Experiments in software modeling

by D. FOX
Fox Computer Services
New York, New York
and
J. L. KESSLER
IBM Corporation
Poughkeepsie, New York

INTRODUCTION

The objectives of a software support package for any computer system can quite readily be defined by the potential user of such a system. He will quickly point out that he wants the system to provide all the features and functions that he requires and that it should occupy minimal storage space and consume minimal time. Obviously, the designer of a software support system faced with these impossible objectives finds his life to be one of constant decision making as he comes up with the compromises that give the best possible approach to this solution.

In this paper, we will concentrate on techniques to aid in one aspect of that decision making: the effect the inclusion of a given feature has on the performance characteristics (time and space) of the final hardware/software systems. Indeed, what is the effect on performance of particular design alternatives that one might choose in implementing these features?

In the past, these decisions have been made purely on an ad hoc basis. Experienced system programmers have dashed off kernels of the code representing the final implementation, then extrapolated from these what they thought would be the size and speed of the final product. With sequential processors and with software systems implemented by small groups (where each individual understood in detail the operations of the complete software package), such an approach was acceptable. However, with the increasing degree of parallelism allowed by our more recent computer system designs, and with the exploitation of that design by systems programmers, performance prediction has become a problem of almost overwhelming complexity.

Only in the last few years have hardware measuring and monitoring techniques permitted the detailed study of performance characteristics of such systems after they were in operation. However, postponement of the availability of any performance data until a checked-out hardware and software system is ready for measurement can be catastrophic. Too many irrevocable software-design decisions will already have been made. Thus, it seems obvious that some technique for predicting the performance of systems programs and for evaluating the impact of various design alternatives on that performance must be considered.

For the past three years, a group within IBM Programming Systems has been working in cooperation with Fox Computer Services to study and develop techniques that will permit the prediction of performance characteristics of such systems under a variety of hardware environments. In addition to providing preliminary design analysis, such a performance model can permit the study of comparative configurations, aid in the preliminary analysis of proposed input/output devices, and can aid the designer in optimizing performance within a given configuration.

Within IBM, earlier attempts in the direction of systems modeling emphasized the hardware aspects of modeling. In particular, major importance was attached to central processor cycle times, peak transmission rates, and so on. The primary objectives were to determine how much core storage, how much direct-access device capacity, and how many lines, terminals, and channels were going to be needed to do the proposed job. The software, if it was described in any detail at all, was generally represented as a series of queueing algorithms describing how the terminals were to be polled, how the accesses to the mass storage devices were to be queued, and so on. It became apparent, however, that in order to provide a useful software support package, a radical departure from this total hardware orientation must be made.

Requirements

Now, let's consider just what is required of a set of techniques and tools for aiding the software designer in this way. Looking at any of our major systems today, we realize that the effective use of hardware is largely under the control of the software. So, primary importance should be attached to the software model. This is not to be interpreted as a lack of concern for the hardware performance; rather, it is a realization that the efficiency with which the software utilizes the potential of the hardware is a major factor in determining total system performance.

First, it became obvious that we needed an integrated model of the complete hardware and software sytem. We found that, although some subsystem modeling (for example, modeling only a storage-location algorithm or only a disk-queuing algorithm) can give considerable aid to a designer in a specific area, subsystem models taken alone and not evaluated against the design of the total system often give extremely misleading results. Some subsystem modeling is of value, but final decisions can only be made after the incorporation of that subsystem model into the total system model.

Secondly, if the model is to be constructed by (and used by) software designers, its structure should be meaningful to them. That is, the model should appear to have the same logical structure as the software being modeled.

A third requirement is that a model, or collection of models, must be highly parameterized. That is, the designer must be able to change the entire environment or parts of the environment without recompiling or reprocessing the entire model. For example, hardware characteristics and configurations should be readily changeable, as should the description of the layout of data on the modeled devices. But, all this parameterization and flexibility leads immediately towards the requirement for large volumes of data to describe the environment in which the model is to run. To free the user who is not interested in all this flexibility from having to provide such a great amount of data, a method of storing default cases and readily accessible standard descriptions of this environment is required.

