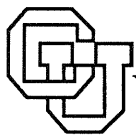
Collected Papers on Olympus

Gary J. Nutt

CU-CS-518-91 February 1991



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Collected Papers on Olympus

Gary J. Nutt†

CU-CS-518-91 February, 1991

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

†This work was supported by NSF Grant CCR-8802283. The author's electronic mail address is nutt@cs.colorado.edu.

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE NATIONAL SCIENCE FOUNDATION

ABSTRACT

Collected Papers on Olympus

Olympus is an architecture for interactive, visual, distributed modeling and programming systems. Since it began as a member of a more general class of distributed systems named the *Strawberry* systems (see Nutt, "An Experimental Distributed Modeling System," *ACM Transactions on Office Information Systems*, 1, 2 (April, 1983), pages 117-142), it was named after a variety of strawberry:

> [Description of the "Olympus strawberry"] *Average-sized light red fruit in mid-season. No runners; vigorous plants bear on branching crowns.*
> From *Sunset New Western Garden Book*, Sunset Books, Lane Publishing, 1979, page 475.

The Olympus study began in 1987 and is still in progress at the time of this writing. The original goals of the project were to consider systems to support the bilogic precedence graph (BPG) model, much as the predecessor *Quinault* system (described in Nutt and Ricci, "Quinault: An Office Environment Simulator", IEEE Computer, 14, 5 (May, 1981), pages 41-57) had been designed to support information control networks.

As the project progressed, the research first evolved to address models for studying process partions (see the ParaDiGM model and VISA system in CU-CS-488-90), and then to using the basic system to support visual, parallel programming (see the Phred model and system in CU-CS-512-91). We also began to understand that we were developing a useful architecture for interactive systems; the study had evolved from one focusing on specific models to the design of systems to support many different models.

This collection of papers includes three papers:

(1) G. J. Nutt, "A Simulation System Architecture for Graph Models," *Advances in Petri Nets*, Springer Verlag, to appear, 1990.

(2) G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter, B. Sanders, "Olympus: An Interactive Simulation System," *1989 Winter Simulation Conference Proceedings*, December 1989, pp. 601-611.

(3) G. J. Nutt, "Distributed Simulation Design Alternatives," *Proceedings of the SCS Conference on Distributed Simulation*, January 1990, pp. 51-55.

The first paper describes the Olympus architecture (an earlier version of the same paper was presented at the *Tenth International Conference on Application and Theory of Petri Nets*, in Bonn, West Germany, June 1989), the second paper describes how it was used to support the BPG model, and the third paper speculates about how the architecture could be used to support distributed simulation systems. Each of the papers has been published elsewhere, so this collection is just a mechanism for grouping the reprints.

Olympus research continues into areas related to automatic system generation, other applied languages, and adaptive load balancing. Please contact the author for current results.

A SIMULATION SYSTEM ARCHITECTURE FOR GRAPH MODELS*

Gary J. Nutt†
University of Colorado (USA)

ABSTRACT

The paper describes a distributed modeling system architecture, designed to support various graph models of computation, including predicate/transition nets and colored Petri nets. To demonstrate the utility of the architecture, we describe an implementation for a specific graph model (one that is related to, but distinct from, Petri nets). The architecture provides for interactive editing and interpretation facilities employing a graphic point-and-select user interface. A user can define a model, then mark it with tokens and observe the operation of the net through real time animation. The model and the marking can be rapidly altered, even as an interpretation is in progress. The system also supports simultaneous use among multiple users, including concurrent editing and interpretation. Thus the system supports cooperative model design and interpretation by a group of designers at different nodes in a network of workstations.

1. INTRODUCTION

The paper describes a distributed modeling system architecture, designed to support various formal graph models of computation. To demonstrate the utility of the architecture, we describe an implementation for a specific graph model (one that is related to, but distinct from, Petri nets -- see [9, 20, 21, 26]).

Formal graph models are an invaluable tool for analyzing the behavior of a concurrent system prior to its implementation. A sound model can be used to predict performance, represent concurrency and synchronization, and to impart fundamental knowledge about the relative merits of alternative designs for the system.

A Petri net can represent control flow, concurrency, synchronization, and nondeterminacy in systems. Petri nets represent considerable more detail than static models such as precedence graphs. Execution of the Petri net is represented by the marking sequence of the Petri net, thus there are dynamic aspects represented by the static net and an initial marking.

Petri net theory is devoted to the study of properties of Petri net variants and their behavior under various markings. Petri net application focuses on the use of the model family to represent prospective concurrent systems and to derive an understanding of the behavior of the system based on the properties of the model.

There is a spectrum of applications, ranging from model instances that can be analyzed for fundamental behavior such as safety and liveness; to timed Petri nets that predict system performance in terms of resource utilization, throughput, and turnaround time; to predicate/transition and colored Petri net models that can be used to simulate the system activity.

The *Olympus Modeling System Architecture* described in this paper can be used to build tools that support the application of Petri nets and other graph models to systems analysis and simulation. In particular, the architecture can be used to define systems that create and analyze the static version of a model as well as to exercise the dynamic execution of the model under various initial conditions.

The goal of the paper is to describe the Olympus Architecture for modeling systems, and to illustrate how the architecture can be used to support dynamic models of computation.

1.1. Background

There are a number of machine-supported, interactive modeling systems in existence, e.g., see PAWS/GPSM [3, 14], the Performance Analysis Workstation [17], PARET [19], Quinault [22], Raddle/Verdi [13], and GreatSPN [1].

Each of these systems has a specific, underlying graph model. The companion system is built to implement the syntax of the graph model and to analyze or simulate the semantics determined by the formal behavioral specification of the model.

---

* This research has been supported by NSF Grant No. CCR-8802283, NSF cooperative agreement DCR-8420944, and a grant from U S West Advanced Technologies. This paper is a substantial revision of a paper entitled "A Flexible, Distributed Simulation System" presented at the Tenth International Conference on Application and Theory of Petri Nets.

† Department of Computer Science, Campus Box 430, University of Colorado, Boulder, CO 80309-0430, (303) 492-7581, nutt@boulder.colorado.edu

These systems are implemented in bitmap graphics environments in which a designer constructs models in the language supported by that system. Typically, the user employs a graphic point-and-select editor to construct a visual model; he may then provide annotation for the model through a wide variety of annotation mechanisms, ranging from popup windows through preparing separate files in a distinct editing session. Systems such as PAWS/GPSM will also predict performance of the model using queueing network analysis techniques (a GPSM model is a queueing network).

The dynamics of model operation can be observed by supplying data to the model, then causing the model to execute in the supplied environment. Most of the systems provide an animation of the simulation execution. The modeling system may allow the user to observe the operation of the model through the changing state of the model (in the case of marked models such as Petri nets), or through the display of distributions, gauges, etc.

The operation of a model is ordinarily controlled through the use of checkpoints. That is, the model can have a flag set at some particular point in the model. When the interpretation encounters the flag, then it suspends execution and interacts with the user, allowing him to interrogate the state of the execution, change internal conditions in the model, etc.

The power of these modeling systems is in their ability to allow a user to quickly and easily build a model and then to view the dynamics of the operation of the model. The designer is able to quickly converge on "correct" models through this interactive, experimental testbed approach.

Because of the importance of these "what if" type experiments, the modeling system must be very easy to use for constructing and editing the model, it must allow the designer to easily alter the loading conditions of an individual experiment, and it must provide intuitive feedback to the designer by providing appropriate measures of the activity in a language that is familiar to the designer.

Most of today's modeling systems can be criticized on the following grounds: They often leave the user in a particular *mode* during a session, e.g., the user cannot edit if he is in the process of interpreting. It is awkward to change loading conditions while the model is in execution, since the interpretation can only be interrupted through checkpoints. Once the model is being interpreted, any editing change to the model requires that the model be halted and the editor be started (mode change) in order to accomplish the change. The modeling language is specific to the system rather than to the designer. Only a single user can interact with any particular model at any given time.

In the Olympus Architecture, we attempt to address these issues. There are no modes in a modeling session; the user is allowed to edit the model or the load characterization at any time -- including during interpretation. The interpreter can be suspended or halted at any time, then restarted or resumed later. Multiple users can be involved in an individual modeling session.

The logical interpretation and the presentation of the model and its execution are separated into model syntax and semantics. The model syntax -- the appearance of the model at a workstation screen -- is independent of the logical specification of the model -- the model's semantics.

We have built an instance of the Olympus Architecture, bound to a particular graph model. Next we provide an overview of the example system.

1.2. An Experiment

The *Olympus BPG Modeling System* (Olympus-BPG) provides machine support for a specific formal graph model, Bilogic Precedence Graphs (BPGs) [26]. It provides a model simulation environment, so that a group of designers can use interactive graphic-based tools to create, maintain, modify, and exercise an interpreted, marked network model.

Olympus-BPG is an instance of the Olympus Architecture, and is intended to address the general problems described above with architectural solutions. In addition, Olympus-BPG addresses various other modeling problems, through specific solutions as opposed to architectural solutions. For example, Olympus-BPG supports hierarchical refinement of elements of BPG models -- a solution that is specific to BPGs, but which is applicable to other models that support hierarchy.

Further, the BPG semantics are implemented in Olympus-BPG by a distinct process (optionally on a distinct machine) from the process that implements the BPG syntax. As a result, the BPG user interface contains a BPG graph editor and console, while the logical storage and interpretation of the BPG occur in a different environment interconnected via a well-defined network interface. The presentation syntax of the model is distinct from the semantics of the model's execution. The Olympus-BPG interpreter can interact with a BPG viewer/console or any of a number of types of viewer/consoles.

Instances of Olympus are designed as *distributed* modeling systems, one aspect of which we have just described. These systems support groups of users interacting with a single model, thus allowing the system to be a common, interactive design environment. The concurrency aspects mentioned above (i.e., simultaneous editing and interpretation) provides a very general modeling facility for cooperative model construction.

2. THE OLYMPUS SYSTEM ARCHITECTURE

2.1. Characteristics of Models Supported by the Architecture

The Olympus Architecture is designed to create, store, and interpret a wide class of graph models. The class is large enough so that the architecture is applicable to many systems, yet specific enough so that there is some advantage to employing a common architecture for various modeling systems.

Marked graph models supported by the Olympus architecture are of the form

$$\Gamma = ((\Xi, \Phi, M), \Delta)$$

where Ξ is a control flow graph with interpretation Φ and marking M, and Δ is a data flow graph. More specifically,

$$\Xi = (\Pi, E_C)$$

$$\Pi = \Pi_1 \cup \Pi_2 \cup \ldots \cup \Pi_K$$

where

$\Pi_i = \{p_{i,1}, p_{i,2}, \ldots\}$ is a finite set of tasks of type i (for $1 \le i \le K$)

$E_C \subseteq \Pi \times \Pi$ is a finite set of edges interconnecting tasks

$\Phi = \{f_i \mid 1 \le i \le |\Pi|\}$ is a set of interpretations for each task,

$f_i: \Pi \rightarrow$ *interpretation language* $\cup \Gamma$'

$M: \Pi \cup E_C \rightarrow \{\text{null}, \tau_i\}$ is the marking of the graph.

τ_i is a name for a token of unspecified type

$$\Delta = ((R \cup \Pi), E_D)$$

where

$R = \{r_1, r_2, \ldots\}$ is a finite set of data repositories

$E_D \subseteq (\Pi \times R) \cup (R \times \Pi)$ is a finite set of edges interconnecting tasks and repositories

Informally, the graph has one component to represent the flow of control among a set of task nodes, Ξ. Nodes in the graph, each representing a task, can be any of K different types, e.g., nodes with disjunctive input and output logic, or perhaps nodes with disjunctive input logic and conjunctive output logic. Every task node can have an interpretation. The interpretation is specified in an arbitrary procedural language, or it can be defined by another marked graph. The data flow graph adds nodes to represent data storage; write operations can be performed by a task if and only if there is an edge (in E_D) from the task to the repository, and read operations are represented by an edge from the repository to the task node. The marking of the control flow graph represent a distribution of tokens on task nodes and control flow edges, i.e., this model allows tokens to reside on a node or on an edge.

The graphs supported by the architecture are very general, too general for extensive analysis. However, the architecture is intended to provide a framework in which specific systems for specific graph models can be easily constructed. In particular, the architecture can be used to support Petri net models.

Murata [18] defines a Petri net as a 5-tuple, (P, T, F, W, M_0) where

$P = \{p_1, p_2, \ldots p_m\}$ is a finite set places

$T = \{t_1, t_2, \ldots t_n\}$ is a finite set of transitions

$F \subseteq (P \times T) \cup (T \times P)$ is a finite set of arcs (flow relation)

$W: F \rightarrow \{1, 2, 3, \ldots\}$ is a weight function

$M_0: T \rightarrow \{0, 1, 2, 3, \ldots\}$ is the initial marking

$P \cap T = \varnothing$ and $P \cup T \neq \varnothing$

There are at least two ways that the marked graph model could be used to represent Petri nets: By mapping places into nodes, or by mapping them into multiarcs. It is necessary to map the places to nodes if the Petri net is to represent predicate/transition nets [12] or colored Petri nets [15], since they rely on interpretations for places. Our example maps places to nodes with disjunctive (OR) input and output logic, and transitions to nodes with conjunctive (AND) logic.

Considering only the case that $W:F \rightarrow 1$, a marked graph can represent a Petri nets as follows. Let:

$$\Gamma = ((\Xi, \Phi, M), \varnothing)$$

$$\Xi = (\Pi, E_C)$$

$$\Pi = \Pi_1 \cup \Pi_2$$

where

$\Pi_1 = P = \{p_1, p_2, ...\}$ is a finite set of tasks with disjunctive input and output logic (places)

$\Pi_2 = T = \{t_1, p_2, ...\}$ is a finite set of tasks with conjunctive input and output logic (transitions)

$E_C = F$

$f_1 = f_2 = \varnothing \Rightarrow \Phi = \{\varnothing\}$

$M = M_0$

Type 1 nodes represent places (nodes with OR logic) and type 2 nodes represent transitions (nodes with AND logic). As stated above, the general graph model semantic of firing for type 1 nodes is degenerate, inasmuch as there is no corresponding semantic in Petri nets. To complete the semantics of transition firing (as it reflects on the activity at a place node), it is necessary to make place "firing" be passive, yielding tokens to downstream transition nodes as required by the transition's activity. (This is particularly obvious in situations involving forward conflict.)

It is possible to describe the characterization of Olympus graph models more precisely, and to make the mapping to Petri nets more specific. However, the goal of this paper is to illustrate an architecture for simulation systems that can be used to support Petri nets and other models.

The Olympus Architecture isolates the semantics of the firing rules, thus firing rules can be encoded as required without affecting the operation of other parts of the architecture. The node interpretation mechanism, hierarchy in node interpretations, data flow, token data, editing, storage, etc. are all independent of the details of the firing rules.

