Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

OBJECT ORIENTED PERFORMANCE MODELS WITH KNOWLEDGE-BASED DIAGNOSTICS

Mohsen Pazirandeh
Jeffrey Becker
Advanced System Technologies, Inc.
12200 E. Briarwood Ave. Suite 260
Englewood, Colorado 80112

ABSTRACT

The performance modeling of computer systems plays an important role in the system engineering life cycle. Historically, however, this important role has not been recognized, and often it has been treated as an obligation. The reason for this diminished stature has many causes, but foremost is that performance modeling has failed to position itself as an indispensible tool to system designers. This single cause itself has many contributors, two of which are the most visible. First, modeling has come to be viewed as an esoteric exercise because of its inability to reflect the system architecture and other system characteristics with ease. Second, the output of most models is a set of statistical data, hardly pointing to a specific design deficiency or operational failure.

Two important events in software and language development have come to offer potential solutions to these problems. Object oriented languages allow hierarchical and graphical definition of system architecture, and languages such as PROLOG facilitate the development of knowledge-based systems. We will show how the combination of an object oriented language (Smalltalk) and PROLOG can be used to develop a tool containing hierarchical system description, graphical system entry, performance prediction algorithms, and knowledge-based diagnostic capabilities at an order of magnitude reduction in development costs over standard high order languages such as PASCAL.

1. INTRODUCTION

There are two basic deficiencies in the development of computer system models, resulting in their diminished applicability and usefulness:

- o The system description, including hardware, software and the workload is done using a high level programming language. This has the disadvantage of having no visual benefit and the user has to pore over many lines of descriptive statements to 'see' the system architecture. This deficiency has made the use of model description quite cumbersome and awkward.

- o The models, generally, produce a set of performance data which make very little sense to the designer; its main user. It takes a seasoned modeler to interpret the results, derive inferences, identify the sources of the problems, and recommend corrective actions.

The above two deficiencies have done much to put both the modeling activity and the modeler in an unfavorable position with the rest of the design group. Managers and system designers have tended to treat modeling as an end and an obligation rather than a tool for designing an efficient system. These problems have plagued the modeling activity for a long time resulting in the exclusion of the modelers as a major component of the system design and development process.

We will show how the aforementioned deficiencies can be alleviated using recently-developed languages. Specificaly, we will show how an object oriented language such as Smalltalk [2] can be used in conjunction with PROLOG to achieve the following three objectives:

1. Define the system architecture graphically. This will include hardware components, their connectivities, workload and system functions, eliminating the need for defining the system in a textual manner.

2. Allow the insertion of any method for the calculation of the performance parameters, including analytical techniques or discrete event simulation satisfying predefined interface requirements. The user can develop his own module or tie it to another performance package. This modularity offers the advantage of incremental development of an eventual tool.

3. Develop a diagnostic rule base to quickly detect and isolate the causes of system bottlenecks and response time failures. We will show how a set of rules can be designed to achieve this and recommend remedial actions which can solve the performance problems. The rules will take into consideration the impact of workload arrival rate, service rate, the operating system and all application and system functions. They will also include the impact of transactions on each other and consider inter-device interface problems.

Our approach is to use Smalltalk to define the system architecture, including hardware connectivity, software functions, and the workload. We will decompose possible system problems into logical, yet independent groups. The groups are defined hierarchically so that simple problems are uncovered before attempting to diagnose those requiring more detailed analysis. A set of rules is developed using PROLOG to perform diagnostics, isolate problems and recommend corrective actions.

The major consequences of this paper will be:

a. That graphical performance modeling tools can be built using object oriented languages.

Furthermore, our test cases will show that this has the added advantage of producing highly accurate models in a fraction of the time required using the traditional approach.

b. That models and modeling activity can be moved from the exclusive domain of modelers and made readily available to system designers. This will make modeling an indispensible tool of the system design process.

2. TECHNICAL APPROACH

Hierarchical System Description

Computer systems are comprised of several types of hardware which have distinct properties but share several others. For example, two CPUs might have different operating systems (with different characteristics) but they share the property of speed. A natural way to represent this is by a hierarchy of hardware types, which can not be accommodated by most high order and simulation languages. We chose to use the Smalltalk language because its object oriented paradigm allows the implementation of a hierarchy of classes. In this paper, we use the hierarchy of hardware classes represented by Figure 1.

