated. Our technique integrating of our monitoring system with an existing single-process debugger is the key to making multi-level debugging available.

The results of this project were reported at the 5th International Conference on Distributed Computing Systems in a paper by R. J. LeBlanc and A. D. Robbins, entitled "Event-Driven Monitoring of Distributed Programs".

1.5 Report Overview

The following three sections describe various aspects of the design of the prototype monitor, called RADAR. They are extracted from Arnold Robbins' M.S. thesis. They are followed by sections on the PRONET implementation, the monitor implementation and the conclusions we have drawn from our research.

The RADAR monitor is structured to support PERQ [MRC87], a message based language designed as a part of previous research on distributed computing at Georgia Tech. However, it could be easily adapted to support other message-oriented programming systems. The relevant features of Pronet will be discussed in section 3.1.

2.2 The RADAR System

The RADAR system takes a partial snapshot of running distributed programs. Because it is interactive the term "monitor" is somewhat misleading; it could also be called the term "debugger."

RADAR is designed to support PERQ on 386/286 computers [BRCC82]. The PERQ is a single user machine with a high resolution bit graphics display and a mouse.

Pronet consists of two sublanguages: NETSLA for describing communication networks, and ALSTEN for describing processes. The PRONET program provides the monitor with information concerning the connectivity of the processes. This information is collected from the NETSLA runtime system. ALSTEN programs use ALSTEN, with a special communications library which records every standard or user-defined event being executed, and makes a copy of every message sent. The exact nature of the interface required by the NETSLA runtime system and the structure of ALSTEN event records will be described in section 3.2. This component of RADAR is known as the RADARLOG.

After the program has completed executing, the REPLAY component of RADAR is invoked to provide a graphical "replay" of the execution. Each message or event is stamped with a global event number. This imposes a partial ordering on events. The monitor then displays events one at a time. The programmer is able to watch the communications traffic

Section 2

RADAR DESIGN

2.1 Distributed Programs

The RADAR monitor is intended to support Pronet [Macc82], a message based language designed as a part of previous research on distributed computing at Georgia Tech. However, it could be easily adapted to support other message-based programming systems. The relevant features of Pronet will be discussed in section 3.1.

2.2 The RADAR System

The RADAR system takes a passive approach to monitoring distributed programs. Because it is interactive the term "monitor" is used to describe it, and not the term "debugger."

RADAR is designed to support Pronet on PERQ computers [3RCC82]. The PERQ is a single user machine with a high resolution bit-mapped display and a mouse.

Pronet consists of two sublanguages: NETSLA for describing communication networks, and ALSTEN for describing processes. The Pronet compiler provides the monitor with information concerning the connectivity of the processes. This information is collected from the NETSLA runtime system. ALSTEN programs are loaded with a special communications library which records every standard or user-defined event during execution, and makes a copy of every message sent. The exact nature of the information supplied by the NETSLA runtime system and the structure of ALSTEN event records will be described in section 3.2. This component of RADAR is known as the RADARLOG.

After the program has completed executing, the REPLAY component of RADAR is invoked to provide a graphical "replay" of the execution. Each message or event is stamped with a global event number. This imposes a partial ordering on events. The monitor then displays events one at a time. The programmer is able to watch the communications traffic

amongst the processes. Processes have names in Pronet, so it is easy for the programmer to see which process is communicating with which other processes.

REPLAY provides the user with the ability to view the contents of any message currently represented on the screen. Messages are represented on the screen as small boxes. The user places the PERQ's mouse over the message which he wishes to examine. REPLAY then opens a new window in which the contents of the chosen message will be displayed in a formatted fashion. For instance, if the message contained an integer and two floating point numbers, the message would be displayed as an integer and two floating point numbers, not as 10 octal bytes. When the user is through with the message the new window disappears.

REPLAY also provides the ability to replay a certain number of events which have already happened. This can be done at any point during the display. The user can "rewind the video tape," so to speak. This replay is limited to a reasonable maximum number of previous events. This feature is known as an "Instant Replay."

Finally, as a separate utility, the user can name a given process and have all of the messages which were sent to that process selected from the recorded message traffic. This single process may then be run by itself with its messages derived from the stored messages. This feature is designed to facilitate single process debugging using real input data (messages). This way, it is possible to observe a process' behavior under realistic conditions, without having to worry about controlling the rest of the processes of the distributed program.

Section 3

COLLECTING INFORMATION

RADAR is intended to support Pronet, a language designed for writing programs which can execute in a distributed processing environment. Pronet stands for **Processes and Networks**. The introduction to Chapter 2 of [Macc82] summarizes the description and design goals of Pronet:

PRONET is composed of two complementary sublanguages: a network specification language, NETSLA, and a process description language, ALSTEN. Programs written in PRONET are composed of network specifications and process descriptions. Network specifications initiate process executions and oversee the operations of the processes they have initiated. The overseeing capacity of network specifications is limited to the maintenance of a communication environment for a collection of related processes. The processes initiated by a network specification can be simple processes, in which case the activities of the processes are described by ALSTEN programs, or they can be "composite processes", in which case their activities are described by a "lower-level" network specification.

ALSTEN is an extension of Pascal which enables programmers to describe the activities of sequential processes. During their execution, processes may perform operations that cause events to be announced in their overseeing network specification. Network specifications, written in NETSLA, describe the activities to be performed when an executing process 'announces' an event... Two principles have influenced the design of these features: independence of process descriptions and distributed execution of network specifications.

This section first describes the features of Pronet relevant to interprocess communication. Then it describes the information provided to the monitor by the NETSLA and ALSTEN compilers. Finally, it presents the format of the information collected at run-time by the special communications library.

3.1 The Features of Pronet

This presentation is derived from Chapter 2 of [Macc82].

3.1.1 ALSTEN

ALSTEN is essentially an extension of Pascal [Jens74]. The file concept has been removed

entirely from the language. Processes communicate only through locally declared *ports*, using *send* and *receive* statements which are analogous to Pascal's *read* and *write*. Ports have a direction, either *in* or *out*. Ports may be combined into *port groups*. One could define a duplex channel as:

port channel (incoming *in* bit; outgoing *out* bit);

To accomodate the notion of a *server process*, which serves many other processes, ALSTEN provides *ports sets* and *port tag* variables. A port set is a collection of port groups or simple ports identified by one name. For instance, if a port set is a set of port groups, a *receive* on a port set would set a port tag variable to indicate which element of the set was actually used for communication. This tag may then be used in a *send* operation for sending replies to the process which originated the message.

The syntax of the *send* and *receive* statements is shown in Figure 1.

```
<send stmt> ::=
    send [<expr>] to <bound port denoter>

<receive stmt> ::= <simple receive>
    | <conditional receive>

<simple receive> ::=
    receive [<variable>] from <free port denoter>

<conditional receive> ::= when
    {<receive part>}
    [<otherwise part>]
    end

<receive part> ::= <simple receive> [do stmt]

<otherwise part> ::= otherwise <stmt>
```

Figure 1 -- Send and Receive Statements in ALSTEN

A type is associated with every port. Only expressions of the type associated with a given port may be sent to or received from that port.

The `<expr>` is optional. In these forms of the *send* and *receive* statements, the port is of type *signal*. A signal is a message with no contents. Signals are often useful for sending control information, such as telling a process to start a particular task.

The syntax for port declarations is shown in Figure 2.

```
<port decl> ::= <simple port decl>
               | <port group decl>

<simple port decl> ::=
    port <port id> <direction> <msg type>

<port id> ::= <id>

<direction> ::= in | out

<msg type> ::= <type id>

<port group decl> ::=
    port [set] <port id> '(' <subport list> ')'

<subport list> ::=
    <subport decl> {';' <subport decl>}

<subport decl> ::=
    <subport id> <direction> <msg type>

<subport id> ::= <id>

<port tag type> ::= tag of <port id>
```

Figure 2 --- Port and Port Tag Declarations in ALSTEN

3.1.2 NETSLA

As stated earlier, the purpose of NETSLA specifications is to initiate and control the communications environment of ALSTEN processes:

The features of NETSLA are aimed at specifying the initial configuration and subsequent modifications of a communication environment for processes. The overriding principle followed in the design of these features is that of "centralized expression--decentralized execution" [Live80]. Centralized expression is important in presenting the abstraction to be supported by network specifications. All of the inter-process relationships that describe a communication environment appear in a single network specification. However, this communication environment is not maintained in a centralized fashion. Processes maintain their communication environment indirectly. When they execute send or announce operations, processes

perform the activities specified by their overseeing network specifications; however, the nature of these activities is unknown to the process (since network specifications are not visible to processes). [Macc82]

The syntax of network specifications is shown in Figure 3.

```
<network specification> ::= <network header>
    {<process class specification>}
    {<event handling clause>}
    [<initialization clause>]
end <identifier>

<network header> ::= network <net id> ';'
    {<port decl>}
    {<event decl>}

<process class specification> ::=
    process class <process id>
    [<process attributes>]
    {<port decl>}
    {<event decl>}
end <process id>

<process attributes> ::= attributes
    <field list>
end attributes
```

Figure 3 --- Network Specifications in NETSLA

When a network starts to run, its initialization clause is executed. The initialization clause is used to create instances of processes and connect the output ports of one process to the input ports of another. A simple network specification is presented in Figure 4; a graphical representation of the network is shown in Figure 5.

If one output port is connected to more than one input port, the messages sent out on it are replicated. This occurs in a manner invisible to the process sending the message. This allows one-to-one, one-to-many, and many-to-one connections between ports.

Processes may define events. These events can then be announced by the processes in their overseeing network specifications. NETSLA provides features for handling these events when they are announced. The programmer specifies what actions to take, such as starting processes

```
network static_net
  process class proc_class
    port input in integer;
    port output out integer;
  end proc_class

  initial
    create proc1 : proc_class;
    create proc2 : proc_class;
    create proc3 : proc_class;
    connect proc1.output to proc3.input;
    connect proc2.output to proc3.input;
    connect proc3.output to proc1.input;
    connect proc3.output to proc2.input;
  end static_net
```

Figure 4 --- A Simple Network Specification

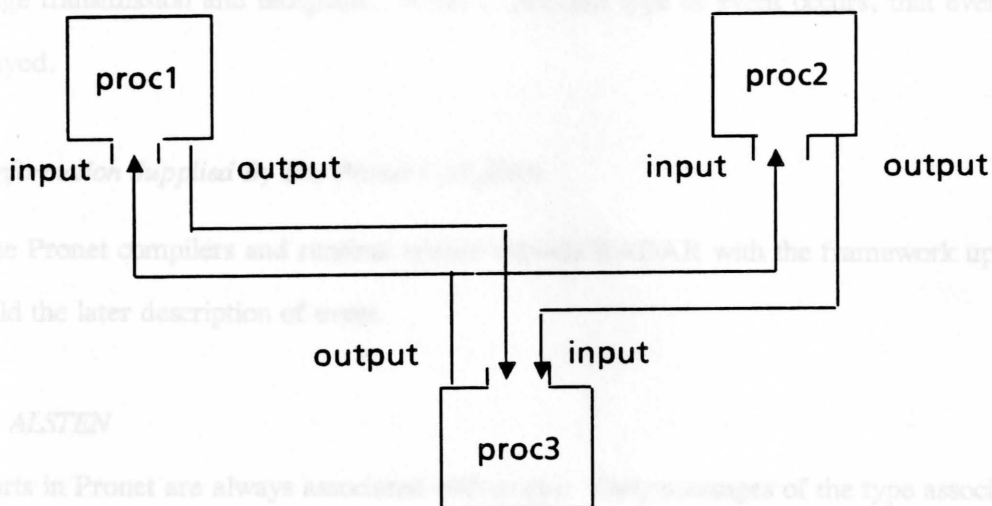


Figure 5 --- A Graphical Representation of a Simple Network

If one output port is connected to more than one input port, the messages sent out on it are replicated. This occurs in a manner invisible to the process sending the message. This allows one-to-one, one-to-many, and many-to-one connections between ports.

Processes may define events. These events can then be announced by the processes in their overseeing network specifications. NETSLA provides features for handling these events when they are announced. The programmer specifies what actions to take, such as aborting processes

or creating new ones. Other actions are also possible.

Pronet predefines several standard events. For instance, when a process terminates normally, the standard event 'done' is announced in its network.

Message transmission and reception are considered to be events. They simply have a separate syntax. The other standard events and the syntax of event declarations and handlers are discussed fully in [Macc82].

Since Pronet is oriented around events, so is RADAR. The special runtime routines record all the events and messages. The REPLAY program presents the user with a visual replay of the events that occurred during the execution of the program. The majority of events will be message transmission and reception. When a different type of event occurs, that event will be portrayed.

3.2 Information Supplied By The Pronet Compilers

The Pronet compilers and runtime system provide RADAR with the framework upon which to build the later description of event.

3.2.1 ALSTEN

Ports in Pronet are always associated with a type. Only messages of the type associated with a port may be sent to or received from that port.

In any given ALSTEN program, there will be a fixed number of different message types, i.e. the types associated with ports.

The ALSTEN compiler will generate a file with a list of *message templates*. A template looks like

Identifier	size	total no elements	list of elements
------------	------	-------------------	------------------

Figure 6 --- Message Templates

The list of elements is simply an order listing of the fields in a message. For instance,

real	array character 19	nt	long
------	--------------------	----	------

Figure 7 --- Fields In A Message

If a field of a message is itself a record with further subfields, the compiler will expand it in line down to its basic elements. Elements can be bytes, integers, long integers, reals, or one dimensional arrays of these types. Bytes are treated as unsigned integers, even though they may have actually been signed quantities. If necessary, RADAR may be modified to allow specifying whether or not such numbers were signed or unsigned. Elements smaller than one byte occupy a byte to themselves. This implies that the Pascal keyword *packed* has no effect. Admittedly, this is a constraint on the compiler; see Section 5 of the thesis for further discussion of this constraint.

The purpose of the list of message templates is to allow the decoding of individual messages. A user can select any message on the screen with the PERQ's mouse. When he does so, RADAR will open a separate window and format the contents of the message in it. Each message carries its type with it. The message is decoded according to the corresponding template and printed accordingly. One dimensional arrays are allowed, principally for use in displaying character strings. REPLAY will treat arrays as if they are indexed from 1.

3.2.2 NETSLA

NETSLA controls process and port creation and the interconnecting of output ports to input ports.

The information generated by the NETSLA system is a file describing each process. A process is described as follows:

```
machine proc_num proc_name number_port_groups
  number of simple ports in each group
    direction number name type { DESTINATIONS }
    direction number name type { DESTINATIONS }
  number of simple ports in each group
    direction number name type { DESTINATIONS }
    direction number name type { DESTINATIONS }
```

Figure 8 --- Description Of A Process

The { } pairs enclose optional information. Only if a port is an output port does it have one or more destinations associated with it. The DESTINATIONS field in Figure 8 above represents the number of destinations to which an output port sends its messages, and the destinations themselves. A destination is uniquely identified by the destination machine, the process number on that machine, and the port number of the process to which the message is directed.

Machine and process id's are hidden from the programmer, but the NETSLA runtime system and the underlying global operating system must know about them, since they actually arrange for execution of the processes.