Fourth, a very important requirement of such a system is brought about by its intended use as a design tool. If a designer is to gain anything from such a tool, he must have it at his disposal at all times. This implies that the designer is the modeler. In a large operating system environment, where many groups are designing and implementing components, this implies that there are a great many modelers. Their models are interrelated and have communication problems just as their component counterparts in the real system have communication problems. So, the need for modeling techniques capable of combining small modular units with minimal interface problems is evident.

Fifth, if the modeling system is to be used as the design tool and used effectively, it must be run constantly, many times per day. It must be detailed, for to influence design it must closely parallel the logical design of the system. This implies that we must have an extremely detailed model that runs very fast.

The sixth goal is one that developed out of early experiments, when we discovered that the modeling technique must be able to encompass all potential hardware and software systems that a designer might be interested in. Specifically, it must be able to handle multiple central processors (not necessarily homogeneous in performance characteristics), handle multiple memories (not necessarily homogeneous in their characteristics), be accessible from a number of processors, and must be able to simulate multiprogramming activity within the central processors.

The first thing one discovers in looking at a set of results from a simulation effort is that some of the results trigger an interest in statistics that do not appear on the first report. Often, simulation must be redone, or at least rerun, to produce the results that are now of interest, and, again, the second set of results probably leads to a desire for a third and a fourth, and so on. Therefore, the seventh requirement is that the modeling techniques must allow, first, that such additional data can be acquired and, ideally, that it be acquired with a minimum of model rerun time.

Design criteria

The approach to modeling a total hardware/software configuration for the purpose of design support and evaluation must differ from previous simulation efforts in several important respects. It is software-oriented and makes no simplifying or averaging assumptions about resource demand. Modules of the subject system are reflected as modules in the model, so that the precise effect of module replacement can be calculated by inclusion of the new module in the existing model.

Since we wish to model the total system and, at the same time, reflect each significant module of the subject system, it is clear that the modeling language must emphasize subroutine capability with explicit and easily described interface procedures. This approach eliminates a difficulty that has plagued modeling systems up to now—the requirement that a modeler who is interested in only one part of the system understand and simulate the rest of the system and its environment before he is able to get any results about his particular part. Our techniques permit modular construction of the model, with an emphasis on explicit interface design

and evaluation. System design changes can be isolated and their effects evaluated without changing any other parts of the model.

In order to study the behavior of the software models under a variety of conditions, one must be able to vary both the hardware and software environment in which the model is executed as well as change the simulated work load. Our techniques provide for an input language to describe these environmental considerations and permit the model easy access to the descriptions.

A word or two about choice of a simulator would seem appropriate at this point.[2] Ours was not the first attempt at total system simulation, and it is important to understand the reasons for the rejection of previously used, more standard simulation techniques and languages. We needed a tool for designers and design evaluators. Therefore, the language must follow as closely as possible the methodology of systems designers and analysts. This is accomplished by isolating software events and bringing to the fore the obvious characteristic of software design—that it is composed of modules of logic, each of which is designed to operate sequentially, but whose interaction may be sequential or parallel, synchronous or asynchronous, consequential or logically independent. Another constraint on our design was an historical rather than a technical one. It was not within the scope of the project to implement a new compiler or translator, nor to modify an existing one in any way. In addition, we were aiming towards a technique which would permit us to perform simulation experiments using System/360.

A study of existing simulation techniques led us to the conclusion that, in terms of scope, capacity and performance, existing simulators would have severely limited the size of the system we could model, the length of model runs which could be achieved economically, and the level of detail to which systems could be studied. But the most important characteristic that appeared lacking in existing models was modularity. Our choice, then, was to outfit FORTRAN with mechanisms for implementing simulation models via CALLs, because it offered a most flexible program and subroutine linking structure that was known and understood almost universally among designers.

