Olympus: An Extensible Modeling and Programming System *

Gary J. Nutt

CU-CS-412-88

University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

* This research has been supported by NSF cooperative agreement DCR-8420944, NSF Grant No. CCR-8802283, US West, and Sun Microsystems.

OLYMPUS: AN EXTENSIBLE MODELING AND PROGRAMMING SYSTEM

Gary J. Nutt

CU-CS-412-88 October, 1988

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

(303) 492-7581
nutt@boulder.colorado.edu

This research has been supported by NSF cooperative agreement DCR-8420944, NSF Grant No. CCR-8802283, U S West, and Sun Microsystems.

ABSTRACT

Olympus: An Extensible Modeling and Programming System

In this paper, we describe Olympus, a system we have built to interpret programs and formal models of computation. Olympus employs a graphical, interactive user interface that enables the user to control the system as it interprets the program or model.

The system is built using a combination of libraries, type hierarchies, and distributed client and server processes. The system "in-the-large" is built around a client-server style architecture. Libraries and the object-oriented programming paradigm are used "in-the-small" for implementing individual clients and servers.

The architecture of the system allows various parts to be reused and extended without rebuilding the entire system. As a result, the basic interpretation system can be used to interpret a variety of formal models of computation and programming languages, and the user interface can be extended with type hierarchies to represent a spectrum of visual network interfaces.

1. INTRODUCTION

1.1. The Problem

Our research program is concerned with studying means by which programmers can construct software to make effective use of distributed hardware systems. The fundamental assumption behind our approach is that distributed software is sufficiently complex that a designer can benefit considerably from interactive support systems. For example, it is difficult to partition a computation into schedulable units of computation without *a priori* knowledge of the result of any particular partitioning strategy. It seems likely to us that the structure of the partition will have a far larger effect on the ultimate performance of a distributed computation than will localized tuning of code within a schedulable unit of computation.

Choosing a particular model on which to base the user interface is very important to the success of a support system, and very difficult to do so that it is acceptable to a wide range of applications and users. We each have a set of preconceived notions about modeling primitives, often based largely on aesthetics. One goal of our general modeling system is that it should support a number of different formal models which have similar semantics. Thus, the system should provide fundamental operation as a simulation and animation system with a somewhat arbitrary user/model interface.

As a result, our research projects center around the application of several different formal models of computation as bases for interactive system support. One project is concerned with interactive and distributed simulation systems [18]. Another project looks at multi-tiered models to study partitioning tools [8, 19]. Two other projects attempt to use the underlying family of models of computation as a basis of interpretive programming systems [1] or as a frontend for generating functional languages.

All of these projects benefit from the existence of computer tools to explore feasibility and to compare approaches; however, each system is different. This common need for similar tools has led us to employ technologies for reusing software.

The *Olympus*† system is an interactive, distributed model/language interpretation environment for bilogic precedence graphs (BPGs). While Olympus is based on a particular formal model of computation, the architecture of the system allows parts of it to be reused in various contexts to support other languages and models of computation.

BPGs are interpreted precedence graphs that incorporate conjunctive (AND) and disjunctive (exclusive OR) logic. Thus, very general control flow patterns (alternative, select, fork, and join) are supported by the model. BPGs also describe possible data flow among interpreted nodes. Like Petri nets, BPGs represent the status of the model through a distribution of tokens on nodes and edges. A formal and complete description of BPGs can be found in [20].

The distributed computation partitioning project [8] employs two levels of formal models: Distributed Computation Precedence Graphs (DCPGs) and an abstraction called the Process Architecture Model (PAM). DCPGs are similar to BPGs, so the Olympus system is an obvious starting point for providing machine support for the model. PAMs are an abstraction of DCPGs in which one studies the partitioning of function into collections of schedulable units of computation. Machine support for PAM also requires interactive editing and interpretation, just as is the case for BPGs and DCPGs. However, specific meaning of constructs in a PAM differ considerably from those defined for a DCPG (or a BPG).

The hypercube programming project [1] focuses on visual paradigms for constructing programs for parallel local memory machines that employ the binary n-cube interprocessor routing mechanism. There are two sublanguages used in this project, one based on a three-dimensional spreadsheet paradigm, and the other on BPGs. The BPG-

† Olympus is a variety of strawberry. It is also a member of our family of *fragaria* (strawberry) distributed software systems.

based language is used to express programs as refinements of control flow diagrams; fine-grained and large-grained programs may be generated. The spreadsheet language is intended to be a simplified user interface to hypercubes; it inherently generates fine-grained programs which can be translated into BPGs. Both projects require a user interface that has many of the same functions as the BPG user interface, but which are unique to the particular language. The projects use a common interpreter, which is a version of the BPG interpreter tailored to hypercube scheduling.

There is another class of projects being investigated in which a visual formal model of computation is used as an alternative form for expressing programs written in linear languages, including both procedural and functional languages. In these studies, a linear language program is either translated to a network form or generated as a network (such as a BPG). The Olympus interactive editor and analysis tools are used to apply heuristics to the detection and explicit identification of parallelism in the program. The resulting graph model is then translated (back) into a linear form that can be compiled and interpreted in a conventional software environment.

There are other interactive modeling systems that could reuse the software we have built, including extended queueing networks, Petri net modeling systems, CASE systems, and special purpose modeling systems (e.g., telephone network modeling).

1.2. Interactive Interpretation Systems

An interactive language and model interpreter should provide several basic functions:

(1) It should have a mechanism for interactively creating and editing model instances.

(2) The user should be able to exercise a model with complete control over the interpretation, e.g., the user should be able to interrupt the interpretation at any moment (without setting breakpoints *a priori*).

(3) When an interpretation is interrupted, the user ought to be able to browse the state of the interpretation and even change the state prior to continuation.

(4) If the system is interpreting a model in scaled real time, then the user ought to be able to change the time scale while the model is in operation.

(5) The interactive system should also allow editing and interpretation to proceed in parallel, even though there will be times during which the user might leave the interpreter in an unusual state.

The Olympus interactive system is split into a *frontend* and a *backend*. The purpose of the frontend is to implement the user interface, and to encapsulate as much of the syntax of the model itself as is possible. The backend provides storage and interpretation of the program or model; it should implement the semantics of the model independent of the frontend design. The overall system operation can be described in terms of the two components.

The Frontend

The frontend serves two main purposes: First, it must implement the human factor, and many of the cognitive, aspects of the interaction between the user and the machine, i.e., it must act as a user interface. Second, it must implement the syntax of the model or program specification, e.g., if the model of computation uses boxes to represent basic blocks of computation, then the frontend is responsible for drawing boxes, interconnecting them, etc.

