DISTRIBUTED SOFTWARE VIA PROTOTYPING AND SIMULATION - JADE
(Invited Paper)

Brian Unger
Department of Computer Science
University of Calgary
Calgary, Alberta, Canada T2N 1N4

UUCP: ...!{ihnp4,ubc-vision}!alberta!calgary!unger
ARPA: unger.calgary.ubc@csnet-relay
CDN: unger@calgary

ABSTRACT

Jade is an environment that supports the development of distributed software. Components may be written in any of a number of different languages, with a common inter-process communication protocol providing a uniform interface among the components. A window system allows the user to interact with many different processes at once. A hierarchical graphics system is provided for use with documentation and programming, and for support of monitoring. Monitoring in Jade is also supported by an extensible mechanism which allows for multiple views of the same process. The non-determinism of distributed systems may be controlled in order to provide repeatability of executions and to aid in testing and debugging. Finally, the formal specification of inter-process events in Jade is supported by a communications protocol verifier, allowing run-time consistency checking. Together, these tools provide a powerful environment for software prototyping and simulation. This paper is a summary of work that has been described in [Unger 85], [Lomow 85] and [Joyce 87].

1. INTRODUCTION TO JADE

The Jade environment may be seen as consisting of four levels: the hardware level, the kernel level, the programming level, and the prototyping level. Each level provides support for the next. The levels are described below, proceeding bottom-up.

The central unifying concept underlying the Jade environment is the Jade Inter-Process Communication protocol, or Jipc (pronounced as "gypsy") [Neal 1984], which is a Thoth-like protocol [Cheriton 1979]. Jipc interfaces have been written for Unix 4.2 and for a stand-alone multi-tasking kernel. The Unix version is currently running on the Vax 11/780 and SMI Sun. The stand-alone kernel runs on 68000-based systems: the Corvus Concept, the MTU, and the Cadlinc Sun. A port has recently been completed to the Mesh Machine [Cleary 1983]. Jipc messages can be sent over any TCP/IP link, between processes on the same machine, across Omninet, and via shared memory on the Mesh Machine. At the software end, a Jipc interface is pro-

vided for each of five different languages: Ada, C, Lisp, Prolog, and Simula. Each item of a Jipc message has a specific type, which must be integer, real, character, string, atom, block, or process id. Conversion between different representations of these types on different machines is performed automatically by the Jipc kernel.

1.1. Programming level

At the programming level, a number of facilities exist for support of program development. A window system, including virtual terminal windows, allows the user to interact with many different processes, possibly on different machines, at once. A hierarchical graphics system is available for use with graphical applications. Monitoring of Jipc events is provided in such a way that the user may easily write monitor consoles specific to particular applications or debugging techniques. A number of monitors, both textual and graphical, are also provided for general use.

1.2. Prototyping level

Prototyping and simulation of distributed software are supported by a formal specification language for the description of allowable Jipc events. A protocol-verification tool allows run-time checking to be performed in order to ensure that the executing system conforms to the specification. The distribution of components of a software system on a target architecture may be simulated with the use of a version of Jipc which allows the specification of an actual distribution which may differ from the simulated distribution. The inter-process interactions faithfully conform to the way they would appear in the target system. Finally, work is underway on the implementation of a time-warp [Jefferson 1985] version of Jipc [Cleary 1985].

2. JIPC AND THE WINDOW SYSTEM

The primary communications primitives provided by Jipc include send, receive, receive_any, forward, and reply. A sending process places data into a Jipc buffer, then invokes the Jipc send call to send the buffer to a specified process. The send is a blocking send, and awaits a reply before returning control to the sender.

In order to receive a Jipc message, a process executes a receive or receive_any call. The receive call is for receiving only from one specified process, while receive_any allows a message from any process to be received. If a process executes a receive before the process from which it is attempting to receive has sent it a message, then the receiving process goes blocked until such a message arrives. If such a message has already arrived, execution continues immediately. The receiving process may then extract information from the buffer sent to it, perform some computations, and place new data into its buffer, comprising the information in the reply. A reply call transmits the receiver's buffer to the original sender. When the reply arrives at the sender, the sender is unblocked and allowed to continue execution, with its buffer containing the contents of the reply message.

