OLYMPUS USER'S MANUAL
Version 0.5

Gary J. Nutt
Adam Beguelin
Isabelle Demeure
Stephen Elliott
Jeff McWhirter
Bruce Sanders

CU-CS-382-87 December 1987
(revised June, 1989)

Department of Computer Science
Campus Box 430
University of Colorado,
Boulder, Colorado, 80309 USA

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

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

TABLE OF CONTENTS

1. INTRODUCTION

Description of the Olympus strawberry from *Sunset New Western Garden Book* [9], page 475.

> *Average-sized light red fruit in midseason. No runners; vigorous plants bear on branching crowns.*

Olympus is a distributed system for constructing, editing, and exercising *Bilogic Precedence Graph* (BPG) models. It is implemented on Sun bitmap graphics workstation, where some of the user interface's use a point-and-select interface. Olympus is intended to support a designer who wishes to construct BPG models, observe their qualitative, dynamic behavior, and to subsequently gather quantitative data about the operation of the models. Olympus supports *BPG graph editing*, *BPG animation*, and *BPG simulation* in a single, integrated environment. Reference [2] provides a summary overview of BPGs, the Olympus architecture, and the use of the system for constructing simulation programs.

1.1. The Operating Environment

Olympus Version 0.5 is implemented on Sun workstations, using either the SunView window and graphics environment [5], the Sun NeWS window and graphics environment [8], or a teletype-style interface. All development and distribution of Version 0.5 has been done on Sun 3 (SunOS 3.3) workstations; the same software may or may not be current in Sun 4 (SunOS 4.0) environments.

Olympus is ordinarily executed within a directory named *strawberry*, containing the binary files named *bin/olympusServer* and at least one of the three frontend files *bin/svEditor*, *bin/newsEditor*, or *bin/ttyConsole* for the SunView, NeWS, or TTY frontends, respectively. (These files are distributed on a **tar** tape, as described in Appendix A.) The binary file must have execute permission. No other files are required to use Olympus (other than various Sun run-time routines, some of which may be included in the strawberry/bin directory); the base directory will also ordinarily contain a few demonstration models in strawberry/models, e.g. demo.bpg, demo.c, demo.h, demo, etc.

1.2. BPG Models

BPG models are described in more detail elsewhere [4]. Briefly, BPGs are composed from a set of *tasks* required to implement a system, a set of *control dependencies* among the tasks, and 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.

The control flow subgraph differs from some other precedence graphs in that it specifies conjunctive ("AND") and disjunctive ("OR") input and output logic for each task. In Olympus, AND tasks are represented by small, filled circles, OR tasks are represented by small, open

circles, and tasks with a single input and single output edge are represented by large, open circles. representation of a BPG.

Each task in the model can be thought of as a definition which becomes active when control is given to the task. Tokens, similar to those used in Petri nets, are used to indicate that a particular task is active. Thus, in the language of Petri nets (and other program schemata), one can think of markings and firings of the various tasks in the graph. A BPG is activated by marking appropriate tasks, at which time the BPG firing rules (control flow logic rules) describe sequences of markings corresponding to control flow among the tasks. Thus, while tasks share the firing rule properties of pure Petri net transitions, they also employ the notion of non-zero transition times.

Each task may have an *interpretation,* meaning that whenever the task fires, the interpretation for the task is evaluated. In the simplest case, an interpretation is a specification of the amount of time required to execute the corresponding tasks in the target system, expressed as a distribution function with suitable parameters. Olympus also supports interpretations defined as arbitrary procedures written to be compatible with Sun's RPC protocol [6].

To model the action of tasks with outgoing OR logic, each arc emanating from the task may have an associated probability; the sum of the probabilities is unity. The probability represents the relative frequency with which control will traverse that particular arc when the OR task terminates. The procedural interpretation may also be used to deterministically choose output arcs.

Single-input, single-output tasks are hierarchical. These ideas for hierarchical refinement are experimental, and will change with newer versions of the model and the system.

Data flow in a BPG is represented in two ways: First, tokens can carry data, and second one can add data repository nodes to the control flow graph. Arcs in the data flow graph represent possible references to the corresponding data repository by the relevant task, depending upon the task interpretation.

1.3. The Architecture and Implementation

Olympus is an experimental implementation of an interactive, distributed modeling system. We have used it as a vehicle to experiment with various approaches to building interactive software development systems. The system has been described in detail in a conference paper [3] and a technical report [1]. Here, we summarize the architecture and implementation of the system in a Sun workstation network environment.

Olympus is implemented as a *frontend client* and a *backend server*. The frontend and the backend are viewed as distinct (families of) processes that communicate with atomic transaction messages. The frontend client has been implemented in a number of ways including in the Symbolics Lisp environment as well as the SunView, NeWS, and TTY package implmentations described below.

The backend is a BPG storage and interpretation system. It reacts to messages from the frontend, e.g., messages to load a file containing a model, to store a part of a BPG in the open model, to interpret a BPG, etc. Whenever the backend completes a transaction, it notifies the frontend to that effect.

The storage part of the backend is a straightforward implementation on top of the UNIX file system. BPG semantics are encapsulated in file access routines in the storage subsystem.

BPG interpretation state is represented by the distribution of tokens on the BPG -- called the *marking* of the BPG. The backend has another storage module, distinct from the storage module, to save the interpretation state of the model. The control flow interpreter reads the interpretation state from the marking storage, enables tasks, and then fires those tasks that are enabled. Firing a task means to execute the corresponding interpretation.