For example, the frontend might take the approach that the user interface need only support a keyboard and a 25x80 character screen. In this case, the model or program is specified to the interface by typing some linear description of the model. The human factor aspects of the interface are issues such as keyboard mappings, escape characters for invoking commands, etc. (The first Olympus frontend employed such an interface.)

At the other end of the spectrum, the frontend may be based on a point-and-select graphics interface. Such a frontend may provide a pallet of model primitives that can be placed on a "canvas," and which can be interconnected using arcs.

The frontend generates a structured internal representation of the model stored in the backend. In order to build consistent internal representations, the frontend performs syntactic analysis of the graph as it is being constructed by the user.

To the extent that the syntax of a particular model or language can be separated from the semantics, then the frontend can be made to be independent of the backend. Olympus provides a backend which implements the semantics of BPGs, so it is expected that the frontend will implement any of a number of models or languages of differing syntax that can be mapped onto BPG semantics, see [20].

Finally, the frontend must be the user interface to the backend as well as to the model construction portion of the system. Thus, it is a console for the system in which the user constructs a model, then exercises the model through the same interface.

The Backend

The backend can be thought of as an animator and a simulator for BPGs, i.e., it executes the control flow of the graph, interpreting nodes as dictated by the BPG model. The backend reacts to directives from the frontend, then notifies the frontend of the changing model status as the interpretation process takes place.

The backend supports single-stepped interpretation of a model with its data, as well as continuous operation. A single interpretation step refers to the occurrence of one BPG event, i.e., the initiation or termination of one task firing. Single stepping through an interpretation changes the real time relationship between simulation steps, but it has been found to be useful for observing the order of task execution for tasks that terminate at nearly the same time, or for simply observing the changing state of the BPG at a leisurely pace.

Continuous operation causes the interpreter to move tokens from task-to-task in real time, as determined from the task interpretations. In order to provide more flexibility for observing the dynamics of operation, the interpreter also provides a means by which the frontend can specify the ratio of real time to time used by the interpreter.

To see patterns in the animation of a BPG, it is often desirable to let the animation run long enough to reach steady state operation. This typically requires that a series of tokens be introduced to the input tasks for the model so that the transient internal task activity can stabilize. The model itself can be constructed so that it effectively introduces a new token at desired intervals of time (by creating a small BPG with a loop, then attaching it to an input task). Because of the repeated need for such a construct, the system provides an "auto load" facility for specifying task marking from an infinite population. The user auto loads a location by specifying a probability distribution function describing the interarrival time for new tokens for the task.

At the direction of the frontend, the backend will report summary interpretation information for each task and repository in the model. The report includes information about the activity of each task, expressed in terms of the number of times that the task was activated, and the amount of time that the task was active. Repository reports indicate the number of read and write references for the repository.

1.3. The Olympus Experiment

The frontend and the backend, i.e., the system in-the-large, is designed as a client-server pair of processes. There have been several implementation of the frontend; some of them have been implemented (in-the-small) using object-oriented type hierarchies, while others are implemented as traditional procedural language clients using libraries of tools.

The formal model provides a language for describing how systems behave. Olympus provides a medium for expressing model instances, and for studying these models by observing their reaction to different conditions. By constructing an interactive system to support the model, it is possible to create an environment in which alternatives -- changes in loading conditions, changes in parameters, or changes in the model itself -- are easy to explore.

The first version of Olympus was operational in July, 1987. Since that time, it has gone through several minor modifications and one major modification. The original version was a single-process program implemented on Sun workstations. The current version (October, 1988) is implemented as a distributed client-server program.

The Olympus frontend has been reimplemented four times, i.e., there have been five versions. The original single-process implementation combined the backend and frontend; the frontend was a line-oriented interface used to debug the backend. The next frontend, also in the single-process architecture, was implemented on SunView [28]. This version was intended to replace the line-oriented interface with a point-and-select graphical interface. The first client-server implementation also used SunView as the basis of the frontend client; the backend server was implemented as a single process in the Sun C environment. The second frontend client was implemented in Lisp on a Symbolics workstation, interacting with the backend server using sockets over an Ethernet. The third frontend client, currently still being refined, was implemented in the NeWS environment [29].

The first implementation of the backend used a straight-forward implementation for the interpreter, based on conventional simulation techniques. The first backend server was implemented as a single process translation of the original code, i.e., the original code was modified so that it could operate as a server, interacting with a distinct client process. This server did not incorporate BPG node interpretations other than timing; thus it could only support animation. The current backend server is itself implemented as a collection of processes which are able to interpret nodes with interpretations expressed in C and other compilable languages (see below).

In the remainder of the paper, we will describe the design and implementation of the current Olympus frontend and backend.

1.4. Related Work

There are a number of simulation systems used for performance prediction that provide an individual user interface to specific modeling facilities. In each case, there is a pictorial representation of the model of operation; the analyst uses graphical support tools to describe the model of operation in the particular language of representation. Commercial products are available to support graphical interfaces to simulation software [6, 12]. The representation is then used to define a simulation program of the model.

In some cases, the system focuses only on providing a graphical editor for constructing a machine-readable model; the model can then be translated into a traditional simulation program. SIMF is one example of this type of system; it provides an interactive editor for preparing SLAM programs [26].

In other cases, the system implements the formal model of computation as the basis of the simulation system, but does not provide a graphical user interface, as was done in our original Olympus implementation, e.g., see [22].

Newer systems incorporate a visual editor along with some form of machine to execute the resulting model (either a translator or an interpreter) under the control of the modeling system. The user specifies the model using the editor, then runs the simulation. Generally, the simulation can be invoked to run continuously, or single-stepped through event executions. Some systems allow the simulation to be halted so that the model or parameters can be changed, then the simulation can be restarted. The PAWS/GPSM simulation system [3, 11, 12], the Performance Analysis Workstation (PAW) [13], PARET [16], and Quinault [17] are all examples of this type of system.

GADD [15] is intended to simulate system modules that interact with other modules using messages. The focus of the model is on message traffic analysis, so all modules are simulated by corresponding simulation modules and the message traffic maps one-for-one with the target system message traffic (cf. Misra's discussion of distributed simulation [14]). As a consequence, it is possible to interconnect simulations with fully-implemented components, thus providing a testbed debugging environment.

Visual programming system have an editing capability for expressing a program as a graphical interpreted model. The model can then be translated into target machine code or interpreted in an abstract machine environment. There are several systems that use machines to support visual programming abstractions, including work done by Browne and his students [4,5], MCC's Verdi [9], Upconn [2], and Poker [25]. Longer collections of papers on visual programming can be found in [7,21].

