A FORMAL MODEL FOR INTERACTIVE SIMULATION SYSTEMS

Gary J. Nutt

CU-CS-410-88 September, 1988
(Rev. April, 1989)

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

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

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

ABSTRACT

A Formal Model for Interactive Simulation Systems

Interactive simulation systems have become an important tool in the performance prediction repertoire of systems analysts and designers. Such systems allow the user to interactively construct a graphical model of a system, then to control the model as it is being executed. These systems are capable of providing visual point-and-select interfaces, animation, and scaled realtime simulation in addition to traditional simulation services.

The goal of the research described in this paper is to define a formal model of computation that specifies the semantics of an interactive simulation system independent of the specific syntax of any particular modeling language. Achievement of this goal enables an interactive simulation system to be designed such that a base implementation of the simulation facilities will support many different user interface extensions. Each user interface defines the syntax of a particular user-oriented model that interacts with the facilities provided in the base simulator. Thus, the interactive simulator can be customized to accommodate simulation models that are preferred by a specific class of performance analysts without reconstructing the entire system.

The applicability of the formal model of computation is argued by describing how it is implemented in the base system, and how different user interfaces can interact with the base system to implement simulations of flow charts, Petri nets, queueing networks, precedence graphs, and models such as the UCLA GMB.

1. INTRODUCTION

1.1. Background

Performance is an important requirement of most computer systems. When the specifications and requirements for a system are written, there are bounds on acceptable performance (although they are not always part of the formal specifications and requirements).

System designers have few tools to calibrate their general designs against (implicit or explicit) performance requirements. In some cases, designers rely on their knowledge and intuition in performance prediction, and in other cases there is a special effort made to predict performance during design.

Simulation is a useful technology to employ during the design and implementation. However, the overhead of using flexible simulation languages may be prohibitive. Many simulation tools are difficult to use because they are primitive or have a steep learning curve; the time required to construct the simulation model may be prohibitive to the project schedule.

The goal of the research described in this paper is to study simulation technology that will allow a design team to adopt various syntactic forms of an interactive simulation system. The interactive simulation system must be useful for identifying bottlenecks and other performance attributes of a design, and it must be natural and intuitive to the analyst, i.e., the user interface to the simulation facilities should employ model syntax and semantics that are in the designer's preferred modeling domain.

Interactive Simulation Systems

Traditional modeling has relied on linear languages and hand-drawn representations as fundamental tools. There is a new set of technologies that can be used to make modeling more convenient and useful. The simulation system family members that we describe in this paper are all visual and interactive; it is the visual syntax of the models that differ from interface-to-interface.

Interactive simulation systems are intended to take advantage of modern bitmap workstation environments. The interface to the simulation system should allow the user to construct visual models of operation, often with only limited amounts of detail in the model. The system should provide an interactive graphical editing facility that supports point-and-select creation and editing of the model. (While other technologies may ultimately prove to be better than a point-and-select model of operation, there is considerable evidence that this approach is well-accepted by a wide variety of users.)

The system should allow the user to experiment with the simulation of the system at nearly any phase of model development. For example, a user may create a model of a small portion of the system; the system should make it easy for the user to provide a simple environment in which to test the model. Also, the system should provide for exercising models that are not very well specified, i.e., ones in which considerable detail has been omitted. This requirement follows from the desire to allow the user to easily explore radically different approaches without investing significant amounts of time in the model development.

The interactive simulation system should provide animation facilities. Model animation refers to a pictorial representation of the activity within the model in scaled real time. Animation facilities are useful for obtaining a general understanding about critical aspects of the operation of the target system without resorting to quantitative observations.

The user should always be "in control" of the operation of the simulation. This means that the user should be able to single-step the simulation, set breakpoints, interrupt the operation of the simulation at any given moment, alter the real time scale at an given moment, dynamically alter loading conditions, and possibly even change the model as it is being executed. A "non-interactive" simulation system might provide single-stepping and breakpoints, but would generally not support interaction between the user and the model as the simulation progresses. Thus, interactive simulation systems cannot provide this sort of support to the user by simply compiling a user's model into a simulation language program.

The systems based on the formal model described in this paper are intended to be interactive simulation systems.

1.2. The Approach

Formal graph models of computation have been used to describe the behavior of software and systems for over 20 years [1]. During the early 1960s, Estrin and his students began to refine precedence models for describing parallelism in software and hardware e.g., see [10]; the "UCLA graph model" is a directed, acyclic, bilogic graph that incorporates both AND and OR logic for control flow. The early model was uninterpreted, and did not address data flow properties. In the 1970s, the model (later called the Graph Model of Behavior -- GMB [11,30]), continued to be refined so that it addressed these issues as well as others.

Just as the UCLA GMB model and others have been used to support general software development by representing software systems during design and implementation, formal models can be used to describe simulation models of software and hardware. The formal model for simulation should support timing, interpretation, control flow, data flow, and hierarchical model definition.

The user interface to the simulation system need not be computationally equivalent to the underlying model, although it must have features that map onto a subset of the features in the underlying model. Instead, the designer's interface to the system should be driven by his aesthetics and preferences, i.e., by the models with which the analyst is familiar and comfortable.

The family of simulation systems described here is based on a formal model of computation, called the *bilogic precedence graph*, (*BPG*) that is similar to the UCLA GMB. BPGs are defined in detail in Section 2. We have chosen to use it as the underlying model because it meets the criteria for a formal simulation model mentioned above, and because of our experience with this model as a definition of a simulation system. In Section 3 we describe an architecture, based on the formal model, for implementing a family of interactive simulation systems. While we have implemented a base simulation system and a BPG-specific interface, the focus in this paper is on the use of the formal model to define the core, not to describe the system that we have built. In Section 4, we will discuss the relationship between the formal model and some other models including flow charts, precedence graphs, Petri nets, queueing networks, and software engineering models such as the UCLA GMB.

1.3. Related Work

There are a number of simulation systems used for performance prediction that provide similar functions to that provided by our family of system. In each case, there is a pictorial representation of the model of operation; the analyst uses graphical support tools to describe the model of operation in the particular language of representation. Commercial products are available to support graphical interfaces to simulation software [6, 16]. 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 [34].

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, e.g., see [30].

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 [4, 15, 16], the Performance Analysis Workstation (PAW) [19], PARET [23], and Quinault [25] are all examples of this type of system.