To implement the task interpreter, it is necessary for the backend to be able to execute procedures specified in some programming language. Task interpretations are implemented on top of Sun's remote procedure call (RPC) facility. That is, an Olympus user can write a conventional C procedure (or procedure that is C compatible in any supported language), then pass the name of the procedure (and the host on which it is located) to the task interpreter. The task interpreter will bind the task interpretation into the task interpreter system using the RPC mechanism; as a result, user-defined C task interpretations are executed under the control of the task interpreter, i.e., when the corresponding task fires in the BPG.

2. SERVER OPERATION

2.1. Running the Server

The server is a C/UNIX program that is executed on a Sun workstation or server machine. The server communicates with the frontend and various task interpreters using Berkeley sockets. The server is executed like any conventional UNIX process, although it does not use *stdin* nor *stdout*. Ordinarily, one would run the server as a background process with a command such as:

```
% olympusServer &
%
```

from a standard shell.

2.2. Task Time Distributions

Task time distributions are specified as distribution funtions. Whenever the time for the task to execute is to be determined by stochastic means, the server will sample one of the distributions specified in Table 1. No special action is required for the server to execute task time distribution interpretations.

Constant(value)
: This distribution always returns *value* whenever it is called.

Uniform(low, high)
: This distribution returns an integer value between the values *low* and *high*. The random variable is uniformly distributed.

Normal(mean, variance)
: Sampling this distribution will return a random number from the normal distribution with a mean of *mean* and variance *variance*.

Exponential(mean)
: Sampling this distribution will return a random number from the exponential distribution with a mean value of *mean*.

Hyperexponential(parameter1, parameter2)
: This distribution function returns an integer valued from the hyperexponential distribution with parameters *parameter1* and *parameter2*.

Erlang(mean, variance)
: This distribution function returns an integer varialbe from the Erlang-k distribution with a mean value of *mean* and a variance of *variance*.

Table 1: Time Specifications

2.3. Task Procedure Interpretations

The node interpretation facility allows the user to write procedural code in a high-level language, which is associated with the nodes of the BPG being modeled in Olympus. This code will be executed whenever a node in the BPG is fired, i.e. when a token arrives at the node. The facility further provides for the active use of data repositories for storing and communicating information between the various task nodes, as specified by the data flow subgraph of the BPG. Such data may be of arbitrary structure and type as specified by the user. Nodes may also communicate by placing data on the tokens themselves, rather than using data repositories. This capability can be used to treat the control flow graph of the BPG as a pure dataflow machine.

This Subsection describes the use of the node interpretation facility. Since the facility is based on the Sun RPC library the user may find it useful to be familiar with this library. If the node interpretations are run on different machines and machine independence is required for data passed between nodes (either via repositories or on tokens), then the user will also need to be familiar with the Sun XDR protocols. For more information on these, see [6]. As noted in the RPC and XDR documentation any language could be used, but this documentation assumes that the language in which node interpretations are written is C.

Adding the Procedure Reference to the BPG

Each task node may have an attribute specifying information about the procedural interpretation.† In the SunView editor, task node property sheet fields specify:

- Host name
- Procedure

and data repository node property sheet fields specify:

- Host name
- Input procedure
- Output procedure

For both types of nodes, the *Host name* field specifies the name of the host on which the procedural code for the node will be executed. This requires that the code be running on the named host (as described below under *Running Tasks Interpretations*). If no host is specified, the local host on which the server is running will be used.

For task nodes, the *Procedure* field specifies the name of the procedure to be executed when the node fires. This is the same as the name of a C function written in the interpretation code (see *Writing Node Interpretation Code* below).

For repository nodes, the *Input procedure* and *Output procedure* fields are analogous to the *Procedure* field for task nodes. Two procedures are required, one for handling input (write) to the repository and the other for handling output (read) from the repository. These fields normally do not need to be filled in, and if left blank will cause default repository routines to be used from the node interpretation library (see discussion below).

† This explanation assumes the SunView editor user interface, although a similar explanation applies to the NeWS editor. The TTY console does not support editing.

Writing Node Interpretation Code

Node interpretation code is conventional C code, entered into a file whose base name will normally (but need not) be the same as the name of the file in which the BPG model is stored. Thus, if the model is *mymodel.bpg* the associated code will be *mymodel.c*. This file may be created using any conventional editor.††

The first line of all interpretation files should be:

```
#include "mymodel.h"
```

where the name of the include file is *always* the same as the name of the BPG model (for reasons described under *Compiling Node Interpretations* below). This file is created by Olympus, and includes all the other header files needed. These are:

<rpc/rpc.h>	RPC, XDR header file
"libtcp.h"	Non-blocking, TCP-based RPC library
"libinterpret.h"	Olympus node interpretation library

The reason that it is most convenient to edit the node procedure for a given node while the property sheet is displayed is that the display of the property sheet also causes all arcs incident to the node to be numbered. The numbering gives unique identification to the input control arcs, output control arcs, and data arcs. These numbers are used, as described below, to relate the behavior of the code to the BPG with which it is associated. Thus, writing node interpretations is partly done textually, and partly done graphically.