When REPLAY first starts up, it builds a table of records describing processes with all these structures attached to each element in the table. Later, when a *send* event occurs, REPLAY determines which process is the destination and depicts a message moving from the source process to the destination process.

3.3 Information Collected At Run-Time

Most of the information that RADAR needs is collected at run-time. Special runtime routines log every event that occurs. These routines are kept in a separate module called RADARLOG.

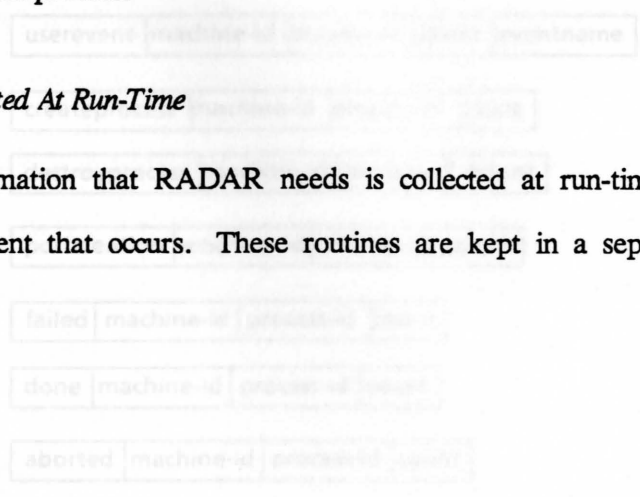


Figure 10 --- Event Records

Events may be one of the following:

type

```
eventtype = (createprocess, destroyprocess,
             message_transmission, message_reception,
             portcreation, failed, done
             aborted, userevent);
```

Figure 9 --- Types of Events

The 'message_transmission' and 'message_reception' events are logged by the *send* and *receive* routines respectively. The other events are logged by the *announce* routine.

The ALSTEN compiler inserts a procedure call to the routine *makelog* as the very first executable statement in a program. This routine creates the log file and announces the process creation event. Before the final *end* of the ALSTEN main program, the compiler inserts a call to the routine *closelog*, which closes the logfile and announces the standard event 'done'.

message-transmission	machine-id	process-id	count
UniqueMesgId	success	checkpointing	mesg-type
bufsize	','	buffer	
message-reception	machine-id	process-id	count
	success	{ UniqueMesgID }	
userevent	machine-id	process-id	count
	eventname		
createprocess	machine-id	process-id	count
destroyprocess	machine-id	process-id	count
portcreation	machine-id	process-id	count
failed	machine-id	process-id	count
done	machine-id	process-id	count
aborted	machine-id	process-id	count

Figure 10 --- Event Records

Each process keeps a count of the events it has announced, including message transmission and reception. The event count starts at one and is incremented with each event.

When a process sends a message, it includes the value of its local event counter. If the receiving process' event count is lower than that of the sender's, the receiver sets its count equal to that of the sender. After receiving the message, the process logs the `message_reception` event. If the message reception succeeded, the process logs the UniqueMesg Id of the message it received. Since `message_reception` is an event like any other, the local event count is incremented before the event is logged. Thus, the `message_reception` event's sequence number will be one greater than the event count of the sender. This insures that there will be at least a partially correct ordering on events. In particular, interrelated events will always be correctly ordered.

Placing an ordering on events in a distributed system is a difficult task. One solution is to use the times on local clocks to time-stamp each event. This method is not acceptable since it is impossible to synchronize all the clocks on all the machines. This introduces the possibility of recording events out of order. For example it would be possible, due to synchronization errors among clocks, to record the reply to a message as having occurred "before" the sending of the initial message.

By having the receiver of a message set its event count equal to that of the sender, and then incrementing the count before logging the message reception, the synchronization problem is avoided. The reply to a message will always be sent "after" the sending of the initial message.

Using this method, it is possible to have several events occurring at the same "time," i.e. several events might all have the same event number. In this case, it is impossible to determine the ordering of these events, but in fact, the ordering is unimportant. The fact that these events all have the same number indicates that they are *not* interrelated, since if one event depended on another to precede it, its event sequence number would have been greater than the sequence number of its predecessor.

Furthermore, this method makes no extra demands on the underlying global operating system to keep clocks synchronized across machines. It also fits in well with Pronet, which has no concept of global time.

3.3.0.1 Summary

Keeping a record of every event, along with a description of message contents and the interconnectivity of every port, provides a complete record of what went on.

Copying all the message allows the user to view what was actually sent; the message description makes the message contents understandable, and the connectivity data allows graphically depicting the movement of a message from its source to its destination.

REPLAY uses the Sapphire protocol, which provides a higher-level, more usable interface to control the screen.

This section discusses the algorithm REPLAY uses, describes the view of the program REPLAY presents to the user, and presents the user interface.

4.1 Outline of the Algorithm

The overall algorithm is fairly simple. It is based on the notion of events as defined previously. Since each event is scheduled, time ordering of events is automatically made possible.

The general algorithm for event replaying is given in pseudo code in Figure 11.

```
get first event
while more events
  if event in ( send_a_message, receive_a_message )
    do something visible with the message
  else
    announce the event audibly
  end if
  get next event
end while
```

Figure 11 -- Top Level REPLAY Algorithm

Section 4

REPLAYING PROGRAM EXECUTION

The major component of the RADAR system is the REPLAY program. After a Pronet program has executed and all the information described above has been collected, REPLAY is invoked to graphically display event occurrences. More importantly, it also displays the message traffic amongst processes.

The PERQ's screen is a high resolution, bit-mapped black and white display. The PERQ has hardware and firmware instructions, called Raster Ops, for manipulating the screen. REPLAY uses the Sapphire graphics package which provides a higher-level, more usable interface to control the screen.

This section discusses the algorithms REPLAY uses, describes the view of the program REPLAY presents to the user, and presents the user interface.

4.1 Outline of the Algorithm

The overall algorithm is fairly simple. It is based on the notion of events as defined previously. Since each event is numbered when recorded, an ordering of events is automatically made possible.

The general algorithm for event replaying is given in pseudo-code in Figure 11.

```
get first event
while more events
  if event in { send_a_message, receive_a_message }
    do something visible with the message
  else
    announce the event conventionally
  end if
  get next event
end while
```

Figure 11 --- Top Level REPLAY Algorithm

Most of the work is involved with displaying events. REPLAY basically has to keep track of four things.

- 1) Which processes are represented on the screen and where they are.
- 2) Which messages are represented on the screen and where they are.
- 3) Rate of event display (see below).
- 4) How full the screen is; i.e., is there room for more processes?

Processes and the messages waiting in input queues take up the majority of the room on the screen. Most of the other events can be displayed simply by printing out a line on the screen of the form "Process P announces Event E as event Number N," in a prominent place. During the interval that the process is announcing an event, it changes color (actually a different shade of gray) so that it is clear which process is involved.

In fact, REPLAY provides a running narrative of this form. However, when a process is created or destroyed, or a message is sent or received, REPLAY will depict this graphically. Newly created processes will be drawn into a free spot on the screen. Messages are depicted as small boxes moving from the sender's output port to the receiver's input port. When each message is received, its box disappears.

Much of the work involves doing all the bookkeeping necessary in as efficient a manner as possible. (It should be "efficient" in terms of both space and time).

4.2 The User Interface

This section discusses various aspects of the operation of REPLAY's user interface.

4.2.1 What the User Sees

The user sees processes and messages queued on input ports. A process with one input port,

one output port and a message just leaving the output port, is shown in Figure 12.

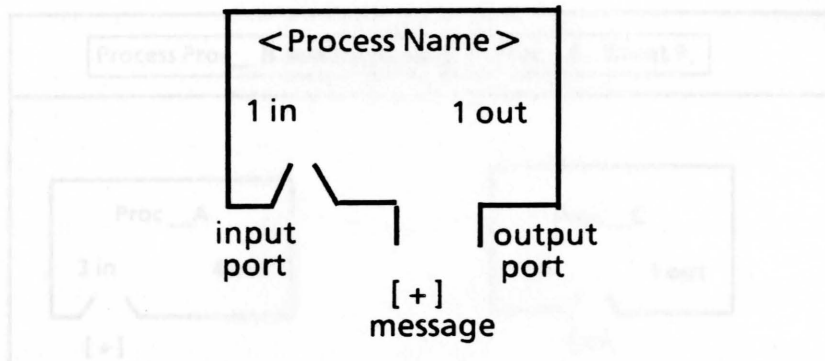


Figure 12 - Picture of a Process and a message

The drawing of a process indicates the number of input and output ports associated with that process. It is not possible to draw each port, since the notion of port sets allows a process to have a very large number of ports. When an output port sends a message, the port appears on the process' border. It closes up after the message arrives at its destination. Similarly, when a message arrives for an input port, the port opens up, and messages queue up in front of it. When all the queued messages have been received, the input port closes back up. The process name and identification appear inside the box, so that it is clear at a glance which process it is.

for the user to follow.

Figure 13 depicts an event replay on the PERQ's screen. The process **Proc_B** is shown sending a message to **Proc_A**, while process **Proc_C** is shown with one message waiting at its input port. The event narration at the top of the screen indicates what is happening. Appendix event takes the full n seconds (whatever the user wants to measure). This is to allow the process to change color, and to remain on the screen in a different color for enough time to make an impression on the user before it changes back to normal.

J contains a sequence of figures portraying a more extended example.

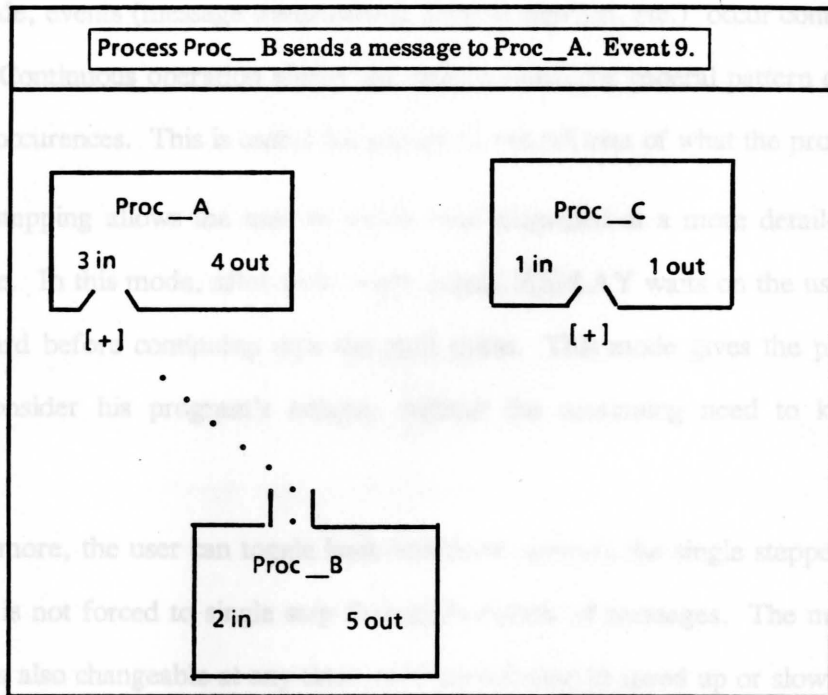


Figure 13 --- A Process Sending A Message

An interesting problem concerns the speed at which the replaying occurs. If events are described and messages move across the screen without any delays, events will happen too fast for the user to follow.

To solve this problem, REPLAY asks the user how many seconds to take to display each event. The default is three seconds per event. Even in single step mode (see below), each event takes the full n seconds (whatever the user entered) to transpire. This is to allow the process to change color, and to remain on the screen in a different color for enough time to make an impression on the user before it changes back to normal.

4.2.2 Single Stepping

REPLAY gives the user the choice of either single stepped or continuous operation. In the second mode, events (message transmission, process creation, etc.) occur continuously, without stopping. Continuous operation allows the user to watch the general pattern of message traffic and event occurrences. This is useful for getting an overall idea of what the program did.

Single-stepping allows the user to watch what happened at a more detailed level and at a slower pace. In this mode, after each event occurs, REPLAY waits on the user to hit a key on the keyboard before continuing with the next event. This mode gives the programmer more time to consider his program's actions, without the continuing need to keep up with his program.

Furthermore, the user can toggle back and forth between the single stepped and continuous modes; he is not forced to single step through hundreds of messages. The number of seconds per event is also changeable at any time, to allow the user to speed up or slow down the rate of event display.

4.2.3 Displaying Messages

Messages on the screen are simply small boxes, queued on the input ports of their destination processes. In this form, the only information that they convey is the fact of their existence. This is only minimally useful.

REPLAY allows the user to actually see what his processes are sending to each other. Using the mouse, the user places the cursor over the particular message he wants to see and interrupts the event display. REPLAY will prompt with a menu of actions available. The user will select the option for viewing a message.

REPLAY first finds the message indicated by the mouse. The message's type is an element in the Pascal *record* describing messages. This type indicates which of the message templates is to be used in decoding the contents of the message.

REPLAY then opens a new window on the screen. It steps through the message buffer and formats the raw bytes into characters, integers, or reals, as dictated by the message template. Enumerated types are treated as integers. Although this is not perfect, it is no more unreasonable than the restriction in standard Pascal against reading and writing enumerated types to and from text files. Message templates were described in Section 2.2.1.

When the user is through looking at the message, he issues the command to close the window. REPLAY then goes back to displaying events.

The value of this "Freeze Frame" facility should be clear. The user can verify not only that processes are sending messages to the right places, but that those messages have the right contents. Formatting message contents is absolutely necessary. Simply displaying the values of integers, characters and reals in octal gives the user no immediately understandable information (except in the rare case of the true hacker who can decode octal into its equivalent floating point or ASCII values). Furthermore, messages are displayed as a unit, unlike Schiffenbauer's system which displays small data packets in octal.

4.2.4 *Selective Replaying of Events*

It is possible while watching a program's actions that a particularly interesting sequence of events will occur which warrants further review. To accomodate this, REPLAY keeps a history of a fixed number of events which have occurred. At any time, the user can stop the normal replay and ask to see an "Instant Replay" of n previous events. The maximum number of events that can be replayed is a compile-time constant in one of the Pascal source code modules.

When this facility is invoked, REPLAY saves the screen state and marks those processes that were on the screen at the time. It clears the screen and starts as if the first event requested were the very first event to occur. Processes and messages are drawn as needed.

Some information which was on the screen but which may not relate to the n events being replayed will be lost during the instant replay. This loss is not permanent, since REPLAY

restores the screen at the end of the instant replay. The user can run the instant replay as many times as desired before returning to the regular display. This facility is analogous to the rewinding of video tape and replaying an interesting series of events during a sports broadcast, hence the name "Instant Replay."

When the instant replay is through, the screen is restored and the processes which were marked as being saved are unmarked. Display then continues as before.

As a final possibility, the user may choose to restart the entire program replay from scratch. This provides the convenience of not having to quit the program and then start it again from the command level. Such small conveniences are often the most useful.

4.2.5 REPLAY Menu Options

At any time during the event replay the user can stop execution by causing a keyboard interrupt.

This invokes an interrupt handler which presents the menu shown in Figure 14.