2.2. Characteristics of the Modeling System

The graph model provides a language for describing target systems behavior. A simulation system provides a medium for expressing models, and for studying these models by observing their reaction to different conditions. An interactive system (using bitmap workstation technology), creates an environment in which alternatives -- changes in loading conditions, changes in parameters, or changes in the model itself -- are easy to explore.

Olympus systems provide the following specific features to their users:

(1) It provides a *simple* mechanism to interactively create and edit model instances.

(2) Detailed behavior of a model can be expressed as a hierarchical model or a procedural interpretation.

(3) The presentation and the semantics of the model can be represented independently.

(4) The user can 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*).

(5) When an interpretation is interrupted, the user can browse and change the state of the interpretation prior to continuation.

(6) If the system is interpreting a model in scaled real time, then the user can change the time scale while the model is in operation.

(7) The interactive system allows editing and interpretation to proceed in parallel.

(8) It is possible to reuse large parts of a system instance, within the architecture, to build a comparable modeling system for related models of computation.

These are general goals, but they are useful guidelines for constructing the modeling system. We now explain how these goals are addressed in the architecture.

2.3. The Architecture

Any instance of Olympus is a collection of copies of the following modules: *Console*, *Model Storage*, *Marking Storage*, *Task Interpreter*, and *Repository Interpreter*. Optionally, it may also include a *Model Editor* or an arbitrary *Observer*.

Each module can be thought of as an object *class*, where any particular implementation incorporates specific instances of the different classes. For a simple Olympus configuration, there may be only one instance of each class type statically generated when Olympus is started. For example, the Sun implementation of Olympus-BPG implementation allows for multiple instances of the Console, Model Editor, Task Interpreter, and Repository Interpreter classes to operate on distinct workstations.

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 may also act 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 model -- a portion small enough to fit into a window; it also need not operate synchronously with the Marking Storage.

Model Storage

This module provides information about the model that defines the model or program 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."

Model Editor

This module is used to define a graph 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 model types into a specific model by translating the model syntax that it supports prior to storage.

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).

Observer

An Observer is a module that performs system-specific computational tasks. It is characterized by its absorbing information from other parts of the system without providing any particular commands or information back to the system. Thus an Observer is similar to a Console with no input operations. Observers are used to analyze a model, to display performance statistics, and other similar tasks.

Task Interpreter

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 Task Interpreter will notify the Marking Storage. The Task Interpreter will repeat the interpretation cycle on successor tasks as determined by the marking.

Repository Interpreter

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.

An Olympus instance may be a statically bound set of the modules described above; it is convenient to divide the groups into the *frontend* and *backend* groups as indicated in Figure 1.

The backend -- also called the Olympus server -- is a persistent process that is started independently of any particular frontend -- also called an Olympus client. (The boxes with rounded corners in the Figure can be roughly equated to processes, although that view will be refined below.)

The client establishes a network communication 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 model 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.

2.4. The Olympus Server

In our example instance, four of the modules are implemented in the server, making it necessary for the server to provide a demultiplexing function to call different modules as required by the client. The server dispatcher is a

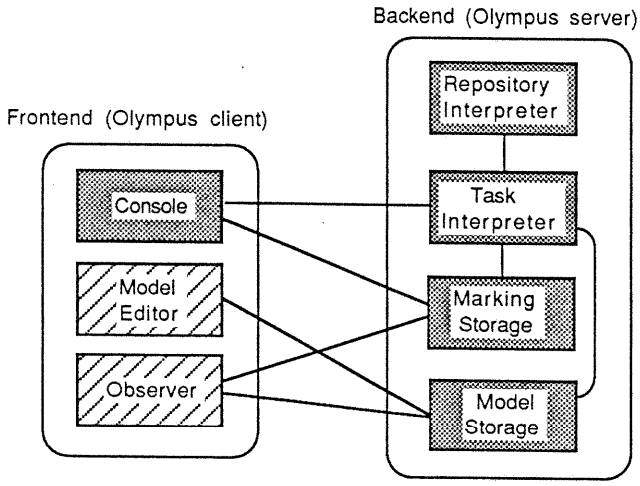


Figure 1: Olympus System Architecture

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 the single client(s)-server socket pair -- it dispatchs the message to the appropriate module.

The set of messages recognized by the server is listed in [24]. 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.

The modules that implement the server need not have been implemented as a single process. For example, the procedure call interface among the modules can be replaced by a remote procedure call interface to achieve functional distribution. 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.

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. 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. There are *add_<atom>* and *delete_<atom>* messages 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.

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 model. 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.

Task and Repository Interpreters

The semantics of the firing and interpretation of a model class are implemented in the Task Interpreter. The graph portion of a model 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.

Node 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 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 corresponding to the *access-repository* procedure.

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. Since such times are often determined by a probability distribution function, Olympus provides a special facility for obtaining a firing time from a probability distribution function without actually executing an arbitrary procedure. The more general interpretation specification -- a procedure declaration -- may be expressed in an arbitrary, precompiled language invoked using Sun's RPC/XDR (Remote Procedure Call/eXternal Data Representation) protocol [30].

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.

After carefully evaluating the cost of using remote procedures versus statically-bound procedures [8], we chose to use RPC 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, node interpretations can be written in C, Fortran, 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.

Depending upon the implementation of the firing policy, Uninterpreted models may be 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

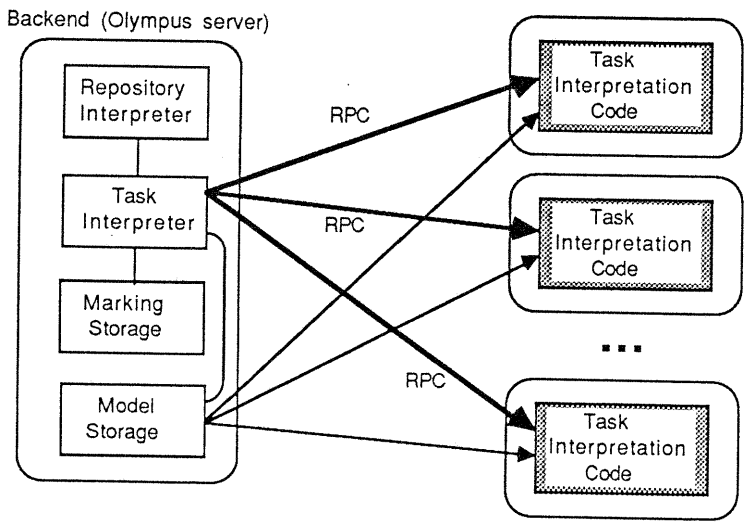


Figure 2: RPC Task Interpretation

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 control flow portion of the system how to place the output token when the interpretation terminates. It is also possible to simply use stochastic functions to randomly choosing an output arc in Olympus.

As described above, the family of graph models may incorporate data flow between tasks and nodes representing data repositories. 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, and thus to each *access-repository* procedure call that can exist in a task interpretation.

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.

The server also collects statistics on the operation of any model. Actual measures are only meaningful to the Olympus user, since some of the tasks and repositories represent resources and queues in the target system, while others are modeling overhead. Olympus provides a facility for instrumenting any arc or node in the graph, enabling data gathering on the modeling atom during interpretation. The resulting data are kept in a file for subsequent analysis. There are currently no additional facilities for analyzing the interpretation of a model.

2.5. Olympus Clients

An Olympus client is used to implement the Console, Model Editor, and various Observers. The server is an engine that stores and interprets graph models, and the client is the mechanism for implementing any function that can use the engine. For example, a client that implements a Console and Editor provides a user interface to the server engine. Because of the separation of the frontend and the backend, and because of the nature of the protocol that is used to allow the frontend and backend to communicate, the client user interface is almost completely independent of the operation of the server. In particular, the client Editor is free to use any visual representation of the graph model stored in the server that fits the need of the user.

One result of the approach is that any Editor that conforms to the client-server intercommunication protocol can be used with the server. Thus, if the server is implemented to support Petri nets, several different clients can be developed to present different visual representations of the Petri net (e.g. representations with bars or rectangles to denote transitions), and to provide different man-machine interaction paradigms. The operation of the server is oblivious to differences in these client frontends.

Because of the limited assumptions that the server makes about the operation of the client, it is possible for the client to perform different functions than the Console and Editor tasks. For example, suppose that one wished to implement a syntax-directed editor for the graph model supported by the server. The conventional approach to constructing such an editor is to either implement the parser in the server, or in the editor itself. If the parser is implemented in the server, then the interactions between the client and the server will become inefficient. If the parser is implemented in the editor, then the editor will become slow (and annoying to use, since the user will tend to be waiting for the parser to complete even though the current graph may be only an interim state.) Beguelin has used the Olympus architecture as a framework in which he implements a *critic* Observer client in addition to an editor client for his server instance [2] (see Figure 3). The critic is an asynchronous client that parses his graph model independent of the actual operation of the client that implements the editor. This results in a system with a critic that is independent of the editor, yet which parses (and otherwise analyzes) the model in the server as the editor is used to create and modify models.

2.6. General Remarks about the Architecture

While we have not described how hierarchical refinement is handled, it can be seen from the graph definitions that the server is required to allow task nodes to be defined in terms of a procedural interpretation or by a refined graph. The server implements *functional hierarchy* by allowing any interpretation to be specified as a graph. When the server is interpreting a model and it encounters such a node interpretation, then the server will use the refined node definition recursively, up to a predefined depth. We will report extensively on our research with hierarchy in a separate paper.

We have used the remote procedure call idea to interconnect the frontend and the backend, and to connect the Task Interpreter manager with the actual Task Interpretations. Sun's RPC is built on top of Berkeley UNIX *sockets* [16], which provide a network-wide means for processes to identify remote processes in terms a simple address -- a *well-known address*. Since a process's IPC port is identified by a simple address, any other process can "connect" to the

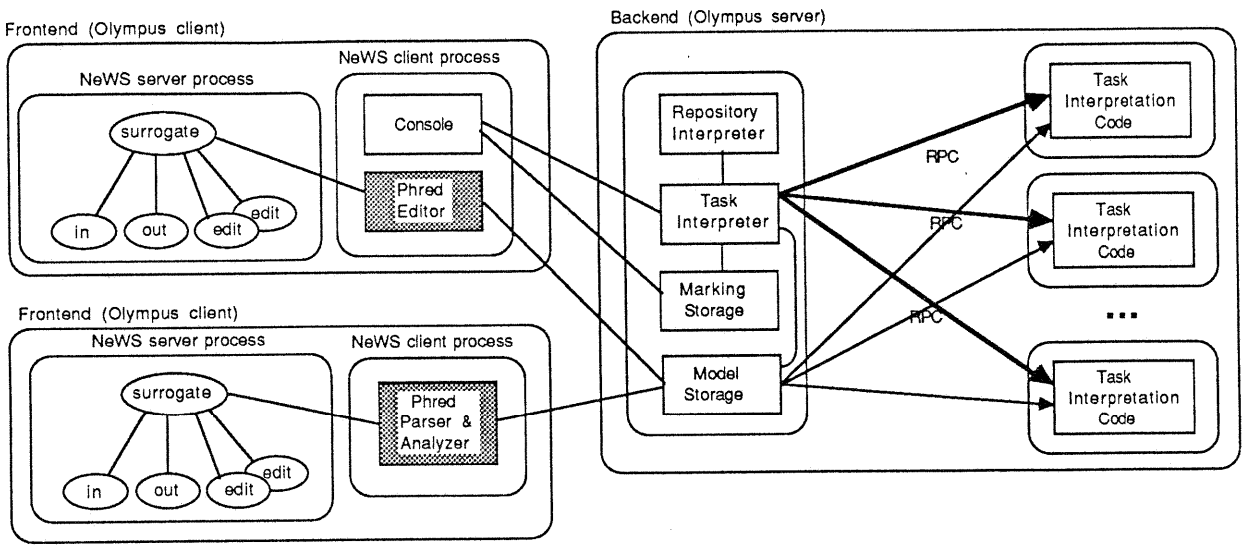


Figure 3: Beguelin's Phred System

process with the well-known socket, i.e., establish communication with the first process using the well-known address. The Olympus server has a well-known socket address which can be connected to by any number of client processes at any given time. Each client can send the server a message over the socket, causing the server to take some action, e.g., update the Model Storage, etc. *as an atomic operation.* When a client wishes to update its picture to show token movement, net changes, etc., then it sends a request for an update to the server; the server responds by placing the update on the socket. Every client is designed to react to updates from the server whether they were requested or not, thus each client has the most current information sent out by the server. As a consequence, any Olympus server supports multiple clients, the number depending only on the number of independent client processes that connect to the server.

Since the client(s) are independent of the Olympus server, 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. Any change in the status of the model, as maintained by the server, results in an update being sent to the client(s). Therefore, if one client edits the model, then all clients see the change at the same time.

Olympus is a very general distributed system, enabling it to provide important interactive simulation facilities such as multiple users with simultaneous updating of screens. The separation of the work into clients and servers along with a carefully designed protocol between them allows the frontend to be asynchronous with the backend, yet allows close interaction of the console with the server.

3. THE OLYMPUS-BPG MODELING SYSTEM

The Olympus architecture has been used to support three different models extensively: BPGs, ParaDiGM [5-7], and phred [2]. In addition, we have experimented with a Petri net frontend to the BPG backend.

In this Section, we describe the instance of Olympus that has been used to support BPGs. (BPGs are a subclass of the general modeling system described above; while the details of BPGs are not especially relevant to the architectural discussion, a brief description of the model is provided in the Appendix.)

The Olympus-BPG server has been implemented in a SunOS environment (on Sun 3 and Sun 4 workstations), and on an Encore Multimax shared memory multiprocessor [25]. Various clients have been built to work with the server, including a line-oriented console, a SunView interactive editor, a Sun NeWS interactive editor, a Sun NeWS performance statistic display, a Sun NeWS interactive editor that employs Petri nets at the user interface, and a Symbolics LISP interactive editor.

Most of the editor clients are window-based interfaces that employ pointing devices to implement visual BPGs (or Petri nets). 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.

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 [29]). Each operation on a property sheet will result in messages being sent to the Model Storage portion of the server.

The SunView BPG Editor Implementation

The SunView implementation uses standard facilities provided in the SunView library to implement the Console and a BPG Editor. When the frontend process is created, it opens a window on the desktop, see Figure 4. 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-BPG server.