Task Nodes

All task node procedures return a value of type **caddr_t**. They are called with a single argument of type **USER_INPUT ***. which is a pointer to an instance of the following structure, defined in "libinterpret.h":

```
typedef struct user_input {
        u_short input_token;
        u_int num_inputs;
        caddr_t *data;
} USER_INPUT;
```

The *input_token* is a number indicating the input arc (as numbered on the BPG while the property sheet is displayed) which contained the token that caused the node to fire. Note that this information is only useful in the case of disjunctive logic nodes. Conjunctive nodes receive the value zero in this field. The *data* field is an array of pointers to any data that may have arrived on input tokens. The data associated with the arc numbered *n* is pointed to by *data*[*n*]. Thus, for disjunctive nodes a pointer to the data for the (single) token that caused the node to fire can be found in *data*[*input_token*]. For conjunctive nodes, pointers to the data for all the input tokens are found in the *data* array locations corresponding to the numbers of the arcs on which the tokens arrived. The *num_inputs* field is the value of the maximum subscript containing

†† It may also be created from within the SunView editor while the property sheet for a node is being displayed. The *Edit* button in the property sheet may be used to create a window with an editor running in it for this purpose. The editor executed is determined by the EDITOR

meaningful data in the *data* array. Thus, a conjunctive node will receive data in subscripts 1 through *num_inputs* of the *data* array. If a token was not carrying data the corresponding pointer will be NULL. Thus, if the interpretation loops through the *data* array up to *num_inputs*, it should also check for NULL pointers before dereferencing to find the data.

Task nodes *must* call a library routine in order to return. This routine is:

```
task_return(arc_number, datap, size, time)
        u_short arc_number,
        caddr_t datap;
        u_int size;
        u_long time;
```

where *arc_number* is used to indicate the output arc which should receive the token (only relevant for disjunctive logic nodes), *datap* is a (possibly NULL) pointer to some data that should be returned on the token, *size* is the size in bytes of the data (which may be zero if the *datap* pointer is NULL), and *time* is the number of seconds that the model should consider the node interpretation to have taken. If this time is given as zero, Olympus will sample the node's time distribution function to determine the time required for the task. Fractional numbers of seconds for time are not currently supported.

Task nodes may access repositories to which they are connected in the BPG using a library routine:

```
repository_access(arc_number, data, size)
        u_short arc_number;
        caddr_t data;
        u_int size;
```

The *arc_number* specifies the data arc that connects the task node to the repository of interest. The system determines whether the access is a read or a write by the direction of the arc, and thus task nodes may not violate the data flow properties of the BPG when accessing repositories. When reading a repository, the *data* argument is the address of a pointer to storage where the data that is read is to be placed. If **data* is NULL, then the system will allocate the necessary space and modify **data* to point to this space. When reading, the *size* argument is not used and may be omitted. When writing a repository, the *data* argument is a pointer to the data to be written, and the *size* argument is the size in bytes of the data.

Repository Nodes

Repository nodes require two routines, one for input and the other for output. In many cases, the user only requires that a repository store any data written to it, and then return this data when it is read. Since this is a common usage, the node interpretation library provides two generic repository routines that accept and return data of arbitrary structure. These routines are used for all repositories for which the input and output routine names are left blank in the repository node property sheet.

---

environment variable, or defaults to the *ed* editor (as a least common denominator).

In certain specialized cases, the user may want to provide custom functionality for a repository. For example, the generic routines provide a model which is similar to the behavior of computer memory, i.e. writes are destructive while reads are not. Also, the generic routines do not distinguish between accesses via different data flow arcs incident to the repository. These properties can be changed by writing custom repository input and output procedures.

The type of a repository procedure, like other node procedures, is **caddr_t**. (Actually, repository input routines do not return a value, and would thus be more properly typed as **void ***. Making them **caddr_t** simplifies the implementation.) For both input and output procedures, the single argument is of type **REPOSITORY_ACCESS *** which is defined in "libinterpret.h":

```
typedef struct byte_array {
        u_int size;
        caddr_t bytes;
} BYTE_ARRAY;

typedef struct repository_access {
        u_long nodeID;
        u_short arc;
        BYTE_ARRAY *data;
} REPOSITORY_ACCESS;
```

The *nodeID* identifies the particular repository node which is involved in the transaction. This field is useful if the same routine is named in several nodes' property sheets, in order to keep the data for the different nodes separated. In particular, this is how the generic routines are able to be used for many nodes in a BPG; they keep an array of pointers to the data, indexed by node ID. The *arc* field identifies the arc (as numbered when the property sheet for the repository node is displayed) on which the request is being made. This allows the repository to behave differently depending on the task node making the access. The *data* field is a pointer to a BYTE_ARRAY structure, which contains the size in bytes of the data to be written into the repository and a pointer to the data itself; it is not meaningful in the case of the output procedure.

As noted, input procedures do not return a value. Output procedures return the address of a pointer to a BYTE_ARRAY structure containing the size of and pointer to the data being read from the repository. If this address is NULL, or the pointer at the address is NULL, it is taken to mean the repository contains no data, and the user will receive a non-zero error return from **repository_access()**.

Machine-independent Data

Version 0.5 treats all data (both for repositories and tokens) simply as a stream of bytes. If the user requires that data be transferred in a machine-independent way this must be handled in the node interpretation code. One way of doing this is to create a memory XDR stream and write XDR routines that describe the data. The interpretation code writes the data to the memory XDR stream, and then passes the address of the stream to either **task_return()** or **repository_access()**.

For details on using memory XDR streams, refer to the XDR manual.