2. THE SYSTEM ARCHITECTURE

Any instance of the Olympus modeling system is some collection of the following set of modules: *Console*, *Model Editor*, *Model Storage*, *Marking Storage*, *Task Interpreter*, and *Repository Interpreter*.†

Console

A Console is a window onto the model which illustrates the activity of the other modules; it is used to control all parts of the system and as a viewport onto the model that is being interpreted. During animation, it is the medium for displaying the dynamics of the interpretation. The Console determines the details of the model from the Model Storage and the status of the interpretation from the Marking Storage. Notice that a Console display is only interested in the marking of a small portion of the BPG -- a portion small enough to fit into a window; it also need not operate synchronously with the Marking Storage.

Model Editor

This module is used to define a BPG model and place it in the Model Storage. The Model Editor is responsible for implementing the visual aspects of the model, thus it can be used to map various other models into BPGs by translating the model syntax that it supports into a BPG prior to storage.

Model Storage

This module provides information about the BPG that defines the model or program (BPG) to be interpreted. For example, the module responds to messages such as "return the identity of the task that is connected to the head of this arc," or "return the body of the procedure to be interpreted when this task is fired."

Marking Storage

This module responds to a message to update the BPG marking, or to indicate the current marking of an arc (as part of an atomic transaction).

Task Interpreter Class

The Task Interpreter evaluates a task procedure. Whenever tokens enable a task, the Task Interpreter will query the Model Storage to obtain a procedure definition, then interpret the procedure on the token. Upon completion of the interpretation, the instance will notify the Marking Storage. The Task Interpreter will repeat the interpretation cycle on successor tasks as determined by the marking.

Repository Interpreter Class

A Repository Interpreter is a passive interpreting machine, i.e., it will interpret a procedure for a BPG repository when it is requested to do so, but it does not enable any subsequent activity (other than response from the request). Repository interpretation can be viewed as (remote) procedure call into a monitor. Once the Task Interpreter calls a Repository Interpreter, the Repository Interpreter cannot respond to another request until it completes the current operation.

The current invocation of Olympus binds the modules described above into the frontend and backend as indicated in Figure 1.

† In general, these modules can be thought of as classes, where any particular implementation incorporates specific instances of the classes. For a simple Olympus configuration, such as the type discussed in this paper, there is only one instance of each class type statically generated when Olympus is started . We expect to take advantage of these modules as classes as the basis of further distribution of the facilities in other parallel environments.

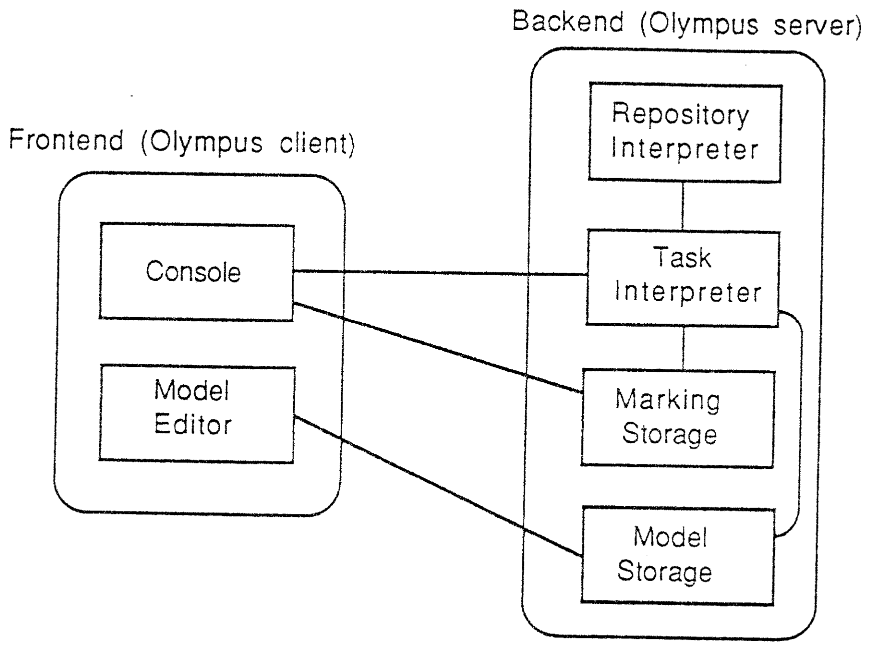

Figure 1: Olympus System Architecture

The backend -- the Olympus server -- is a persistent process that is started independently of any particular frontend -- an Olympus client.

The client establishes a socket with the Olympus server when it is initiated; all intercommunication between the client and the server take place over the socket. The server obtains commands to perform storage and interpretation control operations from the socket, and sends display instructions to the client via the same socket.

Since the client, contains the editor, the BPG can be modified even as the Task Interpreter is in execution; the Console need only be able to multiplex control information to the appropriate module.

3. THE OLYMPUS SERVER

Since four of the modules have been implemented in the server, it is necessary for the server to provide a demultiplexing function to call different modules as required by the client. The server dispatcher is a cyclic program that blocks on a socket read for the socket connecting the client and the server. Whenever a message arrives from a client -- the server can support multiple clients simultaneously using sockets unique to a client -- it dispatchs the message to the appropriate module.

The set of messages recognized by the server is listed in Appendix A. A quick review of the Appendix indicates that there are three classes of messages: Editing directives to modify the Model Storage, editing directives to modify the Marking Storage, and control messages to the Task Interpreter. The dispatcher parses the incoming message, then passes the message to the appropriate module using procedure call.

Notice that the modules need not have been implemented as a single process. The procedure call interface among the modules can be replaced by a remote procedure call interface. Notice, also, that the modularization is intended to isolate different aspects of the model; the two storage modules need not know the semantics of interpretation, and the Task Interpreter need not know any of the details of storage.

3.1. Model Storage

The Model Storage is required to remember a model definition from session-to-session, and to remember the details of a particular model while it is being interpreted. The long term storage is accomplished by saving the definition on secondary storage (using standard Unix files). The *load_model* and *save_model* messages are used to cause the Model Storage to load/save a model image into primary store from/to the file system. Notice that when a model is loaded into the Model Storage from the file system, the client does not know anything other than the name of the model. The client can send a *redraw* message to the Model Storage, which will cause it to send a full description of the model to the client.

Thus, loaded files are stored entirely in the server process's virtual memory. The *add_<atom>* and *delete_<atom>* messages are intended to create atoms in the model; the details of the atom definition can be added and changed using other messages, e.g., *arc_label* is used to add a label to an existing arc.