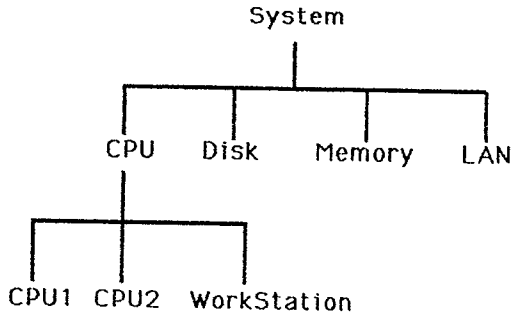

Figure 1. Hierarchy of Hardware Architecture

The principal feature of the class hierarchy is that any class inherits all the properties of its superclasses. For example, all CPUs have the properties of speed and memory. Thus, whenever a subclass of CPU is defined, the system assumes that it has these two properties. Each subclass may have properties which are unique and further define it. For example, operating system characteristics are properties that all CPUs have, yet their impact on response time is machine-dependent. Smalltalk has a powerful way of defining class properties called 'Methods'. A Method is a Smalltalk procedure which performs a predefined operation or defines a property on an instance of a class (object). For example, the statement,

```
icon computeUtilization
```

sends the object 'icon' the message 'computeUtilization' which invokes the Method for computing the utilization of that device. The Method computeUtilization must be defined in the class from which icon is defined, or one of its superclasses since each class also inherits the Methods of its superclasses. Thus, all CPU subclasses will inherit the Methods for accessing the properties speed and memory from class CPU. Similarly, by specifying 'compute operating system', the impact of the operating system overhead on a transaction response time is computed. The advantage of Smalltalk is that the application of this Method to a specific CPU invokes the unique procedure used to determine how the device's overhead is computed and affects response time. This facility can be used to implement operations specific to any class; e.g. utilization and response time calculations and setting of parameters. The beauty of this method is that a given procedure is executed by the receiving object according to its own internal definition. The hierarchy is useful because each hardware item in the system can be specified as an instance of the appropriate class. This makes the system specification extremely simple as each object in the actual system corresponds to an object (class instance) in the model. The use of class-specific Methods also makes the code defining it modular since each class can have several Methods, each comprised of a small number of lines of code. Hence, it is very easy to change and/or add a Method without affecting other Methods.

Graphical System Entry

Smalltalk's bit-mapped and high resolution graphics capability makes it ideal for defining a system using icons. It also offers the user various options using pop-up menus. This capability is further enhanced by the use of a mouse to select items (objects) and pop up menus. We use these capabilities to present the user with a three pane screen comprised of a system drawing pane, a hardware icon pallette and a function (software) list pane. To enter a system description, hardware items are selected from the hardware icon pallette by clicking on the desired icon, and depositing it on the system drawing pane. After an icon is deposited, the program prompts for its properties, e.g., operating system characteristics for CPUs and I/O characteristics for disk devices. The user is then prompted for the definition of the software architecture. The software architecture is specified by defining its functions. A function can be any piece of software. Each time a function is defined the user is prompted for its charateristics, including its host device. A function can be moved to another device just by redefining its host device. This provides the mapping of the software onto the hardware for use in the workload analysis and response time calculations. The software functions are then deposited on the function list pane.

Hardware connectivity is specified by selecting the 'connectNodes' choice from the main system menu and clicking on pairs of icons to be connected. Each pair specified is connected together on the screen and the connection is stored in an internal connection 'dictionary' for future reference during analysis. A fully populated screen is shown in Figure 2.