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

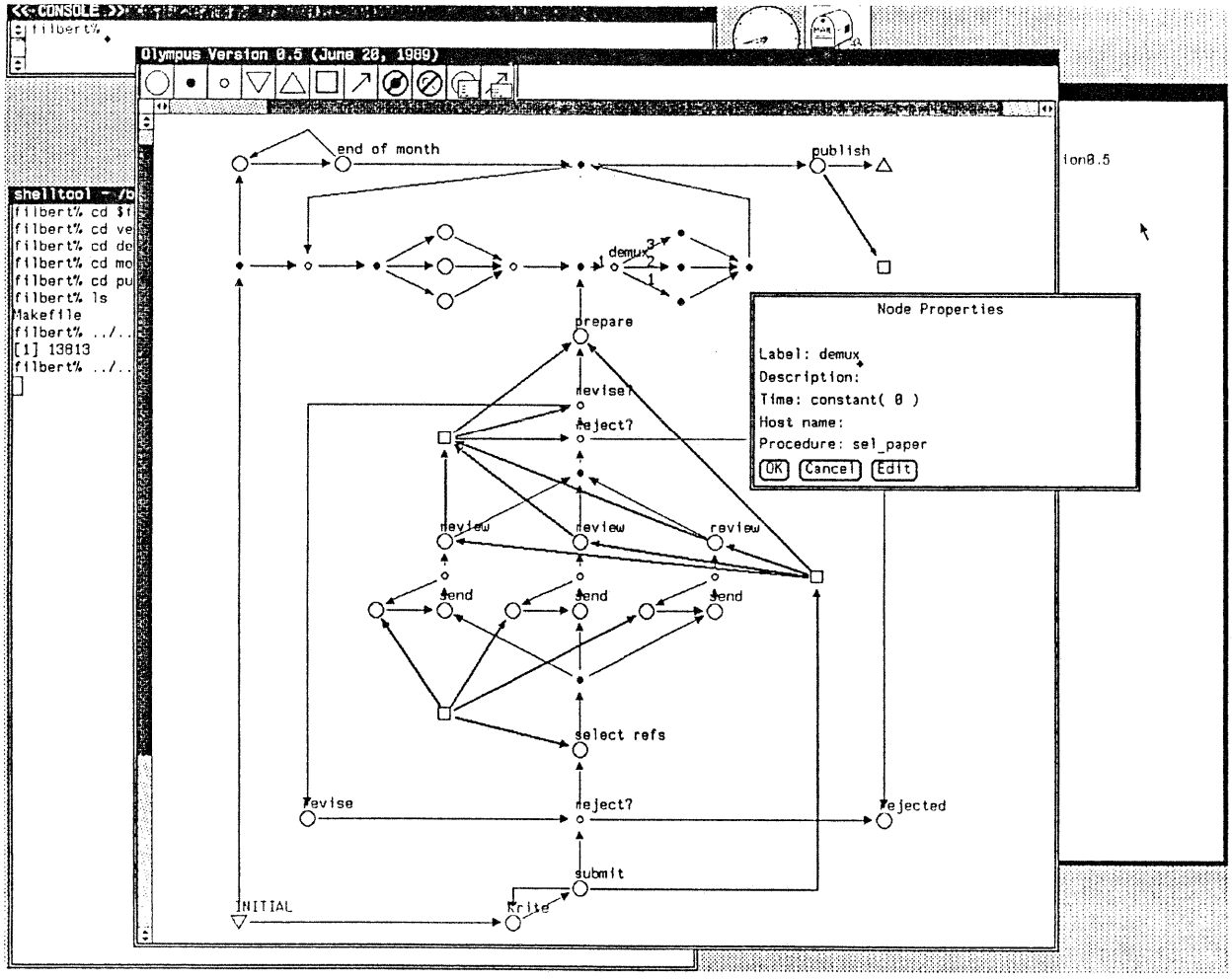


Figure 4: The SunView Client Window

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 [22]. 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-driven 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.

Because of the replacement of SunView by NeWS in Sun environments, the emergence of X Version 11, and because of the limitations in the design, the SunView client was not a good base from which to build other clients. Therefore, we have built subsequent frontends in the NeWS environment.

The NeWS BPG Editor 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

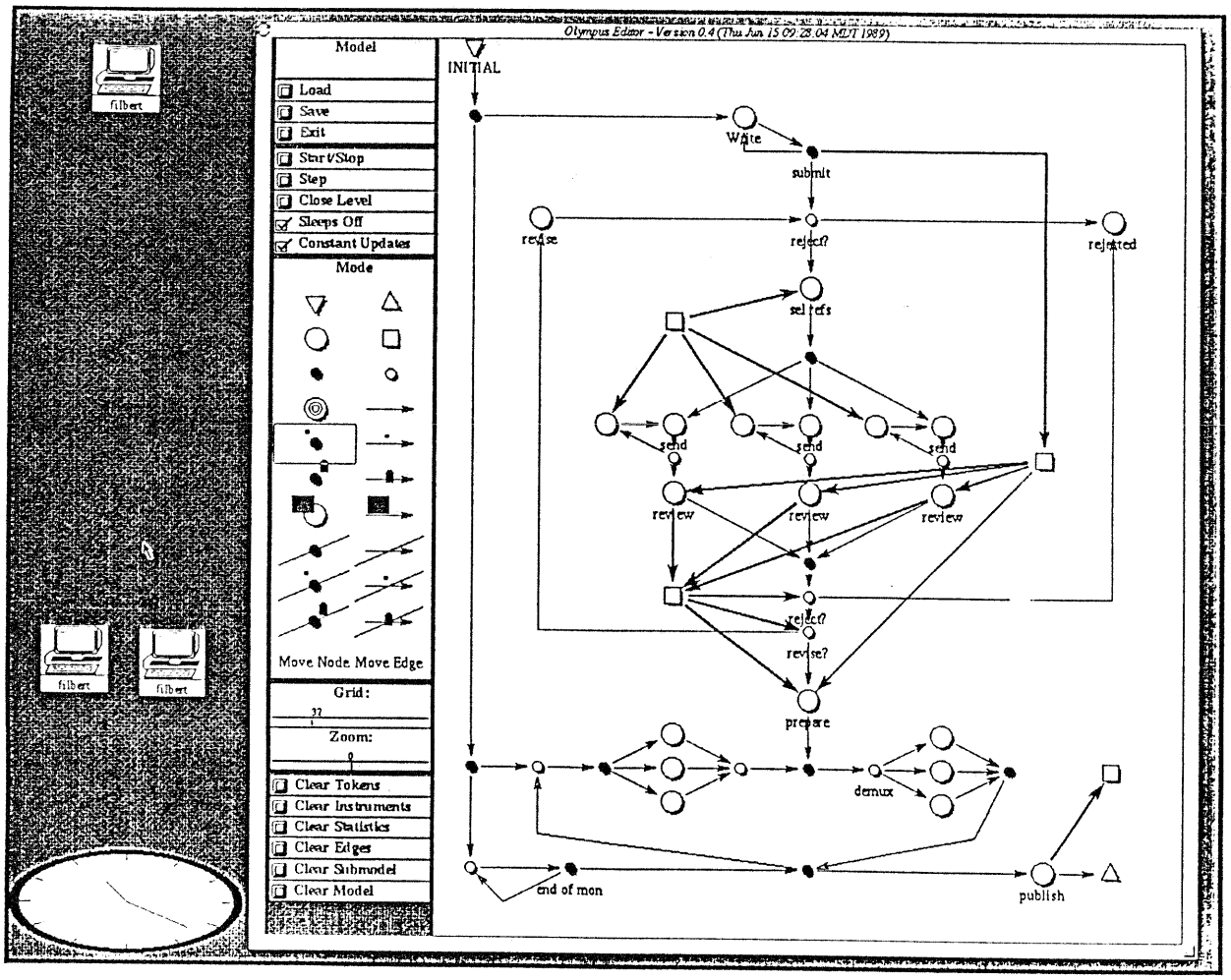


Figure 5: The NeWS Client Window

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) -- see Figure 5.

The Model Editor is a new design, based on type hierarchies of visual network models. The Model Editor 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 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 process 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").

Our experimental Petri net editor (used with the BPG server) was built by modifying the NeWS editor. The modifications did not require server changes (even though the "place firing semantics" were specialized for Petri nets). The BPG server is the same for the BPG frontend as for the Petri net frontend in this experiment.

3.1. Using Olympus-BPG

Traditional simulation modeling breaks down into a number of phases: Target system studies, model design, model construction, model validation, parameter sensitivity analysis, and data collection. The phases overlap as new knowledge is gained in the overall process. For example, it is common to begin the design of the model before the target system is completely understood; in fact, model construction guides the designer to questions that he did not think of during the pure study phase. Similarly, validation generally causes the designer to return to model construction, model design, or even target system study.

Olympus systems attempt to provide support to the analyst during all of the phases of the study. During system study, an editor is used to construct an uninterpreted model of the parts of the target system. As new knowledge is discovered about the target, it is incorporated into the BPG model. The initial parts of the model design proceed concurrently with the study of the target system, where BPGs serve primarily as a documenting device.

As the study shifts into model design, the uninterpreted BPG becomes the focus; timing, hierarchical refinements, and procedural interpretations are provided. The editor is also used to alter the model as the analyst reviews it. The act of deriving interpretations and refining the control and data flow for the BPG constitutes the model construction phase.

Detailed validation can be accomplished in a number of ways, almost all of which are outside the scope of this paper. The animator is the facility that provides the analyst with his initial intuition as to the validity of the model. It points the analyst at critical parts of the model and leads him or her to ask more detailed question about the operation of the target system, or to increase detail in the model. Validation almost inevitably leads to modification of the model, primarily to the specifications of the task interpretations, eg., tuning distribution parameters.

Once the model is judged to be valid, it may be executed on many different loads with many different parameters to investigate the sensitivity of those parameters on different aspects of the performance. This phase often leads to the desire to modify the model, since it may not accurately characterize a parameter in the system that exists in the model. Ordinarily, this means returning to the model design or implementation phase and starting again. With Olympus, that process is rapid because of the form of the model and because of the integration of the tools in the model design environment.

3.2. An Example Session

Suppose that we had constructed the single-server queueing system shown in Figure 6. Each node in the graph is created with a default (unit) time to execute. It is useful to draw the graph, mark the idle nodes, then begin experimenting with arrival and service times. These times are specified by time distributions for p_1 (the arrival distribution), p_4 (the CPU service time distribution), and p_9 (the device service time distribution). Initially, the "depart" node (p_6) will randomly choose output edges for a token, hence jobs will randomly request device I/O or be complete.

The model is refined by attaching probabilities to the two arcs (p_6, p_7) and (p_6, p_8) corresponding to the probability that a job is complete when it finishes a time quantum. A more precise interpretation can be supplied by writing a C

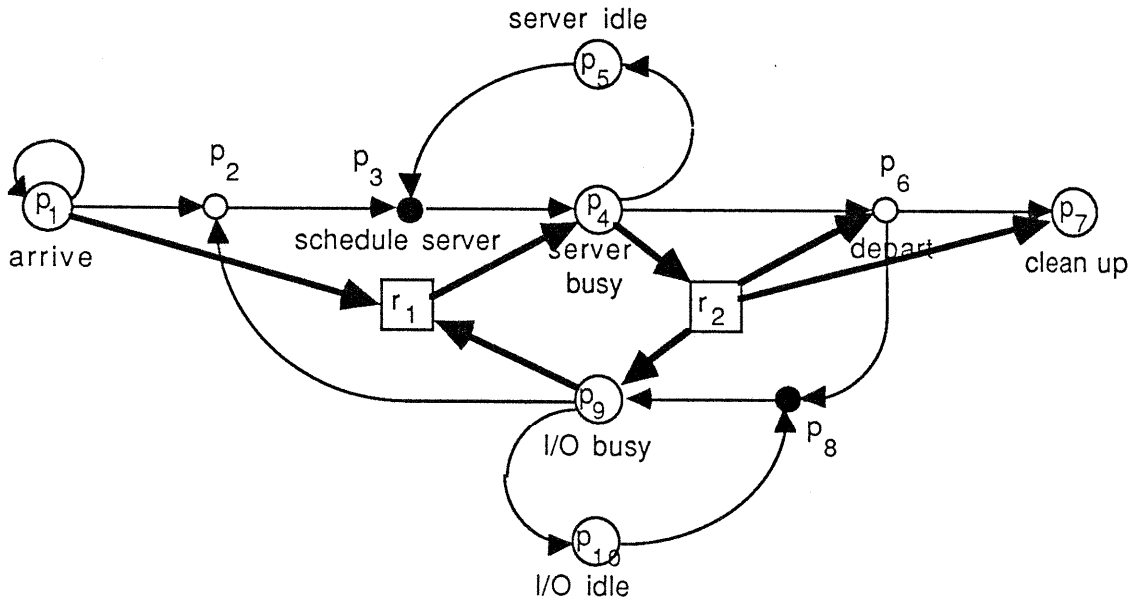


Figure 6: A BPG Queueing Model

procedure for p_6 such that a specific output arc is selected to receive a token representing a job (thus BPGs can be made deterministic by specifying the action in forward conflict situations).

The graph model suggests that there is one server and one I/O device in the system, however, we can simulate two different I/O devices by initially marking the "I/O idle" node with two tokens. This will allow two different jobs to be performing I/O -- reside on the "I/O busy" node at one time. Any of these changes to the model be implemented as the model is being interpreted.

As we continue to use the model, we may wish to distinguish between the two I/O devices, perhaps one is slow and the other is fast. A stochastic model of such operation would simply suggest that a bimodal distribution be used to specify the firing time for the "I/O busy" node. However, such a model will not prevent us from simulating two instances of the slow device being in operation at one time.

An alternative is to change the graph to the one shown in Figure 7. The graph editor has been used to copy p_8, p_9, and p_{10}, to add p_{15}, and to add new arcs connecting the subgraph. It is also necessary to either specify the stochastic conditions under which a job uses device 1 or device 2 (by annotating arcs (p_{15}, p_8) and (p_{15}, p_{13}) with probabilities) or by writing a deterministic procedure for node p_{15} to specify conditions for a job choosing one device or the other.

4. SUMMARY

We have described the Olympus Modeling System Architecture, 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.