1. Change To/From Single-Step/Continuous Operation
2. Change The Number of Seconds Per Event
3. Skip Ahead To A Specific Event Number
4. Display contents of the Message Under the Mouse
5. Instant Replay
6. Start Displaying From Scratch
7. Exit REPLAY
8. Help
9. Never Mind

Figure 14 --- REPLAY Menu Options

The user may skip ahead to a given event, specified by the event sequence number.

REPLAY will then skip to the first event which has the sequence number entered by the user. This is useful if the user knows that his program stopped working after a given event. He can make his changes, rerun the program, and then skip directly to where the change should have an effect.

The help subsystem provides general information on how to use the RADAR monitor.

The 'Never Mind' option allows the user to recover in case he accidentally caused a keyboard interrupt. It does nothing.

In all cases, after the interrupt handler does what the user wishes, the program returns to where it was executing before the interrupt occurred.

3.1 The Preprocessors

The preprocessor actually consists of two parts: a scanner and a parser, both are table-driven. The table-driven approach makes the preprocessor very language independent, i.e., it can translate either ALSIEN or NETSLA as soon as the appropriate tables are provided.

The scanner tables are generated by the LEX token generator from a description of each token that may occur as input to the scanner. LEX is similar to the standard Unix LEX program except that it produces no output, just tables. These tables may then be used in a scanner written in any language (Pascal, in this case). Tokens are described by using a standard regular expression syntax. The parser tables are generated by ZUSE from LL(1) grammars (see Appendix A and Appendix B) which have action codes embedded into them. ZUSE is similar to the Unix YACC generator, except that it generates a parsing program in Pascal rather than C. The action codes give the parser fragments of code to be executed as the parser recognizes syntactic structures in the input. In the case of this preprocessor, appropriate Pascal codes is generated by these fragments.

Section 5

PRONET IMPLEMENTATION

An implementation of PRONET has been developed for PERQ computers running under revision 2.0 of ACCENT, which is a communication oriented network operating system. The run-time libraries developed for this implementation make use of ACCENT message and process primitives through a procedure-like interface to the kernel.

Two language preprocessors, one for ALSTEN and another for NETSLA, have been developed. These two preprocessors both translate a PRONET source program into a Pascal program. Then, the Pascal program generated can be compiled using the PERQ Pascal compiler.

5.1 The Preprocessors

The preprocessor actually consists of two parts: a scanner and a parser; both are table-driven. The table-driven approach makes the preprocessor very language independent; i.e., it can translate either ALSTEN or NETSLA so long as appropriate tables are provided.

The scanner tables are generated by the LEXGEN scanner generator from a description of each token that may occur as input to the scanner. LEXGEN is similar to the standard Unix LEX program except that it produced no program, only tables. These tables may then be used in a scanner written in any language (PERQ Pascal, in this case). Tokens are described by using a standard regular expression syntax. The parser tables are generated by ZUSE from LL(1) grammars (see Appendix A and Appendix B) which have action codes embedded into them. ZUSE is similar to the Unix YACC program except that it generates a parsing program in Pascal rather than C. The action codes provide program fragments steps to be executed as the parser recognizes syntactic structures in the input. In the case of this preprocessor, appropriate Pascal codes is generated by these fragments.

The preprocessor accepts a scanner table, a parser table and source program as input and generates a sequence of Pascal codes as a result of parser actions. The Pascal code generated can then be compiled by using the PERQ Pascal compiler.

Figure 15 below illustrates the overall structure of the preprocessors.

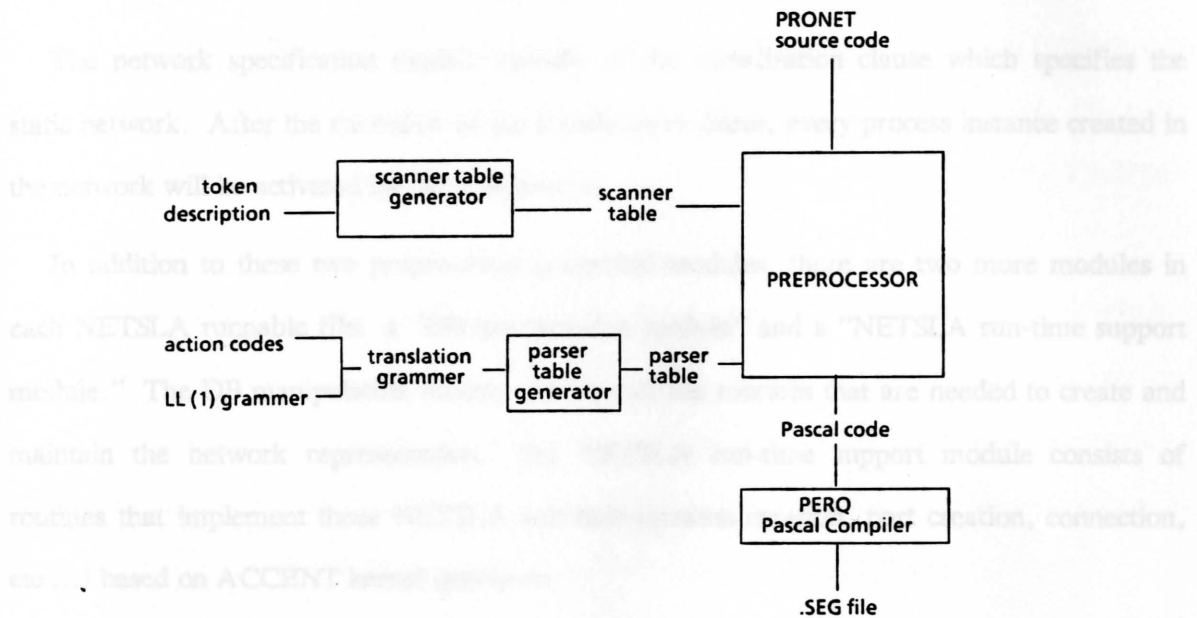


Figure 15 --- Preprocessor Structure

The approach of preprocessing has two important advantages, although it is less efficient than direct compilation. The first is that it was far easier to implement than a compiler would have been. The second is that it makes the full power of PERQ Pascal, particularly access to ACCENT kernel primitives, available to Pronet programmers, since kernel primitives are accessible through calls to kernel interface procedures and functions in the Pascal library. The preprocessors do no type checking, leaving that task to the Pascal compiler.

5.2 Module Structures

The NETSLA preprocessor generates two code modules for each network specification: an "event handler module" and a "network specification module" (see Appendices C, D and E).

The event handler specifies the action that must be performed when a particular event (either predefined or process-defined) occurs. The code in this module is structured as a nested "case" statement. The highest level case statement performs a selection based on the event type argument (message transmission, process-defined event, etc.). Lower level case statements are used to distinguish between process classes, port sets and process-defined events.

The network specification module consists of the initialization clause which specifies the static network. After the execution of the initialization clause, every process instance created in the network will be activated by the root process.

In addition to these two preprocessor-generated modules, there are two more modules in each NETSLA runnable file: a "DB manipulation module" and a "NETSLA run-time support module." The DB manipulation module contains all the routines that are needed to create and maintain the network representation. The NETSLA run-time support module consists of routines that implement those NETSLA activities (process creation, port creation, connection, etc ...) based on ACCENT kernel primitives.

Figure 16 below illustrates the structure of the object module generated for each NETSLA program. It is important to realize that both the event handler module and the network specification module are network specific while the other two modules are common to all network instances. The DB manipulation module and the NETSLA run-time support module are separately precompiled and imported by the main body of the NETSLA program.

DB Manipulation MOdule	common code
NETSLA Run-time Support Module	(libraries)
Event Handler Module	network
Network Specification Module	specific

Figure 16 --- NETSLA Object Module Structure

The ALSTEN preprocessor generates a single code module for each process script (see

Appendices F, G, H and I). This module is a simple translation of the process script which makes use of ALSTEN run-time support facilities for performing ALSTEN operations (send, receive, announce, etc ...).

5.3 Processes and Ports

Both ACCENT and PRONET use the notions of "processes" and "ports", but they are at different levels of abstraction. We implement PRONET processes and ports by using ACCENT processes and ports; the details of this mapping are hidden from PRONET programmers.

A PRONET network specification is implemented as an ACCENT process from which any number of ACCENT child processes can be created to represent the PRONET process instances. Since we do not consider the case of "composite processes" in this implementation, the network can be thought of as a tree of two levels with the network specification process as the root. Composite processes can be implemented without much effort later.

An ACCENT port is a protected kernel object and is used for sending and receiving messages. With each port the kernel associates send and receive (and ownership) rights. The process that creates the port possesses all three rights. In this implementation, we use ACCENT ports for two different purposes.

During the execution of the program, an ACCENT port will be allocated when a CONNECT activity is performed. This ACCENT port is used for transmitting the PRONET messages and will be deallocated when the corresponding DISCONNECT activity is performed. Initially, the receiving process possesses the receive and send rights. Then the send right will be passed to the sending process so that PRONET messages can be transmitted through this port.

There are three ACCENT ports allocated to each child process at the process creation time for the purpose of communicating with the root process. One is for the root process to send the child its process ID, the second is for implementing dynamic port connections and the third is

needed to implement port groups.

5.4 The Network Representation

A representation of the logical network described by a PRONET program is maintained in the address space of the root process. This representation reflects the hierarchical structure expressed in the program by maintaining a tree of network class and network instance representations. The logical network representation also contains information about the connectivity among the ports of network instances. The root of this tree is a network class representation, the leaves are network instance representations which contain information about the currently active processes in the logical network.

The codes for manipulating the logical network representation also reside in the address space of the root process. All creations, updates and reads of the entities in the network representation must be performed by calling from the root process an appropriate procedure in the DB manipulation module.

This centralized approach of maintaining the logical network representation lowers the degree of parallelism but reduces the cost of message transmission.

5.5 Event Generation and Handling

Event generation can be either upward or downward. The term "upward event generation" is used to denote the generation of an event in the overseeing network while "downward event generation" is used to denote the generation of an event in a process instance.

Upward event generation occurs when a process instance announces an event using the "announce" statement of PRONET or transmits a message using the "send" statement. Downward event generation occurs when a network specification creates or removes a port instance on a process instance or sends a message to a process instance.

Event handling codes are generated by the NETSLA preprocessor and reside in the address

space of the root process during run-time. Upward event generation is implemented by sending a message to the root process. This message includes all the information relevant to the event generated. This kind of message arrives at a port which belongs to the root process and holds at most four messages at a time due to the limitation of the size of the backlog for an ACCENT port.

Upon receiving a message from a child process, the root process will call an appropriate event handling routine based on the event type and other information included in the message. Event handler executions are performed in a serial fashion. This centralized approach of event handling has the disadvantage of a low degree of parallelism.

5.6 Implementation Limitations

All of the features of ALSTEN and NETSLA have been implemented and tested on a single machine. However, because of continuing problems with Accent, we have never been able to successfully run a program with processes located at more than one site. Thus all of our testing of PRONET and RADAR has involved programs consisting of multiple processes running on a single machine.

Radar relies on the collection of information by the normal execution of a program. The program runs to completion without any external interference or control. In particular, the data collection is invisible, since it is done inside the ALSTEN message and event primitives.

How much does the extra disk I/O affect the execution of a program? This is the Heisenberg Uncertainty Principle as applied to Debugging, sometimes called the "Heisenbug" Principle [ACM83]. We can present no definite answer here. It is expected that the disk

Section 6

RESULTS AND CONCLUSIONS

The principal results of our efforts, task by task, were:

Task 1 - implementation of Pronet on our Perq workstations, through use of pre-processors which generate Perq Pascal code.

Task 2 - the development of a passive event recording system for multi-process Pronet programs.

Task 3 - the development of a replay system which produces a high-level graphical simulation of distributed program execution.

Task 4 - integration of the replay system with a single-process debugger.

6.1 *Passive Event Recording*

The decision to go with a passive monitor rather than an interactive debugger was one major change in our philosophy during the course of this work. This change in approach resulted from consideration of the basic conceptual problem presented by active interaction with a distributed program: the intrusiveness of interaction might substantially change the behavior of the program being debugged. Thus we chose to minimize the intrusiveness of Radar, but there still remains the question of just how non-intrusive our monitor is.

Radar relies on the collection of information during the normal execution of a program. The program runs to completion without any external interference or control. In particular, the data collection is invisible, since it is done inside the ALSTEN message and event primitives.

How much does the extra disk I/O affect the computation in program? This is the Heisenberg Uncertainty Principle as applied to Debugging, sometimes called the "Heisenbug" Principle [ACM83]. We can present no definite answer here. It is expected that the disk

operations actually buffer to memory until the buffer fills up. If this is the case, there should be little extra overhead since the system will suspend a process only when its I/O buffers must be flushed. The main problem is that while one process is suspended, others can continue to run on other machines.

It can be argued that the fact that one process on one machine has been stopped should not affect the other processes on other machines, since the ALSTEN *receive* is defined to be a blocking operation. The other processes may wait longer to complete the *receive* than they otherwise would have to, but ultimately, the same actions should be accomplished.

Suspending one process for disk I/O can affect other processes which continue to run, in a different manner. The ALSTEN *receive* can specify several alternatives; in effect it can be non-deterministic; receiving from port sets is actually non-deterministic, since the programmer can not know which element of the set will be used. For instance, if there are three processes A, B, and C, and process B was supposed to receive a message from process A, but A was suspended, B could end up receiving a message from process C instead. This should not affect the ultimate semantics of the program, since the *receive* could happen on any specified port. it merely changes the path by which the program arrives at its goal.

One practical problem we encountered in initially using our recording and replay system concerned programs which had to be aborted due to a loop in one or more processes. Simply having Accent abort the processes caused the event files they produced to be discarded. It was necessary to build a special capability into the root process representing the Pronet runtime system to have it terminate the processes in an orderly manner. The basic lesson here is that any passive monitor must make sure that it saves information in a way that will keep that information available under adverse circumstances, because that is just when the information will be needed.

6.2 Graphical Replay of Program Execution

The Perqs have high-resolution, bit-mapped black-and-white displays. This feature gave considerable support to the development of a very effective graphical user interface to our monitoring system. We consider this interface one of the most successful aspects of the project.

In the introduction, we noted that one of the most difficult aspects of designing a tool to support distributed programming debugging was finding a comprehensible way to display information about the program to a user. The graphical replay provided by Radar attacks this problem by providing an abstract view of the behavior of the individual processes. The information provided by the replay involves only activities at the "network" level: process creation and deletion, establishment of connections between ports, message sending and retrieving, etc. None of these activities is exclusively concerned with the internal state of a single process. Thus the replay provides a user with a view of program execution at the "process interaction" level. Only after an erroneous pattern of interaction is identified at that level is it necessary to consider the internal details of any of the processes.

The alternative approach, only possible for a more intrusive debugger active during actual program execution, would be to provide a multi-window display, with each window displaying state information about and allowing interaction with a single process. For programs with more than a few processes, all of the windows wouldn't fit on the screen at the same time. Further, so much detail about individual process activity would be available that it would be virtually impossible to perceive the higher level structure that our replay system makes so apparent. Thus, given our linkage to a single-process debugger, we believe that our more abstract representation of program execution is a superior design choice.

After the program has completed executing, Radar is invoked to provide a graphical "replay" of the execution. Each message or event is stamped with an event number, imposing a partial ordering on events. The monitor then displays events one at a time. The programmer is able to watch the communications traffic amongst the processes. Processes have names in

Pronet, so it is easy for the programmer to see which process is communicating with which other processes.