A receiving process need not reply directly to its sender, but may delay the reply and receive more messages from other processes, or it may forward the received message to another process. In this case, it is the responsibility

of the process to which the message was forwarded to construct a reply.

Jipc provides routines for process creation, destruction, and searching, as well as the above communication primitives. These activities are confined to the specific Jipc system in which the process is executing. Different groups of processes may thus co-exist without interfering with each other, by using different Jipc systems.

The Jade window system is composed of a number of processes which communicate among themselves and with user processes via Jipc. Currently, it runs on top of the stand-alone Jipc kernel and requires in addition a bit-mapped monochrome display. Work is underway to integrate it with the SMI Sun window system under Unix 4.2.

The window system is driven by a mouse with three buttons: a menu button for raising pop-up menus, a help button to provide context-sensitive help, and a point button for indicating positions on the screen or within a window. An arbitrary number of windows may be created for a variety of purposes. Virtual terminal windows allow login access to Unix; a console window allows the user to obtain information about and exert control over various aspects of the workstation; and other windows allow code to be downloaded to run locally on the workstation. Windows may overlap, but must be uncovered in order to receive output. Any output sent to a buried window is buffered until the window is raised.

Application programs may create their own pop-up menus and associated help windows by sending requests to the window manager process on the workstation on which their windows reside. A different set of menus is associated with each window. The program may detect any events occurring in a window by requesting this information from the window manager. These events include menu selections, the cancellation of a selection, pressing or releasing the mouse point button, changing the size of the window, and destroying the window. The window manager also allows direct output, both textual and graphical, to be done to a window.

A toolbox package provides an interface to higher-level routines built on top of the low-level window-manager requests. Included in this toolbox are routines which hide Jipc communication details and allow the programmer to treat window-manager requests as if they were standard C subroutine calls. Routines are also provided for creating rectangles on the screen and dealing with these as if they were buttons. Other tools allow creation and manipulation of slide potentiometers, which may be used for scroll bars. Finally, a Lisp interface to the toolbox routines allows the interactive development of prototype software.

3. GRAPHICS

The Jade graphics system, Jaggies [Wyvill 1984], is based on the Groper system [Wyvill 1977] and provides hierarchical graphics routines for use by application programs and monitors. Jaggies routines may be accessed directly via C or Prolog subroutine calls, or in other languages via Jipc messages to a Jaggies process. The hierarchical nature of Jaggies means that pictures can be created which are comprised not only of graphical primitives but also of other sub-pictures. Recursive inclusion of pictures is allowed, and is controlled at plot time by global and local recursion limits.

The simplest Jaggies primitives are the point, line, arc and circle. Text is available in a number of different fonts, with the user having the capability of installing special-purpose private fonts as well. Boxes may be drawn which are either wire-frame or solid. Raster images are supported to some extent, though Jaggies is primarily a vector graphics system.

Each primitive or sub-picture in a composite picture has associated with it a transform which determines how it is included in the composite picture. Transforms include combinations of the standard translation, rotation, and scaling (X and Y axes separately). A picture may also have an associated color transform. Since the Jade workstations are monochrome, colors are currently represented by different patterns. For instance, a line may be solid, dotted, black, or invisible.

A 32-bit datum is also associated with each Jaggies picture. This datum is not used by Jaggies itself, but may be set by the application program to store additional application-specific information related to the picture. For instance, a picture representing a specific process in a simulation might have its datum set to point to the state variables of that process. Retrieval of pictures by application data is also possible.

Although Jaggies provides only very simple graphical input directly (get the coordinates of a pointer device), it supports a number of routines for manipulating the coordinates it receives. The raw device coordinates may be converted to world coordinates in terms of the displayed picture. Following this, routines may be invoked which will return a set of instances of sub-pictures which have parts close to the indicated point. Thus, picking with a mouse is possible.