Other useful operations are also available through menu choices. In a distributed or parallel processing environment, a set of identical devices are used to attain higher efficiency and other performance improvements. This will require the replication of devices. Icon replication can be done by selecting the appropriate menu choice, clicking on the icon to be replicated and specifying the replicate's position and name. All other properties (including connectivity and operating characteristics) are then transferred automatically and the new icon is drawn. Other menu choices allow the capability of clicking on an icon and redrawing it at a new location, and to

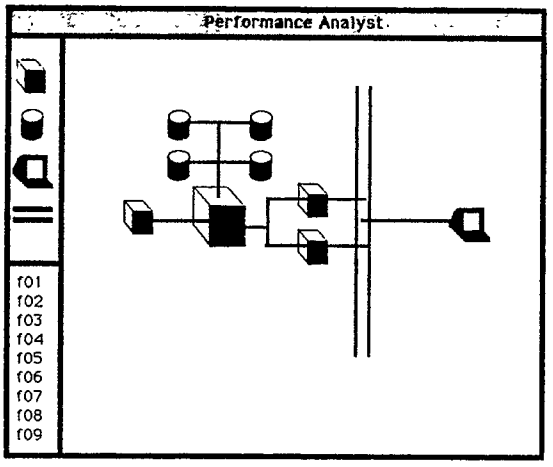

Figure 2. Hierarchy of Hardware Classes

inspect (and alter) the properties of any hardware item on the screen through the use of inspector windows. This feature enables the user to change the mapping of software to hardware from one analysis run to another. Inspectors allow direct access to a CPU's function dictionary so that any function can be deleted from a given device and added to another. Other properties can also be changed in this manner. Figure 3 shows an inspector window opened on the system's Central Processor.

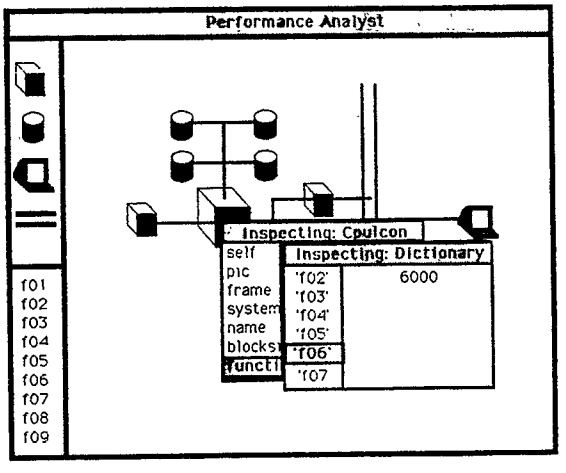

Figure 3. Inspector Window

Workload Description

Once the hardware and software architectures are defined, the user can define the workload. The software architecture is defined as a set of functions, each with a specified host device. The functions are displayed in the function list pane and are used to enter the workload component. As soon as the name of a workload component is specified, the user is prompted for general properties such as arrival rate, number of IO's, and number of database calls. The user can specify the service time by specifying which functions are called, how the functions are called (sequencing), and what percentage of each function is executed in support of the workload component (transaction). This feature enables the user to calculate both transaction response time and functional response time. Thus, a transaction response time failure can be isolated to a specific function in its processing sequence. All transactions and their properties are stored in a system transaction dictionary for use during analysis. A system menu choice allows the user to inspect the properties of any transaction in the dictionary.

Derivation of Performance Parameters

The system is designed to accept any method for the derivation of performance parameters, including analytical modeling or discrete event simulation. It can also accept precomputed performance parameters as output of other methods or packages. The diagnostic rules are designed to be independent of the analysis approach. Thus, the user can insert his own analysis method, or modify existing algorithms, and still take full advantage of the diagnostic capabilities. This modularity feature will make an eventual tool easy to use. For the purpose of expediency we have opted for analytical methods using M/M/m queues with First-Come-First-Served (FCFS) queue discipline.

Rules

We provide a set of PROLOG rules to diagnose system problems based on the predicted performance parameters. These rules were derived from experience gained in many previous modeling efforts including [5, . . .,12]. PROLOG uses a backward chaining paradigm which facilitates the implementation of a diagnostic rule base. A generic PROLOG rule (clause) is given below.

$$A :- B_1,B_2,\ldots,B_n$$

It says that the fact A at the head of the rule is true if the facts in the tail of the rule, $B_1,B_2,\ldots,B_n$ can be solved (are true). Each of the tail facts might also have their own associated rules. Hence, when the PROLOG program containing the above rule is queried to see if the fact A is true, it backward chains to each of the tail facts and tries to solve them. The fact A succeeds if all the tail facts also succeed. It is natural to express the diagnostic procedure for a computer system in this framework since any given diagnosis has associated causes. For example, a transaction will fail its performance requirement if its service time is greater than its response time requirement. Therefore, to verify the truth of a given diagnosis, the existence of the causes must first be established. This is easily expressed in PROLOG.

