THE JADE APPROACH TO DISTRIBUTED SOFTWARE DEVELOPMENT

Brian Unger, Alan Dewar, John Cleary and Graham Birtwistle
Department of Computer Science
The 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. A common inter-process communication protocol provides a uniform interface among the components. A window system allows the user to interact with many different processes at once and allows for multiple views of the same process. A hierarchical graphics system is provided for use with documentation and programming, and for support of monitoring. The non-determinism of distributed systems may be controlled in order to provide repeatability of executions and to aid in prototyping real-time distributed software. Finally, the formal specification of inter-process events in Jade is supported by a communications protocol verifier, allowing run-time consistency checking. We describe these tools and their use in the development of distributed software for the control of a simulated system of parking lots.

I. INTRODUCTION

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. Figure 1 shows how the various components of Jade fit into these levels. Below, we describe the levels, proceeding bottom-up. We use the parking-lot problem as a running example to illustrate the facilities offerred at the various levels.

Hardware and kernel levels

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 is currently underway 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 provided 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.

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. A number of standard monitors, both textual and graphical, are provided for general use. Monitoring of Jipc events is provided in such a way that the user may also write monitor consoles specific to particular applications or debugging techniques.

Prototyping level

Prototyping of distributed software is 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 using a different actual distribution. The inter-process interactions faithfully conform to the way they would appear in the target system. Finally, work is also underway on the implementation of a time-warp [Jefferson 1982, Jefferson 1983] version of Jipc [Cleary 1985].

A distributed parking-lot control system

The examples in this paper are drawn from a simulation of a distributed parking-lot control system, which could serve as a prototype for a real system. In the simulation, there are a number of parking lots, each having a fixed capacity. Each lot has a single entrance/exit gate which is controlled by a processor dedicated to that lot. Cars arrive at the lots at random intervals, wishing to park for a random length of time.

A car arriving at a lot causes a request to be made for space in the lot. If such space is available, the car is allowed to enter and park. If no more parking spaces remain, the parking lot sends a request to each of the other lots in turn, until it is able to reserve a space for the car elsewhere. In the event that there is no available space anywhere, the car simply leaves.

The parking-lot simulation makes a number of simplifying assumptions about the behaviour of its components. Parking-lot computers are assumed to remain working at all times. Communication among cars and parking-lot computers is assumed to be reliable. A car which has a space reserved for it in another lot must eventually show up at that lot, and a car which finds no space available for it must leave the system. While these would not be wise assumptions to make in a real system, they do enhance the simplicity and clarity of the example system. In an actual complete implementation, additional error-checking code would be added to handle violations of these assumptions.

The parking-lot control simulation makes use of a number of different processes. Each car is represented by a single process. Each parking lot is comprised of three processes: a *manager* process which keeps track of the current occupancy and any reservations; an *ear* process which accepts requests to the parking lot and buffers these until the manager asks for them; and a *handler* process which acts as the parking lot's interface to the outside world. The ear process is needed in order to avoid the potential deadlock which would exist if two managers tried to