Radar provides the user with the ability to view the contents of any message currently represented on the screen. Messages are represented on the screen as small boxes. The user places the PERQ's mouse over the message which he wishes to examine. Radar then displays the contents of the chosen message in a formatted fashion. For instance, if the message contained an integer and two floating point numbers, the message would be displayed as an integer and two floating point numbers, not as 10 octal bytes.

Radar also provides the ability to replay a certain number of events which have already happened. This can be done at any point during the display. The user can "rewind the video tape," so to speak. This replay is limited to a fixed maximum number of previous events. The user also has the choice of watching a continuous stream of events (occurring at an interactively settable rate), or single-stepping through events. This prevents information from flowing too fast to be comprehended.

Finally, as a separate utility, the user can name a given process and have all of the messages which were sent to that process selected from the recorded message traffic. This single process may then be run by itself with its messages derived from the stored messages. This feature is designed to facilitate single process debugging under realistic conditions, without having to worry about controlling the rest of the processes of the distributed program.

6.3 Integration with a Single-Process Debugger

Only task 4 went just as it was originally planned. Our program replay system interfaces with the Kraut debugger, which is a standard high-level debugger under Accent. All of the messages to a single process can be collected from the event files. Then that process may be executed again, along with a special driver that simulates the rest of the program. Note that the selected process is actually executed, not simulated; however, the rest of the program is

simulated. The driver simulates the rest of the program by providing messages received by the selected process as they are needed. Thus the process under examination should execute just as it did when the event files were originally collected.

The debugging methodology these mechanisms support works as follows. A program is executed with event files being collected. Its execution is replayed by Radar until the user identifies some particular process as exhibiting inappropriate behavior. Such behavior might be such things as inappropriate or missing message transmission, incorrect contents in a message or any other event visible at the network level. The user then asks for a re-execution of that process and examines its internal state during this replay using Kraut. Whenever the process executes a message receive statement, the Radar driver supplies the appropriate message. Whenever the process sends a message, the driver discards it. This process continues until the cause of the inappropriate behavior can be determined and (hopefully) corrected.

There is only one problem with the above scenario. ALSTEN includes a conditional receive statement which allows the program to go on executing rather than blocking if it tries to receive a message and none is available in its incoming message buffer. Such an unsuccessful attempt to receive is not an externally visible event and thus was not originally recorded in the event files. During re-execution with the special Radar driver and Kraut, messages are always available upon request. Thus a process whose execution originally included unsuccessful conditional receives would not execute in exactly the same way during re-execution. We found it necessary to begin recording unsuccessful conditional receives so that it would be possible to faithfully re-execute processes in this situation.

The ability to examine program execution at the two different levels of abstraction provided by Radar and Kraut provide a very effective technique for tackling the information overload problem of monitoring distributed programs. This idea of replaying a process using stored messages has also appeared recently in a slightly different context: crash recovery in a message based distributed system ([Borg83] and [Powe83]).

6.4 Approaches Taken by Other Researchers

Bates and Wileden [Bate83] take the approach of viewing the 'Behavioral Abstraction' of a program's execution. Basically, the system is viewed 'in terms of its activity rather than its state.' They provide for primitive events such as process creation, page faulty, message transmission, and message reception. Higher level events or 'event abstractions' are built up by designating sequences of primitive events. The debugger then recognizes higher level events and displays these for the programmer, while filtering out other unimportant events.

Gross and Zwaenepoel [Gros83] discuss those aspects of a distributed system both necessary and desirable for easy debugging. They do not present an actual debugging system. The system they propose would support the debugger as a separate process, with kernel facilities which would allow the debugger control over the program's execution, memory and kernel calls. They also make a distinction between the *micro level* of execution, which is the computations made by each process, and the *macro level*, where the overall computation proceeds via messages passed amongst the processes.

Schiffenbauer [Schi81] presents an ambitious project implemented on a network of Xerox Alto minicomputers. He gives an introduction to the problems of distributed debugging and then a discussion of the major issues in designing a debugging facility. The two major issues are *transparency* of the debugger (a practical consideration), and the theoretical consideration of *causality* and logical clocks. He then described the implementation of his debugging facility. One of the more important parts of his work is his implementation of 'logical clocks' and his proof that through the use of logical (rather than actual) clocks, his debugger simulates a valid execution of the distributed program. He further proves that the debugger simulates a probable execution of the program, i.e. that the program behaves the same while being debugged as it probably would have behaved had it been allowed to execute unmonitored.

Curtis and Wittie [Curt82] present their design of a debugging system for parallel programming environments. A parallel programming environment is either a conventional

multiprogramming single processor system, or a 'network computer,' an ensemble of semiautonomous nodes, each with its own memory, peripherals, and communication links. The nodes communicate by passing messages over their links.

The debugging system consists of local event monitors on each node, a central database system, and a user interface. The user interface is based on production rules, which the user expands into sequences of symbols describing what events he wants recorded, what variables saved, and what actions are to be taken upon the occurrence of any given event. This debugger, like that of Bates and Wileden, must be programmed.

Harter [Hart85] proposes a debugging system which includes a standard sequential debugger plus an assertion language, based on temporal logic, to control the automatic monitoring of distributed programs. The system allows a programmer to expand the assertion set interactively. It also includes a graphics interface to display and filter information about program execution.

Our work described below attempts to present a higher level view of message traffic that Schiffenbauer's minimally intrusive view of program execution. We agree with and support the distinction between micro and macro levels of execution suggested by Gross & Zwaenepoel. The interface to our system is simpler than those provided by Harter, Curtis & Wittie and Bates & Wileden, since it need not be programmed.

6.5 Possibilities for Further Research

When a user watches the replay of a program using Radar, he quickly begins to recognize "patterns" consisting of sequences of several events. It would be desirable if Radar had some capability to display execution in terms of such higher-level events. An important question is how such structuring might be made to take place.

6.5.1 A Common Structuring Methodology

One very prevalent and well understood method of imposing structure on incoming information is via lexical analysis and/or parsing techniques. These techniques are well understood, and often easy to use.

Breugege [Breu84] uses Path Expressions, an extension of Regular Expressions. A path expression describes a sequence of events to be looked for, and actions to be executed when that sequence is matched, or not matched. The notation provides good flexibility of description, and would seem to supply a good method for RADAR to use for dealing with its stream of Pronet events.

6.5.2 The Problems with Path Expressions

Path expressions, or more generally, regular expressions and LALR(1) parsing techniques, are a natural first choice for the computer scientist wishing to impose structure on a data stream. Here however, it may be a case of using a useful, but inappropriate, tool for the job.

Why? In this case, the major flaw with these techniques, particularly path expressions, is that they are *predictive*. The debugging programmer must describe what he *expects* the debugger to see, and then what to do. But if a program is bug-ridden, it may never do what the programmer expects it to, even if he is looking for aberrant behavior! So, an interesting and possibly important stream of events could conceivably end up being missed by the debugger, and therefore by the programmer. In sum, a debugger should present a distillation of what happened, not what the programmer expected to happen.

A secondary, although in our view still major, flaw is that this kind of debugger has to be programmed. The user must learn (and remember!) yet another kind of notation, and yet another set of commands. If a debugger is hard to use, it may not get used at all. One of the major RADAR design goals was that it should not have to be programmed.

6.5.3 Using a Data Compression Approach

If regular expressions, LALR(1) grammars, and path expressions are not the answer, what is? For the reasons we are about to present, we feel that an approach based on data compression would be an interesting area for future research.

6.5.4 Why a Data Compression Approach?

When one stops to think about it, it becomes clear that the problem is really one of data compression. We want to replace sequences of low level events with a shorter symbol that represents that sequence. This is exactly what data compression techniques do, although usually they are just acting upon simple byte streams.

The shorter symbol can be given a name that describes the sequence in a "higher level" fashion. For example, replace the sequence "find Fred's number in the phone book", "lift the phone handset", "listen for dial tone", and "dial the number", with, "call Fred".

This approach has several advantages. First, it is not predictive, looking for one thing and missing another. Instead, it is empirical, condensing what is there. It represents *all* the event sequences as they happened. Second, it fits in very well with RADAR's current passive, post-mortem approach to program monitoring. Third, the machine does the work of detecting event sequences and condensing them, not the programmer. There are no new notations or commands to learn.

6.5.5 Possible Implementation Plan

There are numerous data compression techniques. A recently developed, and very powerful technique is the Adaptive Lempel-Ziv Compression described in [Welc84]. On "normal" files of English text it often achieves compression of 50% or greater. One of its strongest points is that it tends to compress the longest possible sequence into a single code.

RADAR gives unique identifies (numbers) to each kind of Pronet message. A single RADAR event would consist of the sending process id, the receiving process id, and the

message type. These events should be representable as unique integers of at most two bytes. A first (conceptual) pass over the recorded data would build a table of event triples (sender, destination, message type) and their corresponding integers.

Next, the second pass performs Adaptive Lempel-Ziv compression on the integer stream, saving the compressed output. As part of the compression algorithm, the Lempel-Ziv method builds a table of codes and what each code represents.

After compression, this table is presented to the user. RADAR presents each sequence and asks for a high level name for that sequence ("call Fred").

Once that is done, the compressed data is then "decompressed"; but not back into an integer stream. Instead, as each higher level code is recognized, the corresponding high level event is displayed graphically on the screen.

6.5.6 Problems with This Approach

The method outlined above is not without its problems. In particular, the ordering of events that RADAR imposes is only a partial ordering. Events are sometimes depicted on the screen in an order different from that in which they actually occurred. Only related events are guaranteed to be ordered. This is because RADAR currently works by merging multiple event streams into a single event stream for display. The problem with this approach is that nonrelated events end up being interleaved with each other. This could conceivably affect the data compression algorithm. Non-related events could be compressed together, i.e. treated as related! (instead of being compressed with their related events).

A major thrust of any future research would be to see if a data compression approach is feasible, and to learn whether or not non related interleaved events would detrimentally affect the data compression, or if the nature of the algorithm is such that it would not matter. Another goal would be to see if some approach could be found to work directly from the original multiple data streams, instead of from the merged single data stream.

As an alternative, some sort of knowledge-based pattern recognition approach might be tried. The data compression approach is essentially syntactic; a pattern recognition mechanism could conceivably work better by making use of information in messages or about network interconnections. Relative computational demands of these two approaches are an obvious tradeoff.

6.6 Conclusions

Finally, we restate our principal conclusions:

- Graphical display of information is an excellent technique for providing information about the execution of a distributed program.
- Passive monitoring and simulated replaying is a successful approach for minimizing the impact of the monitor on the execution of the program under examination.
- Multi-level tools are required to deal effectively with all aspects of distributed program debugging.

We must state that these conclusions are based on relatively little experience with Radar. Because Accent has not been as stable as we had anticipated, there is really no user community on the Perqs other than the people who have worked on the Radar project. A much more extensive evaluation of our tools would be highly desirable.

BIBLIOGRAPHY

- [3RCC82] *Perq System Software Reference Manual*; Three Rivers Computer Corp.; Pittsburgh, Pa., May 1982.
- [ACM83] *Proceedings of the ACM Symposium on High Level Debugging; SIGPLAN Notices*, Vol. 18, No. 8, August 1983.
- [Bate83] "An Approach to High Level Debugging of Distributed Systems"; P.C. Bates, J.C. Wiledon; *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on High Level Debugging SIGPLAN Notices*, Vol. 18, No. 8, August 1983, pp. 107-111.
- [Borg83] "A Message System Supporting Fault Tolerance"; A. Borg, J. Baumbach and S. Glazer; *Ninth ACM Symposium on Operating Systems Principles*, October 1983, pp. 90-99.
- [Breu83] "Generalized Path Expressions - A High Level Debugging Mechanism"; B. Breugege, P. Hibbard; *Journal of Systems and Software*, Vol. 3, 265-276.
- [Curt82] "Bugnet: A Debugging System for Parallel Programming Environments"; R. Curtis, L. Wittie; *Proceedings of 3rd International Conference on Distributed Computing Systems*, Fort Lauderdale, Florida, August, 1982, pp. 394-399.
- [Gross83] "System Support For Multi-Process Debugging"; T. Gross, W. Zwaenepoel; *Conference Preprints from the ACM SIGSOFT/SIGPLAN Symposium on High level Debugging*, March, 1983; pp. 192-196.
- [Hart85] "IDD: An Interactive Distributed Debugger"; P. K. Harter, Jr., D M. Heimbigner, R. King; *Preceedings of The 5th International Conference on Distributed Computing Systems*, Denver, Colorado, May, 1985, pp. 498-506.
- [Jens74] *Pascal User Manual and Report*; K. Jensen, N. Wirth; Springer-Verlag, 1974.

- [Live80] *Run-Time Control in a Transaction Oriented Environment*; N. J. Livesey; Ph.D. Thesis, University of Waterloo 1980.
- [Macc82] *Language Features For Fully Distributed Processing Systems*; A. B. Maccabe; Ph.D. Thesis; Technical Report GIT-ICS 82/12, School of Information and Computer Science, Georgia Institute of Technology, August 1982.
- [Powe83] "Publishing: A Reliable Broadcast Communications Mechanism"; M. L. Powell and D. L. Presotto, *Ninth ACM Symposium on Operating System Principles*, October 1983, pp. 100-109.
- [Schi81] *Interactive Debugging In A Distributed Computational Environment*; R. D. Schiffenbauer; Master's Thesis, Massachusetts Institute of Technology, August 1981.
- [Welc84] "A Technique for High-Performance Data Compression," T. A. Welch, *IEEE Computer*, Vol. 17, No. 6, pp 8-19.
- ```
1 network proc =
2 init_cbase0 =
3 proc_decl_10 =
4 %arrive and enter
5 %proc_decl_11 =
6 %process ;
7 evnt_cbase_10 =
8 %arrive enter leave
9 evnt_cbase_11 =
10 %arrive enter leave
11 init_cbase0 =
12 %initial activity
13 const_pt =
14 const_pt = const con_def_list
```

## APPENDIX A

## The LL(1) Grammar of NETSLA

Grammar productions with selection sets added:

Prod #      Production

- 1    network\_spec = net\_head    const\_pt type\_pt port\_decl\_pt  
                  evnt\_decl\_pt proc\_decl\_10    evnt\_clse\_10  
                  init\_clse0 end identifier  
                  %network ;
- 2    net\_head = network    identifier ;  
              %network ;
- 3    proc\_decl\_10 =  
              %arrive end enter initial leave when ;
- 4    proc\_decl\_10 = process\_decl proc\_decl\_11  
              %process ;
- 5    proc\_decl\_11 =  
              %arrive end enter initial leave when ;
- 6    proc\_decl\_11 = process\_decl proc\_decl\_11  
              %process ;
- 7    evnt\_clse\_10 =  
              %end initial ;
- 8    evnt\_clse\_10 = event\_clause evnt\_clse\_11  
              %arrive enter leave when ;
- 9    evnt\_clse\_11 =  
              %end initial ;
- 10   evnt\_clse\_11 = event\_clause evnt\_clse\_11  
              %arrive enter leave when ;
- 11   init\_clse0 =  
              %end ;
- 12   init\_clse0 = initial\_activity\_lst  
              %initial ;
- 13   const\_pt =  
              %arrive end enter event initial leave  
              port process type when ;
- 14   const\_pt = const    con\_def\_list

```
%const ;

15 con_def_list = const_def next_con_def
 %identifier ;

16 next_con_def =
 %arrive end enter event initial leave
 port process type when ;

17 next_con_def = const_def next_con_def
 %identifier ;

18 const_def = new_const_id = constant ;
 %identifier ;

19 new_const_id = identifier
 %identifier ;

20 constant = signed_const
 %+ - ;

21 constant = unsigned_con
 %char_const identifier int_const real_const string_const ;

22 signed_const = sign after_sign
 %+ - ;

23 after_sign = real_const
 %real_const ;

24 after_sign = int_const
 %int_const ;

25 after_sign = const_id
 %identifier ;

26 unsigned_con = identifier
 %identifier ;

27 unsigned_con = int_const
 %int_const ;

28 unsigned_con = char_const
 %char_const ;

29 unsigned_con = string_const
 %string_const ;

30 unsigned_con = real_const
 %real_const ;

31 scalar_const = identifier
 %identifier ;
```

```

32 scalar_const = non_id_s_con
 %+ - char_const int_const ;

33 non_id_s_con = sign_id_or_int
 %+ - . ;

34 non_id_s_con = int_const
 %int_const ;

35 non_id_s_con = char_const
 %char_const ;

36 id_or_int = const_id
 %identifier ;

37 id_or_int = int_const
 %int_const ;

38 const_id = identifier
 %identifier ;

39 type_pt =
 %arrive end enter event initial leave
 port process when ;

40 type_pt = type typ_def_list
 %type ;

41 typ_def_list = type_def next_typ_def
 %identifier ;

42 next_typ_def =
 %arrive end enter event initial leave
 port process when ;

43 next_typ_def = type_def next_typ_def
 %identifier ;

44 type_def = new_type_id = types ;
 %identifier ;

45 new_type_id = identifier
 %identifier ;

46 types = type_case1
 %identifier ;

47 types = type_case2
 %(+ - array char_const int_const
 packed record set ;

48 type_case1 = identifier type_tail

```

```

 %identifier ;

49 type_tail =
 %); case end ;

50 type_tail = .. scalar_const
 %.. ;

51 type_case2 = non_id_s_con .. scalar_const
 %+ - char_const int_const ;

52 type_case2 = struct_type
 %array packed record set ;

53 type_case2 = (enu_id_list)
 %(;

54 non_id_type = non_id_simp
 %(+ - char_const identifier int_const ;

55 non_id_type = struct_type
 %array packed record set ;

56 simple_type = type_id simp_ty_tail
 %identifier ;

57 simple_type = (enu_id_list)
 %(;

58 simple_type = non_id_s_con .. scalar_const
 %+ - char_const int_const ;

59 simp_ty_tail =
 %), ;] case end ;

60 simp_ty_tail = .. scalar_const
 %.. ;

61 non_id_simp = (enu_id_list)
 %(;

62 non_id_simp = subrange_con .. scalar_const
 %+ - char_const identifier int_const ;

63 pt_class nam = identifier
 %identifier ;

64 enu_id_list = identifier enumer_tail.
 %identifier ;

65 enumer_tail =
 %) ;

```



```
66 enumer_tail = , identifier enumer_tail
 %, ;

67 subrange con = identifier
 %identifier ;

68 subrange con = non_id_s_con
 %+ - char_const int_const ;

69 type_id = identifier
 %identifier ;

70 struct_type = pack_prefix unpacked
 %array packed record set ;

71 pack_prefix = packed
 %packed ;

72 pack_prefix =
 %array record set ;

73 unpacked = array [indx_ty_list] of
 types
 %array ;

74 unpacked = record_head field_list end
 %record ;

75 unpacked = set of simple_type
 %set ;

76 record_head = record
 %record ;

77 indx_ty_list = simple_type index_tail
 %(+ - char_const identifier int_const ;

78 index_tail =
 %] ;

79 index_tail = , simple_type index_tail
 %, ;

80 field_list = rec_sec_list with_variant
 %) ; case end identifier ;

81 rec_sec_list = rec_section rec_sec_tail
 %) ; case end identifier ;

82 rec_sec_tail =
 %) case end ;

83 rec_sec_tail = ; rec_section rec_sec_tail
```

```

 %; ;
84 rec_section = fieldid_list : types
 %identifier ;
85 rec_section =
 %); case end ;
86 fieldid_list = identifier field_id_end
 %identifier ;
87 with_variant =
 %)_end ;
88 with_variant = variant_pref variant_list
 %case ;
89 field_id_end =
 %: ;
90 field_id_end = , identifier field_id_end
 %, ;
91 variant_pref = case tag_type_ids of
 %case ;
92 tag_type_ids = tagfield_id tag_typ_tail
 %identifier ;
93 tag_typ_tail =
 %of ;
94 tag_typ_tail = : scalar_ty_id
 %: ;
95 tagfield_id = identifier
 %identifier ;
96 scalar_ty_id = identifier
 %identifier ;
97 variant_list = variant variant_tail
 %) + - ; char const end
 identifier int_const ;
98 variant = case_1_list : (field_head field_list
)
 %+ - char_const identifier int_const ;
99 variant =
 %) ; end ;
100 field_head =

```

```

 %); case identifier ;
101 variant_tail =
 %); end ;
102 variant_tail = ; variant variant_tail
 %; ;
103 case_1_list = scalar_const caselabelend
 %+ - char_const identifier int_const ;
104 caselabelend =
 %: ;
105 caselabelend = , scalar_const caselabelend
 %, ;
106 port_decl_pt =
 %arrive end enter event initial leave
 process when ;
107 port_decl_pt = pt_decl_list
 %port ;
108 pt_decl_list = port_decl pt_decl_tail
 %port ;
109 port_decl = port_head pt_dir_mtype
 %port ;
110 pt_dir_mtype = in type_id ;
 %in ;
111 pt_dir_mtype = out type_id ;
 %out ;
112 pt_dir_mtype = port_group ;
 % (;
113 pt_decl_tail =
 %arrive end enter event initial leave
 process when ;
114 pt_decl_tail = port_decl pt_decl_tail
 %port ;
115 port_head = port port_tail
 %port ;
116 port_tail = identifier
 %identifier ;
117 port_tail = set identifier

```

```

 %set ;
118 port_group = (sbptdecllist)
 %(- ;
119 sbptdecllist = subport_decl next_subport
 %identifier ;
120 subport_decl = subport_name direct_type
 %identifier ;
121 direct_type = in type_id
 %in ;
122 direct_type = out type_id
 %out ;
123 subport_name = identifier
 %identifier ;
124 next_subport =
 %)- ;
125 next_subport = ; subport_decl next_subport
 %; ;
126 process_decl = process_head attri_decls0 port_decl_pt evnt_decl_pt
 end Identifier
 %process ;
127 process_head = process class identifier
 %process ;
128 attri_decls0 =
 %end event port ;
129 attri_decls0 = attri_head attri_sec_ls attri_tail
 %attributes ;
130 attri_head = attributes
 %attributes ;
131 attri_tail = end attributes
 %end ;
132 attri_sec_ls = attri_sec attri_secl
 %; end identifier ;
133 attri_secl =
 %end ;
134 attri_secl = ; attri_sec
 %; ;

```

```
135 attri_sec = attri_id_ls : types
 %identifier ;

136 attri_sec =
 %; end ;

137 attri_id_ls = identifier attri_id_ls1
 %identifier ;

138 attri_id_ls1 =
 %: ;

139 attri_id_ls1 = , identifier
 %, ;

140 evnt_decl_pt =
 %arrive end enter initial leave process
 when ;

141 evnt_decl_pt = event_decl next_event
 %event ;

142 next_event =
 %arrive end enter initial leave process
 when ;

143 next_event = event_decl next_event
 %event ;

144 event_decl = event identifier about_ptrm0 ;
 %event ;

145 about_ptrm0 =
 %; ;

146 about_ptrm0 = about identifier
 %about ;

147 event_clause = arriv_clause
 %arrive ;

148 event_clause = enter_clause
 %enter ;

149 event_clause = leave_clause
 %leave ;

150 event_clause = when_clause
 %when ;

151 arriv_clause = arrive_head activity_lst close end arrive
 %arrive ;
```

```
152 arrive_head = arrive open arrive_bind do
 %arrive ;
153 arrive_bind = message_id0 on arrive_port from_proces0
 %identifier on ;
154 message_id0 =
 %on ;
155 message_id0 = identifier
 %identifier ;
156 arrive_port = identifier arrive_port1
 %identifier ;
157 arrive_port1 =
 %do from ;
158 arrive_port1 = : identifier
 %: ;
159 arrive_port1 = of port_bind
 %of ;
160 port_bind = identifier port_bind1
 %identifier ;
161 port_bind1 =
 %do from ;
162 port_bind1 = : identifier
 %: ;
163 from_proces0 =
 %do ;
164 from_proces0 = from process_bind
 %from ;
165 process_bind = identifier proces_bind1
 %identifier ;
166 proces_bind1 =
 %about do ;
167 proces_bind1 = : identifier
 %: ;
168 enter_clause = enter_head activity_1st close end enter
 %enter ;
169 enter_head = enter open port_bind do
```



```

 %enter ;
170 leave_clause = leave_head activity_1st close end leave
 %leave ;
171 leave_head = leave open port_bind do
 %leave ;
172 when_clause = when_head activity_1st close end when
 %when ;
173 when_head = when open identifier announced by process_bind
 about_part0 do
 %when ;
174 about_part0 =
 %do ;
175 about_part0 = about port_bind
 %about ;
176 activity_1st = activity activities
 %) ; announce case connect construct
 create disconnect else end find identifier
 range remove send terminate ;
177 activities =
 %) else end ;
178 activities = ; activity activities
 %; ;
179 activity =
 %) ; else end ;
180 activity = simple_act
 %announce connect construct create disconnect identifier
 remove send terminate ;
181 activity = control_act
 %case find range ;
182 simple_act = creation
 %create ;
183 simple_act = termination
 %terminate ;
184 simple_act = removal
 %remove ;
185 simple_act = connection
 %connect ;

```

```
186 simple_act = disconnecton
 %disconnect ;

187 simple_act = msg_transfer
 %send ;

188 simple_act = construction
 %construct ;

189 simple_act = attri_assign
 %identifier ;

190 simple_act = event_trans
 %announce ;

191 simple_bind = object_id : identifier simple_bind1
 %identifier ;

192 object_id = identifier
 %identifier ;

193 simple_bind1 =
 %do where ;

194 simple_bind1 = on proc_denoter
 %on ;

195 obj_denoter = lhs
 %identifier ;

196 port_denoter = obj_denoter
 %identifier ;

197 proc_denoter = identifier
 %identifier ;

198 creation = create create_tail
 %create ;

199 create_tail = identifier : identifier create_tail1
 %identifier ;

200 create_tail1 =
 %); else end ;

201 create_tail1 = on proc_denoter
 %on ;

202 termination = terminate proc_denoter
 %terminate ;

203 removal = remove obj_denoter
```

```

 %remove ;

204 connection = connect port_denoter to port_denoter
 %connect ;

205 disconnecton = disconnect port_denoter from_port0
 %disconnect ;

206 from_port0 =
 %); else end ;

207 from_port0 = from port_denoter
 %from ;

208 msg_transfer = send expr0 to port_denoter
 %send ;

209 expr0 =
 %to ;

210 expr0 = expr
 %(+ - [char_const identifier
 int_const not real_const string_const ;

211 construction = construct_hd [field_as_lst]
 %construct ;

212 construct_hd = construct object_id : identifier
 %construct ;

213 field_as_lst = field_assign fd_assign1
 %identifier ;

214 fd_assign1 =
 %] ;

215 fd_assign1 = ; field_assign
 %; ;

216 field_assign = lhs := expr
 %identifier ;

217 attri_assign = lhs := expr
 %identifier ;

218 event_trans = announce event_id about_port0
 %announce ;

219 about_port0 =
 %); else end ;

220 about_port0 = about port_denoter

```

```
 %about ;

221 control_act = alternation
 %case ;

222 control_act = selection
 %find ;

223 control_act = iteration
 %range ;

224 alternation = alternate_hd case_list else_part0 end case
 %case ;

225 alternate_hd = case expr of
 %case ;

226 case_list = case_element case_list1
 %+ - char_const identifier int_const ;

227 case_list1 =
 %else end ;

228 case_list1 = case_element case_list1
 %+ - char_const Identifier int_const ;

229 case_element = const_list : (open activity_lst close
)
 %+ - char_const identifier int_const ;

230 const_list = scalar_const const_list1
 %+ - char_const identifier int_const ;

231 const_list1 =
 %: ;

232 const_list1 = , scalar_const
 %, ;

233 select_crite = simple_bind where_claus0
 %identifier ;

234 selection = find_head do activity_lst close else_part0 end
 find
 %find ;

235 find_head = find open object_id : find_head1
 %find ;

236 find_head1 = string
 %string ;

237 find_head1 = identifier simple_bind1 where_claus0
```

```

 %identifier ;
238 iteration = range open select_crite do activity_lst close
 else_part0 end range
 %range ;
239 else_part0 =
 %end ;
240 else_part0 = else open activity_lst close
 %else ;
241 where_claus0 =
 %do ;
242 where_claus0 = where expr
 %where ;
243 open =
 %) ; announce case connect construct
 create disconnect end find identifier on
 range remove send terminate ;
244 close =
 %) else end ;
245 id_list = identifier id_list_tail
 %identifier ;
246 id_list_tail =
 % ;
247 id_list_tail = , identifier id_list_tail
 %, ;
248 actual_parms = (actual_parm next_a_parm
 %(;
249 actual_parm = parm_expr field_width
 %(+ - [char_const identifier
 int_const not real_const string_const ;
250 next_a_parm =
 %) ;
251 next_a_parm = , actual_parm next_a_parm
 %, ;
252 lhs = identifier rec_ary_ptr
 %identifier ;
253 vars = identifier rec_ary_ptr
 %identifier ;

```

```

254 rec_ary_ptr =
 %} * + , - ..
 / : := ; =]
 and div do else end from
 in mod noneqrelop of or to ;

255 rec_ary_ptr = . identifier rec_ary_ptr
 % . ;

256 rec_ary_ptr = [index_list] rec_ary_ptr
 % [;

257 index_list = index next_index
 % (+ - [char_const identifier
 int_const not real_const string_const ;

258 next_index = , index
 % , ;

259 next_index =
 %] ;

260 index = expr
 % (+ - [char_const identifier
 int_const not real_const string_const ;

261 expr = parm expr
 % (+ - [char_const identifier
 int_const not real_const string_const ;

262 parm expr = simple_expr parm exp_end
 % (+ - [char_const identifier
 int_const not real_const string_const ;

263 parm_exp_end =
 %) , .. : ;]
 do else end of to ;

264 parm_exp_end = rel_op simple_expr
 % = in noneqrelop ;

265 rel_expr = simple_expr rel_op simple_expr
 % { + - [char_const identifier
 int_const not real_const string_const ;

266 rel_op = =
 % = ;

267 rel_op = in
 % in ;

268 rel_op = noneqrelop

```



```

 %noneqrelop ;

269 simple_expr = char_const add_term
 %char_const ;

270 simple_expr = string_const add_term
 %string_const ;

271 simple_expr = sign term add_term
 %+ - ;

272 simple_expr = term add_term
 %([identifier int_const not real_const ;

273 add_term =
 %) , .. : ; =
] do else end in noneqrelop
 of to ;

274 add_term = add_op term add_term
 %+ - or ;

275 term = factor mult_factor
 %([identifier int_const not real_const ;

276 mult_factor =
 %) + , - .. :
 ; =] do else end
 in noneqrelop of or to ;

277 mult_factor = mult_op factor mult_factor
 %* / and div mod ;

278 factor = identifier var_funccall
 %identifier ;

279 factor = real_const
 %real_const ;

280 factor = int_const
 %int_const ;

281 factor = (expr)
 %(;

282 factor = [elem_list]
 %[;

283 factor = not factor
 %not ;

284 var_funccall = rec_ary_ptr
 %) * + , - .