Compiling Node Interpretations

Once the BPG has been defined and the node interpretation code is written, the code must be compiled and executed before it will be used when the BPG is interpreted. Before compiling, the BPG must be saved either by using the frontend "Save Model" command, or by exiting Olympus. This will create the header file discussed above, which is included in the node interpretation code file. The header file has the same name as the model, except with the suffix changed to ".h" or with ".h" appended. This is done because the Olympus system knows the name of the file in which the model is stored, and thus can create the name of the header file easily in this manner, and because this guarantees that if the name of the model file is unique, then so will be the name of the header file. Once this header file exists, the interpretation code may be compiled. The module must be linked with two libraries: the non-blocking RPC library **libtcp.a** and the node interpretation library **libinterpret.a**. The following Makefile will do the job, provided that the paths to the include file and library directories are properly defined.

```
INC = -Iinclude_file_directory
LIBINTERPRET = library_directory/libinterpret.a
LIBTCP = library_directory/libtcp.a

.c:
        $(CC) $(CFLAGS) $(INC) $(LDFLAGS) -o $@ $< $(LIBINTERPRET) $(LIBTCP)
```

Note that libinterpret.a must precede libtcp.a, since it uses the non-blocking RPC facilities.

Running Task Interpretations

Once compiled, the resulting executable file must be run on all machines which were named in node property sheets under the "Host name" field. Normally this will be done by typing

```
% mymodel &
%
```

so that the execution is placed in the background. When desired, the interpretation code is stopped by sending it a signal (normally SIGINT, although the library catches all catchable signals and will clean up properly and exit upon receipt of any such signal). Olympus behaves properly when a node interpretation process that has been in use (or is expected to be present) disappears or is not running. Also, when the user deletes the marking in Olympus, the node interpretations are essentially reset, so that any processing of node tasks that may be going on will stop. When the marking is deleted, any pending results from node procedures are also discarded.

2.4. Instrumentation

The simulator produces a report summarizing the occupancy and interarrival times of tokens on each task and control arc. It also summarizes the number of times that a read or write reference

is made to a repository.

The server will keep statistics on any arc or node at the request of a client program. The instrumentation includes the number of tokens that have passed through the resource, (the minimum, mean, and maximum occupancy time of the token; and interarrival time minimum, mean, and maximum).

The server will also reset the cumulative counts upon command from a client.

Table 2 is an example of a *.stat* file.

Task	No. of	------- Occupancy -------				-- Inter-arrival Time ---			
Label	Tokens	Total	Min	Mean	Max	Total	Min	Mean	Max
constant	20	40	2	2.00	2	38	2	2.00	2
expontl	20	42	0	2.10	8	38	2	2.00	2
erlang	20	57	1	2.85	4	37	0	1.95	6
hyperexp	20	49	0	2.45	9	37	0	1.95	7
normal	20	53	0	2.65	5	39	0	2.05	6
uniform	20	46	1	2.30	4	39	0	2.05	7

Arc	No. of	------- Occupancy -------				-- Inter-arrival Time ---			
Label	Tokens	Total	Min	Mean	Max	Total	Min	Mean	Max

Repository	No. of	No. of
Label	Reads	Writes

Table 2: Simulation Summary

3. THE TTY CONSOLE

3.1. Running the TTY Console

The TTY console is a simple frontend, intended to be used to put the server into operation with minimal interaction on the part of the user. This facility is executed from a UNIX shell by:

```
% ttyConsole
%
```

The program will respond with the banner

TTY Console Version 0.5 (*<date>*)

then wait for a command to be entered.

3.2. Console Commands

The TTY Console will only interpret a small number of commands to load the model and marking, to collect statistics, and to run/halt the interpretation. The specific commands supported by the TTY Console are shown in Table 3.

L - Load Model
M - Load Marking
D - Delete Marking
R - Run
S - Stop Run
W - Write Statistics
C - Clear Statistics
H - Help
Q - Quit

Table 3: TTY Console Command Summary

Load Model

The command prompts the user for a model file name of the form

model.bpg

This command causes the server to load the specified model into its virtual memory.

Load Marking

If the user has previously specified a marking for a BPG, then the marking can be saved, e.g., in a file named

model.mark

The *Load Marking* command causes the marking file for the specified model to be loaded into the server. Note that the model and marking files must be consistent.

Delete Marking

This command causes the current marking (stored in the server) to be removed, leaving the model in an unmarked state. This command can be used to reinitialize the model.

Run/Stop Run

This command causes the server to interpret the model and marking (as specified using the *Load Model/Marking* commands). The simulation can be terminated with the *Stop Run* command, or by the model/marking coming to a state in which no task is enabled.

Write Statistics

This command causes the server to write a UNIX file, usually with a name of the form

model.stat

with the statistics resulting from instrumentation (see above).

Clear Statistics

This command causes the server to clear its cumulative instrumented values.

Help This command prints the list of commands on stdout.

Quit This command terminates the TTY Console process.

4. THE SUNVIEW EDITOR

The SunView editor is an interactive, graphical program implemented in the SunView programming environment. We assume that the reader is familiar with Suntools operations in our discussion.

4.1. Running the SunView Editor

The SunView editor is started from the Suntools environment. Within a shelltool, type

```
% svEditor
%
```

or if it is desired to initiate the session with a particular file, type

```
% svEditor file_name.bpg
%
```

where **file_name** is the name of the file containing the BPG. The file name *must* be of the form:

model.bpg

Olympus will initialize itself, then place a new 1000 x 1000 pixel SunView window on the screen. The position of the new window is somewhat arbitrary, but the user can move the window using any of the standard Suntools operations, eg., by using the *move* option in the frame menu; or by simply placing the cursor in the banner of the window, depressing the center mouse button and moving the window. The new window will be labeled as

SunView Editor Version 0.5 (<*date*>)

If Olympus was started with a file name, then the corresponding BPG will be painted within the window; otherwise, the window will be blank.

4.2. The User Interface

Since this editor is constructed using the SunView library, many aspects of the user interface are determined by SunView and Suntools. The user should be familiar with this interface before using Olympus; see [7] for tutorial information about using Suntools. Window frame operations and scrolling are supported by SunView.