3. DESIGN OF THE EXPERIMENT

To test the feasibility of our ideas, we consider the following set up, taken from an actual operational system:

Hardware Architecture

The hardware architecture as depicted in Figure 2 is composed of the following components:

a. Front End Processor

The front end processor is assumed to be an IBM Series/1 computer with the EDX operating system. Its main function is to receive message packets from external sources, buffer them, and send the whole message to the Central Processor. It also receives messages addressed to external sources from the Central Processor, breaks them into packets and sends them outside the system.

b. Central Processor

The Central Processor is assumed to be an IBM mainframe with MVS operating system. The operating system also includes the following components:

- o Performance Measure to handle SMF file data gathering function
- o VTAM for block transaction handling
- o CICS for display handling

The Central Processor performs three major functions:

- o Receives message blocks from the front end processor and performs a series of operations.
- o Receives display requests from the workstations, performs the various operations depending on the type of requests, and sends back appropriate responses.
- o Prepares and sends messages to the front end processor for external destinations.

c. Internal Processors

The Internal Processors are also Series/1 computers and work in parallel. Their main function is to act as a message and data buffering medium between the Central Processor and the Workstations.

d. Local Area Network (LAN)

The LAN is the communication link between the workstations and the Internal Processors.

e. Workstations

The workstations are used for communication between the user and Central Processor. Messages addressed to the user are received at the workstation. The user composes messages, and sends queries to the Central Processor from the Workstations.

f. Disk

A disk system with four independent arms is connected to the Central Processor for data and other input/output purposes.

Software Architecture

The software architecture is defined by a set of functions with a default host processor. As we discussed earlier, the functions (or portion of them) can be rehosted to another processor. The major software functions are given in Table 1.

Workload Definition

The arrival and service rates on various devices are model parameters with default values specified herein. The service time of a transaction for multi-function devices such as the Central Processor is specified by the functions it invokes. Hence, the transaction service time is the sum of the service times of its functions.

Name	Function	Host	Timing (Sec)
F1	Block Handler	Front End	.130
F2	Message Processing	Central	.345
F3	Display Handler	Central	.300
F4	Security Interface	Central	.086
F5	Security Manager	Central	.213
F6	Data Base Manager	Central	.100
F7	Statistics Gather	Central	.015
F8	Interface	Central	.100
F9	Channel Interface	Internal	.070
F10	Physical Unit	Internal	.030
F11	X-25 Buffers	Internal	.050
F12	Display Handler	Workstation	.500

Table 1. Software Architecture

The traffic through the system has three sources with distinct characteristics and resource requirements. They are:

o Incoming messages

The incoming messages arrive to the system via the front end processor from external sources. The default arrival rate is 1000 per hour, and they have 60 blocks, and require 80 IOs and 80 database calls. The processing sequence per device is as follows:

External	F1
Central	F2—>F4—>F5—>F6—>F8

o Outgoing Messages

The outgoing messages are composed at the workstations, sent to the Central Processor for processing and to the External Processor for delivery to external destinations. Their rate is 2 per workstation per minute, and they have 12 blocks, and require 40 IOs and 40 data base calls. The processing sequence of functions is as follows:

Workstation	F12
Internal	F11—>F10—>F9
Central	F5—>F6—>F2—>F8
External	F1

o Display Requests

Four types of display requests originate from the workstations with default charateristics given as follows:

	Rate	Block	IO	DB calls
User account	300	1	5	5
Mail	180	8	5	5
Logon	180	1	3	3
Help	300	1	3	3

The processing sequence for all display requests is the same and is as follows:

Workstation	F12
Internal	F9—>F10—>F11
Central	F3—>F5—>F4—>F6—>F8

Analysis

The computed performance parameters are utilization, and average and percentile response times by function, device, and transaction. For this purpose we use M/M/m queues with a First-Come-First-Serve discipline. This will serve our purpose well, as we are interested in demonstrating feasibility of using object oriented languages as a model development environment. Other or more complex algorithms can easily be accommodated. We skip citing the actual formulas used as they appear in almost all standard books on the subject, including [1,3,4].