The only messages that require additional explanation are the XDR-related nodes, and we defer that to a detailed discussion of the Task and Repository Interpreters.

3.2. Marking Storage

The Marking Storage is relatively simple; it is only required to remember the current state of the interpretation in terms of the token distribution on a BPG. The *add_* and *delete_token_from arc/node* place and replace tokens on different parts of the model. The two commands to *delete_all_tokens_from arc/node* are used to reinitialize a model.

3.3. Task and Repository Interpreters

The semantics of the interpretation of a BPG are implemented in the Task Interpreter. The graph portion of a BPG defines the control and data flow of the model of operation, while interpretations may be added to individual nodes.

The Task Interpreter module reads the current marking from the Marking Storage, then determines which task nodes can be fired in their current state. Firable task nodes are scheduled for interpretation. After the interpretation has been completed, then the Task Interpreter updates the marking and again determines the set of firable tasks.

BPG interpretations are procedures that can be executed on a set of local variables and global data obtained from a data repository node in the model. That is, when a task is interpreted, then some procedure is interpreted on its local data; if the BPG graph indicates that the task has read or write access to a data repository, then the procedure may reference that data repository using a built-in *repository access function.*

Each task interpretation also has an associated *time to execute*, which defines the real time to be used by the Task Interpreter if the marking is to be changed in scaled real time, i.e., the client is using the server as an animator. Typically, such times are determined by a probability distribution function.

The remainder of the interpretation specification -- the procedure declaration -- may be expressed in an arbitrary, pre-compiled language invoked using Sun's RPC/XDR (Remote Procedure Call/eXternal Data Representation) protocol [27].

Whenever the Task Interpreter fires a node, it reads the procedural interpretation for the node from the Model Storage. (Actually, it reads *the name and location* of the procedure from the Model Storage.) The Task Interpreter then performs a special nonblocking RPC on the procedure, allowing it to be interpreted by a distinct process, possibly located on another host machine, see Figure 2. The RPC is nonblocking, since the intent is to allow procedures that define tasks which fire simultaneously to be interpreted simultaneously. When the procedure has been evaluated, it notifies the Task Interpreter via another nonblocking RPC call, at which time the Task Interpreter can update the marking.

We chose to use RPC in order to separate the environment in which task interpretations are executed from the environment in which Olympus executes, and to postpone procedure binding until run time. The particular implementation supports parallel interpretation on distributed computers as a bonus. This allowed us to use the standard system facilities for compiling node interpretations, rather than implementing our own facilities. As a consequence, BPG node interpretations can be written in C, Fortan, Lisp, Prolog, or any other language which produces a compiled object module which can be invoked using RPC. The Task Interpreter (a client program, in this case) invokes the appropriate procedure (a server program with the node interpretation) whenever the control flow dictates.

Uninterpreted BPGs are nondeterministic at the OR nodes, i.e., when control flows into an OR node with multiple output edges, the system may place the output token on any of the edges. Task interpretations for OR nodes can make this decision explicit by executing an arbitrary algorithm to make the decision, then by calling a built-in procedure to tell the simulator how to place the output token when the interpretation terminates.

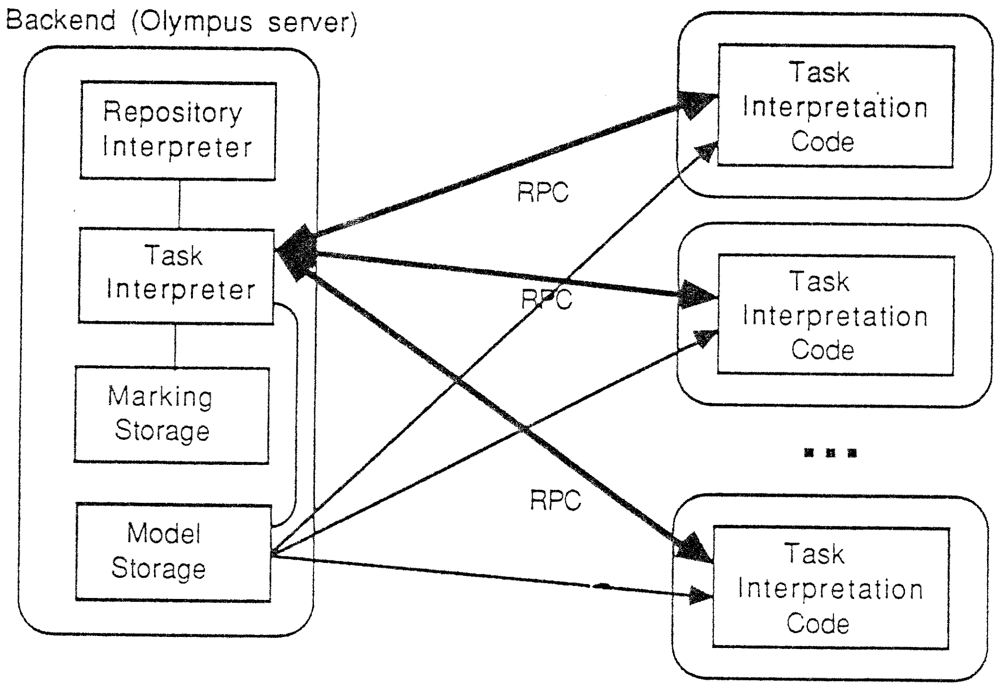

Figure 2: RPC Task Interpretation

As described above, BPGs also incorporate data flow. The meaning of an edge in the data flow diagram is that the task node *may* access any repository to which it is connected (using a data flow arc), although it is not required to do so. The direction of the arc implies the nature of the access, i.e., arcs from tasks to repositories imply write operations and arcs from repositories to tasks imply read operations. Each repository implements an *access procedure* corresponding to the read/write arcs incident to the repository. If a task interpretation is to read the repository, then it will invoke the access procedure for the repository with a given arc name; the call will be treated as a read or write, depending on the orientation of the arc.

A repository is implemented as a set of remote procedures. Each repository has default read and write procedures that are invoked by the task interpretation (using the access function). The user must define repository interpretations in exactly the same manner as he would specify a task interpretation.

Access procedures provide a mechanism for executing arbitrary repository interpretations, but they do not explicitly handle data types. Since Olympus uses Sun's RPC for invocation of these procedures, it also employs the related external data representation (XDR) for defining the types of the information exchanged between the Task and Repository Interpreters.