GADD [22] 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 [20]). As a consequence, it is possible to interconnect simulations with fully-implemented components, thus providing a testbed debugging environment.

Visual modeling systems bear many similarities to visual programming systems. A programming system has an editing capability for expressing a program as a graphical interpreted model. The model can then be translated into target machine code or interpreted in an abstract machine environnment. There are several systems that use machines, much like they are used in our visual modeling system, to support visual programming abstractions, including work done by Browne and his students [3, 5], MCC's Verdi [13], Upconn [2], and Poker [33]. Longer collections of papers on visual programming can be found in [8, 28].

Finally, CASE systems sometimes use similar tools and techniques for representing software during development, e.g., PegaSys [21].

2. THE BILOGIC PRECEDENCE GRAPH MODEL

A bilogic precedence graph, (BPG), is a directed graph, (N, E, M) composed of a set of *nodes*, N; a set of *edges*, E; and a *marking* of N and E where

$$N = T \cup R, \text{ where } T \cap R = \phi$$

and T is a finite, nonempty set of *tasks*, and R is a finite set of *data repositories*. Tasks represent units of processing activity, while data repositories are intended to represent storage media for information. The edge set is also the union of two disjoint sets:

$$E = C \cup D, \text{ where } C \cap D = \phi$$

and

$$C = \{(t_i, t_j) \mid \text{for } t_i, t_j \in T \text{ there is a directed arc from } t_i \text{ to } t_j\}$$

$$D = \{(n_i, n_j) \mid \text{there is a directed arc from } n_i \text{ to } n_j, \text{ where } n_i \in T \Rightarrow n_j \in R, \text{ and } n_i \in R \Rightarrow n_j \in T\}$$

The marking is a mapping of the nodes and edges into the set of nonnegative integers:

$$M = M_N \cup M_E$$

where

M_N: N → nonnegative integers

and

M_E: E → nonnegative integers

The marking indicates a distribution of *tokens* on the nodes and edges in the BPG (cf. Petri nets [27]).

An *unmarked* BPG is one in which $M = \phi$, abbreviated as the ordered pair (T, C).

The BPG is the union of two subgraphs, where the pair (T, C, M) is a *control flow subgraph*, (*CFS*), representing the precedence among the set of tasks, and (N, D) is a bipartite *data flow subgraph*, (*DFS*), representing data flow among the tasks and data repositories.

The properties of the two subgraphs will be discussed individually.

2.1. Control Flow Subgraphs

Let (T, C, M) be a CFS. The CFS represents both conjunctive and disjunctive logic in the control flow. Define

$$T = O \cup A \cup G,$$

where O is a finite set of *disjunctive tasks*, whose input and output control flow edges use disjunctive (exclusive OR) logic. A is a finite set of *conjunctive tasks*, whose input and output control flow edges use conjunctive (AND) logic. G is a finite set of *general tasks*, which employ disjunctive logic on the input edges and conjunctive logic on the output edges.

Task $t \in T$ is *enabled* if $M_E(r,t) > 0$ for at least one (r,t) if t has disjunctive input logic, and for all (r,t) if t has conjunctive logic. An enabled task may *initiate firing*, which will cause the distribution of tokens on the edges and nodes to change. If the node has disjunctive input logic, then exactly one token will be removed from an input arc and placed on the task, i.e., $M_E(r,t)$ will be reduced by one, and $M_N(t)$ will be increased by one. After an amount of time (to be specified below), the task *terminates firing*, causing the token to be removed from the task and distributed to output edges. If the task has disjunctive output logic, then for exactly one (t,u), $M_E(t,u)$ will be incremented by one; If t has conjunctive output logic, then $M_E(t,u)$ will be incremented by one for all (t,u).

For example, suppose that x, y, and z are tasks. Then the model primitives represent the following cases:

(1) If y has only x as a predecessor, then the termination of one instance of a firing of task x will initiate the firing of task y. i.e., there is serial control flow from x to y.

(2) If x is a predecessor of y and z, and x has disjunctive output logic (i.e., $(x, y) \in C$ and $(x, z) \in C$, where $x \in O$) then either y or z (exclusively) may be enabled after x terminates.

(3) If x is a predecessor of y and z, and x has conjunctive output logic (i.e., $(x, y) \in C$ and $(x, z) \in C$, where $x \in A$), then both y and z will be enabled after x terminates.

(4) If x and y are predecessors of z, and z has disjunctive input logic (i.e., (x, z) ε C and (y, z) ε C, where z ε O), then either x or y must terminate before z can be enabled.

(5) If x and y are predecessors of z, and z has conjunctive input logic (i.e., (x, z) ε C and (y, z) ε C, where z ε A) then both x and y must terminate before z is enabled.

Notice that while we have named this family of graph networks "precedence graphs" they allow cycles to appear in the graph. The precedence is suggested by the topology of the graph along with the marking, so that any "instance" of a marking on a graph will provide the same effect as precedence, but the cycles can be included without violating precedence constraints.

Task Refinement

A CFS is one model of a system at a level of detail defined by the nodes in the graph and their interconnectivity. The level of detail can be increased by *refining* the definition of a node, or by providing an *interpretation* for the nodes.

A task is refined by defining another CFS to replace the original task. The new sub-CFS must be a set of (sub)tasks that, together, represent a structure corresponding to the original task in terms of its interactions with the other tasks in the original CFS. Within the sub-CFS, the precedence among subtasks is represented exactly like it is among tasks. However, since the sub-CFS represent a refinement of a task, and since the task meets certain precedence constraints, it is necessary for the sub-CFS to have boundary conditions that match the precedence constraints of the original task.

Let (T, C, M) be a sub-CFS. Consider the subset of *input peripheral tasks*, T_I, and the subset of *output peripheral tasks*, T_O; then the effective input and output logic for the sub-CFS must be either purely disjunctive or purely conjunctive for the sub-CFS. Thus, each sub-CFS can be thought of as being of "type O", "type A", or "type G", depending upon the manner in which the sub-CFS will react to the placement of tokens on T_I in terms of the subsequent placement of tokens on T_O.

While the refinement satisfies the need to represent the structural relationship among nodes, it is not useful for performance prediction. Functional interpretations are required in order to consider performance.