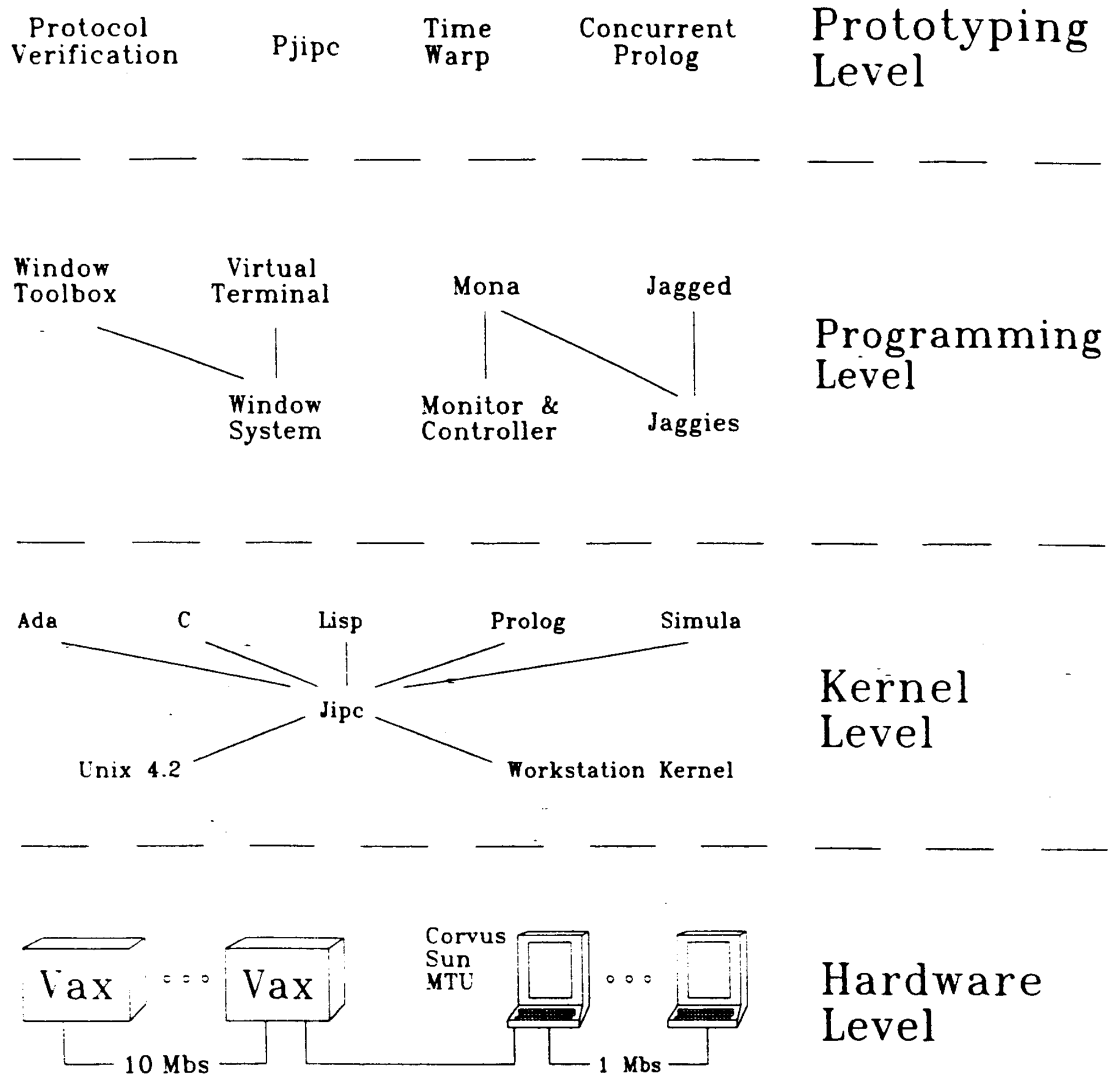

Figure 1. Jade Components and Structures

talk to each other at the same time, each blocking on the other's reply. The handler process allows for the separation of the external-world interface from the internal details of the system.

II. 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. This allows for synchronization among the various processes in the system.

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.

Figure 2 shows an example of how processes communicate via Jipc. Excerpts of the manager and ear processes, as well as the entire handler process, are shown here. The manager process is created by a *creator* process (not shown), and it in turn creates the ear and handler processes. The process id of the ear is sent to the handler, and vice versa. Both of these process ids are then sent to the creator, which eventually replies with a message containing the process ids of all of the ear processes in the system.

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 down-loaded 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.