Appendix A also describes messages related to instrumentation. Olympus is intended to allow the user to measure the performance of the model or language. Thus, the client can request that token traffic be measured in terms of flow rates, dwell times, etc. Ordinarily, the meaning of any particular measure is determined by the user, not by the system.

4. OLYMPUS CLIENTS

Each Olympus client must implement the Console and the Model Editor. In this paper we briefly describe the SunView client and the NeWS Client designs.

Both clients, as well as the Symbolics Lisp client [23], are window-based interfaces that employ pointing devices to implement visual BPGs. Icons are drawn on the screen, under the control of the Model Editor, then stored in the server using the messages described above. When a model is loaded into the server, then the server sends messages to the client (display portion of the client, shared by the Console and the Model Editor) so that it can present the model in its chosen method, see Appendix B.

Node and arc properties are specified through the use of property sheets. Thus, a task can be labeled and an interpretation can be provided by selecting the task, popping up a property sheet form, and filling in the property sheet (cf. the Xerox Star interface [24]). Each operation on a property sheet will result in messages being sent to the Model Storage portion of the server, (see Appendix A).

The SunView Client Implementation

The SunView implementation uses standard facilities provided in the SunView library to implement the Console and Model Editor [28]. When the frontend process is created, it opens a window on the desktop. The window provides scroll bars and a pallet of icons for editing a model. Model editing operations are accomplished by using the pallet and one of the three mouse buttons. The Console operations are all invoked via popup menus from the other two mouse buttons.

Console operations for the Model Editor result in procedure calls, and operations for the remaining modules result in messages being sent to the Olympus server (backend process).

This Model Editor implementation is limited in its abilities. While it is possible to create a rendering of a BPG, and to supply interpretations and other details for each node, the Model Editor does not support editing operations such as moving an object around on the screen. Instead, the original object must be destroyed, then recreated at a new screen location. However, it is possible to completely specify a model in the Model Storage.

The structure of the SunView editor was a direct outgrowth of previous work done on the Alto workstation [17]. In the Alto window environment the mouse was under program control, i.e., it was read like any other device. This structure was also natural for the very first line-oriented interface, since input events all came from the keyboard.

However, SunView (like many modern window systems) is an event-oriented environment. The user defines a number of event routines, then registers them with the SunView window manager. All execution takes place under the control of the window manager; thus, execution is driven by the occurrence of events detected by the window manager, not by the user's program.

As a result, the structure of the SunView client code is rather convoluted. This aggravates the problem of reusing this code to build a client for a different model as required in our research efforts.

Because of the replacement of SunView by NeWS in Sun environments, the emergence of X Version 11, and because of the limitations in our design, the SunView client was not a good base from which to build other clients. The only mechanism for reusing the software was to modify the code, since it does not have a robust set of library routines nor a type hierarchy.

The NeWS Client Implementation

The requirements for the NeWS implementation included one that would make it easier to reuse the code than was the case for the SunView client. We had decided that we would build the next client on top of a set of libraries, at a minimum, and as an object-oriented program if that were feasible (within our other constraints).

The NeWS architecture divides any implementation into two parts: A NeWS server and a NeWS client, i.e., the Olympus client is implemented as another client and a server, see Figure 3. The NeWS client is responsible for implementing the services of the Console and the Model Editor. The NeWS server is responsible for managing the display, based on a specific protocol between the client and the server.

The NeWS client Console implements the same function as the SunView Console, i.e., it accepts commands from the user (via the NeWS server), then routes them to the backend process (Olympus server dispatcher).

The Model Editor is a new design, based on type hierarchies of visual network models. The Model Editor is the main client of the NeWS server; it also implements the syntax of BPGs, even though the server implements the visual aspects of BPGs, e.g., a task is drawn as a circle. The type hierarchy treats model atoms as objects; thus an object may be a node or an arc. Properties that distinguish arcs and nodes are defined in subclasses. Within the node subclass are additional refinements to distinguish between the way the editor treats a task node and a repository node (e.g, arcs between tasks and repositories are data flow arcs and arcs between tasks are control flow arcs; the appearance of the two arc types is different on the screen).

The use of object types allows the editor to be built without being dependent upon specific presentation properties of the nodes. As a result, the same editing functions can be used to construct an editor for BPGs, DCPGs, PAMs, and other models.

In Figure 3, the NeWS server is shown as a collection of *lightweight process*. Each lightweight process can be dedicated to editing tasks without incurring full Unix process context switching costs whenever work is passed among them. There is a surrogate lightweight processes in the server to direct the other lightweight processes on behalf of the (Unix heavyweight) NeWs client editor. The surrogate controls a lightweight process to handle input events, another for output events, and other for specific editing tasks (such as "track the mouse").

More details of the NeWS client design and implementation will appear in a later paper.

Since the user interface process(es) are independent of the Olympus server, then editing operations can take place in parallel with the server's operation. This allows the user to edit a model while the server is in the process of animating or simulating its activity. Because the server handles the Model Storage, and because an editor is the only

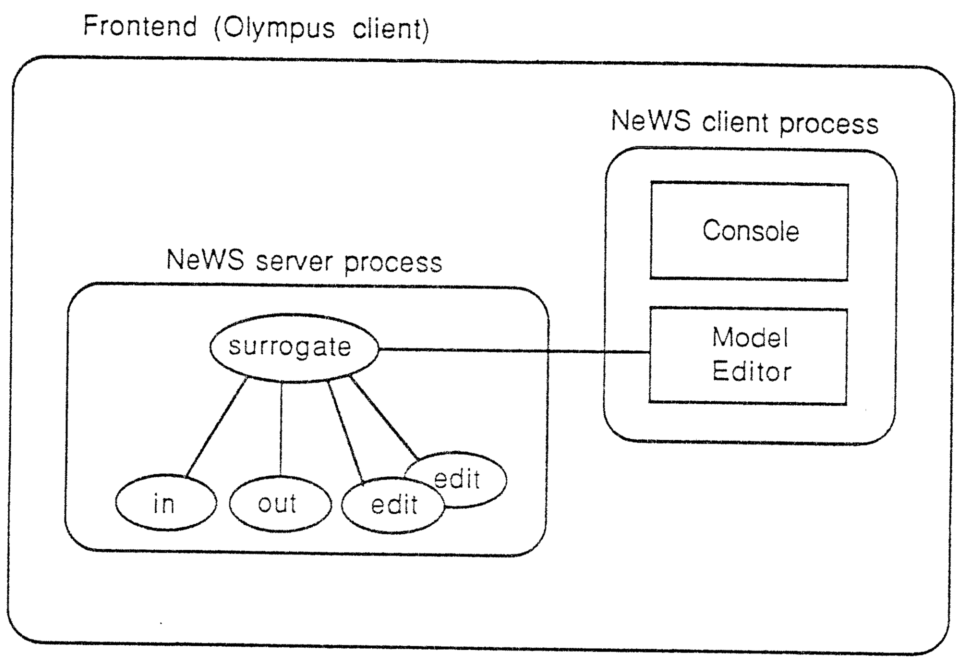

Figure 3: Olympus NeWS Client Design

process that will rewrite the storage, the only race conditions that can arise in the Model Storage would occur when multiple editors are being simultaneously used with a single server.