Evolution (experiments)

Now, let's consider how the need for specific techniques evolved from some of our experiments with software models. One of the earliest was the modeling of the IBM 7010 operating system, using a GPSS III[3,4] This was merely a first attempt to answer two basic questions:

1. Can Programming Systems applications be modeled successfully?
2. What are the problems involved with simulating systems programs, and how would one define a good simulation system to solve this problem?

The result of this experiment was an indication that such techniques could indeed provide promising results. However, the difficulties involved in producing the right kind of model were great, and the process was a very expensive one.* It was evident that much more experimentation and exploration had to be done before we really understood the software modeling problem.

Some modeling was also done in a special-purpose computer simulation language,[1] which proved to be easier to use but suffered from many of the same difficulties found in the GPSS approach. Following this, some of the earlier designs for Operating System/360 language processors, Assembler, COBOL, and FORTRAN, were modeled using a GPSS derivative. These early models did produce reasonable performance predictions. The developers, however, felt that the assistance they were getting from the model was not sufficient to warrant the effort to maintain it.

As these models were being developed toward the end of 1964, Robert Ruthrauff of IBM Programming Systems began to experiment with performance-prediction models that emphasized to a great extent the queuing techniques employed by the input/output supervisor of OS/360. He coded these early performance models in FORTRAN, choosing it primarily because of the familiarity of the programming community with the FORTRAN language. Parallel to this effort, Fox Computer Services had been engaged in producing a performance model for IBM of the IBM 7090 FORTRAN compiler, to be used in evaluating design alternatives that were available to implement that processor. It soon became evident that the fundamental approach was identical, and that much could be gained by combining these efforts in an attempt to develop the more comprehensive set of techniques that were needed to solve the software modeling problem.

By this time, Mr. Ruthrauff's I/O model had undergone a series of six revisions. As an experimental

* Some of the reasons cited in the summary of this project appear below:

1. A model of sufficient detail to be useful was too large and too slow.
2. Techniques of model writing required to represent certain hardware software events and activities in these languages were disagreeable to software designers. (The logic of the model did not "look like" the logic of the modeled program.) Also output traces of events did not follow software events.
3. No dynamic input capability.
4. No model overlay capability.

vehicle, Fox Computer Services and IBM combined to produce another model as an attempt to describe the design of the OS/360 operating in the Primary Control Program environment. Once again, the modeling was done, not by the designers, but by an outside group. Once again, the design of the model did not closely approximate the design of the system. However, a number of things were learned:

1. That a model could be created that would provide valuable performance data to the designer.*
2. That the input/output activity could be closely approximated, and
3. That the particular implementation did not provide enough flexibility for modifying either the hardware or software environment in which various pieces of the model were running.

Thus, in the next two experiments, the OS/360 model was upgraded and further techniques developed. By early 1966, the last of this series of models was completed. The model included a limited multitasking capability, SPOOLING, priority scheduling, and the FORTRAN E and H and COBOL F processors of OS/360. At the close of the OS/360 modeling experiment, two studies were made:

1. To discover the level of accuracy to which such a model could be calibrated.
2. To determine how effective we had been in parameterizing the models. In other words, how much variation in the behavior of the modeled system and in the environment in which those models were executed could be made by merely changing control card values, and how valuable were the results that could be obtained by making such minor variations and reruns?

The validation and calibration experiment proved to be a great success. A set of simulated job streams was run on a series of three System/360 configurations embodying different central processors and different input/output configurations. The results were measured with hardware monitoring devices. The same set of simulated job streams was passed against the model. By varying the input parameters governing the times various portions of the model consumed, the deviation of the model from the actual execution of the operating system was less than 5% in any area studied. The model was validated to the level of overall processor time, phase times and major input/output requests. This included the fetching of program phases, the opening and closing of data files, and all input/output requests, and accounting for both device time and central processor time necessary to initiate the input/output operation. For any one configuration, the results could be readily calibrated to yield zero deviation. The variation caused by moving from one configuration to another could be traced to certain processor phases where assumptions about the consistency of instruction mixes across all system programs was obviously incorrect.