III. 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,

```
/* manager.c */

j_process_id    ear_process[NLOTS];         /* ear process ids of other parking lots */
j_process_id    my_ear, my_handler;         /* ear & handler process ids of this parking lot */
j_process_id    creator_id;                 /* process id of creator of this system */

...

initialize(id)
int     id;
{
        char    ear_name[J_MAX_PROCESS_NAME+1];
        char    handler_name[J_MAX_PROCESS_NAME+1];
        int     i;

        j_initialize();                         /* initialize as a Jipc process */
        creator_id = j_parent_process();

        (void) sprintf(ear_name, "ear%d", id);
        (void) sprintf(handler_name, "handler%d", id);
        my_ear = j_create_process(ear_name, "ear", 0);                  /* create the ear */
        my_handler = j_create_process(handler_name, "handler", 0);      /* create the handler */

        j_putp(my_ear);                         /* inform handler of ear process id */
        j_send(my_handler);

        j_putp(my_handler);                     /* inform ear of handler process id */
        j_send(my_ear);

        j_putp(my_ear);                         /* inform creator of ear & handler process ids */
        j_putp(my_handler);
        j_send(creator_id);

        for (i=0; i < no_of_lots; i++) {        /* get ear process ids of other parking lots */
                ear_process[i] = j_getp();      /*    (from creator's reply)                 */
        }
}


/* ear.c */

...

initialize()
{
        j_initialize();                         /* initialize as a Jipc process */
        my_manager = j_parent_process();
        j_receive(my_manager);                  /* get handler process id from manager */
        my_handler = j_getp();
        j_reply_null(my_manager);
}


/* handler.c*/

#include <jipc.h>

main()
{
        j_process_id    my_ear, sender_id;

        j_initialize();                         /* initialize as a Jipc process */
        j_receive(j_parent_process());          /* get ear process id from manager */
        my_ear = j_getp();
        j_reply_null(j_parent_process());

        for( ; ; ) {
                sender_id = j_receive_any();/* receive any car event */
                j_send(my_ear);             /* direct the message to the ear process */
                j_reply(sender_id);         /* transmit manager's reply to the car process */
        }
}
```