5. CONCLUSION

We have described the Olympus modeling system, designed on a distributed client-server architecture in which the implementations of the client and the server also employ client-server and remote procedure call models of computation, see Figure 4 for a summary of the architecture.

The Olympus architecture supports very general usage; because of the isolation of interpretation in the server, the client need not know any details of the model interpretation. The server will support multiple clients operating on a single model in the server; thus, users can cooperatively construct and analyze a model (or program) using the common server with their individual clients.

Modeling is most useful when the system that supports it is easy to use, and very flexible. The independence of the Console from the server not only allows the user great freedom in applying different loads to the model, it also allows the user to dynamically change the load -- the specification of the load or the specific instance of the load -- while the model is being interpreted. More importantly, Olympus allows the user to "correct" the model during interpretation, instead of requiring that the user halt the model, change it with an editor, recompile it, reinitialize it, and wait for it to get to the loading condition in which it was halted. If alterations of the model should be performed

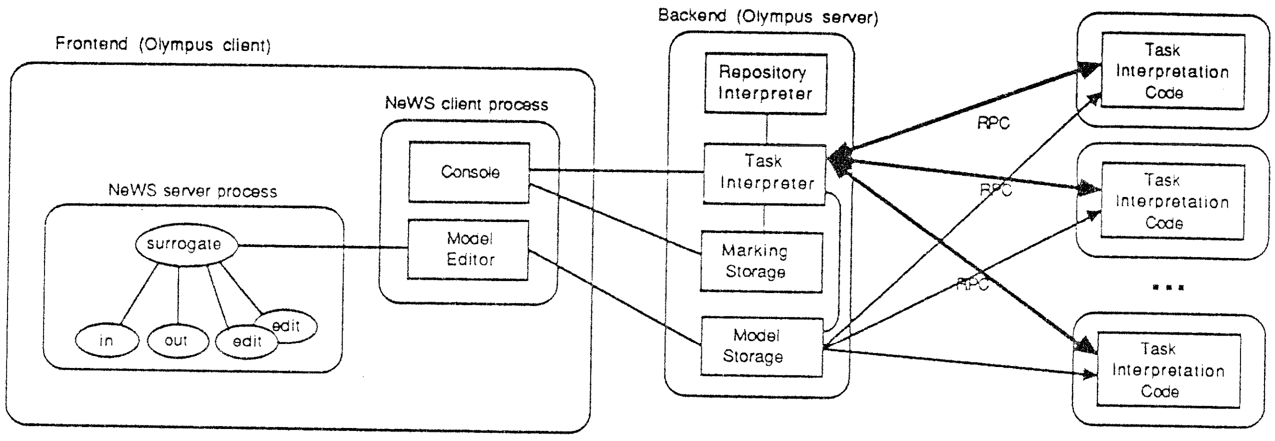

Figure 4: Olympus Architecture Summary

while the interpretation is inactive, then the interpretation can be temporarily interrupted, the model changed, and the interpretation resumed.

The client-server model has allowed us to to focus on the frontend or the backend without tracking versions. The development of the RPC BPG interpretation proceeded in parallel with the NeWS and Symbolics frontends.

An implication is that we have been able to reuse system components for different systems. At this point in the research, the differences in the systems have been limited to providing different user interfaces for BPG interpretation. However, it is clear that we can now implement different frontends to support a number of other models [20].

The client-server architecture has provided us with a means to reuse components in-the-large. We are also experimenting with software reuse in-the-small. The NeWS frontend is expected to be the basis of a family of different frontends to support different models. Preliminary experience with the Symbolics version has suggested a generalization that can be applied to a large number of network-based models in which the customization through type hierarchies is explicit [10]. In the Sun environment, Demeure has also began to reuse the NeWS frontend type hierarchy for implementing her multilevel modeling system [8].

Much of our experience with Olympus has focused on BPG modeling; the next phase of our research (already underway) will also incorporate the study of the Olympus architecture for supporting programming languages. The current implementation already supports distributed programming by virtue of the RPC implementation of task interpretation. Future papers will report our experience with this type of programming language and support system.

6. ACKNOWLEDGEMENTS

Many of the ideas for separating the editor and interface from the fundamental parts of the simulation arose in discussions with Lee Osterweil, Gerhard Fischer, and Roger King. In addition, several people have worked on the Olympus system implementation: Adam Beguelin contributed considerable effort to the SunView versions of the BPG editor. Bruce Sanders has designed and implemented the NeWS version of the frontend; David Redmiles designed and implemented the Lisp Symbolics version. John Hauser performed all of the detailed design and implementation of the Olympus server. Steve Elliott thought of the use of RPC for task interpretation, then subsequently designed and implemented it; he also maintains the current version of the Olympus server. Isabelle Demeure worked on the SunView single-process editor, and is adapting Olympus to support DCPGs and PAMs.

This research has been supported by NSF cooperative agreement DCR-8420944, NSF Grant No. CCR-8802283, and a grant from U S West Advanced Technologies. Some of the equipment used in the research was donated by Sun Microsystems.