The second study, although educational, did not indicate as much success. We found that very little variation in the modeled system's behavior could be made by varying control-card parameters alone. Almost any experiment of interest required a change in the code of at least one of the models.

By this time, we felt that we were beginning to understand what techniques would be required to allow designers of software systems (and components of those systems) to use modeling effectively as a tool for predicting the impact of design alternatives on their final product. From this point, we set off to develop a set of subprograms that would provide the software modeler with the type of functions our experiments indicated were required. It is not a new simulation language or even a simulation system. The user is free to write his own simulation system using the set of subroutines to perform simulation-unique (and software-simulation-unique) functions. While this does place some extra burden on the modeler, he is not hampered by any built-in restrictions or omitted features.

Evolution (techniques)

Having chosen FORTRAN, then, our job was to outfit the software modeler with "CALL"-able mechanisms for the construction of models which would emphasize the events and methodology of total hardware/software system analysis. Again, it must be made quite clear that modification of the FORTRAN language or its compiler was not permitted.

The first mechanism developed was called the Clockworks, whose function was to queue models in time, keep track of events, and pass control to the appropriate submodel when its event time became due. In its simplest form, the Clockworks would accept requests for synchronous delays (in which the requesting model would be suspended until the delay was effected) and asynchronous delays (in which the calling model could continue without suspension while the scheduled delay, when its time completed, could cause the activation of a second model, in parallel with the first).

The scope of the systems to be studied with these techniques includes the entire gamut of hardware/software systems under past, present, or future development. This means that the event-scheduling mechanism must be able to handle multiple CPU's, multiple and

* This model was actually used in support of some design changes which were later incorporated into the OS/360 FORTRAN H compiler. It showed the performance advantages of a certain input buffering/blocking scheme and predicted the advantages of the multiple compile capability.

parallel tasks, flexible hardware configuration descriptions, resource management, task scheduling, and task dispatching. Consequently, the Clockworks mechanism must be open-ended as to the number of tasks to be executed in parallel and the number of future events to be scheduled. This flexibility is achieved by using list processing techniques and dynamic storage allocation for internal Clockworks operation.

This description of the requirements on the time-handling mechanism does not differ markedly from similar mechanisms available in other modeling systems. There are, however, at least two features which are peculiar to hardware/software system modeling. The first is that time itself may not be absolute. That is, a given task may request a resource for a fixed amount of time, but the actual amount of resource time required to satisfy the request may have to be increased because the resource's effective performance is degraded by shared utilization. In particular, a program designed to complete in 100 microseconds on a give CPU-memory combination may actually consume 125 microseconds because concurrent channel activity is "stealing" 20% of the cycle times for data fetch and store activities, which are not related to or known to the requesting program.

Another requirement on this Clockworks is the ability of a single subprogram to represent a model that may be operating at the same instant of simulated time on more than one set of data, in different modeled environments, and in different states of completion. This requirement, known as the reentrability attribute of the model, led us to design a set of mechanisms called the Model Transfer Mechanism, to help the Clockworks relieve the user of the necessity of including logic to test the current state of the system or his model.

Whenever the Clockworks transfers control to a model, it sees to it that all status information and registers are updated in accordance with the particular usage of that model. Thus, for example, at the same instant of simulated time, a model of a compiler could be processing the second input card of deck A on CPU 1, and the same model could be in the middle of processing the tenth input card of deck B on CPU 2. Whenever a request for time on CPU 1 completes, the model is given control in such a way that it simply continues, without any logic to test in which usage it is involved (Figure 1). This is because the CPU number, as well as the pointers to deck A and card 2, are placed in the same storage locations where CPU number 2, deck B, and card 10, respectively, will be placed upon completion of a time request on the second CPU. That is, each usage of the model effectively operates upon the identical variables, which differ only in their values.