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

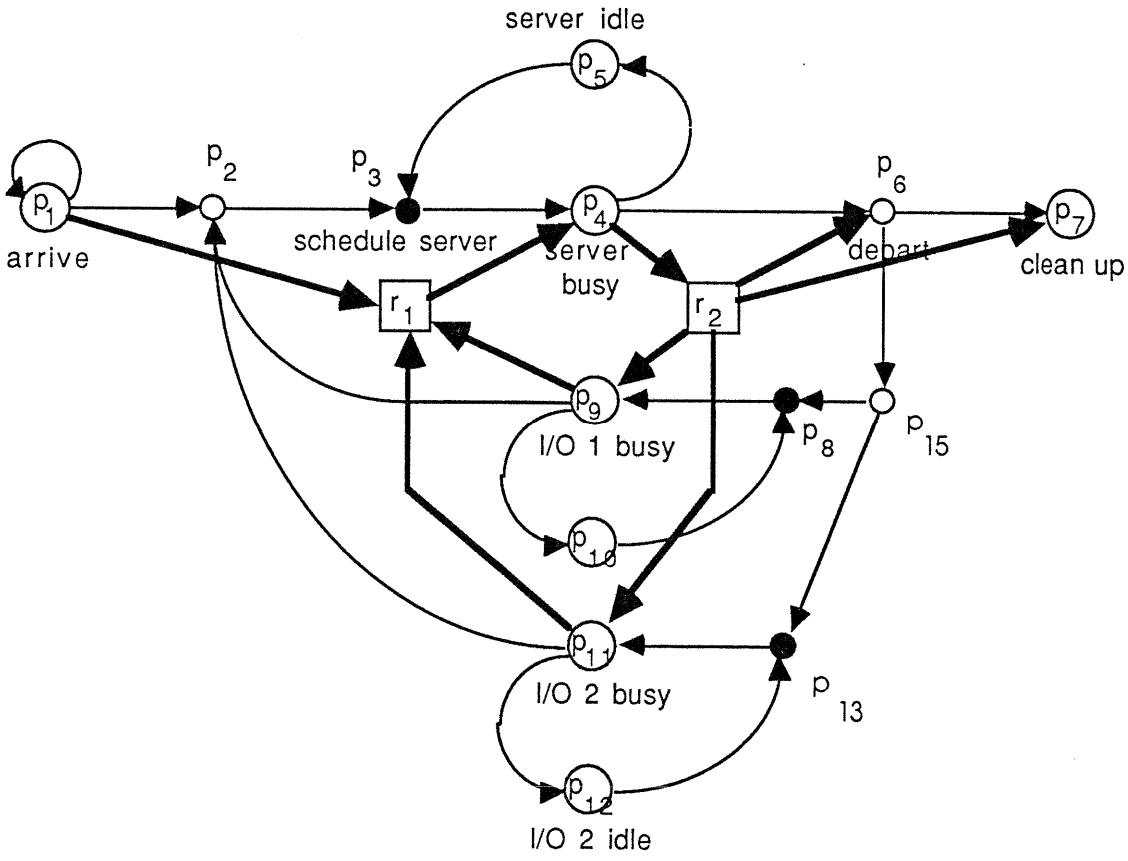


Figure 7: A Two-Device System

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

Our overall research program is concerned with modeling various aspects of complex systems, particularly distributed computer systems. The fundamental assumption behind our approach is that such systems are sufficiently complex that a designer can benefit considerably from interactive support systems to experiment with behavior prior to implementation.

Choosing a particular model on which to base the man/machine interface is very important to the success of the support system, and very difficult to do so that it is acceptable to a wide range of users. We each have a set of preconceived notions about modeling primitives, often based largely on aesthetics. Olympus provides fundamental operation as a simulation and animation system allowing one to use a somewhat arbitrary user/model interface.

Our overall research projects center around the application of several different formal models of computation as bases for interactive system support. In this paper we have described a simulation system while in other projects we study multi-tiered modeling systems [6, 23] and deterministic, visual programming systems [2]. The support systems for these other studies are all based on the Olympus Architecture.

Finally, we expect to use Olympus as the basis of a cooperative, distributed software development environment. The current architecture and implementation support multiple users working on a common model with atomic transactions and simultaneous update of state at the clients. We feel that this is a promising basis for general cooperative problem solving systems.

5. ACKNOWLEDGEMENTS

Several people have worked on the Olympus system including Mohammad Amin, Zuraya Aziz, Adam Beguelin, Mimi Beaudoin, Isabelle Demeure, Steve Elliott, John Hauser, Art Isbell, Nikolay Kumanov, Jeff McWhirter, and Bruce Sanders.

This research has been supported by NSF Grant No. CCR-8802283, NSF cooperative agreement DCR-8420944, and a grant from U S West Advanced Technologies.

6. REFERENCES

1. G. Balbo and G. Chiola, "Stochastic Petri Net Simulation", *1989 Winter Simulation Conference Proceedings*, Washington, D. C., December 1989, 266-276.
2. A. L. Beguelin, "Deterministic Parallel Programming in Phred", University of Colorado, Department of Computer Science, Ph. D. Dissertation, May 1990.
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. M. Broy, *Control Flow and Data Flow: Concepts of Distributed Programmings*, Springer Verlag, 1985.
5. I. M. Demeure, S. L. Smith and G. J. Nutt, "Modeling Parallel, Distributed Computations using ParaDiGM -- A Case Study: The Adaptive Global Optimization Algorithm", *Fourth SIAM Conference on Parallel Processing for Scientific Computing*, Chicago, IL, December 1989.
6. I. M. Demeure, "A Graph Model, ParaDiGM, and a Software Tool, VISA, for the Representation, Design, and Simulation of Parallel, Distributed Computations", University of Colorado, Department of Computer Science, Ph. D. Dissertation, June 1989.
7. I. M. Demeure and G. J. Nutt, "Prototyping and Simulating Parallel, Distributed Computations with VISA", submitted for publication, May 1990.
8. R. S. Elliott and G. J. Nutt, "Remarks on the Cost of Using A Remote Procedure Call Facility", University of Colorado, Department of Computer Science Technical Report No. CU-CS-426-89, February 1989.
9. C. A. Ellis and G. J. Nutt, "Office Information Systems and Computer Science", *ACM Computing Surveys 12*, 1 (March 1980), 27-60.
10. G. Estrin, "A Methodology for Design of Digital Systems -- Supported by SARA at the Age of One", *AFIPS Conference Proceedings of the National Computer Conference 47* (1978), 313-324.
11. G. Estrin, R. S. Fenchel, R. R. Razouk and M. K. Vernon, "SARA (System ARchitects Apprentice): Modeling, Analysis, and Simulation Support for Design of Concurrent Systems", *IEEE Transactions on Software Engineering SE-12*, 2 (February 1986), 293-311.
12. H. J. Genrich, "Predicate/Transition Nets", in *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, W. Brauer, W. Reisig and G. Rozenberg (editor), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, New York, 1987.
13. M. L. Graf, "Building a Visual Designer's Environment", MCC Technical Report No. STP-318-87, October, 1987.
14. *PAWS/GPSM marketing brochures*, Information Research Associates, Austin, TX, 1988.
15. K. Jensen, "Coloured Petri Nets", in *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, W. Brauer, W. Reisig and G. Rozenberg (editor), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, New York, 1986, 248-299.
16. S. J. Leffler, R. S. Fabry, W. N. Joy and P. Lapsley, "An Advanced 4.3BSD Interprocess Communication Tutorial", in *Unix Programmer's Manual Supplementary Documents 1*, Computer Systems Research Group, Computer Science Division, Department of Electrical Engineering and Computer Science, University of California, Berkeley, April 1986.
17. B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer 18*, 8 (August 1985), 87-94.
18. T. Murata, "Petri Nets: Properties, Analysis and Applications", *Proceedings of the IEEE 77*, 4 (April 1989), 541-580.
19. K. M. Nichols and J. T. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer 21*, 5 (May 1988), 39-48.
20. J. D. Noe and G. J. Nutt, "Macro E-Nets for Representing Parallel Systems", *IEEE Transactions on Computers C-12*, 8 (August 1973), 718-727.
21. G. J. Nutt, "The Formulation and Application of Evaluation Nets", Ph.D dissertation, Computer Science Group, University of Washington, 1972.

22. G. J. Nutt and P. A. Ricci, "Quinault: An Office Environment Simulator", *IEEE Computer 14*, 5 (May 1981), 41-57.

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

24. G. J. Nutt, "Olympus: An Extensible Modeling and Programming System", Technical Report No. CU-CS-412-88, Department of Computer Science - University of Colorado, Boulder, October 1988.

25. G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter and B. Sanders, "Olympus User's Manual", Technical Report CU-CS-382-87, Department of Computer Science - University of Colorado, Boulder, December 1987 (revised June, 1989).

26. G. J. Nutt, "A Formal Model for Interactive Simulation Systems", Technical Report No. CU-CS-410-88, Department of Computer Science - University of Colorado, Boulder, September 1988 (Revised May 1989).

27. C. Ramchandani, "Analysis of Asynchronous Concurrent Systems by Timed Petri Nets", Ph.D. dissertation, MIT, 1974.

28. R. R. Razouk and C. V. Phelps, "Performance Analysis Using Timed Petri Nets", *Proceedings of 1984 International Conference on Parallel Processing*, August 1984, 126-129.

29. 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.

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

APPENDIX: THE BILOGIC PRECEDENCE GRAPH MODEL

Bilogic Precedence Graphs (BPGs) are composed from a set of *tasks*, a set of *control dependencies* among the tasks, and a specification of *data references* among tasks. A BPG can be thought of as the union of a control flow subgraph and a data flow subgraph. The control flow subgraph is made up of nodes that correspond to tasks and edges that specify precedence among the tasks. The node set for the data flow subgraph is the union of the task node set with another set of nodes representing data repositories; edges in the data flow graph indicate data references by the tasks.

BPGs are directly descended from Information Control Nets [9], which evolved from our work with data flow models and E nets [20]; and E nets are a derived directly from Petri nets [21]. The control flow subgraph is similar to the UCLA Graph Model of Behavior (GMB) [10, 11] in that it specifies conjunctive ("AND") and disjunctive ("OR") input and output logic specifications for each task. Let small, open circles represent tasks with exclusive OR logic; and small, closed circles represent tasks with AND logic (see the examples on the left side of Figure A1). OR-tasks are enabled by control flow into any input arc, and upon task termination, control can flow out on any output arc. AND-tasks are not enabled until control flows to the task on every input arc, and upon termination control flows out every output arc. Large circles represent tasks with OR-input logic and AND-output logic; ordinarily, we only use single input and single output arcs on these circles since we use them to emphasize the notion of nontrivial processing. Our choice of these primitives is based on our users' preferences (from the E-net and ICN studies). Other logic combinations can be built from these primitives, e.g., AND-logic input and OR-logic output is attained by connecting the output of a multi-input, single-output AND-node with to a single-input, multiple-output OR-node.

As in other marked graphs, the control flow state is represented by tokens (data flow state is not explicitly represented in BPGs). Thus, one can think of markings and firings of the various tasks in the control flow subgraph just as in Petri nets. A BPG is activated by marking appropriate tasks with tokens, at which time the BPG firing rules (control flow logic rules) describe sequences of markings corresponding to control flow among the tasks.

A single-entry, single-exit task directly maps to a Petri net transition with one input and one output place, see Figure A1a. That is, the task fires whenever a token arrives at the input, and a token leaves the task when it has completed firing. A BPG AND-task corresponds to a set of Petri net places and a transition, see Figure A1b. The OR-task is similar to a Petri net with forward/backward conflict, see Figure A1c. Token paths can merge or separate at an OR-task. As in Petri nets, an uninterpreted OR-task in nondeterministic. (However, we will provide interpretations for BPG tasks -- see below.)

Data flow is represented by adding data repository nodes to the control flow graph, and arcs interconnecting nodes and data repositories. (Squares are used to represent data repositories in BPGs.) Data flow in a BPG does not correspond directly to data flow in a traditional data flow graph (see, for example, [4]). That is, task firing is specified by tokens in the control flow subgraph, whereas the data flow subgraph represents data references by the tasks. Thus, an arc from a task to a repository represents the case that the task *may* write information to the repository, and an arc from a repository to the task represents that the task *may* read information from the repository. An interpretation for the task specifies whether or not the task references the repositories to which it is connected for any particular task firing. The resulting model is used to represent storage references and inter-task

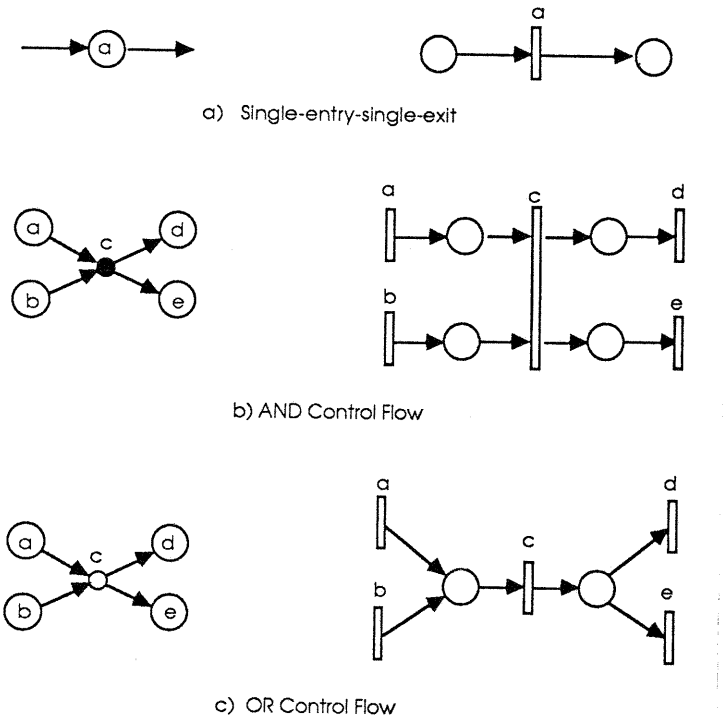


Figure A1: BPG Primitives

communication.

Figure A2 is a Petri net model of a simple queueing system with jobs requesting CPU service and I/O service alternately. The leftmost portion of the Petri net represents the arrival of jobs and the rightmost portion represents their departure. The places labeled "server idle" and "I/O idle" must initially contain a token for the model to behave properly. (Multiple servers and multiple I/O devices could be represented by corresponding numbers of tokens on the respective idle locations.)

Figure 6 is the corresponding BPG. Task p_1 has AND output logic and OR input logic, so the edge from p_1 to itself will cause the task to be cyclic, corresponding to the job arrival portion of the Petri net. When a job is created, a record describing the job is written to repository r_1 by p_1. Task p_2 is used to merge two OR paths, and p_3 fires only when the server is idle (there is a token on the edge (p_5, p_3)) and there is a job to serve (a token is on edge (p_2, p_3)). Task p_4 models the job receiving service by reading the details of the job description from r_1 and writing the updated job status to r_2 when the time slice has been completed (p_4 terminates). When the job leaves the server, then the server becomes idle AND a decision is made as to whether or not the job is done or requires I/O (task p_6). Tasks p_8, p_9, and p_{10} model the I/O device operation, updating the job status in r_1 and r_2 during the process.

While tasks share the firing rule properties of pure Petri net transitions as described in Figure A1, they also employ the notion of non-zero transition time of E-nets, ICNs, and Timed (Performance) Petri Nets [27, 28]. This is the means for introducing time into the simulation. For example, in Figure 6, the job's service time is represented by the amount of time that the token resides on task p_4.