Rules

We provide a set of PROLOG rules to diagnose system performance problems based on the analysis described above. To expedite the feasibility study and keep the approach general enough, we have assumed that a given end-to-end response time has been allocated to various functions or devices, so that each function or device is supplied with a performance budget that it needs to meet. This approach is consistent with the common practice of decomposing a message response time requirement and allocating it to various system components. Thus, we limit our attention to individual functions and devices and assume that the failure of one function or device to meet its budget is tantamount to the transaction's failure to meet its total response time. We can, of course, reallocate the message end-to-end response time requirements and reset individual budgets. Therefore, we proceed to perform diagnostics on this basis, realizing that certain problems are system wide and will be detected only when the system is analyzed as a whole.

The diagnosis has two parts. First, a performance failure is identified. This can be a transaction failing its response time requirement or a device or a function exceeding its performance budget. Then, the problem is examined in order to classify it and find the cause for its failure . We have adopted a hierarchical approach to problem decomposition and isolation. Our approach is to take an increasingly deeper look into the problem. We are driven by the fact that in performing this kind of diagnosis, obvious and simple causes should be disposed of before initiating complex and time-consuming analyses. The modularity and the tree structure for classifying problems will allow us to expand the rules as they are developed. For transaction failures, we have identified three general types of performance problems, but the hierarchical approach allows us to add new ones later. The three types are depicted in Figure 4., and are as follows:

1. Transaction-dependent problems: These problems usually can be resolved by examining the transaction itself. Problems such as excessive service time, high number of data base calls, high number of IO's, and high arrival rate fall into this category. Some of these problems will be easy to detect, e.g., service time is larger than response time requirement. Very subtle problems can also happen. For example, a transaction may fail its required response time because system functions supporting it impose additional wait time.

2. Device-dependent problems: The response time failure is a system problem. These type of problems represent deeper causes of performance failure. Generally, we envision these problems to be independent of a failed transaction. For example, one component of the operating system may be a heavy resource user increasing the total function or device utilization, causing another transaction with low performance margin to fail its required response time. This, however, does not preclude being led back to the failed transaction itself and finding a subtle internal problem contributing to its failure.

3. Design Related Problems: These problems have more subtle causes and require reexamination of the system design concepts to solve them. Problems falling into this category are table locks, a transaction needing another transaction output, yet both calling the same function, and a transaction affecting another transaction's performance.

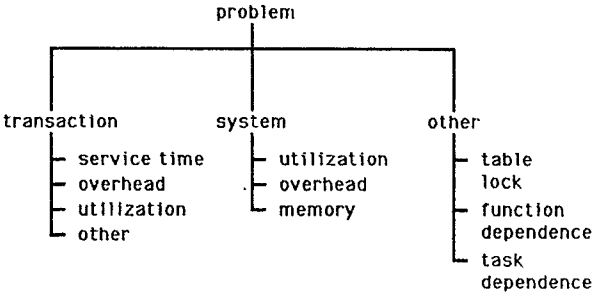

Figure 4. Types of Problems

There are three sets of PROLOG rules corresponding to each of these diagnosis types. If the problem cannot be isolated using these rules, the 'fall-back' diagnosis is that the transaction is using too much of the device.

PROLOG also includes a database feature which allows the user to assert facts and pose queries. There are two basic fact types in our database. The first describes the transactions:

> trans(name, arrival rate, utilization, actual response time, response time requirement, service time, effective service time)

The database includes one such fact for each transaction using the device in question. It also includes a single fact for this device (there is only one device in question):

> cpu(name, speed, average arrival rate, total utilization, operating system utilization, number of transactions using the device).