The basic unit of data around which the Model

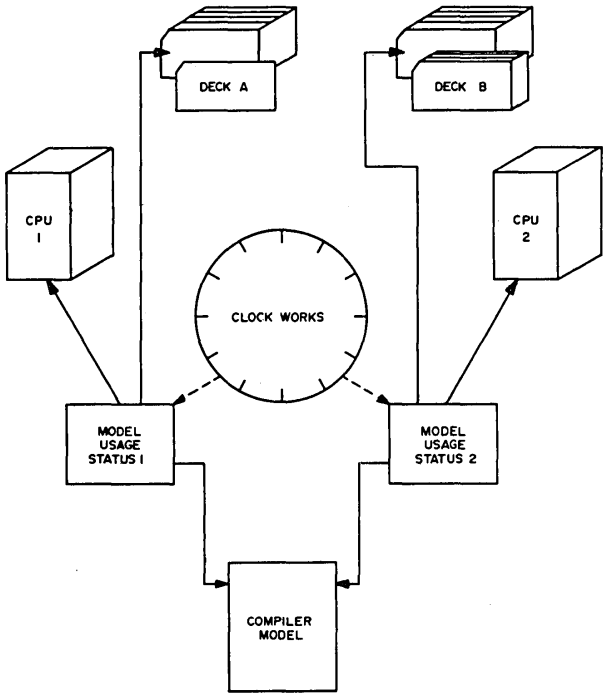

Figure 1—Relationship of clockworks to model usage

Transfer Mechanism operates is the Model Block, or MB. There is one MB for each model usage. The MB serves as a repository for all data and status information required to restore control to a model after its operation has been interrupted or temporarily discontinued. Among the information stored in the MB or, in the case of variable-length information, pointed to by the MB, are:

1. All System/360 general registers, including the address of the next instruction in the component model, usually written in FORTRAN. This is the return point for model resumption;
2. All intermediate or temporary storage currently being used by the model;
3. Call arguments or input parameters to the model; and
4. Return information for model completion.

Every model references data and is controlled only through its Model Block. This permits the Clockworks (via the Model Transfer Mechanism) to restore control to a model by pointing to its currently operative MB. All data (such as card number and deck number in our earlier example) are automatically referenced correctly because the MB is the focal point for such reference.

Until now, we have been using such terms as "task," "model," and "CPU" as if their meaning in a model

were self-evident. Actually, this is far from the case. Our approach implies a specific interpretation of these terms, and it is important to understand these meanings.

It was pointed out earlier that software design consists of modules of sequential logic, each of which can operate as a unit in parallel with others. The term "task" denotes such a unit. It is actually a handle (called the Task Block, or TB) on which is hung descriptions of model operations and events that operate sequentially. Thus, the task is the largest unit of sequential logic in this approach (See Figure 2).