Let (T, C, M) be a CFS. Then define f(t) for each t ε A $\cup$ G to be an arbitrary function (with side-effects) that maps the task into a nonnegative integer:

$$f: A \cup G \rightarrow \text{nonnegative integers}$$

The function f is used for two purposes: First, it defines the firing time for the task, and second, it defines a formal interpretation for the task. The function is defined for all t ε A $\cup$ G; the default definition is that f(t) = 0 with no side-effects, i.e., the tasks requires no time to fire and no computation is performed as a result of firing.

The interpretation for t ε O = T - A $\cup$ G needs to return two values instead of one. The first value is the time, as above, and the second value is an arc identification. The label is used to resolve the nondeterministic choice of output arc, i.e., the interpretation also selects an output arc for OR nodes. Thus

$$f_O: O \rightarrow \text{nonnegative integers x } (R \times T)$$

It is possible to define data variables with scope that spans more than one f(t); the current model does not explicitly preclude this circumstance, although it is likely that it will be restricted at some late time. While global data

variables may simplify the definition of the interpretations, we have not seen cases where they were required. By including global data variables, the design will be more difficult to implement in a distributed memory system in which $f(t_i)$ and $f(t_j)$ are interpreted on different machines.

The other problem with allowing global data variables is that the side-effects of a procedure are not bounded; by limiting the scope of such data variables, one can localize the side- effects. This will allow the interpretations to appear to be functional from an external view.

As long as the global data is not pure modeling overhead, i.e., it maps onto some aspect of the system being modeled, the data can be explicitly represented by a data repository.

For the purposes of the formal model, the language for defining the side-effect and the time value are undefined. In practice, the time value is likely to be specified by a distribution function and the side-effects by a conventional programming language such as C.

Because of the hierarchy of descriptions, one views a precedence graph at some level of refinement. We define the most abstract task level of description to be Level 0, with successive refinements being numbered by succeedingly higher integers.

Suppose that we are interested in inspecting the interpretation of each node in a model at Level i. Then, the following rules hold for finding the interpretation:

(1) If Level i is a precedence graph, then the interpretation is supplied by Level i+1.

(2) If Level i is a procedure, then the interpretation is supplied by the procedure declaration.

(3) If Level i is undefined, then Level i is interpreted to be the same as for the corresponding node at Level i-1.

Therefore, a model can only be interpreted whenever each task ultimately has some procedural definition refinement.

2.2. Data Flow Subgraphs

The overall task model requires that one be able to represent information sharing among tasks. In some cases, the task intercommunication is accomplished by sharing memory, and at other times it might be accomplished through message passing. In order to make that sharing mechanism explicit, data flow edges are added to the CFS graph. And to take into account that the sharing may not occur at the same instant in time, the edges have an interposed data repository.

If task a and task b share information, an edge must be placed in the DFS between a and r, and between r and b, where $r \varepsilon R$, and $a, b \varepsilon T$. If a produces the information and b consumes it, then the edge should be directed from a to r and from r to b, i.e., (a,r) and (r, b) are members of D. If either may produce the information, then four edges must be included in D: (a,r), (r,b), (b,r), and (r,a).

let $t \varepsilon T$ and $r \varepsilon R$, then for each (t,r) and $(r,t) \varepsilon D$, define an *access function*, g, that maps the arc onto a zero-valued function with side-effects.

$g: g_r \times g_w$
$g_r: R \times T \times \text{pointer} \rightarrow 0$
$g_w: T \times R \times \text{pointer} \rightarrow 0$

Edges of the form (t,r) are write access functions and edges of the form (r,t) are read access functions. The

functions are intended to model the action of the data repository under read and write operations. An access function, g(t,r,p), for an edge (t,r) can only be invoked from within an interpretation of t, f(t); similarly, g(r,t,p) can only be invoked from f(t). As with the interpretation function, the language used to define the function is undefined, although it is thought of as a procedural language such as C.

Technically, the "pointer" set is undefined in the formal model; it is intended to be a mechanism for passing parameters between the task and the repository. The task interpretation and the repository interpretation must implement their own protocol for exchanging information. The type of the object passed between the task interpretation and data repositories (via the access functions) is determined by the definitions for g and f. For example, if f passes an integer to a data repository via the pointer, then g must be expecting an integer. Similarly, if f(t) requests data from a repository, the g(r,t,p) must return an object with compatible type.

Notice that data repository access is not required for each firing of a task node that is connected to a data repository node; the nature and frequency of the access are determined by the task interpretation, i.e., by its use of the access function.

2.3. Performance Metrics

There are a number of performance metrics that result from the model definition. While the model itself applies no particular semantic to tokens, tasks, etc., it can be used to represent the activity of tasks, token flow characteristics, etc.

In BPGs, tokens are only blocked at multi-input AND nodes, since a task is not inherently safe.† Queueing times are reflected by the time that such an arc contains at least one token. Similarly, one can compute the maximum and mean number of tokens on such an arc to represent queueing delays.

A safe task can be used to represent a serially reuseable resource, thus the amount of time that a token resides on the task represents resource utilization. Using the statistics for the implied queue for every serially reuseable resource, it is easy to observe the resource competition in the model.

Throughput for any component of system is represented by the number of tokens that fired a particular task corresponding to that component.

2.4. An Extended Example

BPGs define the semantics of a simulation system without necessarily specifying an interface (no figures or pictures were used to describe the model). A BPG defines control flow, data flow, timing, and interpretations for tasks and repositories. These are sufficient to model computer system behavior as a set of discrete events.

For example, suppose that we wished to model a simple queueing system which has an infinite population of jobs that arrive according to an interarrival distribution. Figure 1 is a description of such a model in the language of the formal model.

Process p_1 represents the arrival portion of the model; the token on p_1 causes it to define a job and write a description of it to r_1, the interpretation returns the time at which the next job should be generated, i.e., the interarrival time is modeled as a process firing. After the prescribed amount of time, p_1 terminates firing and places a token on the arc leading to p_2 and another on the arc leading back to itself for the next job.

† Of course one can generate safe tasks such as p_4 in our following example. This is accomplished by adding tasks corresponding to p_3 and p_5 to p_4 as shown.

Process p_2 is included to represent enqueueing from the I/O device, or for new jobs.

Process p_3 is used to reschedule the server whenever it becomes idle. It cannot fire unless there is a job queued (token on (p_2, p_3)) and the server is idle, i.e., there is a token on (p_5, p_3). Upon firing, a token enables p_4.