Each task may have a procedural *interpretation*, to specify the amount of time required for firing the task. Thus, task p_4 can determine the desired amount of service time by evaluating a function such as shown in Figure A3. The evaluation of the procedure results in a simulated time being returned. (Notice that the interpretation for p_4 references repositories r_1 and r_2 using the *access-repository* procedure without specifying read/write commands. The arcs in the graph description, passed as an argument to *access-repository* determine the direction of information flow, i.e., this is the mechanism for data flow in the model.)

The interpretation is evaluated each time the task is fired. Tasks with OR (output) logic may use interpretations to specify deterministic behavior; the procedure evaluates information available to it (from repositories), then selects an output arc to receive the resulting token. We represent this choice as a second value returned from the

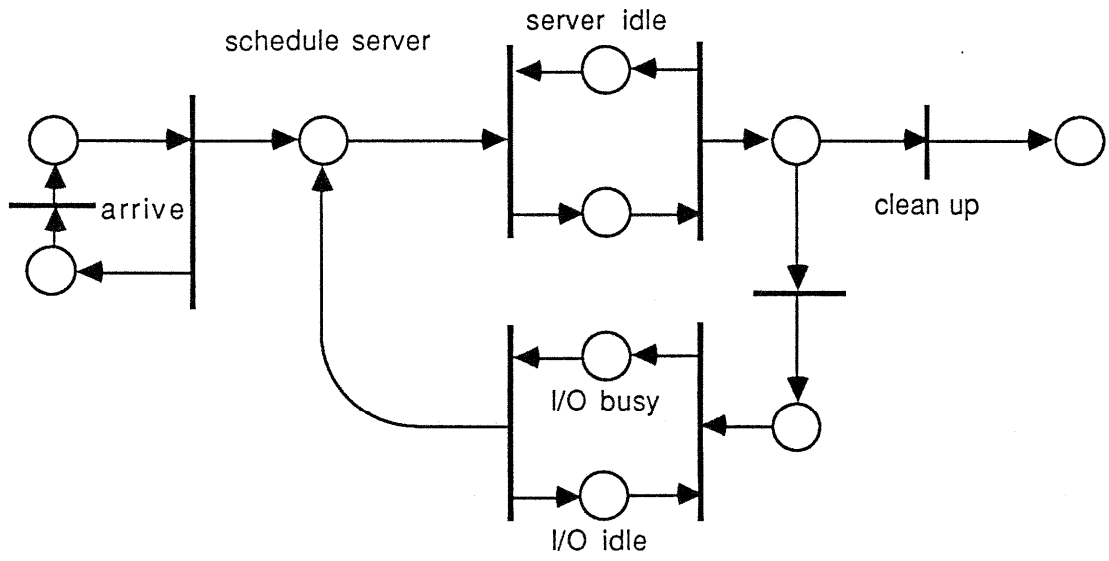


Figure A2: A Queueing System

interpretation of an OR-task, i.e.,

(time, out-arc) = f(OR-task)

BPG tasks are hierarchical. Any task may be refined by defining a new BPG which has the same input/output behavior as the parent task. Using ideas similar to those in macro E-nets [20], it is possible to define subnets that have the same logical input/output properties as an individual task in the BPG. For example, suppose that the server task were actually two subservers, only one of which could be busy at any time (this is a hypothetical example to describe hierarchical nodes in the graph). Then, the sub-BPG shown in Figure A4 would be one hierarchical refinement of p. Task $p_{4,1}$ would be a decision task that decided which of the two units were to be used for this operation. One unit would be represented by task $p_{4,2}$ and the other by $p_{4,3}$. Tasks $p_{4,4}$ and $p_{4,5}$ would be used to produce the correct token response to the outputs of the original p_4.

This is a brief, intuitive description of BPGs, particularly as they relate to Petri nets. Part of the motivation for using BPGs is that they are sufficiently simple, and similar to other commonly used models, that they are natural for representing the individual events involved in the simulation of a system. (A similar argument was used to justify the formulation of E-nets [20, 21]). The other rationale for using BPGs is that they encompass the semantics of several other formal models, including other variants of Petri nets, queueing networks, and several CASE models; by implementing the modeling system so that it interprets the semantics of BPGs, it is possible to provide a user interface that employs the syntax of these other models at a particular user's design workstation.

For the interested reader, a more complete and more formal description of BPGs can be found in [26].

```
p4()
{
        struct *job;

/* Read r1 */
        access_repository((r1, p4), job)
/* Change a field in the record read from the repository */
        update(job.statistics);
/* Write r2 */
        access_repository((p4, r2), job);
/* Return the amount of time required for the task to fire */
        return(job.service_time);
}
```

Figure A3: A Node Interpretation

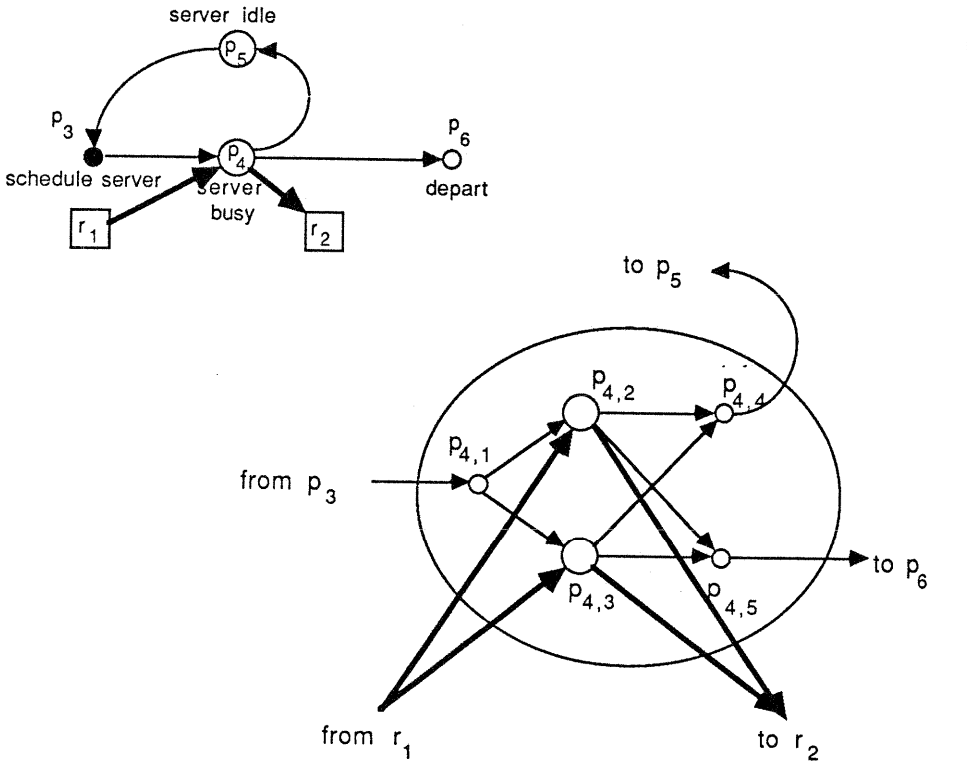


Figure A4: A Refined Task

OLYMPUS: AN INTERACTIVE SIMULATION SYSTEM

Gary J. Nutt
Adam Beguelin
Isabelle Demeure
Stephen Elliott
Jeff McWhirter
Bruce Sanders
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

ABSTRACT

In this paper, we describe the Olympus Modeling System, a prototype modeling and simulation system we have built to interpret formal models of computation. Olympus employs a graphical, interactive user interface that enables one to control the system as it interprets a model. The system supports animation and simulation; animation is used to observe qualitative behavior of a model and simulation is used to obtain quantitative information about the model's behavior. The system supports multiple simultaneous users with fine-grained interaction between the users and the system.

1. INTRODUCTION

The *Olympus Modeling System* is an interactive, distributed model interpretation environment for bilogic precedence graphs (BPGs). BPGs are interpreted control flow 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. An interpreted BPG corresponds to a simulation model of some system.

An interactive 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.

We address these requirements with Olympus by defining an underlying formal model for the simulation, then by implementing the user interface to the simulation interpreter as an asynchronous subsystem. This enables the user interface subsystem to respond immediately to user requests, even while the simulator is "busy" with other tasks. The two subsystems then communicate using conventional interprocess communication mechanisms.

In the remainder of the paper, we will first describe BPGs, then the design and implementation of the current Olympus frontend and backend. Finally, we will provide a simple example to illustrate the use of the system and the model.

1.1. Related Work

There are a number of interactive simulation systems used for performance prediction. 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 [2, 5]. 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 [19].

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 [16].

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 [1,4,5], the Performance Analysis Workstation (PAW) [7], PARET [10], and Quinault [13] are all examples of this type of system.

GADD [9] 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 [8]). As a consequence, it is possible to interconnect simulations with fully-implemented components, thus providing a testbed debugging environment.

2. THE BILOGIC PRECEDENCE GRAPH MODEL

Bilogic Precedence Graphs (BPGs) are composed from a set of *tasks*, a set of *control dependencies* among the tasks, and a specification of *data references* among tasks. A BPG is the union of a control flow subgraph and a data flow subgraph. The control flow subgraph consists of nodes that correspond to tasks and edges that specify precedence among the tasks. The node set for the data flow subgraph is the union of the task node set with another set of nodes representing data repositories; edges in the data flow graph indicate data references by the tasks.

The control flow subgraph is similar to the UCLA Graph Model of Behavior (GMB) [3,16] in that it specifies conjunctive ("AND") and disjunctive ("OR") input and output logic specifications for each task. Let small, open circles represent tasks with exclusive OR logic; and small, closed circles represent tasks with AND logic (see Figure 1). OR-tasks are enabled by control flow into any input arc, and upon task termination, control can flow out on any output arc. AND-tasks are not enabled until control flows to the task on every input arc, and upon termination control flows out every output arc. Large circles represent tasks with OR-input logic and AND-output logic; ordinarily, we only use single input and single output arcs on these circles since we use them to emphasize the notion of nontrivial processing.

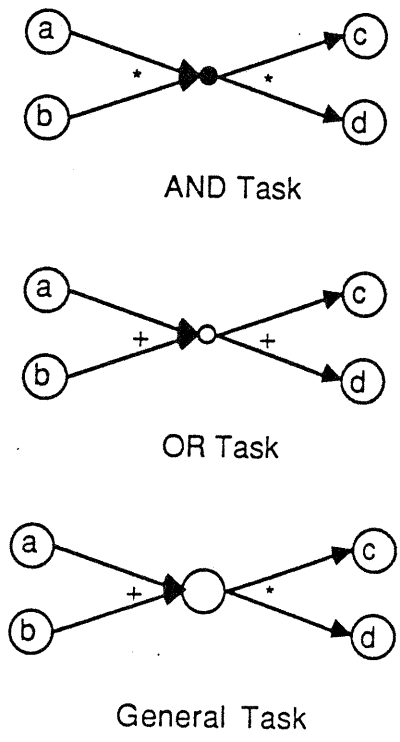


Figure 1: BPG Primitives

Following Petri nets, the control flow state is represented by tokens (data flow state is not represented explicitly in BPGs). Thus, one can think of markings and firings of the various tasks in the BPG just as in Petri nets. A BPG is activated by marking appropriate tasks with tokens, at which time the BPG firing rules (control flow logic rules)

describe sequences of markings corresponding to control flow among the tasks.

Data flow in a BPG is represented by adding data repository nodes to the control flow graph, and arcs interconnecting nodes and data repositories. Data repositories are meant to explicitly represent *possible* flow of data among individual tasks. The task interpretation will ultimately determine if data is read from (written to) a data repository by the task. The resulting model illustrates storage references and inter-task communication. Squares are used to represent data repositories in BPGs.

Figure 2 is a BPG of a simple system with two customers and one server. The server is represented by tasks s_1, s_2, and s_3. Tasks s_2 represents the case that the server is idle; at initialization, this task contains a token. Task s_3 is an AND-task which fires only when there is a token on arc (s_2, s_3) and another on arc (s_2, s_3). Whenever a token resides on task s_1, then the server is busy.

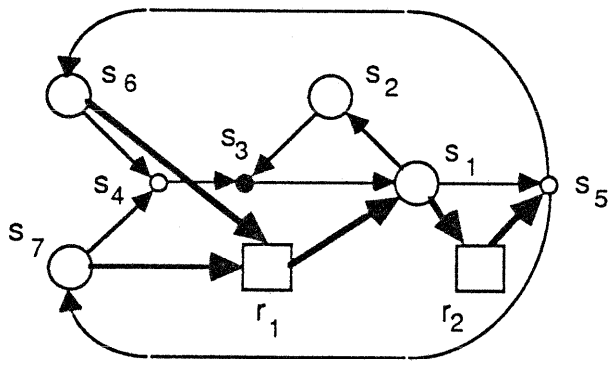


Figure 2: A 2-Customer Server

Tasks s_4 through s_7 model the customer requests for service. When a token is on s_6, then this represents the case when the first customer is "thinking" and does not require the service; s_7 represents a similar state for the second customer. Tasks s_4 and s_5 multiplex tokens (representing requests for service) into and out of the server.

In order to ensure that s_4 has enough information to demultiplex a token to the correct "thinking" task, it reads information from repository r_2 (placed there by s_1) to identify which customer was just serviced. Repository r_1 is used in a similar manner to provide the server busy task (s_1) with the corresponding information.

Each task may have a procedural *interpretation*, to specify the amount of time required for firing the task, and for performing miscellaneous simulation tasks. Thus, task s_1 can infer the desired amount of service time and manage the customer identity by evaluating a procedure similar to:

```
s1()
{
        struct *request;

        request = read_repository(r1);
        wait(request.service_time);
        write_repository(r2, request);
}
```

The interpretation is evaluated each time the task is fired. Tasks with OR (output) logic can use interpretations to specify deterministic behavior; the procedure evaluates information available to it (from repositories), then selects an output arc to receive the resulting token. For example, s_5 might look like:

```
s5()
{
        struct *request;

        request = read_repository(r2);
        if (request.customer == 'first')
                route(s6)
        else
                route(s7);
}
```

BPG tasks are hierarchical. Any task may be refined by defining a new BPG which has the same input/output behavior as the parent task. While this is an important aspect of BPGs (to address scaling problems) we do not discuss it further in this paper.

This is a brief, intuitive description of BPGs. Part of the motivation for using BPGs is that they are sufficiently simple, and similar to other commonly used models, that they are natural for representing the individual events involved in the simulation of a system. (A similar argument was used to justify the formulation of E-nets [11, 12]). The other rationale for using BPGs is that they encompass the semantics of several other formal models, including Petri nets, queueing networks, and several CASE models; by implementing the modeling system so that it interprets the semantics of BPGs, it is possible to provide a user interface that employs the syntax of these other models at a particular user's design workstation.