Jaggies pictures may be constructed and edited interactively by use of the graphics editor, Jagged [Wyvill 1984]. Essentially, Jagged provides the user with a mouse-and-menu interface to the Jaggies subroutines. Primitive and composite pictures may be created and transformed, then written to files for later retrieval by an application program which will make use of them.

A Prolog-based graphics language, Growl [Cleary 1984], has been implemented on top of Jaggies. Growl allows the construction of hierarchical pictures from within a Prolog environment. Growl and Jaggies are very similar in their capabilities, but are different in their semantics. For instance, Growl makes use of the backtracking feature of Prolog to define the different sub-pictures within a composite picture. Growl has been used in conjunction with a graphical debugger for Prolog [Dewar 1985].

4. MONITORING

A recent paper [Joyce 87] describes issues in monitoring distributed systems and the Jade monitoring system in particular. A brief overview is presented here. Monitoring in Jade is provided at the Jipc level in such a way that the programmer need not take any special action whatsoever in order to make application programs monitorable. Instead, the Jipc system automatically passes information on certain Jipc events to monitoring processes, as described below.

A *Jipc system* is a distributed concurrent program consisting of processes running on some combination of machines. No modifications to program source are required for Jipc events to be monitorable. If the user wishes to

disable this feature, this may be done by linking the program with a non-monitored version of the Jipc library.

4.1. Jipc event detection

The Jipc events which are considered monitorable are those which are concerned directly with inter-process interactions. These include sending, receiving, replying, receiving a reply, searching for a named process, and creating or destroying a process. The manipulation of message buffers, in preparation for sending a message or after receiving a message, is considered to be of interest only to the manipulating process and is therefore not monitored by Jipc. A special Jipc primitive is provided whereby a program may explicitly signal the occurrence of a particular application-defined event. Such an event is monitored but has no effect on the executing program.

Whenever a monitorable Jipc event occurs, the event, along with any associated buffer or other information, is sent to a special *channel* process. There is one channel process for each Jipc system on each machine. Channel processes are created automatically, as needed, by the Jipc kernel. The channel process forwards the information it receives to any *consoles* present. Consoles may be written by the user to perform special-purpose monitoring functions. A number of standard consoles are provided. All monitoring communication between user processes, channels, controller (defined below) and consoles is via standard Jipc messages (themselves non-monitored). Thus, all features of the Jade environment are available to anyone constructing a console or controller. In particular, a message can be received from channels on many machines, allowing monitoring of truly distributed systems. Typically, each console displays its output in a separate window.

4.2. Controller processes

The user may insert a *controller* process between the channel processes and the consoles. If such a process is present, each channel will send information only to the controller. The controller can then filter out uninteresting events before forwarding monitoring information to the individual consoles.

Another use for a controller is to exercise control over the executing system of Jipc processes. Since each process awaits a reply from the controller before completing the Jipc call which is being monitored, the controller may delay processes or fix the order in which they are allowed to proceed. This provides a means of controlling the non-determinism inherent in a distributed computing environment. A specific order of execution may be consistently reproduced, allowing debugging and testing of unusual circumstances to take place.

4.3. Textual Traces

A *Text Console* has been developed that reports each event in the event stream with one or two lines of textual output. The name of the process that initiated the event, the event type, and the name of the process that is the subject of the event, if any, are written on the first line. If the event is one in which processes communicate, the contents of the message are printed as the second line of output.

A textual trace provides little more than what would be achieved by printing debugging information at strategic points in each process. However, the user is not required to do this because events are detected and reported automatically by the monitoring system. Thus the possibility of introducing errors while inserting monitoring statements into each process is eliminated, and consistent monitoring information is provided to Consoles.

Facilities for *event filtering, breakpoints,* and *execution histories* are included in the Text Console for assisting the user in dealing with the large quantities of information produced by the monitoring system. Each of these facilities depends on pattern matching in the event stream. There are two types of patterns: process patterns and event patterns. A process pattern is an expression which identifies a process or group of processes. An event pattern identifies an event or group of events and may include a process pattern. Event filtering uses event stream pattern matching to determine which events to display: when the Console receives an event that matches one of the filters which the user has specified, that event is displayed. This enables the user to interactively specify the set of events to be displayed by the Text Console.