P_4 models most of the activity related to the server. This process retrieves a job from the r_1, decides how much time the job will use the server before stopping, then updates the amount of time the job has used. P_5 receives a token when the server finishes to indicate that the server is idle.

When a job leaves the server, it may be ready to depart the system or it may be requesting an I/O operation. P_6 makes this decision based on the information about the job stored in r_2; if the job has finished its service time, then it departs the system.

The I/O part of the model is similar to the server part.

Let the CFS be (N, E, M) where

$N = T \cup R$

$T = G \cup O \cup A,$

$G = \{p_1, p_4, p_5, p_7, p_9, p_{10}\}$

$O = \{p_2, p_6\}$

$A = \{p_3, p_8\}$

$R = \{r_1, r_2\}$

$E = C \cup D$

$C = \{(p_1,p_1), (p_1,p_2), (p_2,p_3), (p_3,p_4), (p_4,p_5), (p_4,p_6), (p_5,p_3), (p_6,p_7), (p_6,p_8), (p_8,p_9), (p_9,p_{10}), (p_9,p_2), (p_{10},p_8)\}$

$D = \{(p_1,r_1), (r_1, p_4), (p_9,r_1), (p_4,r_2), (r_2,p_6), (r_2,p_7), (r_2,p_9)\}$

$M_N(p_1) = 1$

$M_N(p_5) = 1$

$M_N(p_{10}) = 1$

Figure 1a: A BPG of a Simple Queueing System

```
f(p1)
{
        struct job_descript job;

/* Generate a new job, then reschedule self Δt later */
        job.time = sample(total_service_distribution);
        job.IOtime = 0;
        g(p1, r1, &job);
        return(sample(interarrival_distribution));
}

f(p4)
{
        int slice;
        struct job_descript job;

        g(r1, p4, &job);
        slice = sample(interIO_time_distribution);
        slice = (slice > job.time) ? job.time: slice;
        job.time = job.time - slice;
        g(p4, r2, &job);
        return(slice);
}

fO(p6)
{
        struct job_descript job;

        g(r2, p6, &job);
        return(0, (p6, ((job.time <= 0) ? p7: p8)));
}

f(p7)
{
        struct job_descript job;

        g(r2, p7, &job);
        report_generation(&job);
        return(0);
}
```

```
f(p9)
{
        int iot;
        struct job_descript job;

        g(r2, p9, &job);
        iot = sample(IO_time_distribution);
        job.IOtime = job.IOtime + iot;
        g(p9, r1, &job);
        return(iot);
}
```

Figure 1b: Task Interpretations for the Simple Queueing System

```
{
struct job_descr_queue rep_1;

        g(p_1, r_1, job)
        /* job will be treated as a C struct of type job_descript */
        {
                struct job_descr *new_job;

                new_job = allocate_space(sizeof(job));
                new_job = copyof(job);
                enqueue(new_job, rep_1)
                return(0);
        }

        g(p_9, r_1, job)
        /* job will be treated as a C struct of type job_descript */
        {
                struct job_descr *new_job;

                new_job = allocate_space(sizeof(job));
                new_job = copyof(job);
                enqueue(new_job, rep_1)
                return(0);
        }

        g(r_1, p_4, job)
        /* job will be treated as a C struct of type job_descript */
        {
                struct job_descr *next_job;

                next_job = dequeue(rep_1);
                job = copyof(next_job);
                free_space(next_job);
                return(0);
        }
}
```

Figure 1c: Repository Interpretations for r_1

```
{
struct job_descr_queue rep_2;

        g(p4, r2, job)
        {
        /* job will be treated as a C struct of type job_descript */
                struct job_descr *new_job;

                new_job = allocate_space(sizeof(job));
                new_job = copyof(job);
                enqueue(new_job, rep_2)
                return(0);
        }

        g(r2, p6, job)
        {
        /* job will be treated as a C struct of type job_descript */
                struct job_descr *next_job;

                next_job = dequeue(rep_2);
                job = copyof(next_job);
                push(next_job, rep_2);
                return(0);
        }

        g(r2, p7)
        {
        /* job will be treated as a C struct of type job_descript */
                struct job_descr *next_job;

                next_job = dequeue(rep_2);
                job = copyof(next_job);
                push(next_job, rep_2);
                return(0);
        }

        g(r2, p9)
        {
        /* job will be treated as a C struct of type job_descript */
                struct job_descr *next_job;
```

```
        next_job = dequeue(rep_1);
        job = copyof(next_job);
        free_space(next_job);
        return(0);
}
```

Figure 1d: Repository Interpretations for r_2

3. AN INTERACTIVE SIMULATION SYSTEM ARCHITECTURE

We use the formal model as the basis of an interactive simulation system as shown in Figure 2. The BPG Interpreter is given a BPG, (T, E), from the BPG Storage, and a marking, M, from the Marking Storage. The Interpreter reads the model and the marking from the respective storage modules. It then evaluates the marking to determine the set of tasks that can be fired; those that are enabled are scheduled for firing (and interpretation of the corresponding task interpretations). As a result of firing a particular task, the marking is updated and the interpreter continues operation. The performance metrics mentioned above are kept by the Interpreter.

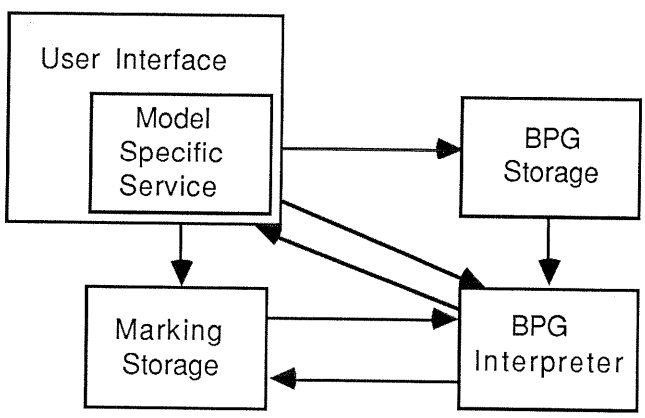

Figure 2: The Simulation System Architecture

The Storage services are used to save a model generated by the input portion of the system. The implementation of such a service need not be complex, although it must support operations such as "store a node," "retrieve the node at the head of this edge," "erase this edge," etc. The Interpreter will interpret the information in the storage service as a BPG; the input module will store information in the storage service as some subclass of a BPG.