Within the Editor window, all operations are instigated with the mouse. Each button on the mouse is used for a different class of operations. The left button is used for *editing* operations, the middle button is used for *animation and simulation* control, and the right button is used for *managing the Olympus environment*, ie., setting parameters, etc.

The middle and right button operations invoke a popup menu. By depressing one of these two buttons, a menu will be displayed at the current location of the cursor. As long as the button is

kept down, the menu will be displayed. To select an entry from a menu, move the mouse over the desired selection on the menu while keeping the button depressed. Menu entries will highlight to illustrate which entry is pending; if the cursor is moved out of the menu area while the button is depressed, then no entry will be pending. To invoke an entry from the menu, move the cursor to the entry and release the button. This will cause Olympus to execute the command corresponding to the entry. (There is one menu that operates in a different manner, and it will be discussed in the specific context below.)

In several cases, Olympus will require information from the user before it can carry out a task. It will place a *popup dialog box* on the screen requesting additional information from the user. Fill-in the information on the dialog box and then select the **OK** or **Cancel** (or **Yes** or **No**) "button" in the popup dialog box (using the left button on the mouse). This will remove the popup box from the screen and return control to Olympus. The **Cancel** button allows the user to cancel the command, while the **OK** button will finish the operation with the supplied input data. Often, these dialog boxes will require the user to enter information into fields using the keyboard instead of the mouse.

The left button is ordinarily used to edit the BPG. The top of the Olympus window -- under the banner -- contains eleven small *panels* with silhouettes. These panels are used to choose editing commands as discussed below. To select a command, move the cursor to the appropriate panel and stroke the left mouse button. When the cursor is moved back into the large portion of the window (containing the image of the BPG), the cursor will change shape to indicate that Olympus is in a mode to edit the BPG.

4.3. Editing Functions

Drawing

The SunView Editor is a point-and-select editor. Objects are placed on the *canvas* by selecting an object from the object pallet, then placing the object on the canvas. Other editing operations are also invoked using the pallet-canvas model.

Drawing Nodes

To draw a node, select one of the six node objects from the pallet: Large, open circle; small, filled circle; small, open circle; inverted triange ("start"); triangle (stop); or square. Move the cursor to the canvas. When the cursor is place at the point where the object is to be deposited, stroke the left mouse button.

All objects are placed on the canvas using an invisible grid (see the **Grid** command below). Therefore, objects may be placed on the canvas in a slightly different place than that addressed by the cursor.

Drawing Arcs

To draw an arc between two objects, select the arc from the pallet, then select a tail object, followed by a head object, both using the left button on the mouse. An arc with a single *joint* in it may be constructed by pointing at the tail, then a space that is not "near" any

object to identify a location for the joint in the arc, then at the object at the head of the arc. The Editor will automatically draw normal width lines to represent control flow, and thick lines to represent data flow.

Changing the Model

Two of the panels on the pallet contain *slashed circles*; a slashed circle over an AND node represents the *erase node* operation. After selecting this operation from the pallet, the user is expected to select a node object with the left mouse button; the object will be removed from the model. A slashed circle over an arc is used to remove arcs; after selecting the operation from the pallet, select an object on the tail of the arc followed by an object at the head of the arc. The arc will be removed from the model.

The SunView Version 0.5 editor makes no provision for moving objects (but see the NeWS version, below). This must be done by erasing the object, then redrawing it at a new location.

Property Sheets

A property sheet is popup dialog box. There are three types of property sheets: A task property sheet, a data repository property sheet, and an arc property sheet.

Task Property Sheets

The task property sheet has editable fields to specify:

Label. An annotation that will appear on the visible graph.

Description. An annotation that appears only in the property sheet, i.e., that is stored with the model for the user's purposes.

Time. The task time distribution (see above). Allowable values for the field are *Constant*, *Uniform*, *Normal*, *Exponential*, *Hyperexponential*, and *Erlang*. Parameters for the distribution are included in parenthesis following the distribution name; all parameters must be integers. For example, an constant distribution with a mean of 25 is specified as

Constant(25)

Host name. The host that will execute a task procedural interpretation for the node (if one is specified in the following field). If the field is left blank, then the interpretation is assumed to be done on the same host machine that supports the server, but in a distinct process.

Procedure. Name of the procedure that provides the interpretation for the given task.

The task property sheet uses **OK** to commit the edited contents of the fields or **Cancel** to abort the field editing. The **Edit** button is used to start an editor in a distinct window (to write interpretations for the task -- see above).

Repository Property Sheets

The data repository property sheet is similar to the task property sheet:

Label. An annotation that will appear on the visible graph.

Description. An annotation that appears only in the property sheet, i.e., that is stored with the model for the user's purposes.

Time. Unused

Host name. The host that will execute a repository input/output interpretation for the node (if they are specified in the following fields). If the field is left blank, then the interpretation is assumed to be done on the same host machine that supports the server, but in a distinct process.

Input procedure. Name of the procedure to be used for repository input operations

Output procedure. Name of the procedure to be used for repository output operations

The **OK**, **Cancel**, and **Edit** buttons behave as for task interpretations.

Arc Property Sheets

The arc property sheet is used to label arcs, and to specify probabilities for arcs emanating from an OR node.

Label. An annotation that will appear on the visible graph.

Probability. An integer annotation between 0 and 99 representing the probability (out of 100) that a particular arc will receive the output token from the OR task. Although Version 0.5 does not check, the sum of probabilities should be 100.

The **OK** and **Cancel** buttons behave as for task interpretations.

4.4. Interpretation Functions

The interpretation functions are invoked with the middle mouse button by selecting an entry from the popup menu.