Breakpoints are also event patterns specified by the user. When an event matching a breakpoint occurs, monitoring is suspended and control is given to the user. The value of breakpoints is that they free the user from having to constantly watch the Console to detect important events. Breakpoints can also be used to detect *impossible* events, or events that signal error conditions. It is easy to determine the program state of any process when stopped at a breakpoint because a sequential debugger can be invoked from the Text Console.

The history mechanism allows the user to re-examine a specified number of previous events, in the order of occurrence, for a set of processes defined by a process pattern. The history facility is a particularly useful adjunct to textual monitoring because it allows the user to easily determine how a process, or set of processes, reached a specific state. It also permits events that have scrolled off of the screen to be re-displayed. The history mechanism maintains a fixed length copy of the Console event stream that is periodically truncated. The most recent event generated by each process is also maintained in a separate history structure which is not truncated, enabling the most recent action of all processes to be obtained at any time.

When trying to understand a large distributed system, it is important for the user to be able to focus on only those processes and events which are of immediate relevance, without having extraneous events cluttering the display. Since both event filtering and breakpoints can be altered while monitoring, the user can readily change the focus of the monitoring session. The addition of these facilities to the basic Text Console makes it a very useful tool, qualitatively different from the graphical Console presented next.

4.4. Graphical State Displays - The Mona Console

Mona provides the user with an animated graphical view of the event stream. Whenever Mona receives an event, it updates a picture which represents the current state of inter-process communication in an application system.

Each update to the picture results in the display of a new *frame*. A frame describes the current state of the application system; successive frames present successive states. A sequence of frames creates an animated display of the executing distributed program. The graphics package only updates the portion of the picture that is actually changed, so the screen is not completely redrawn for each new frame. Mona requires a bit-mapped screen, along with an input device such as a mouse, for pointing to locations on the screen. Many of the details found in the textual trace are not available with Mona, e.g., the contents of messages are not shown.

When a process comes into existence, Mona, by default, places its icon on the circumference of a series of concentric circles. This often results in an arrangement of icons that does not reflect the structure of the system. To alleviate this problem, Mona allows the user to re-position icons using the mouse. An arrangement of icons can be saved and subsequent invocations of Mona can use these previously defined arrangements when deciding where to position icons.

The design of a distributed system is often structured hierarchically so that a collection of processes provides a single service or implements a single function. Furthermore, individual processes and groups of processes are often combined to form larger units. During application system development and debugging, and the demonstration of the system's operation, the user will, at times, want to focus on the internal workings of a collection of processes and, at other times, will want to regard a collection as an indivisible unit. The display management facilities of Mona are able to reflect the system's structure so that its execution can be viewed at levels of abstraction above the IPC level. This supports monitoring and debugging after development has moved beyond the IPC protocol level.

In Mona, a *group* is defined to be a collection of entities in which each entity is either a group, or a Jipc process. A group is created by using the mouse to define the opposite corners of a box which physically encloses the processes and groups which are to constitute the new group. Groups can be created, removed, and incorporated into other groups and a group can be repositioned as an indivisible unit. The grouping of processes is discussed in [Cheriton 85] as a programming and kernel optimisation aid while here it is used as a mechanism to simplify a large, complex display.

A Mona group may be either *open* or *closed*. An open group is delimited by a dashed line box; the interactions among the top level entities of an open group are displayed. A closed group is delimited by a solid line box. None of the internal interactions among the entities of a closed group are shown, and internal process icons are not depicted. An open group corresponds to a collection of entities that the user wishes to view. A closed group encapsulates entities whose internal activities are not of current interest. The rule for displaying events in Mona is to depict all *visible* events, i.e., those in which the participating processes are not in subgroups having a common closed ancestor group, and those in which the participants are both on the screen.