The input and output portions of the system are incorporated into a User Interface module. The User Interface has two responsibilities: It must implement the specifics of the human-machine interface, and it must capture the syntax and certain semantics of the representation of the BPG that is used at the interface. In the figure, we attempt to emphasize this latter responsibility by showing it as a sub module within the User Interface.

The User Interface is specific to the workstation and user of the system. It may be implemented in an environment which provides windows, scrolling, popup boxes, menus, and other presentation facilities. These aspects of the system are of no concern to the Interpreter and Storage portions of the system.

The Model-Specific Service implements a mechanism for defining an instance of the model, e.g., a UCLA GMB, a timed Petri net, a queueing network, etc., and for mapping specific semantics of the model onto BPGs. The Model-Specific Service interacts with the user via the User Interface, and with the Storage services and the Interpreter through direct interfaces. The primary job of the Model-Specific Service is to implement a model editor and an animation/simulation console. While the system is designed to support graphical animation and simulation, the Model-Specific Service and User Interface are not required to be graphical.

The architecture that we have described in Figure 2 has been designed with multiple process implementations in mind. That is, each of the blocks in the diagram can be thought of as an independent process, sometimes acting as a client and other times as a server.

The Olympus† Simulation System is a first implementation of the services described in Figure 2. Olympus has been implemented in a BSD Unix environment on a network of Sun workstations; the implementation is described in detail in another paper [26].

The Olympus implementation combines the storage and interpretation services into a single process, called the *Olympus server* (the model-specific service is viewed as a client process in this implementation).

The User Interface client establishes a socket with the Olympus server when it is initiated; all intercommunication between the client and the server take place over the socket. The server obtains commands to perform storage and simulation control operations from the socket, and send display instructions (when an animation is in-progress) to the client via the same socket.

The Model-Specific Service in Olympus is a BPG interface, similar to that described in the next section. That is, Model-Specific Service includes an editor and viewer; the editor can be used for modifying the BPG model in storage, even as the interpreter operates. The Service also provides for control of the Interpreter, for changing the marking, etc.

Finally, the User Interface has been implemented using three different technologies: The first version was a SunView implementation, the second version is a NeWS implementation, and the third version is implemented in Lisp on a Symbolics workstation.

---

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

4. SOME INSTANCES FROM THE FAMILY OF SIMULATION MODELS

The fundamental hypothesis of our approach is that BPGs can be used to describe the semantics of interactive simulation independent of the syntax of many different models. That is, one can build a new user interface for each individual modeling system without changing the simulation services implied by the BPG. In this section, we discuss various models in terms of BPGs, and hence, in terms of how they might utilize the remaining parts of the interactive simulation service if they were implemented in the User Interface.

We introduce the discussion by providing a graphical interpretation of the BPG model.

A BPG can be represented graphically by choosing distinct representations for members of G, O, A, and R, then constructing the graph composed of vertices interconnected with edges from E. Let members of G be represented by large, open circles; O by small open circles; A by small, filled circles; and R by rectangles. The marking of the BPG can be represented by placing small, filled rectangles on edges and nodes.

Figure 3 is a pictorial representation of the BPG shown in Figure 1.

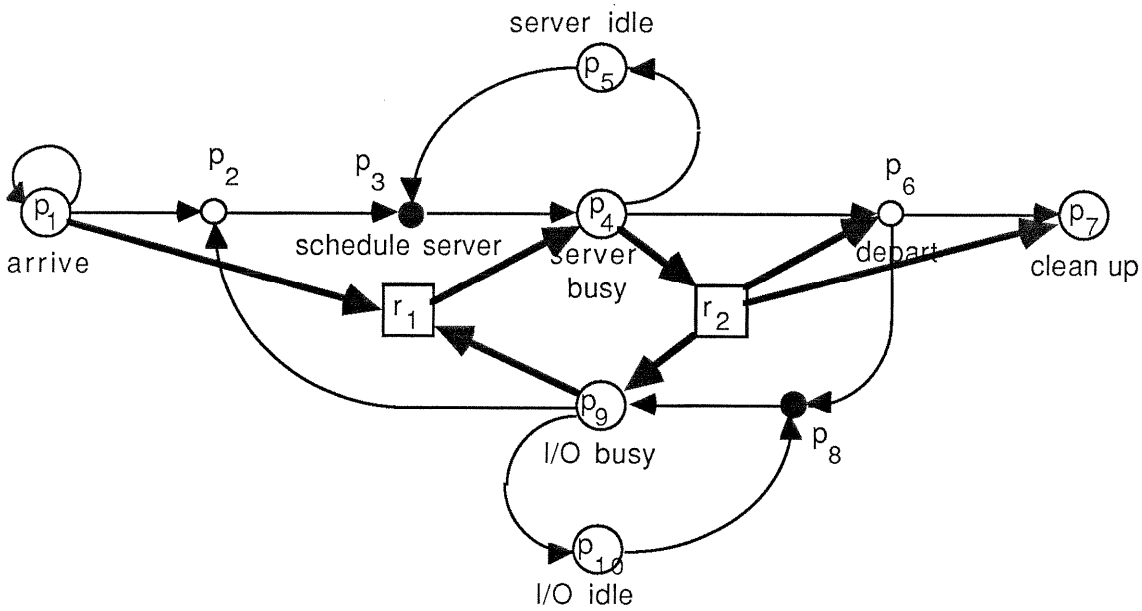

Figure 3: A Pictorial Representation of the BPG in Figure 1

4.1. Flow Charts

Flow charts have often been used to describe systems and software as well as simulations. As formal models of computation, flow chart can be thought of as interpreted directed graphs with disjunctive logic. Flow charts also support hierarchy.

A flow chart can be formally defined using the definition from Section 2. Data flow is ignored, so $R = \phi$ and $D = \phi$. Since only disjunctive logic is used for control flow, $A = \phi$, i.e., $T = O \cup G$, where each $g \varepsilon G$ will have only a single output edge. Every $t \varepsilon O$ is a "decision box" in the flowchart.

A flow chart does not ordinarily support the idea of multiple tokens. Instead, a single execution of program corresponding to the flowchart takes place when the single input task is marked with a token. Thus, the markings are simplified to the point that $\Sigma (M_N(t) + M_E(t_i,t_j)) = 1$.