```

```

 .. / : ; = [
] and div do else end
 in mod nonegrelop of or to ;

285 var funccall = actual_parms)
 %(;

286 add_op = sign
 %(+ - ;

287 add_op = or
 %(or ;

288 mult_op = *
 %(* ;

289 mult_op = /
 %(/ ;

290 mult_op = div
 %(div ;

291 mult_op = and
 %(and ;

292 mult_op = mod
 %(mod ;

293 variable = identifier rec_ary_ptr
 %(identifier ;

294 field_width =
 %(, ;

295 field_width = : expr more_field
 %: ;

296 more_field =
 %(, ;

297 more_field = : expr
 %: ;

298 elem_list =
 %] ;

299 elem_list = elem next elem
 %(+ - [char_const Identifier
 int_const not real_const string_const ;

300 elem = expr elem_tail
 %(+ - [char_const identifier
 int_const not real_const string_const ;

```

```
301 next_elem =
 %] ;
302 next_elem = , elem next_elem
 % , ;
303 elem_tail =
 % ,] ;
304 elem_tail = .. expr
 % .. ;
305 proc_id = identifier
 % identifier ;
306 rec_var_list = variable next_rec_var
 % identifier ;
307 next_rec_var =
 % ;
308 next_rec_var = , variable next_rec_var
 % , ;
309 subport =
 % ;
310 subport = . subport_id
 % . ;
311 pt_class_id = identifier
 % identifier ;
312 subport_id = identifier
 % identifier ;
313 expression0 =
 % ;
314 expression0 = expr
 % (+ - [char_const identifier
 int_const not real_const string_const ;
315 event_id = identifier
 % identifier ;
316 sign = +
 % + ;
317 sign = -
 % - ;
```

## APPENDIX B

## The LL(1) Grammar of ALSTEN

Grammar productions with selection sets added:

Prod #      Production

- 1      comp\_unit =    prog\_head prog  
                  %@ process ;
- 2      prog\_head = process script    prog\_id ;  
                  %process ;
- 3      prog\_id =    identifier  
                  %identifier ;
- 4      prog = port\_decl\_pt label\_pt const\_pt type\_pt evnt\_decl\_pt var\_pt  
                  proc\_fct\_pt stmt\_pt .  
                  %begin const event function label port  
                  procedure type var ;
- 5      block = label\_pt const\_pt type\_pt var\_pt proc\_fct\_pt stmt\_pt  
                  %begin const function label procedure type  
                  var ;
- 6      label\_pt = label    label\_list ;  
                  %label ;
- 7      label\_pt =  
                  %begin const event function procedure type  
                  var ;
- 8      label\_list = labels next\_label  
                  %identifier int\_const ;
- 9      next\_label =  
                  % ; ;
- 10     next\_label = ,    labels next\_label  
                  %, ;
- 11     labels =    int\_const  
                  %int\_const ;
- 12     labels =    identifier  
                  %identifier ;
- 13     const\_pt =  
                  %begin event function procedure type var ;

```
14 const_pt = const con_def_list
 %const ;

15 con_def_list = const_def next_con_def
 %identifier ;

16 next_con_def =
 %begin_event function procedure type var ;

17 next_con_def = const_def next_con_def
 %identifier ;

18 const_def = new_const_id = constant ;
 %identifier ;

19 new_const_id = identifier
 %identifier ;

20 constant = signed_const
 %+ - ;

21 constant = unsigned_con
 %char_const identifier int_const real_const string_const ;

22 signed_const = sign after_sign
 %+ - ;

23 after_sign = real_const
 %real_const ;

24 after_sign = int_const
 %int_const ;

25 after_sign = const_id
 %identifier ;

26 unsigned_con = identifier
 %identifier ;

27 unsigned_con = int_const
 %int_const ;

28 unsigned_con = char_const
 %char_const ;

29 unsigned_con = string_const
 %string_const ;

30 unsigned_con = real_const
 %real_const ;

31 scalar_const = identifier
 %identifier ;
```

```

32 scalar_const = non_id_s_con
 %+ -char_const int_const ;

33 non_id_s_con = sign_id_or_int
 %+ - ;

34 non_id_s_con = int_const
 %int_const ;

35 non_id_s_con = char_const
 %char_const ;

36 id_or_int = const_id
 %identifier ;

37 id_or_int = int_const
 %int_const ;

38 const_id = identifier
 %identifier ;

39 type_pt =
 %begin event function procedure var ;

40 type_pt = type typ_def_list
 %type ;

41 typ_def_list = type_def next_typ_def
 %identifier ;

42 next_typ_def =
 %begin event function procedure var ;

43 next typ def = type_def next_typ_def
 %identifier ;

44 type def = new_type_id = types ;
 %identifier ;

45 new type id = identifier
 %identifier ;

46 types = type_case1
 %identifier ;

47 types = type_case2
 % (+ - array char_const int_const
 packed ptr record set tag ;

48 type_case1 = identifier type_tail
 %identifier ;

```



```
49 type_tail =
 %); case end ;
50 type_tail = .. scalar_const
 %.. ;
51 type_case2 = non_id_s_con .. scalar_const
 %+ - char_const int_const ;
52 type_case2 = struct_type
 %array packed record set ;
53 type_case2 = ptr identifier
 %ptr ;
54 type_case2 = (enu_id_list)
 %(;
55 type_case2 = tag of pt_class_nam
 %tag ;
56 non_id_type = non_id_simp
 %(+ - char_const identifier int_const
 tag ;
57 non_id_type = struct_type
 %array packed record set ;
58 non_id_type = ptr identifier
 %ptr ;
59 simple_type = type_id simp_ty_tail
 %identifier ;
60 simple_type = (enu_id_list)
 %(;
61 simple_type = non_id_s_con .. scalar_const
 %+ - char_const int_const ;
62 simple_type = tag of pt_class_nam
 %tag ;
63 simp_ty_tail =
 %), ;] case end ;
64 simp_ty_tail = .. scalar_const
 %.. ;
65 non_id_simp = (enu_id_list)
 %(;
66 non_id_simp = subrange_con .. scalar_const
```

```

 %+ - char_const identifier int_const ;

67 non_id_simp = tag of pt_class_nam
 %tag ;

68 pt_class_nam = identifier
 %identifier ;

69 enu_id_list = identifier enumer_tail
 %identifier ;

70 enumer_tail =
 %) ;

71 enumer_tail = , identifier enumer_tail
 % , ;

72 subrange_con = identifier
 %identifier ;

73 subrange_con = non_id_s_con
 %+ - char_const int_const ;

74 type_id = identifier
 %identifier ;

75 struct_type = pack_prefix unpacked
 %array packed record set ;

76 pack_prefix = packed
 %packed ;

77 pack_prefix =
 %array record set ;

78 unpacked = array [indx_ty_list] of
 types
 %array ;

79 unpacked = record_head field_list end
 %record ;

80 unpacked = set of simple_type
 %set ;

81 record_head = record
 %record ;

82 indx_ty_list = simple_type index tail
 % (+ - char_const identifier int_const
 tag ;

83 index_tail =

```

```

84 index_tail = , simple_type index_tail
 % , ;

85 field_list = rec_sec_list with_variant
 %) ; case end identifier ;

86 rec_sec_list = rec_section rec_sec_tail
 %) ; case end identifier ;

87 rec_sec_tail =
 %) case end ;

88 rec_sec_tail = ; rec_section rec_sec_tail
 % ; ;

89 rec_section = fieldid_list : types
 % identifier ;

90 rec_section =
 %) ; case end ;

91 fieldid_list = identifier field_id_end
 % identifier ;

92 with_variant =
 %) end ;

93 with_variant = variant_pref variant_list
 % case ;

94 field_id_end =
 % : ;

95 field_id_end = , identifier field_id_end
 % , ;

96 variant_pref = case tag_type_ids of
 % case ;

97 tag_type_ids = tagfield_id tag_typ_tail
 % identifier ;

98 tag_typ_tail =
 % of ;

99 tag_typ_tail = : scalar_ty_id
 % : ;

100 tagfield_id = identifier
 % identifier ;

```

```

101 scalar_ty id = identifier
 %identifier ;

102 variant list = variant variant_tail
 %)+ - ; char_const end
 identifier int_const ;

103 variant = case_1_list : (field_head field_list
)
 %+ - char_const identifier int_const ;

104 variant =
 %); end ;

105 field_head =
 %); case identifier ;

106 variant_tail =
 %); end ;

107 variant_tail = ; variant variant_tail
 %; ;

108 case_1_list = scalar_const caselabelend
 %+ - char_const identifier int_const ;

109 caselabelend =
 %: ;

110 caselabelend = , scalar_const caselabelend
 %, ;

111 port_decl_pt =
 %begin const event function label procedure
 type var ;

112 port_decl_pt = pt_decl_list
 %port ;

113 pt_decl_list = port_decl pt_decl_tail
 %port ;

114 port_decl = port_head pt_dir_mtype
 %port ;

115 pt_dir_mtype = in type_id ;
 %in ;

116 pt_dir_mtype = out type_id ;
 %out ;

117 pt_dir_mtype = port_group ;
 % (;

```

```
118 pt_decl_tail =
 %begin const event function label procedure
 type var ;
119 pt_decl_tail = port_decl pt_decl_tail
 %port ;
120 port_head = port port_tail
 %port ;
121 port_tail = identifier
 %identifier ;
122 port_tail = set identifier
 %set ;
123 port_group = (sbptdecllist)
 % (;
124 sbptdecllist = subport_decl next_subport
 %identifier ;
125 subport_decl = subport_name direct_type
 %identifier ;
126 direct_type = in type_id
 %in ;
127 direct_type = out type_id
 %out ;
128 subport_name = identifier
 %identifier ;
129 next_subport =
 %) ;
130 next_subport = ; subport_decl next_subport
 % ; ;
131 evnt_decl_pt =
 %begin function procedure var ;
132 evnt_decl_pt = event_decl next_event
 %event ;
133 next_event =
 %begin function procedure var ;
134 next_event = event_decl next_event
 %event ;
```

```

135 event_decl = event event_id about_part0 ;
 %event ;
136 about_part0 =
 %; ;
137 about_part0 = about pt_class_id
 %about ;
138 var_pt =
 %begin function procedure ;
139 var_pt = var var_decl_lst
 %var ;
140 var_decl_lst = var_decl var_decl_end
 %identifier ;
141 var_decl_end =
 %begin function procedure ;
142 var_decl_end = var_decl var_decl_end
 %identifier ;
143 var_decl = id_list : types ;
 %identifier ;
144 proc_fct_pt = identifier
 %begin ;
145 proc_fct_pt = pf_decl_list
 %function procedure ;
146 pf_decl_list = pf_decl pf_decl_tail
 %function procedure ;
147 pf_decl_tail =
 %begin ;
148 pf_decl_tail = pf_decl pf_decl_tail
 %function procedure ;
149 pf_decl = pf_head ; blkorfwd
 %function procedure ;
150 blkorfwd = forward ;
 %forward ;
151 blkorfwd = block ;
 %begin const function label procedure type
 var ;
152 proc_start =

```



```

 %(: ; ;
153 pf_head = procedure proc_id_dec proc_start p_head_tail
 %procedure ;
154 pf_head = function func_id_dec proc_start f_head_tail
 %function ;
155 p_head_tail =
 %; ;
156 p_head_tail = (fpsl)
 %(;
157 f_head_tail =
 %; ;
158 f_head_tail = : parm_type_id
 %: ;
159 f_head_tail = (fpsl) :
 parm_type_id
 %(;
160 proc_id_dec = identifier
 %identifier ;
161 func_id_dec = identifier
 %identifier ;
162 fpsl = f_parm_sect fpsl_tail
 %identifier var ;
163 fpsl_tail =
 %); ;
164 fpsl_tail = ; f_parm_sect fpsl_tail
 %; ;
165 f_parm_sect = parm_group
 %identifier ;
166 f_parm_sect = var parm_group
 %var ;
167 parm_type_id = type_id parm_ty_tail
 %identifier ;
168 parm_type_id = struct_type
 %array packed record set ;
169 parm_type_id = (enu_id_list)
 %(;

```

```

170 parm_type_id = tag of pt_class_nam
 %tag ;

171 parm_type_id = non_id_s_con .. scalar_const
 %+ - char_const int_const ;

172 parm_type_id = ptr identifier
 %ptr ;

173 parm_ty_tail =
 %); ;

174 parm_ty_tail = .. scalar_const
 %.. ;

175 parm_group = id_list : parm_type_id
 %identifier ;

176 id_list = identifier id_list_tail
 %identifier ;

177 id_list_tail =
 %: ;

178 id_list_tail = , identifier id_list_tail
 %, ;

179 body_start =
 %announce begin case for goto identifier
 if int const receive repeat send when
196 while with ;

180 stmt_pt = begin body_start stmt_list end
 %begin ;

181 stmt = label_prefix unlabeled_st
 %announce begin case for goto if
199 int const receive repeat send when while
 with ;

182 stmt = stmt_with_id
 %identifier ;

183 stmt_with_id = identifier asgn_cal_lab
 %identifier ;

184 unlabeled_st = begin stmt_list end
 %begin ;

185 unlabeled_st = goto labels
 %goto ;

```

```

186 unlabeled_st = case_head case_list otherwise_pt end
 %case ;

187 unlabeled_st = repeat stmt_list until expr
 %repeat ;

188 unlabeled_st = if_stmt
 %if ;

189 unlabeled_st = for_stmt
 %for ;

190 unlabeled_st = while_stmt
 %while ;

191 unlabeled_st = with_stmt
 %with ;

192 unlabeled_st = receive_stmt
 %receive when ;

193 unlabeled_st = send_stmt
 %send ;

194 unlabeled_st = announcestmt
 %announce ;

195 asgn_cal_lab = rec_ary_ptr := expr
 % := [ptr ;

196 asgn_cal_lab = actual_parms)
 %(;

197 asgn_cal_lab = : unlabeled_st
 %: ;

198 asgn_cal_lab =
 %; else end otherwise until ;

199 actual_parms = (actual_parm next_a_parm
 %(;

200 actual_parm = parm_expr field_width
 %(+ - [char const identifier
 int_const nil not real_const string_const ;

201 next_a_parm =
 %) ;

202 next_a_parm = , actual_parm next_a_parm
 %, ;