Figure 2. Communication via Jipc

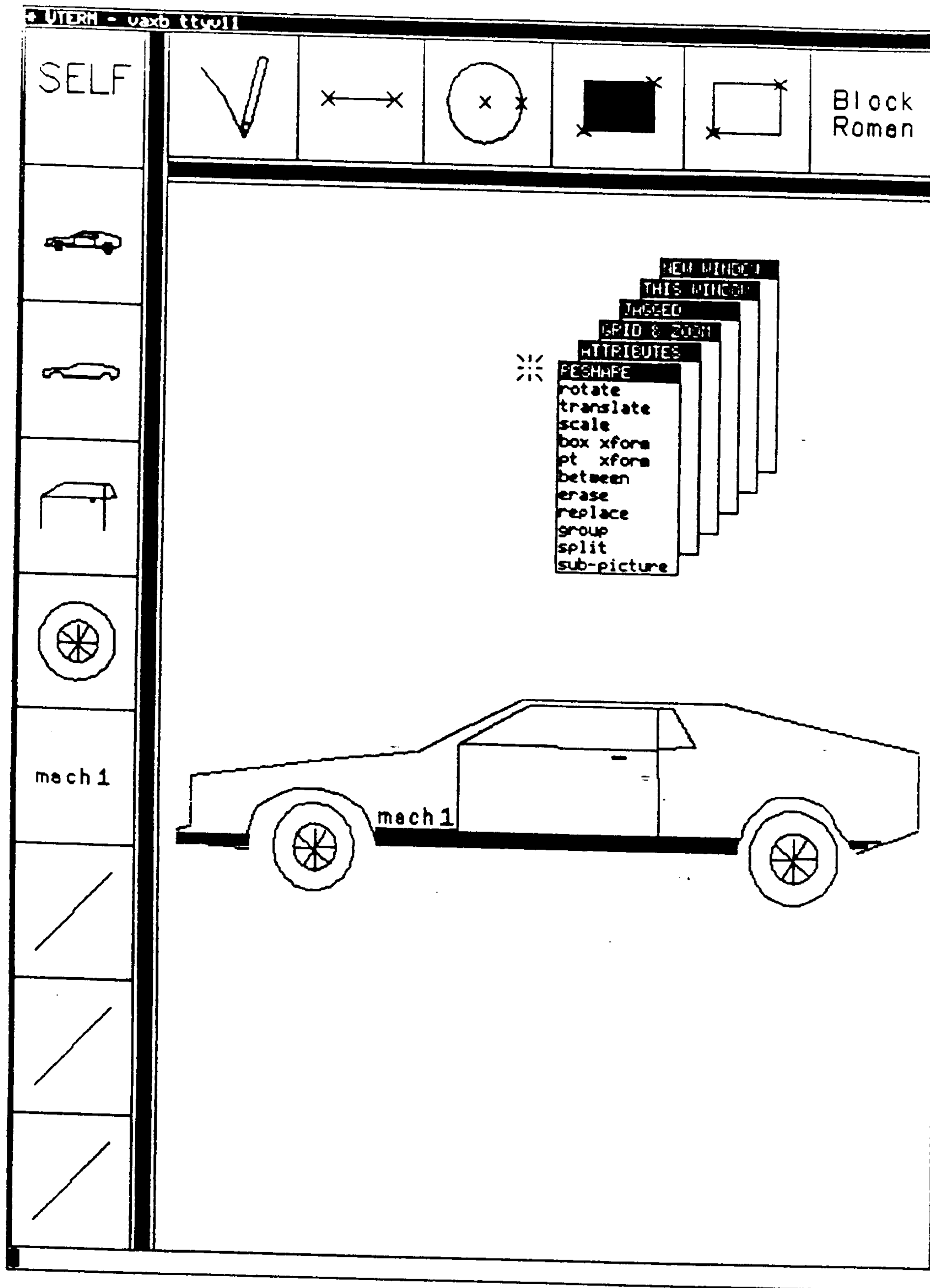

Figure 3. Jagged

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 typical Jagged session is shown in Figure 3.

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].

IV. MONITORING

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 and as shown in Figure 4. In this figure, three processes running on Machine 1 and two processes on Machine 2 comprise Jipc System 28, while three processes on Machine 3 comprise Jipc System 42. A Jipc system is a distributed concurrent program. 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.

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.

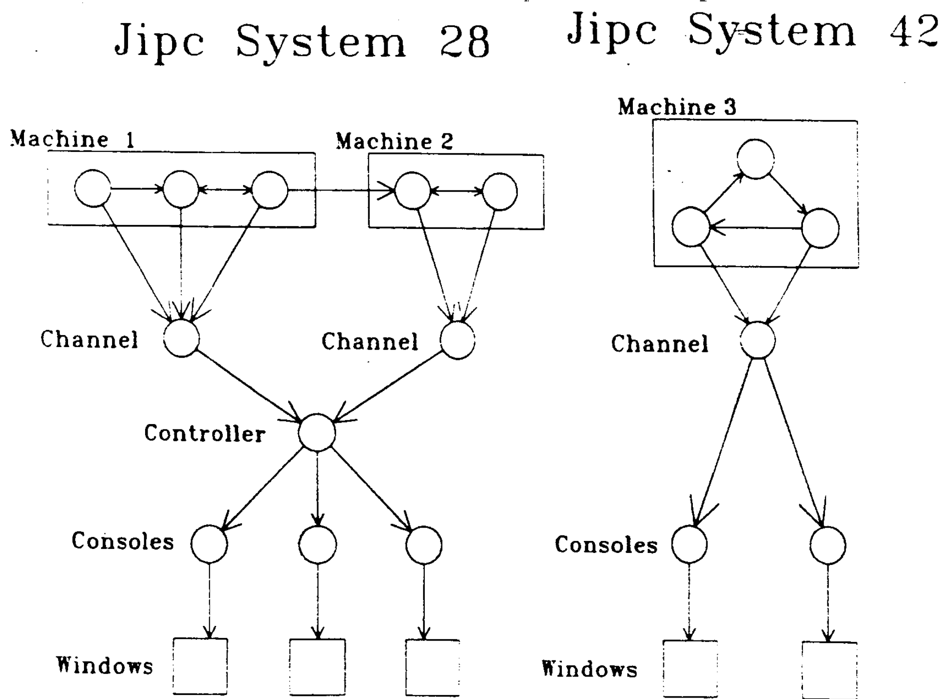

Figure 4. Monitoring Jipc Events

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.