In terms of control flow logic, a flow chart is equivalent to a state diagram or a (disjunctive logic) precedence chart.

It should be clear that this model is a subset of the BPG model, and that an implementation of a user interface to provide simulations based on flowcharts is feasible.

4.2. Petri nets

Petri nets [27] have often been used to describe concurrent systems, since they incorporate logical flow to represent alternative and parallel threads of control. While much of the utility of Petri nets is in their formal representation, it is possible to animate the token flow in a Petri net so that one can watch the flow of tokens through the Petri net graph.

The BPG Storage and Interpreter can be used to support a User Interface for a Petri net modeling system. This is accomplished by constructing the Model-Specific Service such that it maps the Petri net constructs into corresponding BPG constructs.

A Petri net is a four-tuple, (P, T, I, O), where P is a finite set of places, T is a finite set of transitions, and I and O describe the interconnectivity of places and transitions. P and T form two sets of nodes for a bipartite graph. I and O describe the set of input and output places, respectively, for any particular transition.

A marking, μ, of a Petri net is a mapping of P into the nonnegative integers.

Each transition uses conjunctive logic for a firing rule, i.e., a transition can fire if and only if each input place contains at least one token; the firing results in one token being placed on each output place of the transition.

Let (P, T, I, O) be a Petri net with marking μ. Define a BPG, (N, E, M) such that:

$$N = (O' \cup A \cup G) \cup R$$

where

$O' = P$

$A = T$

$G = \phi$

For each

$p_i \varepsilon I(t)$, add (p_i, t) to E

and for each

$p_i \varepsilon O(t)$, add (p_i, t) to E

The BPG is marked as follows:

$$M_E = \phi$$

and

$$M_N(t) = \mu(p), \text{ for } t \varepsilon O \text{ and } p \varepsilon P$$

$$f(t) = \phi \text{ for all } t \varepsilon A$$

The remaining problem is to get the t ε O' to behave properly; in the Petri net, there may be forward conflict, i.e., the outputs of a place may lead to two or more different transitions that may be simultaneously enabled. If there is a single token on the shared input place, then it can only cause one or the other of the two transitions to fire; the choice is nondeterministic in a Petri net. We define

```
f_O(t)
{ /* for all tεO' = P */
        while (no downstream task, t', would be enabled);
        return(0, I(t'));
}
```

This is a busy wait interpretation, but it will cause the OR node to defer a choice of the output arc until there is a downstream arc that will be enabled by the firing. Successful implementation implies that task firing can occur concurrently in the simulation system. (Otherwise, the function becomes more complex, but still possible.)

The resulting BPG will behave like a Petri net.

Timed Petri Nets

Timed Petri nets (TPNs) are Petri nets which introduce time and conflict resolution; this allows one to study performance characteristics of a system represented by the model. Over the years, there have been many variants of TPNs [14, 24, 29, 31, 36]; we arbitrarily select a simplified version of this class of models.

Time is introduced to the model by providing a distribution function describing the amount of time required to fire a transition; it is allowable to use a constant distribution of zero. Thus, timing in the BPG tasks is analogous to timing in TPNs; the definition above need only provide interpretations for tεA that correspond to the distributions in the TPN. Conflict resolution is also specified by assigning probabilities to output arcs in forward conflicting situations. These probabilities are used to stochastically resolve the conflict; this approach can be encoded into the BPG model as illustrated above.

Figure 4 is a Petri net representation of the simple queueing system introduced in Figure 1.

4.3. Queueing Networks

Queueing networks (QNs) have been derived as a formal model by which one can describe interconnected service centers; analytic solution techniques exist which can be used to derive exact or approximate performance characteristics of the model in a relatively efficient manner. Extended queueing networks (EQNs) are QNs that incorporate additional modeling primitives to represent synchronization operations [12, 15, 18, 32]).

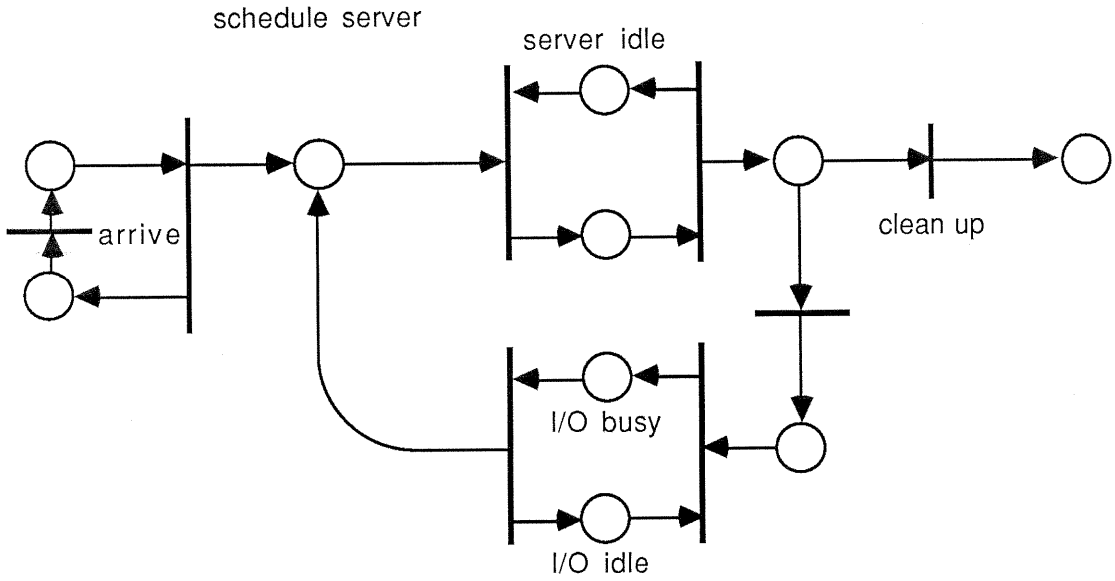

Figure 4: Timed Petri Net Example

Vernon, et al have compared EQNs and performance Petri Nets (a variant of TPNs) as they are used to predict the performance of systems using analytic techniques [35]. The comparison is with respect to descriptive power, efficiency in deriving an analytic solution to a model, and applicability to different problem domains. The conclusion from their study is that QNs are more efficient models than TPNs in terms of the effort to construct a model and the effort to analyze the model once it is constructed. However, TPNs appear to have an edge in representing synchronization -- a property that Vernon, et al believe may grow in importance as the applications (systems) employ more parallelism.