7. REFERENCES

1. A. Beguelin, "(No title -- in preparation)", University of Colorado, Department of Computer Science, Thesis proposal, in preparation.
2. M. Bhattacharyya, D. Cohrs and B. Miller, "A Visual Process Connector for Unix", *IEEE Software 5*, 4 (July 1988), 43-50.
3. J. C. Browne, D. Neuse, J. Dutton and K. Yu, "Graphical Programming for Simulation of Computer Systems", *Proceedings of the 18th Annual Simulation Symposium*, 1985.
4. J. C. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach", *14th International Conference on Parallel Processing*, August 1985, 624-631.
5. J. C. Browne, "A Unified Approach to Parallel Programming", *1987 Summer Workshop in Parallel Computation*, June 23, 1987.
6. *CACI Simscript II.5 marketing information*, CACI Product Company, La Jolla, California, 1988.
7. S. Chang, T. Ichikawa and P. A. Ligomenides, *Visual Languages*, Plenum Press, 1986.
8. I. M. Demeure, "A Model (DCPG/PAM) and a Graphic Tool (VISA) for Distributed Computations", University of Colorado, Department of Computer Science, Thesis proposal, May 1988.
9. M. L. Graf, "Building a Visual Designer's Environment", MCC Technical Report No. STP-318-87, October, 1987.
10. S. Henninger, A. Ignatowski, C. Rathke and D. Redmiles, "A Knowledge-based Design Environment for Graphical Network Editors", University of Colorado, Department of Computer Science, submitted for publication.
11. S. Iacobovici and C. Ng, "VLSI and System Performance Modeling", *IEEE Micro*, August 1987, 59-72.
12. *PAWS/GPSM marketing brochures*, Information Research Associates, Austin, TX, 1988.
13. B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer 18*, 8 (August 1985), 87-94.
14. J. Misra, "Distributed-Discrete Event Simulation", *ACM Computing Surveys 18*, 1 (March 1986), 39-65.
15. M. Moser, "GADD -- A Tool for Graphical Animated Design and Debuggin", *ICC '87 Conference Record*, 1987, 38.2.1-38.2.5.
16. K. M. Nichols and J. T. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer 21*, 5 (May 1988), 39-48.
17. G. J. Nutt and P. A. Ricci, "Quinault: An Office Environment Simulator", *IEEE Computer 14*, 5 (May 1981), 41-57.

18. G. J. Nutt, ''A Rapid Simulation Modeling System'', Technical Report submitted for publication, Department of Computer Science - University of Colorado, Boulder, September 1987.

19. G. J. Nutt, ''Visual Programming Methodology for Parallel Computations'', *MCC-University Research Symposium Proceedings*, Austin, Texas, July 1987.

20. G. J. Nutt, ''A Formal Model for Interactive Simulation Systems'', Technical Report CU-CS-410-87, Department of Computer Science - University of Colorado, Boulder, September 1988.

21. G. Raeder, ''A Survey of Current Graphical Programming Techniques'', *IEEE Computer 18*, 8 (August 1985), 11-25.

22. R. R. Razouk, M. Vernon and G. Estrin, ''Evaluation Methods in SARA -- The Graph Model Simulator'', *Proceedings of the Conference on Simulation, Measurement and Modeling of Computer Systems*, August 1979, 189-206.

23. D. Redmiles, ''Vesuvious Editor for Olympus (working title)'', University of Colorado Department of Computer Science technical report (in preparation), 1988.

24. D. Smith, E. Harslem, C. Irby and R. Kimball, ''The Star User Interface: An Overview'', *Proceedings of the AFIPS National Computer Conference 51* (1982), 515-528.

25. L. Snyder, ''Parallel Programming and the Poker Programming Environment'', *IEEE Computer 17*, 7 (July 1984), 27-36.

26. K. L. Stanwood, L. N. Waller and G. C. Marr, ''System Iconic Modeling Facility'', *Proceedings of the 1986 Winter Simulation Conference*, December 1986, 531-536.

27. ''Networking on the Sun Workstation'', Document Number 800-1345-10, Sun Microsystems, Inc., September 1986.

28. ''SunView Programmer's Guide'', Document Number 800-1324-03, Sun Microsystems, Inc., February 1986.

29. *NeWS: A Definitive Approach to Window Systems*, Sun Microsystems, Inc., 1987.

Appendix A: Messages Recognized by the Server

Model Storage Messages

add_arc.
: Add a new non-jointed arc from node <tailNodeID> to node <headNodeID>. Nodes <tailNodeID> and <headNodeID> should already exist.

add_jointed_arc.
: <headNodeID>, with the joint at (<x>, <y>). Nodes <tailNodeID> and <headNodeID> should already exist.

add_node.
: Add a new node of type <nodeType> at the location (<x>, <y>).

arc_joint_at.
: If arc <arcID> is not jointed, make it jointed with the joint at (<x>, <y>). Otherwise, change the location of the arc's joint to (<x>, <y>). Arc <arcID> should already exist.

arc_label.
: Change the label of arc <arcID> to <label>. Arc <arcID> should already exist.

arc_probability.
: Change the probability of choosing arc <arcID> to <probability>. Arc <arcID> should already exist.

delete_arc.
: Delete arc <arcID>. Arc <arcID> should already exist.

delete_marking.
: Delete all tokens.

delete_node.
: Delete node <nodeID>. Node <nodeID> should already exist.

load_model.
: Delete the entire current model; then load the model previously saved under the name <name>.

node_at.
: Node <nodeID> should already exist.

node_description.
: Change the description of node <nodeID> to <description>. Node <nodeID> should already exist.

node_host.
: Change the name of the host on which to execute the interpretation for node <nodeID> to <hostname>. Node <nodeID> should already exist.

node_label.
: Change the label of node <nodeID> to <label>. Node <nodeID> should already exist.

node_read_interpretation_procedure.
: Change the name of the read procedure for repository node <nodeID> to <procedure name>. Node <nodeID> should already exist.

node_read_XDR_procedure.
: Change the name of the XDR procedure used for decoding input to the node procedure for node <nodeID> to <procedure name>. Node <nodeID> should already exist.

node_time_distribution.
: Change the time distribution for node <nodeID> to be of type <distributionType> with parameters <param1>

and <param2>. If one or both of the parameters are not needed, they will be ignored. Node <nodeID> should already exist.

node_write_interpretation_procedure.
Change the name of the interpretation procedure for task node <nodeID>, or the write procedure for repository node <nodeID> to <procedure name>. Node <nodeID> should already exist.

node_write_XDR_procedure.
Change the name of the XDR procedure used for encoding output from the node procedure for node <nodeID> to <procedure name>. Node <nodeID> should already exist.

redraw.
End all current changes to the model. (Used to inform clients when it safe to do a complete refresh of the screen.)

save_model.
Save the entire current model under the name <name>.

Marking Storage Messages

add_token_to_arc.
Add one token to arc <arcID>. Arc <arcID> should already exist.

add_token_to_node.
Add one token to node <nodeID>. Node <nodeID> should already exist.

delete_all_tokens_from_arc.
Delete all tokens from node <nodeID>. Node <nodeID> should already exist.

delete_token_from_arc.
Delete the next token from arc <arcID>. Arc <arcID> should already exist.

delete_token_from_node.
Delete the next token from node <nodeID>. Node <nodeID> should already exist.