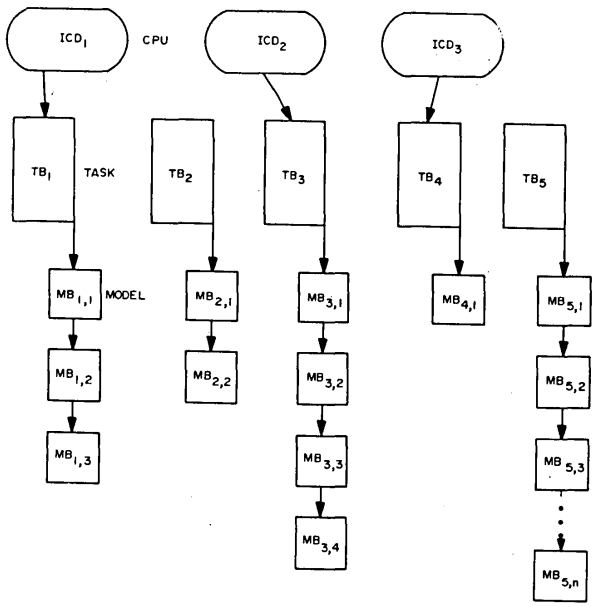

Figure 2—Independent control domains and related tasks

The CPU, on the other hand, is a special case of a more general modeling concept, the Independent Control Domain, or ICD. An ICD is an environment in which tasks operate. It has two properties: first, each ICD can contain a task in operation, regardless of task operations in any other ICD; second, no more than one task can operate in an ICD at any one time. A CPU is an ICD insofar as task seizure of a Central Processing Unit implies that no other task can use that Central Processing Unit at the same time. On the other hand, a multiprocessing system permits one task to operate in each CPU concurrently. (Of course, Central Processing Units that do permit multiple, simultaneous instruction sequences would be modeled as multiple ICDs.)

Another kind of ICD that can arise in systems where operator or user interaction is important in evaluation is the operator domain. This ICD can be a conceptualization of the human part of the system, which presumably will be occupied with just one task at a time.

A "model" is a smaller unit of sequential processing, usually a single subroutine that corresponds to a load module in the subject system. Tasks usually consist of several models invoked either sequentially or in push-down fashion, with the model invoked for the task returning to its predecessor upon completion. Each active task has exactly one model currently operative at any time; it is called the "top" model for the task.

Several other mechanisms are available to the software/hardware system modeler, but we will only be able to touch upon them briefly. The first is dynamic storage allocation, called the WORK mechanism. This consists of a single array of COMMON storage named WORK. Both users and mechanisms can get and free WORK blocks as needed. Addresses and pointers used by, and passed between, models are always understood to be subscripts of the WORK array. We have written a general List Processing Mechanism that is used by both our modeling subroutines themselves and by their users. This mechanism constructs lists in WORK.

Finally, we have provided generic input and output capabilities that allow the user to define multiple input streams, to request statistical as well as snapshot and trace output, and to read and write table or data overflow onto direct access data sets.

To illustrate the use of some of these mechanisms, let us follow the action of a typical model statement. Suppose that the top model of a task is a compiler that wishes to invoke a supervisor function, such as opening a file. The compiler model would

CALL SVC ('OPEN ', arguments to OPEN)

Control is passed to the OPEN model by calling the model transfer service "SVC" to mimic the System/360 supervisor call interruption. Since OPEN itself is a model, it would operate under an MB. Prior to the OPEN CALL, the MB list of this task is simply as shown in Figure 3.

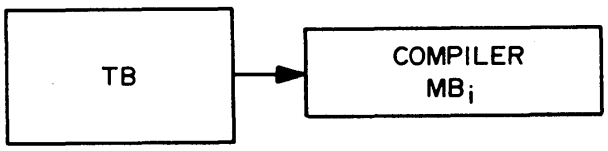

Figure 3—MB list prior to OPEN CALL

After the call, the processor MB is pushed down and OFTEN becomes the top model as in Figure 4. Notice that for as long as the OPEN model is in use, it constitutes the operation of this task, and its events are the events of the task. As soons a it issues a

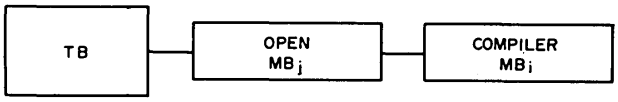

Figure 4—MB list following OPEN CALL

RETURN

the list is restored as shown in Figure 5 and the processor model resumes at the model statement just beyond the call for OPEN (this location having been saved in registers in the MB). Any data or intermediate variables used by the processor are attached to its MB and, therefore, will be undisturbed by OPEN's operation.