Specific variants of EQNs have a number of different types of nodes, e.g., IPGs have 16 node types [15]. Roughly, these nodes represent servers and queues, passive resources, and control flow activity, see Figure 5.

The server and queue nodes represent an active resource from which a job may request certain service; the queue is used to keep track of the pending requests for service. Individual service requests are represented by a probability distribution function; when a customer is allocated the resource, then the distribution function is used to specify the length of time that the job will utilize the server.

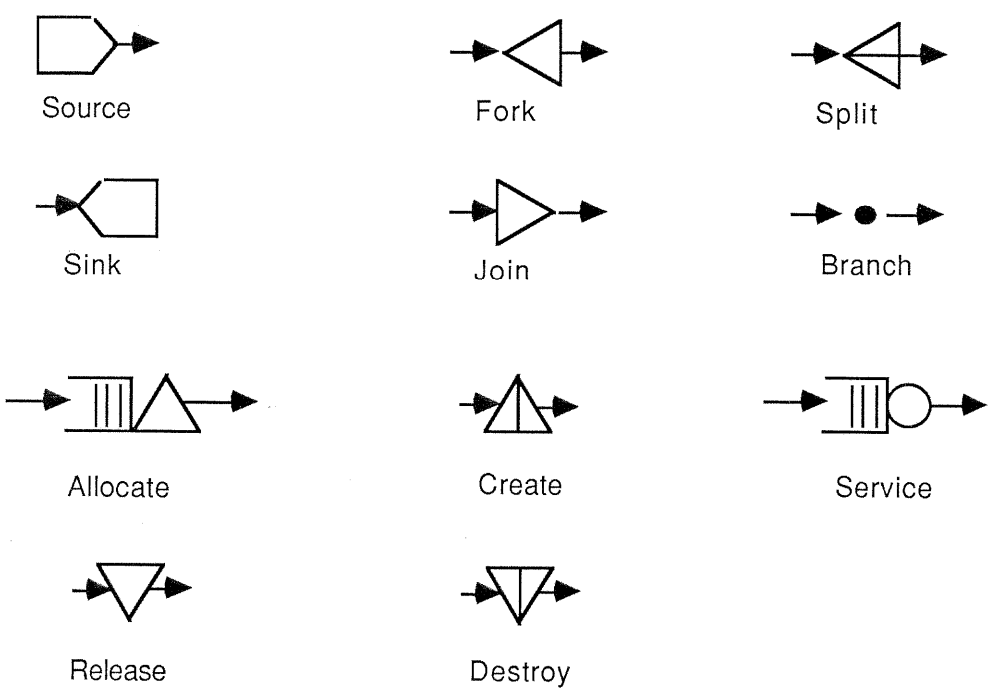

Figure 5: Extended Queueing Network Nodes

Each server node and queue can be represented in the BPG model as follows: Assume that *arrive* is the name of the task upstream from the queue. Define an A (AND) node, *sched*, and two G (general) nodes, *busy* and *idle* such that

(idle, sched), (arrive, sched), (sched, busy), (busy, idle) ε T

Let

```
f(busy)
{
        return(sample(service_distribution))
}
```

Now the BPG behaves as a server node with a FIFO queue. For proper operation, it is necessary that initially, M(idle) = 1.

Passive resources are held by a job as it moves through the queueing network. In order to obtain passive resources, the job must compete with other jobs. Thus there is the idea of a passive resource allocation node and associated queue of competing jobs in the model.

The BPG to model a passive resource allocation node is a subset of the server example. Each instance of an EQN allocate node corresponds to the "sched" node above, and each release node is represented by a degenerate AND node such as "busy" in the previous example. The "idle" node represents the set of idle resources; and initially M(idle) = n, where n is the number of units of the given resource. No tasks have non-zero times associated with them.

The EQN control flow primitives are used to implement *fork*, *join*, *split*, and *branch*. The *fork* and *join* nodes model conjunctive control flow, and thus can be supported by the conjunctive tasks in the BPG. Similarly, the *split* and *branch* nodes model disjunctive control flow and can be supported by the disjunctive tasks in the BPG model.

Figure 6 illustrates the use of the components to specify the simple queuing network introduced in Figure 1.

4.4. The UCLA GMB

The UCLA GMB is a composition of the Graph Model of Control, a data graph, and a PLIP interpretation for data graph components [11, 30].

The control flow graph is made up of nodes representing event, interconnected by directed edges representing control flow among the events. BPG tasks are analogous to GMB events. The control flow rules allow input and output logic to be compositions of AND and OR; thus, GMB control nodes may map to an equivalent composition of BPG tasks. For example, if a GMB event is enabled by the termination of events (a OR b) AND c, then the BPG would represent this situation with two tasks: An OR node with a and b as successors, which in turn was a successor to the subject AND task with c also acting as a successor. As with BPGs, the control state of the model is represented by a distribution of tokens on the graph, and the tokens are redistributed by the firing rules for the events.

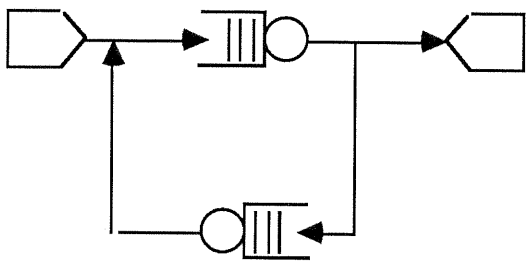

Figure 6: Extended Queueing Network Example

The GMB datagraph is made up of processors and datasets. Uncontrolled processors react to conditions in the data-graph, while controlled processors are activated by conditions in the control flow graph. Datasets represent static collections of data in the system.

BPGs can emulate the datagraph behavior of controlled processors by encoding the access functions within the task interpretations. Similarly, the behavior of the datasets is modeled using the g functions described in Section 2. Uncontrolled processors are implemented in the BPG model as autonomous models, interconnecting with the control flow model only through the conditions that exist within the data repositories.

Finally, GMB employs a PL/I declarations to specify the interpretation of task nodes; it would be easy to allow this PL/I subset (PLIP) to be a language for specifying the f interpretations for each task.