For the interested reader, a more complete and formal description of BPGs can be found in [15].

3. THE OLYMPUS MODELING SYSTEM

The BPG model provides a language for describing target system behavior, while 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 (using bit-map workstation technology), we have created an environment in which alternatives -- changes in loading conditions, changes in parameters, or changes in the model itself -- are easy to explore. Furthermore, the design decouples the user interface from the simulation itself, allowing the user to exercise highly interactive control over the simulation.

Olympus has been implemented in a network of Sun workstations, using Sun graphics and network software. We briefly describe the architecture and one of the implementations; additional details can be found in [14].

3.1. The Architecture

Olympus is an interactive system composed from a *frontend* and a *backend*, see Figure 3. The frontend implements the user interface, while the backend provides storage and interpretation of the model (independent of the frontend implementation).

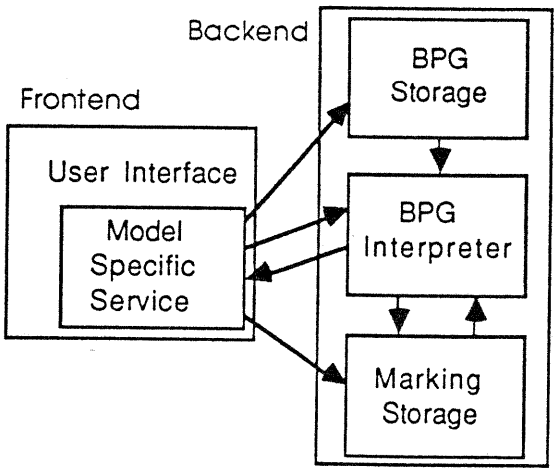


Figure 3: The Olympus Architecture

The Frontend

The separation of the system into a frontend and a backend allows the user interface to be independent of the simulation details. It also enables us to decouple the user interface console tasks from the simulation tasks, yet provide explicit means for the two parts to interact.

The frontend serves two main purposes: First, it implements the human factor (and many of the cognitive) aspects of the interaction between the user and the machine, i.e., it acts as a user interface. Second, it implements 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.

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

The frontend generates a structured internal representation of the model that is 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 could implement any of a number of models or languages of differing syntax that can be mapped into BPG semantics, see [15]. However, in this paper we only discuss a BPG frontend in use with the backend.

The Backend

The backend is an interpretation engine 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 is decomposed into parts to handle storage

of the model, storage of the marking, task interpretation, and repository interpretation.

Model and marking storage manage records representing atoms in the model, e.g., a task, and edge, or an interpretation for models and a token/task for markings. The task interpreter implements the BPG control flow semantics. It moves tokens around on the graph, evaluating interpretations as required. During evaluation, the task interpreter may invoke the repository interpreter as a function of the interpretation of a specific task. Thus, the repository interpreter acts as a server to the task interpreter client.

The frontend and backend communicate using a client-server protocol unique to Olympus. The protocol is based on asynchronous message passing primitives in which messages are formatted for efficient transformation of editing, interpretation, and control information between the two components. That is, the backend is a server which responds to commands from the frontend (a client). For example, when the user creates a new node at the frontend, the backend server is told to store the node; the server responds by updating the model storage and telling the frontend that there is a new typed object in the model. (The frontend can treat the object as it sees fit.)

The backend supports single-stepped interpretation of a model, 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.

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.

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.

3.2. An Implementation

We have built several implementations of the frontend and backend; here we summarize only the most recent ones. Figure 4 illustrates an implementation of the architecture in a network of Sun workstations, using Unix (R) processes, graphics, and network protocols.

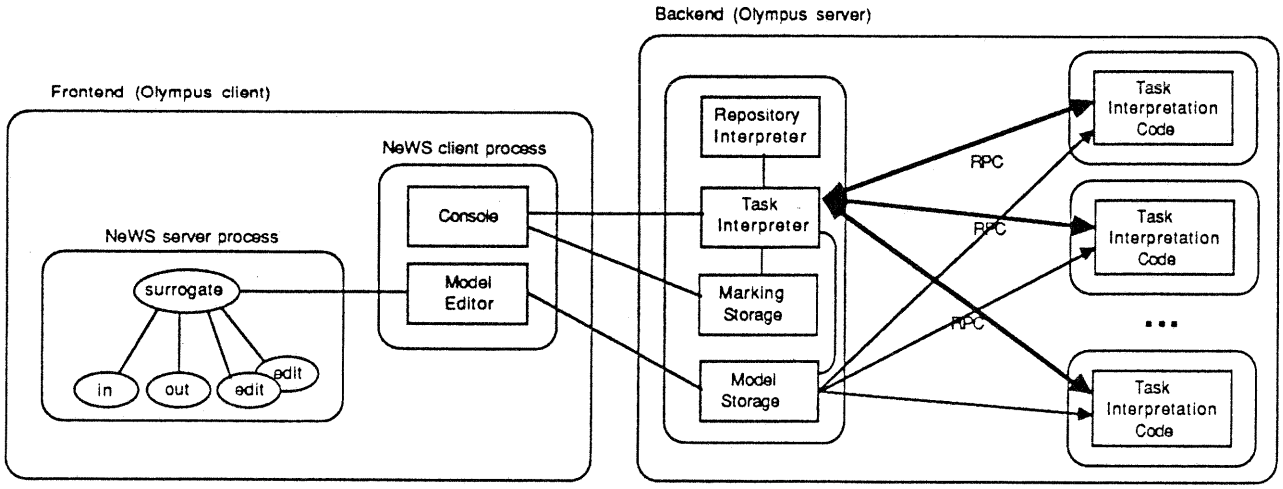


Figure 4: An Olympus Implementation

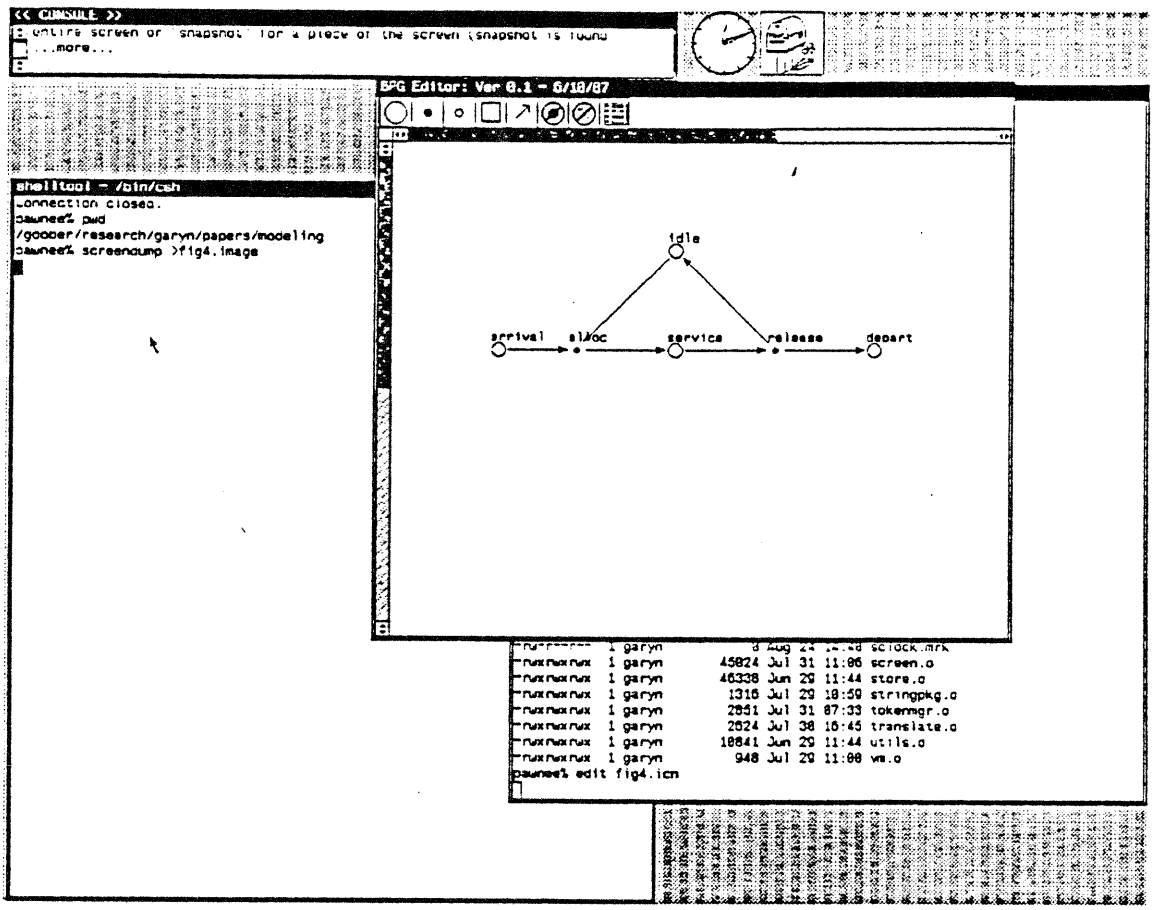


Figure 5: The SunView Olympus Frontend

There are several different versions of the frontend in use: The first point-and-select version is the most functionally complete, although not the most aesthetically pleasing. It was built on the SunView [20] window package as a single Unix process. Figure 5 is the display for this frontend. As an experiment, we also developed a version on a Symbolics Lisp workstation [17], however we have not continued to maintain this version.

The newest frontend is being implemented as two Unix processes that conform to Sun's NeWS model [22]. That is, the Olympus client process is actually implemented as a NeWS client and a NeWS server process. The NeWS client implements the logical aspects of the user interface, while the NeWS server process -- implemented as a community of lightweight processes called "in," "out," and "edit" in Figure 4 -- is responsible for placing images on the display. Thus, the model editor is the only part of the frontend that needs to interact with the model storage in the Olympus server. Also, NeWS allows its client and server processes to be in execution on distinct machines.

The Olympus server (backend) is implemented as n+1 Unix processes: The first process multiplexes among the four interpretation and storage "subprocesses." The other n processes are used to evaluate BPG interpretations.

BPG interpretations can be defined in any language, provided that the definition can be viewed as a procedure callable in C. The task interpreter uses the Sun Remote Procedure Call (RPC) facility [21] to invoke the interpretation procedure whenever the corresponding task is fired. In order to promulgate concurrency in the simulator, we have used the RPC facility as a remote fork rather than as a procedure call; thus, n tasks can be interpreted at one time by starting n RPC servers on different machines.

The frontend-backend interface is implemented on top of sockets [21]. This allows the frontend and backend to be executed on distinct machines in a network environment.

The separation of the frontend from the backend has allowed us to implement the interface so that there can be an arbitrary number of clients connected to the same server. Each client can send editing or console requests to the server, and the server will respond to all clients as if they were one, since they are connected to the server via a single, shared socket. This allows multiple users to view a single server session with full access and viewing rights. Since the server serializes each transaction, the users will not cause the server to violate critical sections or otherwise violate concurrency constraints.

4. AN EXAMPLE

Suppose that we were configuring an internet as a composition of four different networks. There are several different configurations that one might consider, e.g., see Figure 6. In Figure 6a, three gateway machines interconnect the four nets; two gateways are used in in Figure 6b, and a single centralized gateway is used to interconnect the machines. in Figure 6c.

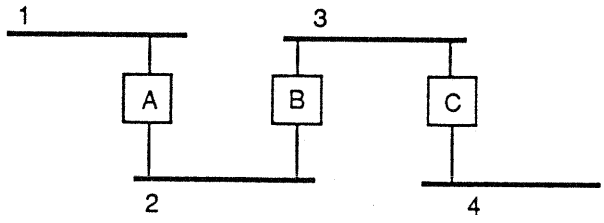
The gateway machines in the configuration for Figure 6a should not be as expensive as those in Figure 6b, and the single gateway in Figure 6c should be the most expensive (highest performance).

The single-gateway configuration may be the most cost-effective solution, particularly in cases where hosts on each subnet need to communicate an equal amount with all other hosts in the internet. However, a machine that is fast and large enough to support this configuration may be too costly. In this case, other configurations should be considered

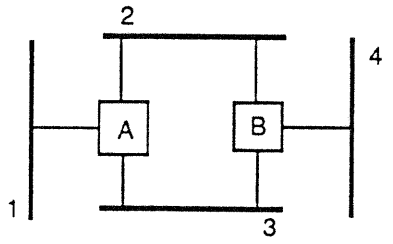
We will use this general scenario to describe how BPGs and Olympus can be used to quickly examine various scenarios (our discussion is far from complete, due to space limitations).

4.1. Modeling the Internet Configurations

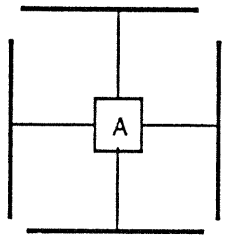
We can reuse part of the model shown in Figure 2 to represent a gateway machine. In this application it is not necessary to re-enable two customers, so tasks s_5 through s_7 are not needed. Consider the BPG shown in Figure 7: The service request population is now modeled by four submodels (g_1 through g_4) representing the four subnets.



6(a) Triple-Gateway Configuration



6(b) Double-Gateway Configuration



6(c) Single-Gateway Configuration

Figure 6: Possible Internet Configurations

Each subnet produces a token by firing, then reenables itself to produce a token at some later time. Each g_i has the form:

```
g_i0
{
        wait(sample(interarrival_distribution));
}
```

The service time of the gateway machine can be modeled by a constant distribution, or by something more complex if the gateway machine is more complex. (For example, the gateway may not be reliable if it is saturated, i.e., it may drop packets. Our simple model does not take this into account.)

The BPG shown in Figure 7 is easily constructed in Olympus, and can be viewed as an animation to gain a qualitative feel for the relative power required from the single gateway machine to satisfy various loads (as determined by the g_i distributions) As the gateway becomes saturated, tokens will build up on the arc from s_4 to s_3.

Olympus allows multiple users to view the animation, and allows the graph to be edited while the animation is in progress. For example, one could add a fifth network to the simulation model without halting the animator.

Figure 8 is a model of the configuration shown in Figure 6b. In this model, some fraction of the load from network 2 goes to gateway (server) A and the other portion goes to gateway B. The relative speeds of the two gateways can be decreased by increasing the mean of the service time distributions.

Figure 9 is a model of the configuration shown in Figure 6a. Again, the loads are split for networks 2 and 3, since they each have two gateway machines.

Olympus is used to create the models, to observe their behavior in qualitative terms, then to obtain quantitative performance data about their behavior. (The average number of tokens on the arcs incident into $s_{x,3}$ reflects the amount of packets that need to be buffered at a gateway.)

It is easy to explore a wide variety of gateway machine performance considerations by adjusting the service time distribution of the gateway server tasks. The performance is again reflected by the token dwell time at arcs that represent the gateway queues.