Standard monitoring consoles

A textual monitor is available which simply displays each event that occurs, along with the contents of any associated buffer. An example of output from this monitor is shown in Figure 5. A more sophisticated monitor, Mona [Joyce 1985], provides a graphical representation of the executing system. Processes are represented by labelled boxes, with inter-process communication denoted by arrows connecting the processes. A dashed arrow indicates that a send has taken place, with a solid arrow indicating that the message has been received. When the reply is completed, the arrow disappears. If a process does a receive before a message has been sent to it, Mona places a diamond or a small rectangle within the box for the receiving process. As can be seen from Figure 6, this graphical representation gives the user a much more complete and intuitive view of the overall system state at any time. In this figure, for instance, it can easily be seen that ear2 has just received a message from manager1, while manager2 and handler2 have each sent ear2 a message which has not yet been received. The Jade window system, along with menus and help windows, may also be seen in this figure.

In addition to these consoles which portray Jipc events to the user, a console also exists for recording a sequence of events for later playback. Playback is achieved by simply sending the recorded events to the channel process and invoking any monitor consoles desired. The channel process does not distinguish between real Jipc events and those sent to it as part of playback, so a faithful reproduction of the events is guaranteed.

Detection of communication deadlock is possible and has been implemented as a console. The sole function of this console is to keep track of the communications taking place and report to the user whenever deadlock occurs. Similar consoles could easily be implemented to perform other high-level error-detection functions.

Application-specific monitoring consoles

Although Mona provides a good graphical overview of the communications taking place in a Jipc system, a user may wish to have a console which displays the execution in terms more specific to the particular application. Jaggies and a library of monitoring routines make this possible. An example is shown in Figure 7. Here, parking lots 0 and 1 are full, while parking lot 2 has two spaces left. A car has arrived at each of lots 0 and 1, and these lots are each in the process of sending a request to lot 2 to reserve a parking space. (Prior to sending this request, each lot sent a similar request to the other lower-numbered lot, only to discover that it had no space either.)

```
...
varb.manager0: send: to varb.ear0
varb.ear0: receive_any
varb.manager1: send completes: reply received (null message)
varb.ear2: reply: to varb.manager2
varb.ear0: receive_any completes: message from varb.manager0
varb.manager1: send: to varb.ear1
varb.manager2: send completes: reply received
varb.ear2: receive_any
varb.ear0: reply: to varb.manager0
varb.ear1: receive_any completes: message from varb.manager1
varb.manager2: send: to varb.ear2
varb.ear0: receive_any
varb.manager0: send completes: reply received
varb.ear1: reply: to varb.manager1
varb.ear2: receive_any completes: message from varb.manager2
varb.manager0: send: to varb.ear2
varb.manager1: send completes: reply received
varb.ear1: receive_any
varb.ear2: reply: to varb.handler2
varb.manager1: send: to varb.ear2
varb.handler2: send completes: reply received
varb.ear2: reply: to varb.manager2 (null message)
varb.handler2: reply: to varb.simulator2
varb.ear2: receive_any
varb.manager2: send completes: reply received (null message)
varb.handler2: receive_any
varb.simulator2: send completes: reply received
...
```