5. CONCLUSION

Interactive simulation systems are important tools in the repertoire of today's system designer. A performance analyst may favor queueuing networks, timed Petri nets or other specific simulation and analytic models; as part of this familiarity, the analyst understands the mechanisms for using distributions to specify properties of models, how to interpret results, etc. While most system designers have some knowledge of service and interarrival time distributions, there is a spectrum of other graphical models used to support software design, e.g., see [9]. Simulation tools will be far more useful to the analyst if the model is natural to him.

Choosing a model that is "natural" is not an objective task; analysts pick and choose among models depending upon their experiences, sense of aesthetics, and the task at hand. Instead of attempting to choose the "best" model to support the analysts, our approach is to implement the interactive simulation facilities and then adapt any of a wide variety of interfaces to the facilities via a customer-selected model.

Animation has become popular in interactive systems, e.g., see [28]. As a performance analysis tool, the value is in the qualitative feedback from the model rather than traditional quantitative results from simulations. Our approach attempts to support the quantitative results early in the design process and qualitative results from the same model as the design matures.

We think this is a more difficult problem than translating a modeling language into a simulation programming language (namely, the interactive aspect of the problem prevents straight-forward translation). We have proposed a formal model to capture the semantics of the simulation and described a means by which a variety of models can be adapted to the underlying system. While our descriptions of BPG "implementations" of various external models may suggest that the technique is merely providing an underlying simulation language with various translations, the resulting system preserves the fine-grained interaction between the User Interface and the Simulation Service.

Our research is not complete, but the formal model and the approach are worth describing at this point. In the future, we intend to reimplement the storage and interpretation services as distinct processes, since there is no logical reason why the two services should be implemented within a single process. We expect to achieve better separation of model-specific semantics from the simulation server by this implementation.

We also see that our distribution of function is one form of distributed simulation, i.e., the control, storage, and interpretation are distributed across distinct processes. Chandy and Misra, and others have described techniques for distributing the task interpretation across local memory machines [7, 17, 20]. We will apply these techniques for distributing the simulation service to the precedence models. It may be that this approach will provide new insight into the problem of distributed simulation.

6. ACKNOWLEDGEMENTS

Several people have worked on the Olympus system implementation: Adam Beguelin contributed considerable effort to a SunView version of the BPG editor. Bruce Sanders has designed and implemented a NeWS version of the BPG editor and interface. David Redmiles designed and implemented the Lisp Symbolics User Interface to Olympus. John Hauser, Steve Elliott, and Jeff McWhirter have performed all of the detailed design and implementation of the Olympus server.

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.

7. REFERENCES

1. J. L. Baer, "A Survey of Some Theoretical Aspects of Multiprocessing", *ACM Computing Surveys 5*, 1 (March 1973), 31-79.
2. M. Bhattacharyya, D. Cohrs and B. Miller, "A Visual Process Connector for Unix", *IEEE Software 5*, 4 (July 1988), 43-50.
3. J. C. Browne, "Formulation and Programming of Parallel Computations: A Unified Approach", *14th International Conference on Parallel Processing*, August 1985, 624-631.
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. J. C. Browne, "A Unified Approach to Parallel Programming", *1987 Summer Workshop in Parallel Computation*, June 23, 1987.
6. *CACI Simscript II.5 marketing information*, CACI Product Company, La Jolla, California, 1988.
7. K. M. Chandy and J. Misra, "Asynchronous Distributed Simulation via a Sequence of Parallel Computations", *Communications of the ACM 24*, 4 (April 1981), 198-205.
8. S. Chang, T. Ichikawa and P. A. Ligomenides, *Visual Languages*, Plenum Press, 1986.
9. E. J. Chikofsky, ed., *Software, Special Issue on CASE*, IEEE, March 1988.
10. G. Estrin and R. Turn, "Automatic Assignment of Computations in a Variable Structure Computer System", *IEEE Transactions on Electronic Computers EC-12*, 5 (Dec 1963), 755-773.
11. 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.
12. V. Fernandes, J. C. Browne, D. Neuse and R. Velpuri, "Some Performance Models of Distributed Systems", *Proceedings of the CMG XV International Conference*, 1984.
13. M. L. Graf, "Building a Visual Designer's Environment", MCC Technical Report No. STP-318-87, October, 1987.
14. M. A. Holliday, "Deterministic Time and Analytic Models of Parallel Architectures", Ph.D. thesis, (Technical Report #652), Computer Sciences Department - University of Wisconsin, Madison, July 1986.
15. S. Iacobovici and C. Ng, "VLSI and System Performance Modeling", *IEEE Micro*, August 1987, 59-72.
16. *PAWS/GPSM marketing brochures*, Information Research Associates, Austin, TX, 1988.

17. 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.

18. J. T. Jennings and C. U. Smith, "GQUE: A Tool Using Workstation Power to Aid Software Design Assessment", draft technical report from Performance Engineering Services, Austin, TX, 1988.

19. B. Melamed and R. J. T. Morris, "Visual Simulation: The Performance Analysis Workstation", *IEEE Computer 18*, 8 (August 1985), 87-94.

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

21. M. Moriconi and D. F. Hare, "The PegaSys System: Pictures as Formal Documentation of Large Programs", *ACM Transactions on Programming Languages and Systems 8*, 4 (October 1986), 524-546.

22. M. Moser, "GADD -- A Tool for Graphical Animated Design and Debuggin", *ICC '87 Conference Record*, 1987, 38.2.1-38.2.5.

23. K. M. Nichols and J. T. Edmark, "Modeling Multicomputer Systems with PARET", *IEEE Computer 21*, 5 (May 1988), 39-48.

24. 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.

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

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

27. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, Inc., 1981.

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

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

30. 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.

31. 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.

32. C. H. Sauer, E. A. MacNair and J. F. Kurose, "The Research Queueing Package: Past, Present, and Future", *AFIPS Proceedings of the National Computer Conference*, 1982.

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

34. 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.

35. M. K. Vernon, J. Zahorjan and E. D. Lazowska, ''A Comparison of Performance Petri Nets and Queueing Network Models'', Technical Report #669, Computer Sciences Department - University of Wisconsin, Madison, September 1986.

36. W. M. Zuberek, ''Timed Petri Nets and Preliminary Performance Evaluation'', *Proceedings of the Seventh Annual Symposium on Computer Architecture*, 1980, 88-96.