These facts allow the rules to pose queries about the system. For example, one rule checks to see if a transaction is a major resource user resulting in degradation of total device performance.

```
transHog(roT):- cpu(_,_,_,_,_,n),trans-
(name,_,roM,_,_,_,_), is(x, roT/n)gt(roM,x),
deepTransProb(name).
```

This rule evaluates to true if the number of transactions using the device in question is n AND there exists a transaction 'name' in the database with utilization roM AND roM > roT/n where roT is the maximum utilization threshold supported by the device. The last item in the rule chains to another set of rules to check for problems of type 3 above for this transaction. The underscore '_' is used to mark the place of a value which is not important to the rule.

After the rules have isolated a problem, the screen displays an appropriate message to inform the user. Only a single diagnosis is made for each problematic transaction, and hence, one message is displayed for each. Highlighting the probable cause of transaction response time failure in this fashion enables the user to focus the forthcoming effort to alleviate the problems.

4. TEST PLAN AND RESULTS

We have adopted a two phased incremental approach to testing the program. The purpose of testing is to ensure that individual modules of the program provide the intended results. Our approach, therefore, is to prepare test cases with known results to validate individual modules.

a. Algorithm Validation

The algorithm validation is accomplished by choosing test cases with known results, running them through the model and checking the model output against these results. The process is continued with proper correction until the two sets of data agree.

b. Rule Validation

The rule validation is accomplished by preparing a set of data known to have no performance problem and running it through the system to ensure that the program provides the desired system response. We then change individual parameters to invoke specific performance problems leading to a specific rule invocation. The system response is then compared against the known result to assure agreement.

The system in figure 2 was constructed and the workload was specified using the transactions described earlier. We computed the relevant performance parameters (response times and utilizations) and checked these against the results produced by the program's analysis routines. We then ran ten test cases to evaluate the rule base. The test results are summarized In Table 2.

Each of the above cases shows the change made to the basic data used to test the program's analysis routines along with the corresponding diagnosis provided by the rules. Cases 8 and 9 bear a strong resemblance but are slightly different. The rule base offers two paths to reach the same conclusion and the cases were created to test both paths. The main result of these cases is that the diagnoses are intuitively correct based on the causes. These were also checked with performance experts at AST.

5. Summary

o Conclusions

The prototype tool has indicated that object oriented languages in conjunction with a rule base (expert system) provide an ideal environment for developing performance models. Further, our experience has shown that the models developed using object oriented languages not only have features not available with the traditional approach, but also can be done in a fraction of the time. We developed the prototype tool with the extra features in only three weeks, while a similar model of the system developed in PASCAL required an order of magnitude increase in time and costs.

An even more far reaching consequence of this paper is the possibility of building performance models in a radically new way. To see how this can be accomplished, we need to discuss two limitations of performance models in addition to the ones discussed earlier:

1. The development of a performance model always requires moderate to extensive programming, most of which is application specific with very little re-usability. This is true whether one uses a modeling package, a simulation language, or a standard programming language.

2. Almost all models are 'hard-wired', in the sense that the flexiblity and the reusability of the models are limited by the language used and the foresight of the modeler. For example, it is very difficult to develop models of supercomputers using the present day approaches, as the operational concepts of the supercomputers are dynamic and hardware dependent.

Case	Action	Diagnosis
0	- (base data presented above)	no problem.
1	raise service time of logon	logon fails due to service time>response time requirement
2	raise # IO's of logon	logon fails because effective service time > response time requirement
3	raise arrival rate of incoming	incoming fails due to high arrival rate
4	raise number of blocks of incoming	incoming fails due to high ratio of overhead to service time
5	raise service time of all transactions	outgoing fails due to high device utilization
6	raise service time of outgoing and arrival rate and number of blocks of incoming	outgoing fails due to high arrival rate and overhead of incoming
7	raise overhead of all transactions	outgoing fails due to unacceptable wait time
8	slightly raise service time and number of # blocks of transaction incoming	incoming fails due to excessive use of the device
9	same as (8) for transaction mail with different amounts of change	mail fails due to excessive use of

Table 2. Results of Test Cases