Zooming is the counterpart of grouping, enabling the user to focus on part of an application system. There are two types of zooming, physical and conceptual. In a physical zoom the mouse is used to define a rectangular area of the screen. In a conceptual zoom a group is selected. In both cases the area

or group is enlarged to fill the entire screen. Successive zoom operations are placed on a stack, so the user can zoom in and out in a hierarchical manner. Zooming in on a closed group causes it to be opened (the assumption being that the user is zooming in on the group in order to see its internal activities).

Shrinking is a display management aid. When the user shrinks a group, it is scaled into a small box. This physical operation does not change whether the group is open or closed. Shrunken groups may be moved, opened, closed, removed, and included inside of new groups. When a shrunken group is expanded it regains its former size. If a shrunken group is removed, its subcomponents are automatically expanded to their previous size. A *step mode* also helps the user manage the display by requesting confirmation before the depiction of the next visible event. This alleviates events being portrayed so quickly that they flicker past, leaving the viewer unsure of what just happened.

4.5. Other monitoring consoles

Neither the Text Console nor Mona is able to simultaneously display both the current state of the system and the sequence of events that led to that state. In response, we have developed a facility which displays process evolution versus events.

The *Event Line Console* displays the current state and history of each process in a compact form and, at the same time, defines the relative ordering of events. The display is divided into three sections:

(1) The name of each process is listed along with a single letter abbreviation for that process. The abbreviation is also repeated on the left and is used to identify the process in event descriptions.

(2) There is one *event line* for each process. Each event line is divided into an equal number of *event intervals*. An event interval demarcates adjacent events in the console event stream; it has no relationship to the passage of real time. Each event interval displays an event; events are inserted at the right of the display and scroll to the left. The relative ordering of events is shown by their location on the horizontal axis of the event line. This information is lost for the events as they are scrolled off of the event line.

(3) The current state of a process is always available, even if a process has had all of its events scrolled off of the event line (because the process is blocked or has not generated an event recently).

A process's event line is blank before it enters the Jipc system or is created, and after it leaves the Jipc system or is killed. While a process is executing and not generating monitorable events its event line is dashed (---). A dotted (...) line signifies that the process is blocked by a Jipc call.

This Event Line Console could be extended in several ways:

(1) A history function could be provided by permitting the user to scroll the event lines both left and right.

(2) A breakpoint facility could be provided.

(3) The user could point to an event and have the message or parameters associated with that event displayed.

(4) The user could move event lines vertically so that the event lines of related processes are adjacent, or grouped.

We have little experience using the Event Line Console. Its development was motivated by the apparent preference of most users for the Text Console during debugging. With extensions such as those listed above, we anticipate that the Event Line Console will be more useful than the Text Console.

4.6. Functions of monitors

The role of monitoring in the development of distributed systems can be extended by mechanisms which perform computations on an event stream, which enable non-determinism to be controlled, and which utilise application specific information to interpret an event stream. Mechanisms able to operate without any knowledge about the specific application system being monitored include the analysis of inter-process communication patterns and the control and re-creation of specific execution paths. Example Consoles which implement such mechanisms are outlined below.

During the monitoring process it is inevitable that the system developer will need to observe behaviour that is specific to the current application. Information can be presented to a monitoring system that enables it to interpret an event stream in a way which is relevant to a particular application distributed system.

The Consoles described above display a stream of events. Tools which accept a stream of events, perform computations on this stream, and then present computed results to a user can also be implemented as Consoles. One that we have implemented collects statistics on inter-process interactions and another determines whether the state of communication among a set of processes is deadlocked.

A *Statistics Console* records the number and type of events which occur during the execution of an application system can be recorded for each process, as well as, additional information available to the monitoring system such as message lengths. At any time, the statistics console can be interrupted and data can be displayed either for individual processes or for the entire system. For IPC events, the statistics can be separated into local calls (the initiating and destination processes are on the same machine) and remote calls (the initiating and destination processes are on separate machines). The type and number of errors generated by each process are also recorded.

This Console assists in optimising a system at the inter-process communication level. Statistics concerning which processes communicate, how often they communicate, and average message length can aid in making decisions about system and process decomposition, and the assignment of processes to processors.