Figure 5. Textual Monitor

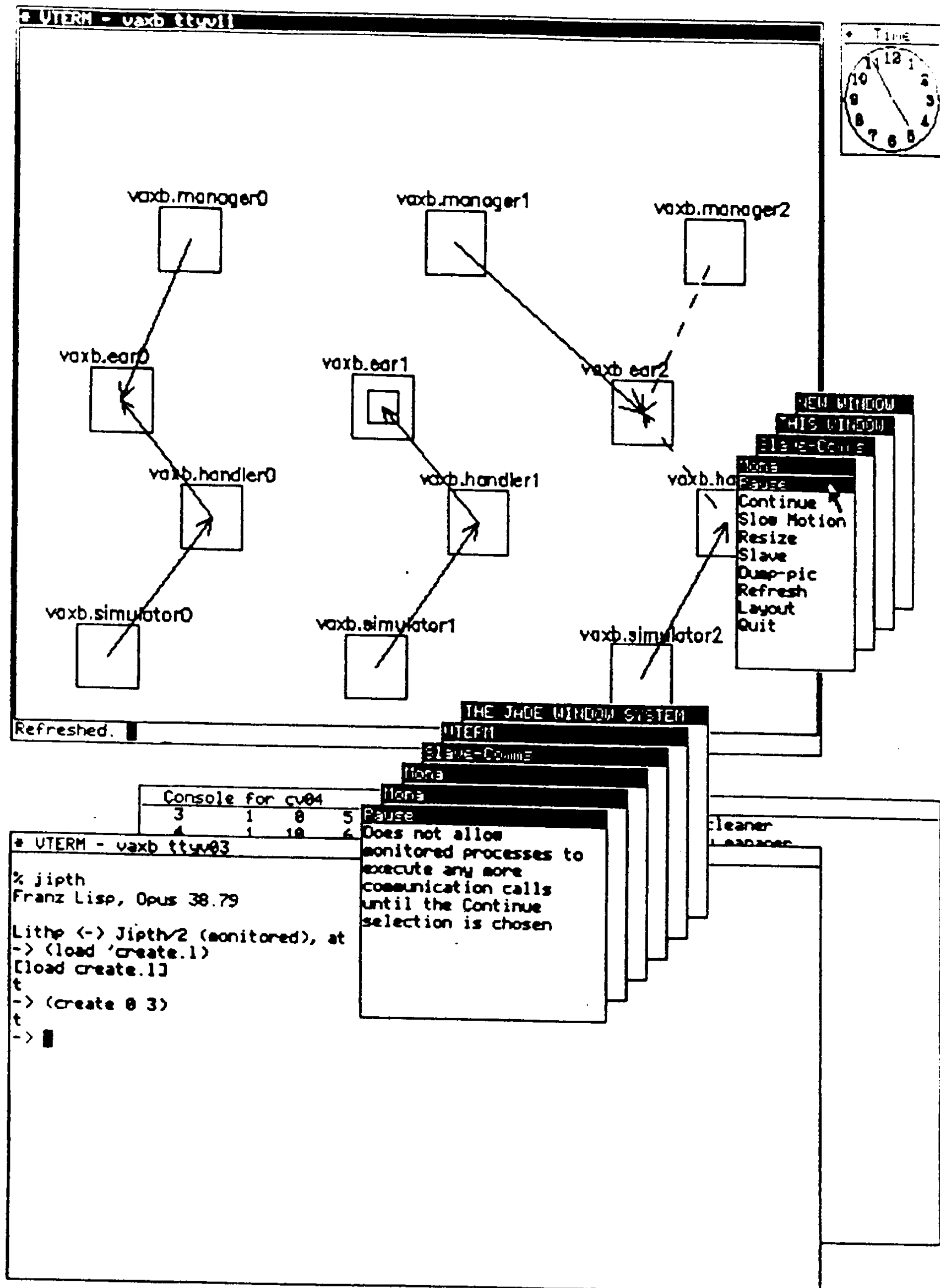

Figure 6. Graphical Monitor

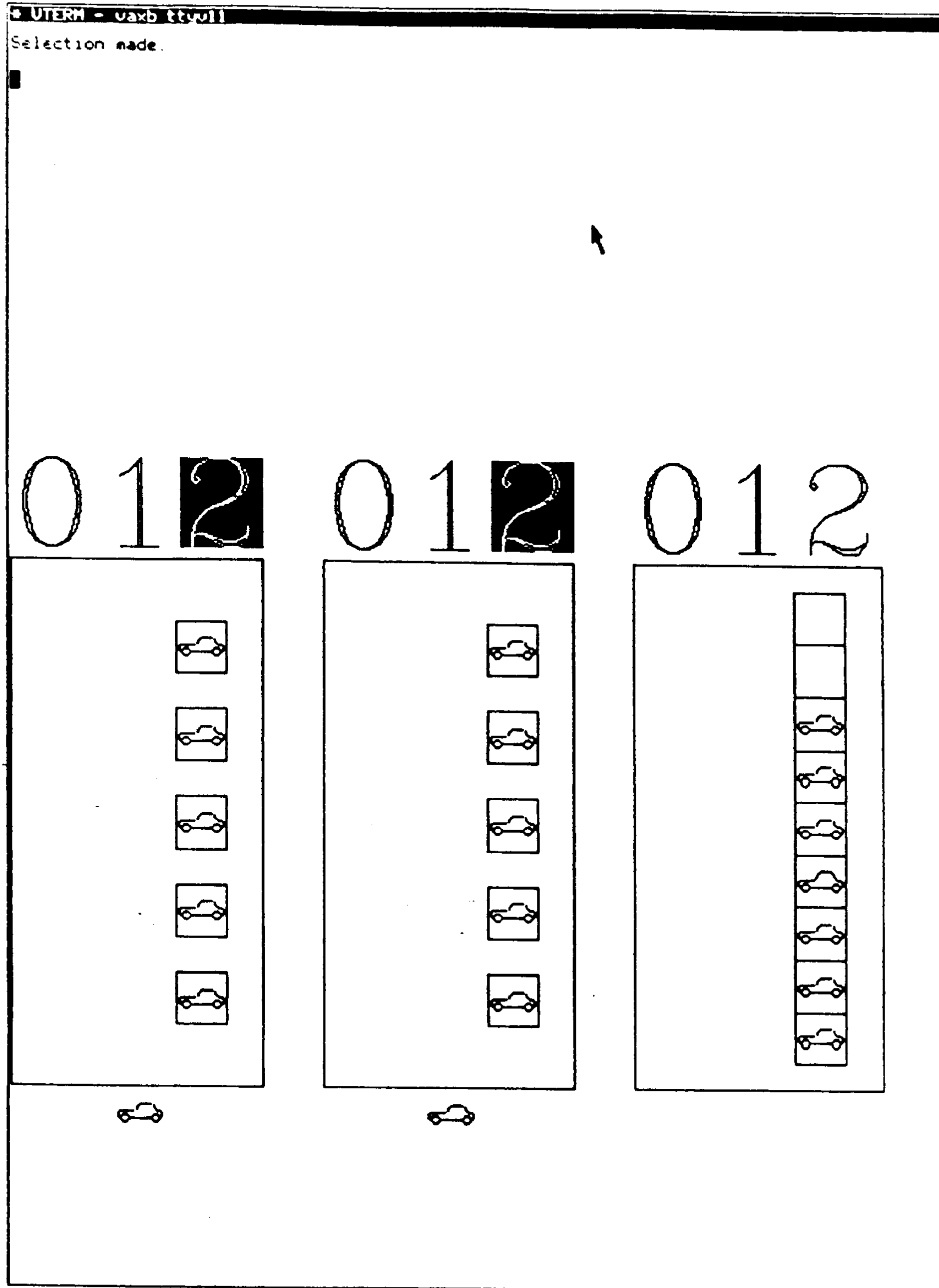

Figure 7. Special-Purpose Monitor

V. FORMAL SPECIFICATION

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. Figure 8 shows an example of the JDL specification for the parking-lot system.

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. For an example of a more sophisticated protocol description model and its use in refinement of specifications, see [Merlin 1983].

VI. 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. 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. A five-volume Jade User's Manual has been produced [Jade 1985], Volume IV of which provides full details of the distributed parking-lot simulation described here.

In future, we hope to pursue a still higher level of program-development support. A programmer should be able to move from a formal specification to an implementation by a number of refinements, each provably consistent with the previous level. Such a support environment should greatly enhance the software-development process.

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 Naval Research Laboratory for providing us with the use of an SMI Sun workstation. 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. We would particularly like to acknowledge the many significant contributions of Radford Neal and Greg Lomow.

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" in *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" in *Proceedings of the Association of Simula Users Conference*. Calgary, Alberta.