Task Interpreter Messages

bye.
Disconnect this client. (This message should be sent to the server and a 'bye' received back from the server before the client exits.)

clear_statistics.
Clear all statistics.

delete_instruments.
Delete all instruments.

instrument_arc.
If <instrumentFlag> is nonzero, add an instrument to arc <arcID>. Else if <instrumentFlag> is zero, remove the instrument from arc <arcID>. Arc <arcID> should already exist.

instrument_node.
If <instrumentFlag> is nonzero, add an instrument to node <nodeID>. Else if <instrumentFlag> is zero, remove the instrument from node <nodeID>. Node <nodeID> should already exist.

set_sleeps.
: If <sleepFlag> is nonzero, turn on sleeps. Else if <sleepFlag> is zero, turn off sleeps. (The default is 'on'.) Used to disable real time operation.

set_updates.
: If <updatesFlag> is nonzero, set the update rate to be 'constant'. Else if <updatesFlag> is zero, set the update rate to be 'occasional'. (The default rate is 'occasional'.) This command is currently not used. Used to determine the frequency with which the client scans the Marking Storage for new state.

start_model.
: If the model is not already running, start the model running. If sleeps are on, run the model in real time; otherwise, run the model as fast as possible.

step_model.
: If the model is not being interpreted, step the model by one event.

stop_model.
: If the model being interpreted, stop it.

sync.
: Respond to sync from the client.

write_statistics.
: Save all gathered statistics under the file name <name>.

Appendix B: Messages Recognized by the Client

Model Editor Messages

add_arc.

Add the non-jointed arc <arcID> from node <tailNodeID> to node <headNodeID>. If arc <arcID> already exists, update the tail and head node information, and make the arc non-jointed. If either of the nodes <tailNodeID> or <headNodeID> don't exist, ignore this command. Also update the numbering of the arc at the head and tail nodes. This is used to identify arcs to node interpretations.

add_jointed_arc.

Add the jointed arc <arcID> from node <tailNodeID> to node <headNodeID>, with the joint at (<x>, <y>). If arc <arcID> already exists, update the tail and head node information and the joint location information. If either of the nodes <tailNodeID> or <headNodeID> don't exist, ignore this command.

add_node.

Add the node <nodeID> with type <nodeType> at the location (<x>, <y>). If node <nodeID> already exists, update the location and type information.

arc_joint_at.

If arc <arcID> is not jointed, make it jointed with the joint at (<x>, <y>). Otherwise, change the location of the arc's joint to (<x>, <y>). If arc <arcID> doesn't already exist, ignore this command.

arc_label.

Change the label of arc <arcID> to <label>. If arc <arcID> doesn't already exist, ignore this command.

arc_probability.

Change the probability of choosing arc <arcID> to <probability>. If arc <arcID> doesn't already exist, ignore this command.

arc_statistics.

Change the statistics for arc <arcID> to be:

Number of tokens fired on this arc:	<count>
Minimum occupation time:	<occupMin>
Mean occupation time:	<occupMean>
Maximum occupation time:	<occupMax>
Minimum inter-arrival time:	<arrivMin>
Mean inter-arrival time:	<arrivMean>
Maximum inter-arrival time:	<arrivMax>

If arc <arcID> doesn't already exist, ignore this command.

delete_arc.

Delete arc <arcID>. If arc <arcID> doesn't already exist, ignore this command.

delete_instruments.

Remove all instruments from nodes and arcs.

delete_marking.

Change the number of tokens on every node and arc to zero.

delete_model.

Delete all nodes and arcs.

delete_node.

Delete node <nodeID>. If node <nodeID> doesn't already exist, ignore this command.

node_at.

Change the location of node <nodeID> to (<x>, <y>). If node <nodeID> doesn't already exist, ignore this command.

node_description.

Change the description of node <nodeID> to <description>. If node <nodeID> doesn't already exist, ignore this command.

node_host.

Change the name of the host on which to execute the interpretation for node <nodeID> to <hostname>. If node <nodeID> doesn't exist, ignore the command.

node_label.

Change the label of node <nodeID> to <label>. If node <nodeID> doesn't already exist, ignore this command.

node_read_interpretation_procedure.

Change the name of the read procedure for repository node <nodeID> to <procedure name>. If node <nodeID> doesn't exist, ignore the command.

node_read_XDR_procedure.

Change the name of the XDR procedure used for decoding input to the node procedure for node <nodeID> to <procedure name>. If node <nodeID> doesn't exist, ignore the command.

node_statistics.

Change the statistics for activity node <nodeID> to be:

Number of tokens fired on this node:	<count>
Minimum occupation time:	<occupMin>
Mean occupation time:	<occupMean>
Maximum occupation time:	<occupMax>
Minimum inter-arrival time:	<arrivMin>
Mean inter-arrival time:	<arrivMean>
Maximum inter-arrival time:	<arrivMax>

If node <nodeID> doesn't already exist, ignore this command. This command is currently not used.

node_time_distribution.

Change the time distribution for node <nodeID> to be of type <distributionType> with parameters <param1> and <param2>. Ignore one or both of the parameters if they are not needed. If node <nodeID> doesn't already exist, ignore this command.

node_write_interpretation_procedure.

Change the name of the interpretation procedure for task node <nodeID>, or the write procedure for repository node <nodeID> to <procedure name>. If node <nodeID> doesn't exist, ignore the command.

node_write_XDR_procedure.

Change the name of the XDR procedure used for encoding output from the node procedure for node <nodeID> to <procedure name>. If node <nodeID> doesn't exist, ignore the command.

Console Messages

bye.

Exit client.

instrument_arc.

If <instrumentFlag> is nonzero, add an instrument to arc <arcID> Else if <instrumentFlag> is zero, remove the instrument from arc <arcID>. If arc <arcID> doesn't already exist, ignore this command.

instrument_node.

If <instrumentFlag> is nonzero, add an instrument to node <nodeID> Else if <instrumentFlag> is zero, remove the instrument from node <nodeID>. If node <nodeID> doesn't already exist, ignore this command.

no_more_tokens.

Report that there are no more tokens to fire.

number_of_tokens_on_arc.

Change the number of tokens on arc <arcID> to <count>. If arc <arcID> doesn't already exist, ignore this command.

number_of_tokens_on_node.

Change the number of tokens on node <nodeID> to <count>. If node <nodeID> doesn't already exist, ignore this command.

maximums.

Verify that <maxNodeID> nodes and <maxArcID> arcs can be handled by this client.

redraw.

Refresh the screen if necessary.

repository_statistics.

Change the statistics for repository node <nodeID> to be:

Number of reads:	<readCount>
Number of writes:	<writeCount>

If node <nodeID> doesn't already exist, ignore this command.

sync.

Respond with a sync.