A *Deadlock Detection Console* is a debugging tool which uses the event stream to maintain a model of the state of a Jipc system. As the Deadlock Detector receives each event it updates the model and checks to see if any cycles of blocked processes exist in the model. When deadlock is detected, the user is informed, information regarding the current state of the deadlocked processes is displayed, and the system's execution is halted.

The advantages of the Deadlock Detector are:

(1) It actively monitors for deadlock. Mona, by contrast, depends on the user to recognise deadlock.

(2) In a distributed system with many processes, it can detect and identify deadlock amongst a small subset of the processes even though the rest of the system is operating normally.

(3) It requires no attention from the user until deadlock is detected.

One of the difficult problems in developing distributed systems is their inherent non-determinism. A correct execution of an application system corresponds to a partial ordering of the communication events. Events which can occur in arbitrary order are "independent" and it is possible to control the order in which these events occur during monitoring. This control can also be used to automatically recreate a specific execution path from a recorded trace.

5. SPECIFICATION AND PROTOTYPING

5.1. Formal specification of Jipc systems

Having a formal specification of the allowable interactions between processes provides a number of advantages to the developers of a distributed software system. The specification serves as documentation and as a means of communication between developers. Different modules may be developed independently, with each module being designed to conform to the specification. Test cases for the system may be suggested both by the specification itself and by the process of writing it. Finally, run-time checking of an executing system may be performed to ensure that its actual behavior is consistent with its expected behavior. These functions are provided by the Jipc Description Language, JDL, and its associated run-time verifier.

JDL allows a system designer to specify formally a number of aspects of a Jipc system. A JDL specification consists of three parts: process descriptions, buffer descriptions, and event descriptions. The process descriptions specify all processes present in the system, and allow related processes to be grouped together and referred to as a class for purposes of event descriptions. Buffer descriptions specify the buffers which may be used in Jipc messages. Finally, event descriptions make use of process and buffer descriptions to describe the Jipc events which are allowed.

There are some subtleties of inter-process communication that JDL cannot represent. For instance, two classes of processes may exist with each process in the first class having a single corresponding process in the second class with which it may communicate. JDL could specify only that communication between processes of the two classes must conform to certain constraints. The more restricted communication could be specified by explicitly enumerating all possible process names in each class and providing identical rules for each pair of processes, but this is not always feasible or even possible in a real system. A more sophisticated extension to JDL is needed if this level of refinement is required of specifications.

Since the run-time verifier is written in Prolog and translates JDL specifications into Prolog clauses, it would be a straight-forward matter to allow the full power of Prolog expressions to be intermixed with JDL descriptions. However, this would introduce the temptation to write large portions of the description in Prolog, and would thus detract from the simplicity of JDL. Since a simple specification may result in the design of a simpler and clearer system, it is beneficial to keep JDL as simple as possible. Nonetheless, it is recognized that some more highly refined description techniques are needed.

5.2. Prototyping process distribution

A system of processes may be developed for a target environment which differs from the development environment in a number of ways. One such difference is the assignment of processes to machines. While the target environment may have a large number of different machines on which processes are to be run, it may not be possible or desirable to have the prototype system distribute processes to actual machines in the same way. A prototyping version of Jipc allows distributions of processes to be simulated on different physical sets of machines.

Embedded-system software developed in the Jade environment is intended to be runnable directly in the target environment without requiring any changes to the source code. However, Jipc routines may behave differently if processes involved are distributed differently. For instance, searching for a process by name is done on a per-machine basis; thus, it makes a difference where the processes reside. The prototyping Jipc provides an intermediary between the simulated target environment and a physical distribution of processes in the development environment. The user may specify the correspondences.

6. PROTOTYPING AND SIMULATION TOOLS

The Jade prototyping system is an embedded system simulator, called *Jems* [Lomow 85], that supports the design, implementation, debugging, testing, maintenance, and performance evaluation of a set of cooperating processes. This is accomplished by integrating systems prototyping and computer systems simulation tools. The development of distributed software is accomplished as part of the development of a target system simulation. A crucial requirement in this process is that software components of the target system be identical to the corresponding software components in the simulation.