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. and Sowizral, H. (December 1982) *Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control*. Technical Report, The Rand Corporation, San Diego, California.

Jefferson, D. and Sowizral, H. (August 1983) *Fast Concurrent Simulation Using the Time Warp Mechanism, Part II: Global Control*. Technical Report, The Rand Corporation, Santa Monica, California.

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

Merlin, P. and Bochmann, G.V. (January 1983) "On the Construction of Submodule Specifications and Communication Protocols" *ACM Transactions on Programming Languages and Systems*, 5 (1) 1-25.

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" in *Proceedings of the Canadian Information Processing Society Session '84*. Calgary, Alberta.

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" in *Proceedings of the Canadian Information Processing Society Session '84*, pp 214-217. Calgary, Alberta.

```
PROCESS_CLASSES:
        <manager> = manager?;                                  -- '?' matches any single character
        <ear> = ear?;
        <handler> = handler?;
        <parkinglot> = <manager> OR <ear> OR <handler>;
        <simulator> = simulator?;
        <creator> = creator;

BUFFERS:
    EXTERNAL:
        null = ();                                             -- an empty buffer
        initialize1 = (P: <handler>);                          -- any handler process
        initialize2 = (P: <ear> ; P: <handler>);
        initialize3 = (P*: <ear>);                             -- zero or more ear processes
        initialize4 = (P: <handler> ; I);                      -- a handler process and an integer
        check_message = (S: "check_info");                     -- an exact string
        event = (S: ONE OF "enter","leave", "transfer");
        reply_event = (S: ONE OF "reject","accept","ok","direct"; P*: <handler>);
        request_space = (S: "request"; P: <ear>);
        reply_space = (S: ONE OF "reply_yes", "reply_no"; P*: <handler>);

EVENTS:
        creator CREATE <simulator>;
        creator CREATE <manager>;
        <manager> CREATE <handler>;
        <manager> CREATE <ear>;

        <manager> SEND creator USING initialize2;
        creator RECEIVE <manager> USING initialize2;
        creator REPLY <manager> USING initialize3;
        <manager> REC_REPLY creator USING initialize3;

        creator SEND <simulator> USING initialize4;
        <simulator> RECEIVE creator USING initialize4;
        <simulator> REPLY creator USING null;
        creator REC_REPLY <simulator> USING null;

        <manager> SEND <handler> USING initialize3;
        <handler> RECEIVE <manager> USING initialize3;
        <handler> REPLY <manager> USING null;
        <manager> REC_REPLY <handler> USING null;

        <manager> SEND <ear> USING initialize1;
        <ear> RECEIVE <manager> USING initialize1;
        <ear> REPLY <manager> USING null;
        <manager> REC_REPLY <ear> USING null;

        <simulator> SEND <handler> USING event;
        <handler> RECEIVE <simulator> USING event;
        <handler> SEND <ear> USING event;
        <ear> RECEIVE <handler> USING event;
        <ear> REPLY <handler> USING reply_event;
        <handler> REC_REPLY <ear> USING reply_event;
        <handler> REPLY <simulator> USING reply_event;
        <simulator> REC_REPLY <handler> USING reply_event;

        <manager> SEND <ear> USING check_message;
        <ear> RECEIVE <manager> USING check_message;
        <ear> REPLY <manager> USING event;
        <manager> REC_REPLY <ear> USING event;
        <ear> REPLY <manager> USING request_space;
        <manager> REC_REPLY <ear> USING request_space;
        <ear> REPLY <manager> USING reply_space;
        <manager> REC_REPLY <ear> USING reply_space;

        <manager> SEND <ear> USING reply_event;
        <manager> SEND <ear> USING request_space;
        <manager> SEND <ear> USING reply_space;
        <ear> RECEIVE <manager> USING reply_event;
        <ear> RECEIVE <manager> USING request_space;
        <ear> RECEIVE <manager> USING reply_space;
        <ear> REPLY <manager> USING null;
        <manager> REC_REPLY <ear> USING null;
```

Figure 8. JDL Specification