```

```

203 if stmt = if_head stmt if_tail
 %if ;
204 if tail = else stmt
 %else ;
205 if tail =
 %; end otherwise until ;
206 for stmt = for_head do stmt
 %for ;
207 while stmt = while_head stmt
 %while ;
208 with stmt = with_head stmt
 %with ;
209 if head = if expr then
 %if ;
210 while head = while expr do
 %while ;
211 label_prefix =
 %announce begin case for goto if
 receive repeat send when while with ;
212 label_prefix = int_const :
 %int_const ;
213 lhs = identifier rec_ary_ptr
 %identifier ;
214 vars = identifier rec_ary_ptr
 %identifier ;
215 rec_ary_ptr =
 %) * + , - ..
 / : := ; =]
 and div do downto else end
 from in mod noneqrelop of or
 otherwise then to until ;
216 rec_ary_ptr = . identifier rec_ary_ptr
 %. ;
217 rec_ary_ptr = [index_list] rec_ary_ptr
 %[;
218 rec_ary_ptr = ptr rec_ary_ptr
 %ptr ;

```

```

219 index_list = index next_index
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

220 next_index = , index
 % , ;

221 next_index =
 %] ;

222 index = expr
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

223 expr = parm_expr
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

224 parm_expr = simple_expr parm_exp_end
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

225 parm_exp_end =
 %), .. : ;]
 do downto else end of otherwise
 then to until ;

226 parm_exp_end = rel_op simple_expr
 %= in noneqrelop ;

227 rel_expr = simple_expr rel_op simple_expr
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

228 rel_op = =
 %= ;

229 rel_op = in
 %in ;

230 rel_op = noneqrelop
 %noneqrelop ;

231 simple_expr = char_const add_term
 %char_const ;

232 simple_expr = string_const add_term
 %string_const ;

233 simple_expr = sign term add_term
 %+ - ;

234 simple_expr = term add_term

```

```

 %([identifier int_const nil not
 real_const ;

235 add_term =
 %), .. : ; =
] do downto else end in
 noneqrelop of otherwise then to until ;

236 add_term = add_op term add_term
 %+ - or ;

237 term = factor mult_factor
 %([identifier int_const nil not
 real_const ;

238 mult_factor =
 %)+ , - .. :
 ; =] do downto else
 end in noneqrelop of or otherwise
 then to until ;

239 mult_factor = mult_op factor mult_factor
 %*/ and div mod ;

240 factor = identifier var_funccall
 %identifier ;

241 factor = nil
 %nil ;

242 factor = real_const
 %real_const ;

243 factor = int_const
 %int_const ;

244 factor = (expr)
 %(;

245 factor = [elem_list]
 %[;

246 factor = not factor
 %not ;

247 var_funccall = rec_ary_ptr
 %)* + , - .
 .. / : ; = [
] and div do downto else
 end in mod noneqrelop of or
 otherwise ptr then to until ;

248 var_funccall = actual_parms)

```



```

 %(;
249 add_op = sign
 %+ - ;
250 add_op = or
 %or ;
251 mult_op = *
 %* ;
252 mult_op = /
 %/ ;
253 mult_op = div
 %div ;
254 mult_op = and
 %and ;
255 mult_op = mod
 %mod ;
256 variable = identifier rec_ary_ptr
 %identifier ;
257 field_width =
 %) , ;
258 field_width = : expr more_field
 %: ;
259 more_field =
 %) , ;
260 more_field = : expr
 %: ;
261 elem_list =
 %] ;
262 elem_list = elem next_elem
 %(+ - [char_const Identifier
 int_const nil not real_const string_const ;
263 elem = expr elem_tail
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;
264 next_elem =
 %] ;
265 next_elem = , elem next_elem

```

```

 %, ;
266 elem_tail =
 %,] ;
267 elem_tail = .. expr
 %.. ;
268 proc_id = identifier
 %identifier ;
269 stmt_list = stmt more_stmt
 %announce begin case_for goto identifier
 if int_const receive repeat send when
 while with ;
270 more_stmt =
 %end until ;
271 more_stmt = ; stmt more_stmt
 %; ;
272 case_head = case expr of
 %case ;
273 case_list = case_elem case_elems
 %+ - char_const identifier int_const ;
274 case_elems =
 %end otherwise ;
275 case_elems = ; case_elem case_elems
 %; ;
276 case_elem = case_labels : stmt
 %+ - char_const identifier int_const ;
277 otherwise_hd = otherwise :
 %otherwise ;
278 case_labels = scalar_const next_scalar
 %+ - char_const identifier int_const ;
279 next_scalar =
 %: ;
280 next_scalar = , scalar_const next_scalar
 %, ;
281 otherwise_pt =
 %end ;
282 otherwise_pt = otherwise_hd stmt_list

```

```

283 for head = for identifier := expr
 to part expr
 %for ;

284 to part = to
 %to ;

285 to part = downto
 %downto ;

286 rec_var_list = variable next_rec_var
 %identifier ;

287 next_rec_var =
 %do ;

288 next_rec_var = , variable next_rec_var
 % , ;

289 with head = with rec_var_list do
 %with ;

290 receive stnt = simple_rcv
 %receive ;

291 receive stnt = when stnt
 %when ;

292 simple_rcv = receive variable0 from
 port denoter freebinding0
 %receive ;

293 variable0 =
 %from ;

294 variable0 = variable
 %identifier ;

295 port denoter = pt_class_id subport
 %identifier ;

296 subport =
 %; do else end otherwise set
 until use ;

297 subport = . subport_id
 % . ;

298 pt_class_id = identifier
 %identifier ;

```

```
299 subport id = identifier
 %identifier ;

300 freebinding0 =
 %; do else end otherwise until ;

301 freebinding0 = use variable
 %use ;

302 freebinding0 = set variable
 %set ;

303 when_stmt = when_head receives else_part0 end
 %when ;

304 when_head = when
 %when ;

305 receives = receive_pt next_receive
 %; end otherwise receive ;

306 next_receive =
 %end otherwise ;

307 next_receive = ; receive_pt next_receive
 %; ;

308 receive_pt =
 %; end otherwise ;

309 receive_pt = simple_rcv do stmt
 %receive ;

310 else_part0 =
 %end ;

311 else_part0 = otherwise stmt
 %otherwise ;

312 send_stmt = send expression0 to port_denoter
 use_part0
 %send ;

313 expression0 =
 %to ;

314 expression0 = expr
 %(+ - [char_const identifier
 int_const nil not real_const string_const ;

315 use_part0 =
 %; else end otherwise until ;
```

```
316 use_part0 = use variable
 %use ;

317 announceamt = announce event_id about_bind0
 %announce ;

318 event_id = identifier
 %identifier ;

319 about_bind0 =
 %; else end otherwise until ;

320 about_bind0 = about pt_class_id use_part0
 %about ;

321 sign = +
 %+ ;

322 sign = -
 %- ;
```

## APPENDIX C

## An Example NETSLA program - Broadcasting

```

network broadcast;
process class sender
port inport in integer;
port outport out integer;
end sender

process class receiver
port inp in integer;
port outp out integer;
end receiver

initial
 create sender : sender;
 create receiver1 : receiver;
 create receiver2 : receiver;
 connect sender.outport to receiver1.inp;
 connect sender.outport to receiver2.inp
end broadcast

Gr := a_creation_pr (throot, "sender", "sender", "sender.RUN", 1, 1, 1, 1);
Gr := a_creation_pr (throot, "receiver1", "receiver1", "receiver.RUN", 1, 1, 1, 1);
Gr := a_creation_pr (throot, "receiver2", "receiver2", "receiver.RUN", 1, 1, 1, 1);
Gr := connection(throot, "sender", "outport", "receiver1", "inp", 1, 1);
Gr := connection(throot, "sender", "outport", "receiver2", "inp", 1, 1);
wakeUp;
end; (*init*)

```

## APPENDIX D

## A Network Specification Module

This code was generated by the Netsla preprocessor."

```

quit := false;
procedure init;
begin (*init*)
 p_id := 0;
 alive := 0;
 total_procs := 0;
 initialized := false;
 Gr := AllocatePort(KernelPort, ChildtoParPort, MAXBACKLOG);
 Gr := AllocatePort(KernelPort, EventPort, MAXBACKLOG);
 build_net('broadcast');
 build_proc('sender');
 build_port('inport');
 build_port('outport');
 build_proc('receiver');
 build_port('inp');
 build_port('outp');
 Gr := a_creation_pr (theroot,'sender','sender','sender.RUN',p_list_head);
 Gr := a_creation_pr (theroot,'receiver','receiver1','receiver.RUN',p_list_head);
 Gr := a_creation_pr (theroot,'receiver','receiver2','receiver.RUN',p_list_head);
 Gr := connection(theroot,'sender','outport','','receiver1','inp','');
 Gr := connection(theroot,'sender','outport','','receiver2','inp','');
 wakeup;
end; (*init*)

3: begin (* enter event *)
 enter_evt;
end;

4: begin (* leave event *)
 leave_evt;
end;

5: begin (* when evt *)
 when_evt;
end;

6: begin (* when evt. w/ about part *)
 when_evt;
end;

19: begin (* connectivity inquiry *)
 writein('Conn Inq Request Received');
 Gr := inquiry(theroot, EventMsg);
 if Gr=SUCCESS then
 writein('Conn Inquiry Completed');
 else
 writein('Conn Inquiry NOT Completed');
 end;
end;

99: begin (* termination of a process instance *)
 with yarray:=map(EventMsg.Data2) do

```



## APPENDIX E

## The Event Handling Module

```

begin
 EvntMsg.Head.LocalPort := EventPort;
 quit := False;
 while (quit=FALSE) do
 begin
 writeln('Events before receive req');
 Gr := Receive(EvntMsg.Head, 0, LOCALPT, RECEIVEIT);
 if Gr=SUCCESS then
 case shrink(EvntMsg.Head.ID) of
 1: begin (* message transmission. *)
 writeln('Send_Msg Request Received. ');
 Gr := send_msg(theroot, EvntMsg);
 if Gr=SUCCESS then
 writeln('Send_Msg Request Completed. ');
 else
 writeln('***Send_Msg Request NOT Completed. ');
 arrive_evnt;
 end;
 2: begin (* message transmission. w/ tag *)
 writeln('Send_Msg(w/ Tag) Request Received. ');
 Gr := send_msg_tag(theroot, EvntMsg);
 if Gr=SUCCESS then
 writeln('Send_Msg(w/ Tag) Request Completed. ');
 else
 writeln('***Send_Msg(w/ Tag) Request NOT Completed. ');
 arrive_evnt;
 end;
 3: begin (* enter event *)
 enter_evnt;
 end;
 4: begin (* leave event *)
 leave_evnt;
 end;
 5: begin (* when evnt *)
 when_evnt;
 end;
 6: begin (* when evnt. w/ about part *)
 when_evnt;
 end;
 19: begin (* connectivity inquiry *)
 writeln('Conn Inq Request Received');
 Gr := inquiry(theroot, EvntMsg);
 if Gr=SUCCESS then
 writeln('Conn Inquiry Completed')
 else
 writeln('Conn Inquiry NOT Completed');
 end;
 99: begin (* termination of a process instance *)
 with vpparray[vppmap[EvntMsg.Data2]] do

```

```
begin
 DeleteCanvas(canvs);
 PaintRectangle(UserCanvas,White,x0+1,x0+xlen-3,y0+2,y0+ylen-2);
 Used := False;
end;
alive := alive-1;
if alive=0 then
begin
 quit:=TRUE;
 EraseCanvas(UserCanvas,White);
 {DeleteCanvas(UserCanvas);}
end;
end;
(* more come here *)
otherwise:
begin
end
end; (* case *)
end; (* while *)
```

## APPENDIX F

## A Script for Sender Processes

```

process script sender;
port input in integer;
port output out integer;
var

```

```

 i:integer;
begin
 while i<>999 do
 begin
 write('Integer: ');
 readln(i);
 send i to output
 end
 end
end.

```

```

 signal = boolean;

```

```

 accentuary = record

```

```

 head : set;
 ipname1 : string(12);
 arg1 : integer;
 ipname2 : string(12);
 arg2 : integer(12);
 ipname3 : string(12);
 arg3 : integer(12);
 ipname4 : string(12);
 arg4 : integer(12);
 ipname5 : string(12);
 arg5 : integer;
 ipname6 : string(12);
 arg6 : integer(12);
 ipname7 : string(12);
 case of
 1 : (signal : boolean);
 2 : (arg1 : integer);
 3 : (arg2 : integer);
 end;
end;

```

```

var

```

```

 xsignal : boolean;
 gr : generalreturn;
 whenflag : boolean;
 xsignal : signal;
 comports : port;
 p array : array[12];
 petr : string(12);

```

```

($INCLUDE state_spt.pas)

```

```

begin

```

```

($INCLUDE state_spt.pas)

```

```

InitMsgn(12, 12);

```

```

Gr := Child ark;

```

```

while i<>999 do

```

```

begin

```

```

 write('Integer: ');

```

## APPENDIX G

## The Preprocessor-generated Code for Sender Processes

```

program sender;
imports Child_lib from Child_lib;

var
i:integer;
var
pinport : port;
poutport : port;

type
signal = boolean;

accentmsg = record
 head : msg;
 ipcname2 : TypeType;
 arg2 : integer;
 ipcname3 : TypeType;
 arg3 : string[10];
 ipcname4 : TypeType;
 arg4 : string[10];
 ipcname5 : TypeType;
 arg5 : integer;
 ipcname6 : TypeType;
 arg6 : string[20];
 ipcname1 : TypeType;
 case integer of
 1 : (msignal : signal);
 2 : (msginport : integer);
 3 : (msgoutport : integer);
 end;
end;

var
xxmsg : accentmsg;
gr : generalreturn;
whenflag : boolean;
xxsignal : signal;
commport : port;
p_array : PortBitArray;
pstr : string[12];

{$INCLUDE Alsten_supt.pas}
begin
{$INCLUDE AlstenInit.pas}
InitMsgn(NullPort);
Gr := Child ack;
while i<>999 do
begin
write('Integer: ');

```

```
readln(i);
begin (* send *)
 xxmsg.head.id := 1;
 xxmsg.head.remoteport := InPorts^[1];
 xxmsg.head.localport := DataPort;
 xxmsg.msgoutport:=i;
 xxmsg.arg2 := p_id;
 xxmsg.arg3 := 'outport';
 xxmsg.arg4 := '';
 gr := send(xxmsg.head,0,wait)
end (* send *)
 end
;goaway;end.
```

## The Preprocessor-generated Code for Receiver Processes

## APPENDIX H

## A Script for the Receiver Processes.

```

program receiver;
imports Child lib from ChildLib;

process script receiver;
port inp in integer;
port outp out integer;
var

```

```

 j:integer;
begin
 while j<>999 do
 begin
 receive j from inp;
 writeln(j)
 end
 end.
end.