This relationship between target system software components and components of the simulation has two aspects. First, all interactions between components must be identical, and second, the internal implementation of each component should also be identical. Both of these requirements can be satisfied.

Component interactions can be identical only if the communication and synchronisation required between components is accomplished via the same mechanisms. This is achieved in Jade by requiring that all interactions between processes be accomplished via Jipc. This must be true of the target system, as well as, of the simulated system.

Thus we assume that in the target system a set of processes is distributed over a network of computers and that these processes interact only via Jipc. Jems supports this within the simulated environment by providing a different version of Jipc that appears identical to standard Jipc from the viewpoint of a target process. The Jems version of Jipc intercepts all messages and can thus monitor, control, and even alter the contents of all delivered messages.

The Jems version of Jipc directs all messages that it intercepts to a central Jems controller. The Jems controller then can cause the simulation to progress in a way which is consistent with the advance of a global *simulation time*. The controller can also interact with the user to selectively view the progress of the simulation, stop the simulation, alter or create messages, alter which portions of the system are being viewed, and then continue system operation. It is also possible to change back and forth between standard Jipc and the Jems simulated Jipc at run time.

Identical component implementations can be maintained if the simulated version does not require statements that specifically alter simulation time. The simulated execution of software components usually requires the insertion of statements like Hold(t) to characterize the passage of t units of simulation time. This can be avoided by measuring the real processor time consumed by the component between Jipc calls.

The Jems version of Jipc makes this measurement and sends these elapsed times to the Jems controller. The controller can then maintain a global simulation time and determine which processes of the target system can be enabled to run. Since the Jipc protocol is blocking, the Jems controller allows target processes to proceed by controlling the delivery of Jipc messages. The blocking Jipc protocol provides greater control and flexibility in the simulation and prototyping process than would an asynchronous protocol.

Jems also supports simulation of the external system in which the target computer system is embedded, as well as, peripherals of the embedded system. These model components will require simulation functions such as Hold(). The use of Hold() is also possible from within a software component to represent code that has not been implemented yet. Thus in a target software component, the Hold() becomes a place holder, or notice to the designer, that more detailed code will still be required at this location.

Integrated simulation and prototyping has many advantages which should now be apparent. Very abstract versions of target software components can be implemented and tested within models of the target system hardware. Arbitrary configurations of this target system can be represented and alternative software architectures can be examined. Examples of a preliminary prototyping tool applied to actual system and software development problems are presented in [Unger 82a] and [Unger 82b]. In Jade these abstract preliminary versions can become progressively more detailed until a final implementation of target software that meets performance objectives is complete.

Not only is the user assisted during the design and implementation phases, but maintenance can also be supported. Only one version of the target software need be maintained for both the simulation and the actual system. The simulation can then be used to support the ongoing modification and maintenance process. New hardware configurations can be tested and evaluated before installation. Proposed modifications can be tested in the

simulation before being installed in a target system. Finally, the non-determinism inherent in the operation of the target system can be avoided in the simulation to support debugging during both development and maintenance.

7. DISCUSSION

Jade has been in existence for three years. In this time, two major releases have been produced. The first concentrated mostly on lower-level components, such as the Jipc kernel and the programming level, while the second placed more emphasis on the higher-level aspects of distributed software development, such as specification and prototyping. A five-volume Jade User's Manual has been produced [Jade 1985]. The current Jade environment is used at the University of Calgary in the senior-year undergraduate Computer Science program, and by graduate students and research staff. Jade has also been licenced to several universities and research institutions across the United States and Canada.

Currently we are using Jade to support research and tool development in distributed simulation. This work is based on optimistic synchronisation mechanisms first described in [Jefferson 85]. Recent progress is reported in [Lomow 88], [Cleary 88] and [Li 88].

ACKNOWLEDGMENTS

Funding for the Jade project has been provided by the Natural Sciences and Engineering Research Council of Canada. We are also grateful to the US Naval Research Laboratory for providing us with the use of an SMI Sun workstation and funding for optimistic distributed simulation research. The Jade staff and affiliated faculty members and students at the University of Calgary have contributed greatly, providing a stimulating research environment within which Jade was developed. I would particularly like to acknowledge the many significant contributions of Alan Dewar, Radford Neal and Greg Lomow; and Larry Mellon for his editing assistance.