Mark Node

After selecting this function from the menu, the cursor changes shape to indicate that the user must next select a task on which a token should be placed. The task is selected by moving the cursor to the desired task and stroking the left mouse button.

Step One animation/simulation event will occur (in real time), then the animation will be halted, and the user may remark the graph, **Step** it through another event, or perform any other command. (Editing commands will result in unpredictable results; before editing, the user

should select the **Animate** command to allow the model to complete prior to changing the model.) The events that Olympus interprets essentially amount to the initiation or termination of a task firing; thus a step may produce delayed activity on the screen or it may not produce any visible activity. The first case is a task initiation, where the delay is the amount of time required for the task to be interpreted. The second case is a task termination event.

Animate

This command causes Olympus to fire tasks in the BPG as long as possible, based on the extant token marking. That is, markings schedule events to occur in the future; the animation will continue in real time as long as there are future events scheduled. When no additional events are scheduled, then a popup box is placed on the screen indicating that **There are no more activities to fire.** Olympus will not proceed until the user signals **OK** to the message.

Simulate

This command causes Olympus to interpret the model as rapidly as possible, and to gather statistics about the behavior of the model under the specified marking. (Before the model can be simulated, it must be marked.) The simulator performs the same logical token movement as the animator; when it completes the simulation, ie., there are no more events scheduled to execute, then it places the termination popup dialog box on the screen indicating that **There are no more activities to fire.**

Stop This command halts the interpretation that is currently in progress; if the server is not interpreting, then the command has no effect.

Delete Marking

This command clears the current marking, leaving the model unmarked.

Instrument Node

This command enables instrumentation for the selected node.

Instrument Arc

This command enables instrumentation for the selected arc.

Delete Instruments

This command disables all instrumentation settings, leaving the model uninstrumented.

Write Statistics

This command causes the server to write its cumulative instrumentation data to the named file.

Clear Statistics

This command clears the cumulative counters for each instrumented part of the model.

4.5. Environment Functions

The functions on the right mouse button menu are used to control the model parameters. This includes file manipulation, display control, producing Postscript files appropriate for printing on a Postscript printer, and terminating an Olympus session.

When the SunView Editor is started, a file may be specified on the command line, as explained earlier. This causes Olympus to open the named file, and to read it into primary memory. Alternatively, a session can be initiated with no file named; in this case, the data describing the model is built up in primary memory. The result of creating or editing a BPG model will not be saved in a file until an explicit command is given to save the file (or until the session is terminated; see below).

Load Model

This function first invokes the **Put File** operation to save the file that is currently loaded. The screen is then cleared. After this action is completed, a second popup box is placed on the screen to specify the name of the new file to be loaded. The user employs the mouse to place the cursor in the **File name:** field, then the file name may be entered using the keyboard. After the user signals completion of the file name entry by selecting the **OK** button at the bottom of the popup box, the file is written to the UNIX file system.

Save Model

This operation will cause a popup box to be placed on the screen with the current name of the file containing the model as a default. If no file has been named, then the default name is null. The user may choose to save the loaded file, or ignore it, using the **OK** and **Cancel** buttons, respectively.

Load Marking

This command behaves similarly to the **Load Model** command, except that it loads a marking for a model.

Save Marking

This command behaves similarly to the **Save Model** command, except that it saves a marking for a model.

Open Node Level

This command waits for the user to select a task node for refinement. Once a node has been selected, the screen is cleared and the user can define a task refinement as a BPG with a unique START and a unique STOP node.

Close Level

This command clears the current BPG (if it is a refinement) and restores the superior level. It has no effect if there is no abstraction of the current BPG.

Set Filters

A special popup box is placed on the screen. The box contains five buttons: **Data**, **Control**, **Labels**, and **OK**. Each button acts as a toggle to control the visibility of certain aspects of the model. **Data** toggles the visibility of the data flow subgraph of the BPG. By selecting **Data**, any data flow arcs and repositories are removed fromt he screen; selecting **Data** again will repaint the data flow information for the model. Similarly, the **Control** and **Labels** buttons toggle the visibility of the control flow subgraph and the object labels. The popup box is removed from the screen when the **OK** option has been selected.

Hardcopy

Olympus will produce Postscript files when this function is invoked. Before producing the

file, a popup box is placed on the screen, indicating the default Postscript file name. If a file name has been specified in the session, then the default Postscript file name is the base file name with a suffix of *.ps*. The scale factor is used to change the size of the image written to a Postscript device. The default size is 1.0; to produce a graph that is reduce to three quarters of the original graph, the scale factor should be set to 0.75. The user can back out of the **Hardcopy** command using the **Cancel** option.

Quit In order to terminate an Olympus session, select the **Quit** option from the right-button menu. This will provide the user with an opportunity to save the current model in a specified file, using a popup dialog box as described in the **Put File** function.

5. THE NeWS EDITOR

5.1. Running the NeWS Editor

The NeWS editor is started from the NeWS environment. Within a shelltool, type

```
% newsEditor
%
```

or if it is desired to initiate the session with a particular file, type

```
% newsEditor file_name.bpg
%
```

where **file_name** is the name of the file containing the BPG. The file name *must* be of the form:

model.bpg

Olympus will initialize itself, then the editor will prompt the user to create a variable-sized window. window on the screen. The new window will be labeled as

NeWS Editor Version 0.5 (<*date*>)

If Olympus was started with a file name, then the corresponding BPG will be painted within the window; otherwise, the window will be blank.