```

```

head : seq;
ipnames : seq;
arg1 : integer;
ipnames : seq;
arg2 : integer;
ipnames : seq;
arg3 : integer;
ipnames : seq;
arg4 : integer;
ipnames : seq;
arg5 : integer;
ipnames : seq;
arg6 : integer;
ipnames : seq;
case integer of
 1 : (arg1 := ipnames[1]);
 2 : (arg2 := ipnames[2]);
 3 : (arg3 := ipnames[3]);
end;

```

```

var
 xmsg : accoutmsg;
 gr : generalreturn;
 whenflag : boolean;
 xsignal : signal;
 comport : port;
 p array : seq;
 patr : string(12);

```

```

(SINCLUDE Alacm_msg.pas)
begin
(SINCLUDE Alacm_out.pas)
InitMsgs(1000, 1000);
Gr := Child;
while j<>999 do
begin
 begin (* receive *)

```

## APPENDIX I

## The Preprocessor-generated Code for Receiver Processes

```

program receiver;
imports Child_lib from Child_lib;

var
 j:integer;
var
 pinp : port;
 poutp : port;

type
 signal = boolean;

 accentmsg = record
 head : msg;
 ipcname2 : TypeType;
 arg2 : integer;
 ipcname3 : TypeType;
 arg3 : string[10];
 ipcname4 : TypeType;
 arg4 : string[10];
 ipcname5 : TypeType;
 arg5 : integer;
 ipcname6 : TypeType;
 arg6 : string[20];
 ipcname1 : TypeType;
 case integer of
 1 : (msignal : signal);
 2 : (msginp : integer);
 3 : (msgoutp : integer);
 end;
end;

var
 xxmsg : accentmsg;
 gr : generalreturn;
 whenflag : boolean;
 xxsignal : signal;
 commport : port;
 p_array : PortBitArray;
 pstr : string[12];

{$INCLUDE Alsten_supt.pas}
begin
 {$INCLUDE AlstenInit.pas}
 InitMsgn(NullPort);
 Gr := Child ack;
 while j<>999 do
 begin
 begin (* receive *)

```



```
rcv('inp','',999,1,rcv_result);
if rcv_result then
j:=xxmsg.msginp;
end (* receive *)
;
writeln(j) end
;goaway;end.
```

The following figures show a sequence of screens taken from a multiprocess interactive expression interpreter program developed during this project for testing the open-architecture. The SCANNER process reads the input from the keyboard and then generates messages containing token values which are sent to the PARSER and not other parts. These values (of constants are identified by the SCANNER). The PARSER sends a message to the INTERPRETER, sending the semantic values of the expression. This message drives the interpretation.

Dotted lines are included on the figures to indicate connections. These are not present in the actual presentation done by the system.

Sending a message is represented on two screens. The first shows a message box moving on output port, while the second shows a message box on input port. The second part represents the state of the display when the message box is on the screen. These two parts show the beginning and end of the presentation of the message. In the actual presentation, the message box moves smoothly across the screen from the output port to the input port.

Receiving a message is represented by a single screen that shows the state of the display after the message box is removed from the input port of the appropriate port.

## APPENDIX J

### Event Replay Example

The following figures show a sequence of 6 events taken from a multiprocess arithmetic expression interpreter program developed during this project for testing and demonstrations. The SCANNER process reads an expression from the keyboard and then produces two messages: one containing token classifications for the PARSER and one containing token values (of constants and identifiers) for the INTERPRETER. The PARSER sends a message to the INTERPRETER describing the syntactic structure of the expression. This structure drives the interpretation.

Dotted lines are included in the figures to indicate port connections. These are not present in the actual presentation done by our replay system.

Sending a message is represented by two pictures. The first shows a message box leaving an output port, while the second shows it arriving at an input port. The second picture represents the static state of the display after completion of the event. These two pictures show the beginning and end of the presentation of the event. In the actual presentation, the message box moves smoothly across the screen from the output port to the input port.

Receiving a message is represented by a single picture that shows the state of the display after the message box is removed from the input queue of the appropriate port.

Figure J - 1

---

SCANNER sends message to PARSER

---

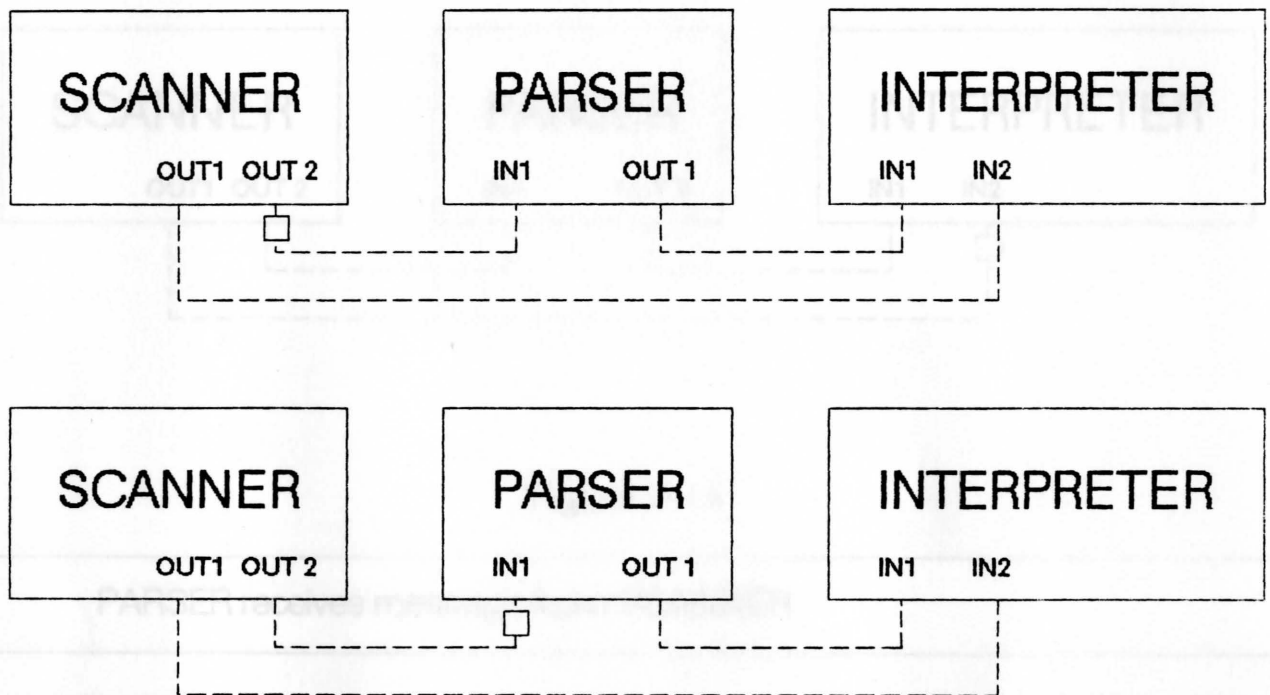
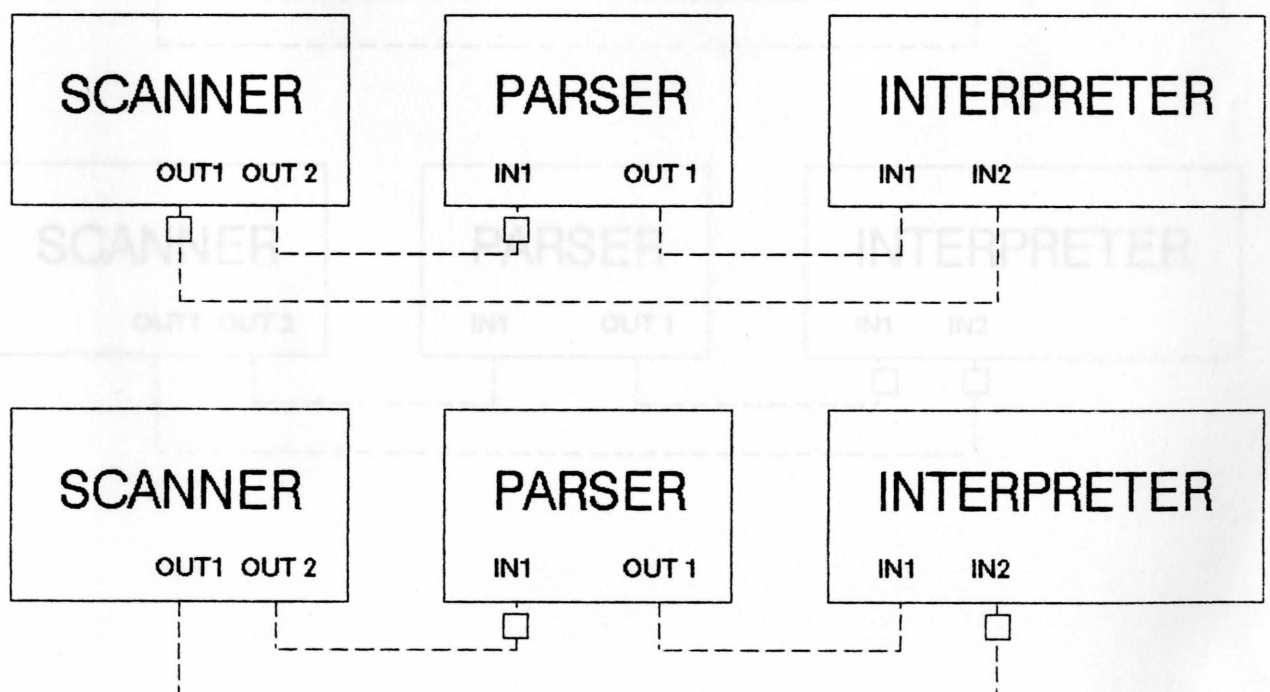


Figure J - 2

---

SCANNER send message to INTERPRETER

---



---

PARSER receives message from SCANNER

---

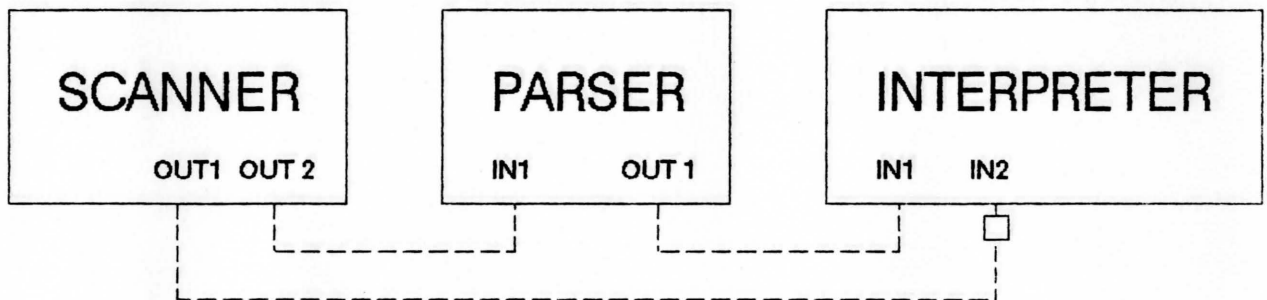


Figure J - 4

---

PARSER receives message from SCANNER

---

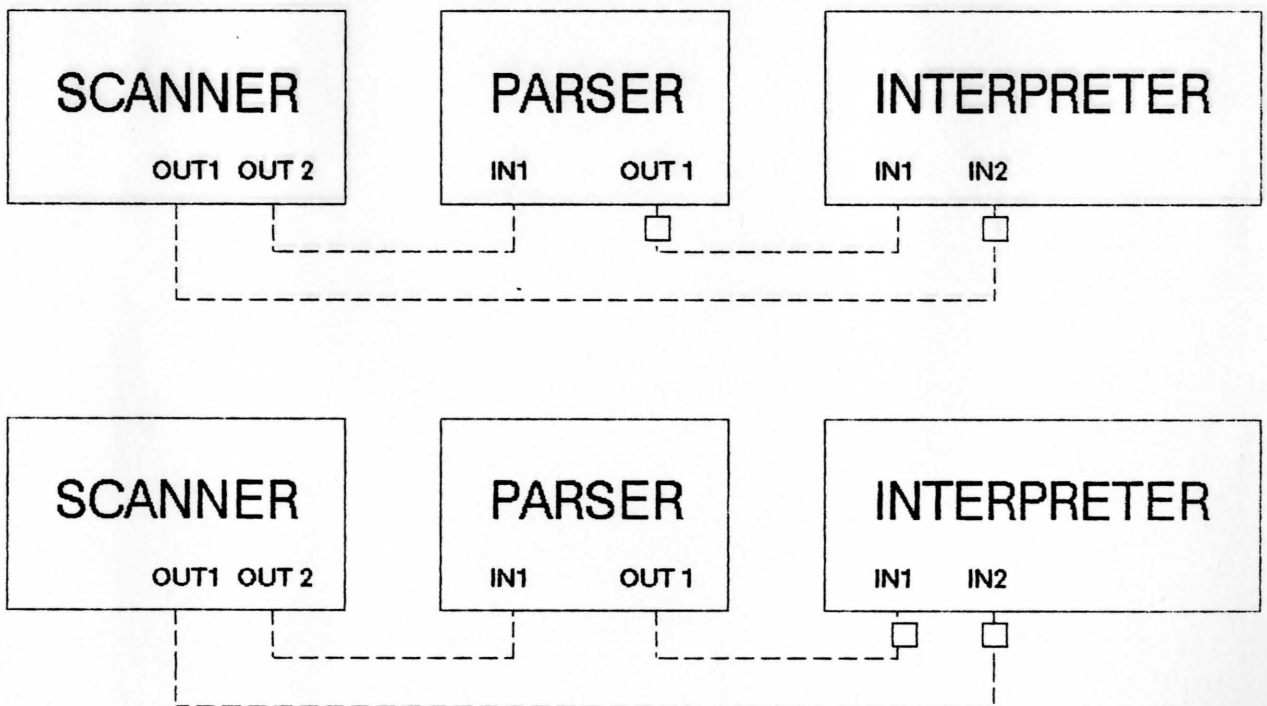


Figure J – 5

---

INTERPRETER receives message from PARSER

---

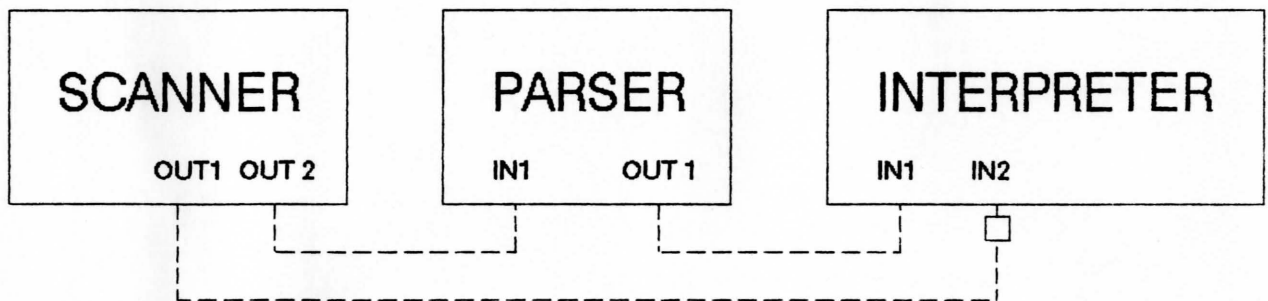


Figure J – 6

---

INTERPRETER receives message from SCANNER

---