REFERENCES

Cheriton, D.R., Malcolm, M.A., Melen, L.S., and Sager, G.R (February 1979) "Thoth: a portable real-time operating system" *Communications of the Association for Computing Machinery, 22* (2) 105-115.

Cleary, J.G., Wyvill, B.L.M., Birtwistle, G., and Vatti, R. (1983) "Design and Analysis of a Parallel Ray Tracing Computer" *Proceedings of the XI Association of Simula Users Conference,* Paris.

Cleary, J.G. (1984) *A distributed graphics system implemented in Prolog.* Research Report 84/173/31, Department of Computer Science, University of Calgary.

Cleary, J.G., Lomow, G.A., Unger, B.W., and Xiao, Z. (August 1985) "Jade's IPC Kernel for Distributed Simulation" *Proceedings of the Association of Simula Users Conference,* Calgary, Alberta.

Cleary, J., Unger, B., and Li, X. (February 1988) "A Distributed AND-Parallel Backtracking Algorithm Using Virtual Time" *To appear in SCS Multi-88 Distributed Simulation Conference.*

Dewar, A.D. (1985) *A Graphical Debugger for Prolog.* MSc Thesis, Department of Computer Science, University of Calgary.

Jade (October 1985) *Jade User's Manual, Volume I: Developing Distributed Systems in Jade, Volume II: The Jade Workstation, Volume III: The Jade Graphics System, Volume IV: An Example System, Volume V: The Workstation Based Editor.* Technical Reports, Department of Computer Science, University of Calgary.

Jefferson, D. (July 1985,) *Virtual Time.* ACM Transactions on Programming Languages and Systems.

Joyce, J. and Unger, B.W. (January 1985) "Graphical Monitoring of Distributed Systems" *Proceedings of the SCS Conference on AI, Graphics, and Simulation,* San Diego, California.

Joyce, J., Lomow, G., Slind, K., and Unger, B. (May 1987) "Monitoring Distributed Systems" *ACM Transactions on Computer Systems, 5* (2).

Li, X., Unger, B., and Cleary, J. (February 1988) "Communicating Sequential Prolog" *To appear in SCS Multi-88 Distributed Simulation Conference.*

Lomow, G.A. and Unger, B.W. (February 1985) "Distributed Software Prototyping and Simulation in Jade" *INFOR, 23* (1) 69-89.

Lomow, G., Cleary, J., Unger, B., and West, D. (February 1988) "A Performance Study of Time Warp" *To appear in SCS Multi-88 Distributed Simulation Conference.*

Neal, R., Lomow, G.A., Peterson, M., Unger, B.W., and Witten, I.H. (May 1984) "Experience with an inter-process communication protocol in a distributed programming environment" *Proceedings of the Canadian*

Information Processing Society Session '84, Calgary, Alberta.

Unger, B., Dewar, A., Cleary, J.G., and Birtwhistle, G. (October 1985) *A Distributed Software Prototyping and Simulation Environment: JADE.* Research Report 85/216/29, Department of Computer Science, University of Calgary.

Unger, B, Bidulock, D., Lomow, G., Belanger, P., Hawkins, C., and Jain, N. (August 1982a) "An Oasis Simulation of the ZNET Microcomputer Networks" *IEEE Micro, 2* (3).

Unger, B and Bidulock, D. (September 1982b) "The Design and Simulation of a Multi-computer Network Message Processor" *Computer Networks, 6* (4).

Wyvill, B.L.M. (March-April 1977) "Pictures-68 Mk.1" *Software--Practice and Experience, 7* (2) 251-261.

Wyvill, B.L.M., Neal, R., Levinson, D., and Bramwell, R. (May 1984) "JAGGIES--a Distributed Hierarchical Graphics System" *Proceedings of the Canadian Information Processing Society Session '84,* 214-217, Calgary, Alberta.