5.2. The User Interface

This editor is constructed using the NeWS windowing facilities, and has considerably different appearance and behavior than the SunView editor. First, the NeWS editor supports color models: The data and control flow submodels are in distinct colors (as are the tokens).

The NeWS Editor has a pallete on the left side of the display, composed of buttons, sliding scales, and silhouettes. The left mouse button is used to select items from the pallette, the middle mouse button is unused except in **Move Edge** operatons, and the right mouse is passed through to NeWS (thus it supports a NeWS popup menu).

The pallete is divided into four regions: Environment operations, Mode, Appearance, and Reset.

5.3. Environment Operations

Model Field

This field contains the name of the file currently being operated upon by Olympus. It is an editable field; editing operations can take place nearly any time by placing the cursor in the field and typing.

Load

Loads the model whose name appears in the **Model Field** into the server.

Save Saves the model currently loaded in the server into the UNIX file whose name appears in the **Model Field.**

Exit Halts the editor and server.

Start/Stop
Causes the server to begin interpretation of the loaded graph under the current marking.

Step Same action as the **Step** function in the SunView Editor.

Close Level
Same action as **Close Level** in the SunView Editor.

Sleeps Off
Changes the mode of the server from animation to simulation (see the distinction between the two in the discussion of the SunView Editor).

Constant Updates
Slows interpretation down (with **Sleeps Off** by keeping the screen updated with token movement.

5.4. Mode Operations

The NeWS Editor is always in some drawing mode, initially the mode to draw task nodes. The current node is indicated in the **Mode** of the pallete by a light rectangle surrounding one of the silhouettes. The left mouse button is enabled to operate upon the graph model in the chosen mode, e.g., the default mode is to be able to draw large, open circles whenever the left mouse button is stroked.

Thus, the **Mode** pallette is used to "set a brush style" for creating START, STOP, task, data repository, AND, and OR nodes, and arcs. The "bullseye" mode is used to open a level of refinement (see the **Open Level** operation in the SunView Editor description). An arc is drawn (in the arc mode) by selecting a tail with the left button, then a head. If the arc is to have a joint, then the joint is selected with the second mouse stroke using the middle button.

Two modes are used to mark nodes and arcs with tokens.

Two modes are used to enable instrumentation on nodes and arcs.

Two modes are used to set properties (using property sheets) for nodes and arcs.

Six modes are used to remove nodes, arcs, tokens, and instrumentation.

The final two modes are used to move nodes and edges. The middle button is used with jointed arcs (see above).

5.5. Appearance

The grid and the scale at which shapes appear on the screen can be controlled with sliding scales labeled **Grid** and **Zoom.** The default grid spacing in the NeWS Editor is 32 pixels. **Zoom** is not implemented in Version 0.5.

5.6. Reset Operations

The last six buttons on the control pallete are used to clear tokens, instrumentation, statistics, edges, submodels, and the model itself.

6. REFERENCES

1. 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.
2. 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.
3. G. J. Nutt, "A Flexible, Distributed Simulation System", *Tenth International Conference on Application and Theory of Petri Nets*, Bonn, West Germany, June 1989.
4. 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).
5. "SunView Programmer's Guide", Document Number 800-1324-03, Sun Microsystems, Inc., February 1986.
6. "Networking on the Sun Workstation", Document Number 800-1345-10, Sun Microsystems, Inc., September 1986.
7. "Windows and Window Based Tools: Beginner's Guide ", Document Number 800-1287-03, Sun Microsystems, Inc., February 1986.
8. *NeWS: A Definitive Approach to Window Systems*, Sun Microsystems, Inc., 1987.
9. *Sunset New Western Garden Book*, Sunset Books, Lane Publishing Company, 1979.

APPENDIX A: THE DISTRIBUTION TAPE

Olympus Version 0.5 is distributed using a UNIX *tar* format. The tape has the following directory structure:

strawberry
- Makefile (*[Used to compile the source code (if delivered)]*)
- bin
  - newsEditor
  - olympusServer
  - svEditor
  - ttyConsole
- build (*[meta make file]*)
- demo
  - *[Miscellaneous example BPGs]*
- doc
  - *Miscellaneous documentation, including this one*
- include
  - *[include files for compiling interpretations]*
- lib
  - *[Library routines for compiling interpretations]*
- miniNeWS
  - *[A minimal NeWS environment]*
- src
  - *[Olympus source code, if applicable]*

To copy the information from the tape, *cd* to the directory where you intend to install Olympus. Then execute

```
tar xvpf /dev/rst0
```

If you have signed an Object License agreement, then you will have all of the described files except strawberry/Makefile, strawberry/build, and those in strawberry/src. If you have signed both an Object and a Source License agreement, then you should have all of the files.

APPENDIX B: HINTS AND LIMITATIONS

Version 0.5 is an early version of the BPG support system. There are some limitations that will be changed in future versions. A few of these are summarized here.

Hierarchical Levels

BPGs are hierarchical models. Version 0.5 supplies only limited support of hierarchical moddling.

Task Interpretations

A node interpretation is executed when a token arrives at the node. There are two ways this can occur: the system is animating the BPG and a token flows along a control arc to the node; the node is marked by the user using the "Mark Node" command from the Olympus menu. Normally, the latter is not what is expected. For example, to start animating a model you probably would mark a starting node and then pick the "Animate" menu choice. However, if this starting node has an associated interpretation, the code for the interpretation will be called via RPC at the time you *mark* the node, not when you animate the graph. The conventional way to avoid this is to have a start node that doesn't have an interpretation, and is simply connected by a control-flow arc to the first "real" node of the graph. You mark this start node, then start the animation, and the token from the start node flows into the graph, perhaps causing execution of code associated with the first "real" node.