5. SUMMARY

Olympus is a working prototype modeling system. It is currently being used to model memory access strategies in MIMD machines [18], even though it continues to be developed.

The frontend/backend architecture has provided considerable implementation freedom for focusing on the simulation or on the user aspects of the system. It has also been the major factor in meeting the basic requirements described in the Introduction. Since the frontend is a separate process from the simulator, it is responsive to the commands and queries of the user. The backend has been designed to field messages from the frontend as part of it's basic simulation cycle, thus it, too, is responsive to the user control without undue complication. Because the frontend and the backend communicate using a socket pair, any number of frontends can connect to the socket at one time. The backend will send all information needed by a frontend to the socket, where each instance of the frontend will react to the information by updating a screen, moving a token, etc. Messages from the frontend to the backend will be serialized by the socket, thus each frontend can act as a console, i.e., backend transactions are atomic.

The BPG model provides a graph model of the computation describing a simulation program. While we have distributed the simulation program interpreter (task interpretations are executable on different machines), we are currently exploring techniques for distributing the model interpretation -- both by distributing the model and by further distributing simulation functions. One goal of this research is to address simulation strategies that lie between the conservative Chandy-Misra technique [8] and the optimistic time warp technique [6].

Our experience with Olympus and BPGs has shown us that the architecture enables us to implement systems that meet the goals described above. However, we continue to refine our language and our architecture. For example, newer languages will incorporate data on tokens, and will place more constraints on the form of the interpretations (currently, they are any C procedure). Our current version of Olympus provides limited support for hierarchical BPGs, which we believe is a fundamental requirement for scalable systems; we are currently designing new facilities to handle hierarchies more generally.

ACKNOWLEDGEMENTS

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.

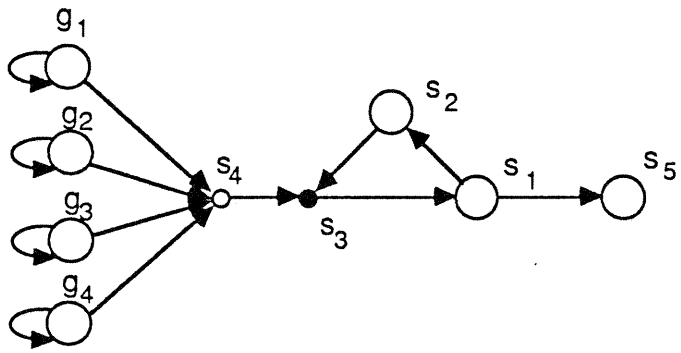


Figure 7: The Single-Gateway Model

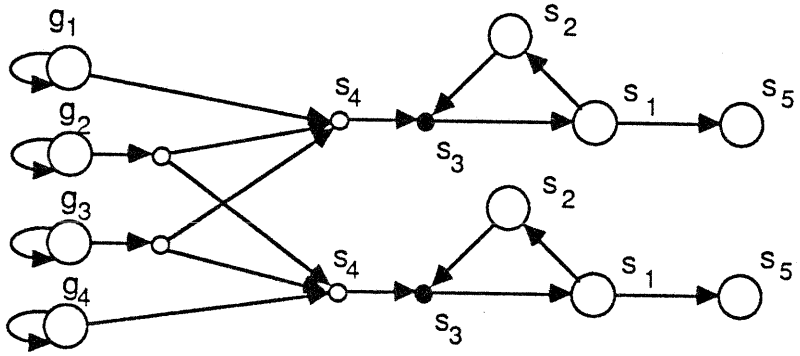


Figure 8: The Two-Gateway Configuration

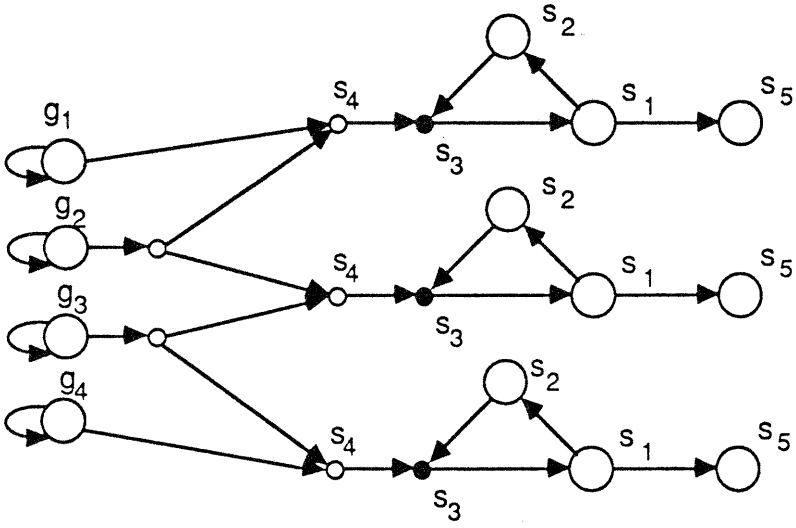


Figure 9: The Three-Gateway Configuration

6. REFERENCES

1. 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.
2. *CACI Simscript II.5 marketing information*, CACI Product Company, La Jolla, California, 1988.
3. G. Estrin, "A Methodology for Design of Digital Systems -- Supported by SARA at the Age of One", *AFIPS Conference Proceedings of the National Computer Conference 47* (1978), 313-324.
4. S. Iacobovici and C. Ng, "VLSI and System Performance Modeling", *IEEE Micro*, August 1987, 59-72.
5. *PAWS/GPSM marketing brochures*, Information Research Associates, Austin, TX, 1988.
6. D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel and H. Younger, "Distributed Simulation and the Time Warp Operating System", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, November 1987, 77-93.
7. B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer 18*, 8 (August 1985), 87-94.
8. J. Misra, "Distributed-Discrete Event Simulation", *ACM Computing Surveys 18*, 1 (March 1986), 39-65.
9. M. Moser, "GADD -- A Tool for Graphical Animated Design and Debuggin", *ICC '87 Conference Record*, 1987, 38.2.1-38.2.5.
10. K. M. Nichols and J. T. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer 21*, 5 (May 1988), 39-48.
11. J. D. Noe and G. J. Nutt, "Macro E-Nets for Representing Parallel Systems", *IEEE Transactions on Computers C-12*, 8 (August 1973), 718-727.
12. G. J. Nutt, "The Formulation and Application of Evaluation Nets", Ph.D dissertation, Computer Science Group, University of Washington, 1972.
13. G. J. Nutt and P. A. Ricci, "Quinault: An Office Environment Simulator", *IEEE Computer 14*, 5 (May 1981), 41-57.
14. G. J. Nutt, "A Flexible, Distributed Simulation System", *Tenth International Conference on Application and Theory of Petri Nets*, Bonn, West Germany, June 1989.
15. G. J. Nutt, "A Formal Model for Interactive Simulation Systems", Technical Report No. CU-CS-410-88, Department of Computer Science - University of Colorado, Boulder, September 1988 (Revised May 1989).
16. 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.
17. D. Redmiles, "Vesuvius Editor for Olympus (working title)", University of Colorado Department of Computer Science technical report (in preparation), 1988.
18. C. J. C. Schauble, "A Memory Access Simulator for MIMD Machines", University of Colorado, Department of Computer Science, PhD proposal, April 1989.
19. 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.
20. "SunView Programmer's Guide", Document Number 800-1324-03, Sun Microsystems, Inc., February 1986.
21. "Networking on the Sun Workstation", Document Number 800-1345-10, Sun Microsystems, Inc., September 1986.
22. *NeWS: A Definitive Approach to Window Systems*, Sun Microsystems, Inc., 1987.

AUTHORS' BIOGRAPHIES

GARY J. NUTT is a Professor of Computer Science at the University of Colorado. He received the Ph.D in Computer Science from the University of Washington in 1972, and has held research positions at Xerox PARC and Bell Labs, and managment positions at NBI and Interactive Systems. His current research interests focus on modeling and performance for distributed systems.

ADAM BEGUELIN is a Ph.D. candidate in the Department of Computer Science at the University of Colorado. He has a B.S. degree with highest honors in Mathematics and Computer Science from Emory University in 1985 and an M.S. degree in Computer Science from the University of Colorado in 1988. His research interests include parallel processing, programming languages,

distributed systems, human computer interfaces, and graphics.

ISABELLE DEMEURE is a postdoctoral researcher in the Department of Computer Science at the University of Colorado. She received a Maîtrise de Mathématiques from University Paris VI, France, in 1981, a Diplôme d'Ingénieur from the Ecole Nationale Supérieure des Télécommunications de Paris, France, in 1983. She worked as a systems and networks software engineer at SESA, in France, from 1983 to 1985. She was granted the Ph.D degree in Computer Science from the University of Colorado in 1989. Her research areas include distributed computations, modeling, and software engineering.

STEPHEN ELLIOTT is a software consultant. He has a B.A. degree in Music from Denver University in 1982 and an M.S. degree in Computer Science at the University of Colorado in 1984. His research interests include programming languages, networks, and software to support music composition.

JEFF McWHIRTER is a graduate student in the Department of Computer Science at the University of Colorado. He has a B.S. degree from Oakland University in 1986. His research interests are in distributed simulation and graph models of computation.

BRUCE SANDERS is a Professional Researcher in the Department of Computer Science at the University of Colorado. For the past two years, he has also held a teaching appointment in the Department, developing and teaching a senior-level software engineering projects course. He received his M.S. in Computer Science from the University of Colorado in 1978. Prior to joining the University, he was a Member of Technical Staff at Bell Laboratories and has held software engineering and management positions at NBI and Integrated Solutions. His interests include graphics, user interfaces, operating systems, modeling, and software engineering.

All authors' address is:

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430
(303) 492-7581

DISTRIBUTED SIMULATION DESIGN ALTERNATIVES

Gary J. Nutt†
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

ABSTRACT

Distributed simulation is a particular instance of distributed computation. Each distributed simulation program is composed of a variable part to describe the specific model, and a fixed part to implement the simulation environment, e.g., event ordering and simulated time. To exploit parallelism, the simulation must be implemented such that the variable part of the simulation maps onto the concurrent facilities of the fixed part of the simulation and onto the computer system that support the simulation. In this paper, we survey several factors that must be considered in order to exploit parallelism in the simulation.

INTRODUCTION

Distributed discrete event simulation is usually intended to satisfy one of two general criteria: The simulator is distributed across distinct locations since it must conform to some external constraints such as testbed modeling. Or, the simulator is distributed across multiple CPUs to decrease the amount of time required for the simulation to complete execution on some set of input data.

Testbed simulations require that the distributed components meet constraints similar to those found in traditional process control and real time systems. High-performance simulations are similar to general parallel programs such as scientific programs, thus there are far fewer constraints on the implementation but there is a focused goal of decreased run time (over serial implementations).

In this paper we address high-performance simulation applications. The goal is to consider techniques for constructing a simulation program so that it can be executed in a multicomputer environment with decreased execution time compared to a corresponding single computer environment.

BACKGROUND

Speedup

Distributed software performance may be measured by the ratio of the time to execute a computation in a serial environment to the time to execute the same computation (after partitioning) in a distributed environment. This *speedup* ordinarily has an upper bound of N for distributed systems capable of supporting N individual serial computations (*processes*). Empirical evidence often shows that the speedup for a particular implementation -- partitioning strategy -- may be small, even for relatively large values of N, (see [5, 14] for distributed simulation examples).

In some cases, the limiting factor in speedup is inherent in the algorithm itself, i.e., any partitioning of the computation results in the processes being strictly ordered. On the other hand, an algorithm may theoretically allow for N-way parallelism, i.e., the computation is partitioned into N processes, each of which may be constantly active on useful computation. A particular implementation of the algorithm is likely to introduce *management overhead* (such as scheduling and context switching), and *synchronization delays* (one process may be delayed waiting for information from another). Synchronization delays can lead to dramatic decreases in the speedup, since they are determined by complex factors related to the process architecture and distribution of function.

Partitioning Issues

The partitioning of a computation determines both the management and synchronization aspects of the performance. Important factors involved in choosing a partition are: The "amount of computation" per synchronization (its *granularity*), the nature of the communication and synchronization (IPC) mechanism, and the identification of specific function to place in a process, e.g., partitioning by data or functional criteria.

Granularity. Granularity is the ratio of the amount of computation that a process can accomplish without synchronizing with some other process. A large-grained process has relatively infrequent synchronization operations with other processes, while a fine-grained process is constantly synchronizing.

Synchronization. Synchronization is accomplished by exchanging data among processes. The underlying computer architecture that supports the computation will have considerable influence on the nature of the synchronizing mechanism. Shared memory multiprocessors provide hardware that will allow N CPUs to read and write the same physical memory. Distributed memory systems (networks of computers) do not support such access, instead, allowing information to be shared through (generally much slower) explicit information exchange operations.

Partitioning Criteria. There are two general approaches to dividing the computation among a set of processes: *Data partitioning* refers to the approach in which computational functions are replicated in each process with a data stream passing through each process. *Functional*

† Supported by NSF Grant No. CCR-8802283, a grant from U S West, and the Center for Software Systems Science.

partitioning is the approach in which data passes from one process to another in order to complete a transaction, i.e., the functionality is distributed instead of the data.

Distributed simulation systems must adjust to these general criteria for distributed computations if they are to be efficient for different simulation jobs [2, 15].

Discrete Event Simulation

Discrete event simulators generally take the form shown in Figure 1: The target system is represented by a physical model, which is represented by a simulation application program. The simulation kernel interprets simulation applications; the kernel, in turn, is an application program on the host system. The simulation kernel is a host-specific application program that interprets arbitrary simulation applications. It provides an abstract machine environment which manages simulated time and schedules activity among the simulation application components (events or processes).

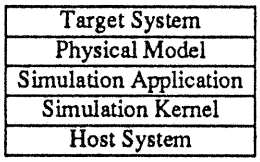


Figure 1: Simulation System Architecture

DATA-PARTITIONED DISTRIBUTED SIMULATION

A distributed simulation is a distributed computation of the simulation application. With the usual meaning, the distributed simulation operates on a parallel or distributed host system. Because the application and the host system are distributed, the kernel is also distributed, possibly using different criteria than that used in distributing the application.

Each of the simulation application, the simulation kernel, and the host system are an important factor in the ultimate performance of the simulation program. One can form a general characterization of the approach by considering the distribution strategies used in the simulation application, kernel, and host system layers.

For example, the Chandy-Misra technique [9] implemented in a message-based host can be characterized as shown in Table 1. The application represents the target system as a functionally-partitioned model where each process has arbitrary granularity. Logical processes synchronize with timestamped messages, using the conservative approach (no logical process can proceed until it is certain that no earlier event can occur).