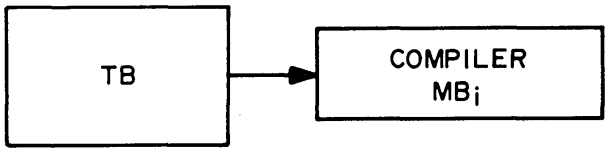

Figure 5—MB list following RETURN from OPEN

Structure of a simulation

The structure of any system can generally be described as consisting of input/output processing components. Simulation is no different. In particular, for ease of implementation and use, our software modeling package is divided into three processors: the setup or preprocessor, the execution processor, and the postprocessor.

The preprocessor digests the data which describe: the hardware device characteristic and hardware configuration, the structure of data on the external storage devices, the simulated job streams* and, finally, the parameterizations of the models themselves.

Since we allow considerable flexibility in the definition of this environment, the amount of data needed as input is very large. To free the user of this software modeling system from providing all this data, standard definitions and default cases are cataloged and put at his disposal. For instance, complete hardware configurations are cataloged so that he can identify them with only one card and invoke all the data defining the devices and the connections of the devices in that configuration. Similarly, standard jobstreams, data set layouts, and model descriptions are also cataloged. The preprocessor extracts this environmental information from libraries as ordered by the user and sets up tables and temporary data sets that communicate this information to the rest of the simulation. In addition, the preprocessor accesses a large library of models, selecting those that describe a specified operating system environment, and adds or modifies models as requested by the user. The output of this processor is a load module, ready for execution, and a set of input data ready to drive the created model.

The execution processor is made up of all the software modeling mechanisms, plus models selected from the library and models introduced by the user for this particular simulation run. The output of this phase is a standard statistical summary, plus a log of simulation events. The standard statistical summary presents, first of all, the amount of time that the central processor was controlled by each of the various tasks, as well as other information about the status of each task. In addition, statistics are maintained on the utilization of all data sets, channels, control units, and devices, and the amount of contention for these entities. For direct-access devices, the amount of time lost due to rotational delays and positioning is also provided. Finally, model usage is summarized. Such data as the amount of time that each model was in control of the modeled system and the number of times each model was used, are recorded. Various levels of logging are available to the user at execution time; indeed, within his own models, he may elect to record an event on the log at any time he chooses.

The postprocessor is really a report generator for processing this event log in a variety of ways. The user may elect to get a complete flow trace of any or all models through the postprocessor. He may be interested in summary or subsummary totals accumulated about selected events on the log. The postprocessor permits generation of many types of reports and, in addition, includes the capability of adding user routines at a number of exit points so that the report capability may be extended by the user in a specialized way, if he so desires.

Summary of techniques

This has been a brief look at some of the important features of an approach in which the total model consists of small modular components. The features are recapitulated here:

1. Total hardware/software modeling with emphasis on software language, events, and characteristics.
2. Explicit modules with explicit interfaces following the structure of the system as closely as possible.
3. Flexible, modeler-defined input/output facilities.
4. Dynamic storage allocation and reallocation.
5. List processing.

* Description of the work load to be handled by the modeled system.

6. Time processing, including dynamic degradation reflecting hardware performance degradations.
7. Wide scope of subject systems.
8. Mechanism to permit reentrant modeling with standard subprograms.

Status

Most of the techniques and mechanisms we have described have been implemented. We certainly do not believe that we have solved all technical problems, or perhaps even scratched the surface. However, the authors are convinced that a tool of this nature is desirable to control performance characteristics during the software design process.

REFERENCES

1 P H SEAMAN
On teleprocessing system design Part IV the role of digital simulation
IBM Systems Journal 5 No 3 175-189 1966

2 D TEICHROW J F LUBIN
Computer simulation-discussion of the technique & comparison of language
Communications of the ACM Vol 9 No 10 723-741 1966

3 R E EFFRON G GORDON
A general purpose digital simulator and examples of its application
IBM Systems Journal 3 No 1 22-34 1964

4 H HERSCOVITCH T H SCHNEIDER
GPSS III—an expanded general purpose simulator
IBM Systems Journal 4 No 3 174-183 1965