Another problem results from the behavior of the sockets on which the RPCs are transported. If remote calls arrive while another call is being processed, then the calls queue in order of arrival. The next call on the queue is executed when the call currently running completes. This can cause timing problems between nodes in the BPG, and at worst can cause deadlock. For example, imagine a producer/consumer model, in which the interpretations for both the producer and consumer are run on the same host. Further, imagine that the two processes are synchronized using a repository as a semaphore. Now consider what happens if the consumer node interpretation is written to busy-wait, reading the repository at regular intervals, until it finds that data has been produced. The problem is that the consumer never relinquishes the processor, and thus the call to the interpretation code for the producer will be queued, but will never run. Since the producer cannot run until the consumer returns, but the consumer isn't going to return until data is produced, deadlock results. There are many other situations in which this kind of contention can occur. However, the solution is fairly simple: node interpretations should *never* busy wait, sleep, or behave in such a way that they can block indefinitely. Rather, looping (such as for a busy wait) should occur in the BPG, repeatedly calling the node interpretation. The interpretation then just makes one check of the condition (by reading a repository) and returns in such a way that it indicates whether it succeeded or failed (by indicating different output arcs for success and failure, for example).

APPENDIX C: LIBRARY PROCEDURES FOR INTERPRETATIONS

caddr_t
build_output(output_token, data, size, execution_time)
u_short output_token;
caddr_t data;
u_int size;
u_long execution_time;

Builds the task return structure and returns the address of a pointer to this structure. This is normally not called directly by user programs, but rather is called by using the **task_return**() macro.

caddr_t
generic_recv(rp);
REPOSITORY_ACCESS *rp;

This is the library default repository input routine.

caddr_t
generic_send(rp);
REPOSITORY_ACCESS *rp;

This is the library default repository output routine.

int
repository_access(arc, data, size);
u_short arc;
caddr_t data;
u_int size;

This is the routine that task node procedures call to read or write a repository. *Arc* indicates the repository to be read or written; *Data* should be the address of a pointer in the case of reading; if the pointer is NULL, space will be allocated by the system for the data. *Data* should be a pointer to the data to be written in the case of writing, and *size* should indicate the size in bytes of the data. This routine returns 0 on success, otherwise it returns an **enum clnt_stat** cast to integer (see <rpc/clnt.h> and the RPC documentation).

void
repository_close();

This is an internal routine that is used to close down repository access after a repository_access() call, or when there is some error. It is not normally needed by users.

bool_t
repository_free(xdr_proc, object);
xdrproc_t xdr_proc;
caddr_t object;

Used to free allocations made by the system in connection with repository reads. *Xdr_proc* is the XDR routine that describes the data pointed to by *object*. This routine returns TRUE if successful, otherwise it returns FALSE (these values are defined in <rpc/types.h>.) It is useful if you write your own repository input and output routines, but is otherwise not needed.

APPENDIX D: SAMPLE LICENSES

Sample Object License Agreement

This License Agreement between _____ ("licensee") and Gary J. Nutt, Department of Computer Science, University of Colorado, ("licensor") relates to the licensee's use of the Olympus Modeling System ("Olympus").

(1) The licensor grants the licensee the nonexclusive right to the use of Olympus for its staff and students. The licensee has no right to sell nor give Olympus to any third party.

(2) The licensee will only use Olympus for noncommercial (educational and research) purposes.

(3) The licensee agrees not to charge a fee for the use of Olympus.

(4) There is no implied support nor legal liability on the part of the licensor for the use of Olympus by the licensee.

(5) The licensee agrees to report bugs in Olympus to the licensor, although the licensor is under no obligation to provide corrections nor alterations to Olympus.

(6) If Olympus is used in any research project, proposal, publication or other system, then the licensee agrees to provide appropriate reference to the licensor.

(7) Any written communications required to be made or given to either party shall be addressed as set forth below, or such other addresses as designated by written notice given to the other party:

_____
_____
_____
_____
_____
_____
_____

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

The licensee and the licensor agree to the terms of this Agreement.

For the licensee:
Signature:

_____

Printed Name:

_____

Title:

_____

Date:

_____

For the licensor:
Signature:

_____

Printed Name:
Gary J. Nutt

Title:
Professor

Date:

_____

Sample Source License Agreement

This License Agreement between _____ ("licensee") and Gary J. Nutt, Department of Computer Science, University of Colorado, ("licensor") relates to the licensee's use of the source code that implements Olympus Modeling System ("the source code").

(1) The licensor grants the licensee the nonexclusive right to the use of the source code for its staff and students. The licensee has no right to sell nor give the source code to any third party.

(2) The licensee will only use the source code for noncommercial (educational and research) purposes.

(3) The licensee agrees not to charge a fee for the use of the source code.

(4) There is no implied support nor legal liability on the part of the licensor for the use of the source code by the licensee.

(5) The licensee agrees to report bugs in the source code to the licensor, although the licensor is under no obligation to provide corrections nor alterations to the source code.

(6) If the source code is used in any research project, proposal, publication or other system, then the licensee agrees to provide appropriate reference to the licensor.

(7) Any written communications required to be made or given to either party shall be addressed as set forth below, or such other addresses as designated by written notice given to the other party:

_____
_____
_____
_____
_____
_____
_____

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

The licensee and the licensor agree to the terms of this Agreement.

For the licensee:
Signature:

_____

Printed Name:

_____

Title:

_____

Date:

_____

For the licensor:
Signature:

_____

Printed Name:
Gary J. Nutt

Title:
Professor

Date:

_____