Layer	Granularity	Synchronization	Partition Criteria
Applic	arbitrary	timestamped messages	functional
Kernel	large	conservative	data
Host	large	messages	-

Table 1: Distributed Memory Chandy-Misra Strategy

Each logical task (application portion and kernel portion) is a full simulation, including a local simulated clock; thus the kernel is replicated at each host node. While the simulator is data partitioned, the simulation applications are functionally-distributed across logical tasks using the physical task partition.

Since applications may have arbitrary granularity, fine-grained partitions are acceptable, even though they are likely to introduce large amounts of management overhead, (since the fixed-cost underlying machinery is exercised on a per task basis). A shared memory implementation can support finer-grained processes than distributed memory systems implementations, since the management overhead is smaller. Empirical evidence suggests that relatively fine-grained applications such as queueing network simulations will suffer, since the distribution strategy is a mismatch between the application and the host system, e.g., see [14].

The Time Warp technique [8] shares many similarities with the Chandy-Misra technique, with its primary differences being in the synchronization strategy in the kernel. Table 2 represents one view of Time Warp as implemented on a distributed memory machine. Again, there will be a performance problems with simulation applications made up of fine-grained tasks.

Layer	Granularity	Synchronization	Partition Criteria
Applic	arbitrary	timestamped messages	functional
Kernel	large	optimistic	data
Host	medium-large	routed msgs	-

Table 2: The Time Warp Distribution Strategy

While it is useful to compare situations in which one of these two techniques -- or some compromise between them [1] -- is better than the other, the differences are not likely to be wide for some classes of simulations, since both techniques use a similar design for the three-layer abstract machine. One could expect the best performance from this class of simulation implementations in cases where the application is constructed as a set of large-grained, message-passing, functional logical tasks in which there is relative balance among the amount of computation among the various logical tasks so that synchronization delays (simulated time synchronization) is minimized.

Extending Data-Partitioned Kernels

Speedup degradation due to synchronization manifests itself as blocked processes in the conservative data-partitioned kernels, and as repeated rollback in optimistic data-partitioned kernels. The synchronization delays occur due to unbalanced computational loads among the logical tasks. Consider a situation in which task A receives synchronization messages from tasks B and C. If B is a fine-grained task and C is a relatively large-grained task, then A will run at the rate of C with either conservative or optimistic techniques.

The problem is that the work for the host processors is not balanced, forcing the synchronization mechanism to slow

a simulation down to the speed of the slowest simulation component. This problem is likely to lead to the very large degradation in speedup.

The alternatives all amount to balancing the load on the processors. In the techniques discussed above, this requires that the simulation programmer be aware of the problem and consequently adjust the run-time loads by changing the granularity of the logical tasks. That is, the simulation application must conform to the host platform. This, in turn, will likely change the criteria for mapping physical to logical tasks.

In some cases, it is possible to statically analyze the workload for the logical tasks and to construct a new mapping which attempts to optimize the granularity on the basis of the analysis. Alternatively, the simulation kernel could be constructed so that adapted to the workload on the various host processors.

Static Analysis. There are a number of static analysis techniques that can be used to accomplish some form of recognition of parallelism, including simulation (e.g., see [3]. Such analyses can assist the simulation programmer in defining the application partition. However, it is important to recognize the difference between executing logical tasks and executing physical tasks in this analysis. Wagner and Lazowska point out that in a single CPU, multiple disk queueing network model that while the time to execute a time slice on the CPU is much smaller than the time to read or write a block on the disk (thus allowing the single CPU to service many jobs that use different disks), the logical tasks that simulate the CPU and each disk all take about the same amount of time in the logical task implementation [16]. This implies that there will be a limit to the realizable parallelism since the synchronization delays among the CPU logical task and the disk logical tasks will become a limiting factor.

The static analysis approach is a tool to assist the simulation programmer in choosing his own task definition. While it can result in significantly better performance than simulation applications that are constructed strictly from the physical process model, it will not address transient behavior within a simulation.

Adaptive Techniques. Adaptive techniques are similar to adaptive load balancing techniques. The simulation kernel monitors the performance of the set of logical tasks, then adjusts the workload as a function of these observations, e.g., see [10].

To accomplish adaptive partitioning, it is necessary to refine each logical task into a set of logical subtasks, each of finer granularity. The initial logical task is, thus, a collection of logical subtasks related by precedence (that can be mapped into messages). The system is loaded with the initial logical tasks and allowed to execute. If a processor supporting a particular logical task exceeds some threshold of inactivity, then it will adapt to the simulation application behavior by migrating subtasks from predecessor logical tasks (busy processors should not worry about load balancing, since they are already saturated).

For example, if tasks B and C are predecessors of task A, and if task A exceeds the idle threshold, then it could attempt to do some of the work of either task B or task C.

FUNCTIONAL-PARTITIONED KERNELS

A functionally-partitioned simulation kernel divides the kernel into subtasks and then distributes their execution over the host system. Each node in the host system is intended to support part of the simulation of more than one logical task, while other nodes support the remaining parts of the kernel's work.

The Olympus System [12, 13] (and the predecessor Quinault system [11]) are functionally distributed kernels, whose operation is summarized in Table 3.

Layer	Granularity	Synchronization	Partition Criteria
Applic	arbitrary	precedence	functional
Kernel	large	precedence	functional
Host	-	-	-

Table 3: The Olympus Distribution Strategy

An Olympus simulation application is expressed as an interpreted, directed graph similar to a predicate-transition (Petri) net [6]. Each node in the graph has firing rules that dictate when it should be activated, and an interpretation that defines the simulation processing that should take place when the node fires.

For example, Figure 2 is a model of a simple system with two customers and one server. The server is represented by tasks s_1, s_2, and s_3. Task s_2 represents the case that the server is idle; at initialization, this task contains a token. Task s_3 is an AND-task which fires only when there is a token on arc (s_2, s_3) and another on arc (s_2, s_3). Whenever a token resides on task s_1, then the server is busy.

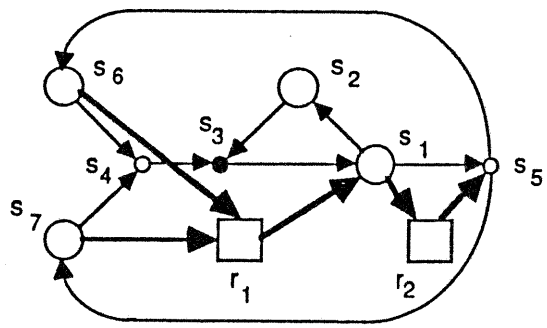


Figure 2: A 2-Customer Server

Tasks s_4 through s_7 model the customer requests for service. When a token is on s_6, then this represents the case when the first customer is "thinking" and does not require the service; s_7 represents a similar state for the second customer.

Tasks s_4 and s_5 multiplex tokens (representing requests for service) into and out of the server.

In order to ensure that s_4 has enough information to demultiplex a token to the correct "thinking" task, it reads information from repository r_2 (placed there by s_1) to identify which customer was just serviced. Repository r_1 is used in a similar manner to provide the server busy task (s_1) with the corresponding information.

Each task may have a procedural *interpretation*, to specify the amount of time required for firing the task, and for performing miscellaneous simulation tasks. Thus, task s_1 can infer the desired amount of service time and manage the customer identity by evaluating a procedure similar to:

```
s1()
{
        struct *request;

        request = read_repository(r1);
        wait(request.service_time);
        write_repository(r2, request);
}
```

The interpretation is evaluated each time the task is fired. Tasks with OR (output) logic can use interpretations to specify deterministic behavior; the procedure evaluates information available to it (from repositories), then selects an output arc to receive the resulting token. For example, s_5 might look like:

```
s5()
{
        struct *request;

        request = read_repository(r2);
        if (request.customer == 'first')
                route(s6)
        else
                route(s7);
}
```

A more complete description of the modeling language (including a refinement of this model) appears in [13].

The IRA PAWS system [4], and others [7], are based on a similar approach for defining the simulation application.

The kernel is decomposed into processes for *precedence management*, *node interpretation*, *model storage*, and *marking storage*. The precedence management process interprets the token flow through the graph, deciding which nodes in a graph are enabled, invoking a task interpreter process to represent firing, and routing tokens as a result of firing. A task interpreter evaluates an interpretation, upon demand by the precedence manager. There may be several task interpretations associated with a precedence manager, corresponding to the amount of concurrent activity in the graph as determined by that precedence manager. It is also possible for multiple precedence managers to exist at any given time, each precedence manager handling some portion of the graph, i.e., a data partitioning strategy is used across the precedence manager functions of the simulator kernel. The two storage processes are used to keep a copy of the model definition (the model storage) and the current state of the graph interpretation, i.e., the distribution of tokens on arcs and nodes in the graph.

This approach uses the same idea as data-partitioned kernels for managing the clock, i.e., within the subgraph assigned to a precedence manager, time is managed by precedence constraints (represented by control tokens in the graph). Conservative or optimistic techniques can be used to synchronize the precedence manager portions of the kernel; our current prototype uses a conservative approach.

A model may be "loaded" on the simulator by assigning sets of logical task nodes from the graph model to processors in the host system. The precedence management portion of the kernel is replicated at each host processor node. Each precedence manager determines the model definition from the model storage, and the current marking from the marking storage, then schedules task interpretation as appropriate. Thus if the precedence manager is replicated at some number, say n, processors, and the i^{th} precedence manager will distribute task interpretation across k_i additional processors, resulting in a partition with from n to $\sum_{i=1}^{n} k_i$ concurrent processes.

This will be effective only when there is a nontrivial amount of computation associated with an interpretation, and when several logical tasks are managed by a single precedence manager. If either of these conditions fail, then the implementation will incur excessive management overhead.

The solution also requires careful assignment of graph nodes to processors, just as was the case for data-partitioned kernels. If the simulation displays transient behavior, then adaptive techniques can be applied.

Adaptive Distribution

A precedence manager is initially assigned a set of logical task nodes, but may be allowed to adjust the set based on behavior of the simulation. If a processor detects excessive idle (or ineffective) time, then it should attempt to absorb more of the work by taking responsibility for interpreting additional nodes in the graph.

This can be achieved by requiring that the model not change during the simulation, and by replicating the model at each host node. Thus, simulation task load balancing is achieved without task loading, but by adjusting a global task assignment table through an appropriate protocol.

Thread Distribution

As an alternative to conventional adaptive load balancing, if the entire graph model is available at each host node processor, the assignment of logical task nodes from the graph can be accomplished by allowing the precedence manager to move through the graph as a thread of control. The effect is that a data-partition application is executed on a functionally-distributed kernel -- a precedence manager and a task interpreter. Since the precedence manager manages only a single task interpreter, it would ordinarily be combined with the task interpreter host process.

The difficulty with this approach occurs when the simulation program initiates simultaneity or when concurrent

threads converge. The concurrency initiation suggests that the host system should supply an additional host process to support a new thread of control. When threads converge, then precedence managers must be designed to merge the multiple threads back to a single thread in an efficient manner.

CONCLUSIONS

Distributed simulation is an important application for distributed and parallel computer systems. We have seen the emergence of sound technology for implementing the simulation kernel for certain classes of simulation applications, but which can be very inefficient for cases in which there is little inherent parallelism in the application or in cases where the application partition is not well-matched to the kernel or the host system platform.

There are a number of techniques that can be applied to distributed simulation in addition to the well-known static data partitioning techniques. First, since we expect complex simulation applications to display variable behavior, depending on the activity in the model, we expect that adaptive techniques will be mandatory for distributed simulation to become a viable production tool. Secondly, complex simulation applications are likely to have nontrivial computation associated with events (or logical processes), thus we expect that functionally-partitioned simulation kernels will contribute to the speedup of model execution for this class of applications. Our initial experiments with functionally-partitioned kernels show promise for this class of problems although we are still relatively early in our experiments.

Just as there is a need for better application programming tools for general distributed computations, there is a need for better simulation tools for distributed simulation systems. There is considerable work to be done in order to find ways for the simulation application writer to be able to make efficient use of the underlying platforms.

ACKNOWLEDGEMENTS

Jeff McWhirter has been a significant contributor to the distributed simulation aspects of Olympus.

This work has been supported by NSF Grant No. CCR-8802283, a grant from U S West, and the Center for Software Systems Science.

REFERENCES

1. Y. Aahlad and J. C. Browne, "Balanced Sequencing Protocols", *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 58-63.

2. D. Baezner, J. Cleary, G. Lomow and B. W. Unger, "Algorithmic Optimizations fo Simulations on Time Warp", *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 73-78.

3. A. M. Baum and D. J. McMillan, "Automated Parallelization of Serial Simulations for Hypercube Parallel Processors", *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 131-136.

4. 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.

5. R. M. Fujimoto, "Performance Measurements of Distributed Simulation Strategies", *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 14-20.

6. H. J. Genrich, "Predicate/Transition Nets", in *Petri Nets: Control Models and Their Properties, Advances in Petri Nets 1986, Part 1*, W. Brauer, W. Reisig and G. Rozenberg (editor), Lecture Notes in Computer Science, Springer Verlag, Berlin, Heidelberg, New York, 1987.

7. T. Y. Hou and M. Y. Chiu, "A Hybrid Model for Distributed and Concurrent Simulation", *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 21-24.

8. D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel and H. Younger, "Distributed Simulation and the Time Warp Operating System", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, Austin, Texas, November 1987, 77-93.

9. J. Misra, "Distributed-Discrete Event Simulation", *ACM Computing Surveys 18*, 1 (March 1986), 39-65.

10. D. M. Nicol, "Dynamic Remapping of Parallel Time-Stepped Simulations", *Proceedings of the SCS Multiconference on Distributed Simulation 21*, 2 (March 1989), 121-125.

11. G. J. Nutt, "An Experimental Distributed Modeling System", *ACM Transactions on Office Information Systems 1*, 2 (April 1983), 117-142.

12. G. J. Nutt, "A Flexible, Distributed Simulation System", *Tenth International Conference on Application and Theory of Petri Nets*, Bonn, West Germany, June 1989.

13. G. J. Nutt, A. Beguelin, I. Demeure, S. Elliott, J. McWhirter and B. Sanders, "Olympus: An Interactive Simulation System", *Proceedings of the 1989 Winter Simulation Conference*, Washington, D. C., December 1989.

14. D. A. Reed, A. D. Malony and B. D. McCredie, "Parallel Discrete Event Simulation Using Shared Memory", *IEEE Transactions of Software Engineering 14*, 4 (April 1988), 541-553.

15. S. V. Sheppard, C. K. Davis and U. Chandra, "Parallel Simulation Environments for Multiprocessor Architectures", *Proceedings of the SCS Multiconference on Distributed Simulation 19*, 3 (July 1988), 109-114.

16. D. B. Wagner and E. D. Lazowska, "Parallel Simulation of Queueing Networks: Limitations and Potentials", *ACM SIGMETRICS Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, May 1989, 146-155 .