The design of a meta-system*

by A. S. NOETZEL

The Moore School of Electrical Engineering of the University of Pennsylvania
Philadelphia, Pennsylvania

INTRODUCTION

The design and implementation of multiprogrammed, time-sharing computer systems continues long after the system is put to use. A tool is needed that will measure and evaluate the computer system while it is in operation, as an aid to further development or optimization for a particular usage. Research into the possibility of developing this tool was undertaken at the University of Pennsylvania's Moore School of Electrical Engineering. The research led to the design of the tool, which is presented in this report. It is called the *Meta-system.*

The uniqueness of the Meta-system is due to the coalescing of two widely used techniques—on-line measurement, and simulation—into one system. Measurement is performed by extracting raw representations of a computer system's operation (from that system) using software techniques only. Evaluation of the system is based on input of the measured performance characteristics to a simulation model that exercises modified hardware-software versions of the system. All the potential modifications to the system are evaluated in the context of the task load of the system, as extracted from the operational system.

PRELIMINARY DESIGN

Because of the novelty of the Meta-system, the description of the system will be preceded by a discussion of the design requirements of the system, and of the capabilities and limitations of various design alternatives. This should also make clear the area of applicability of the Meta-system.

The Meta-system was first conceived of as a feedback loop on an operating computer system, in which the functions of measurement, evaluation, and modification take place. These functions are discussed in the following paragraphs.

The measurement function of the Meta-system

The study of measurement techniques of operational systems resulted in the following set of requirements for the measurement function of the Meta-system:

1. It should be implemented by software techniques. The recognition and measurement of the logical or decision-making functions of the operating system will require decision-making capabilities in the measurement devices. Also, the measurement device must be capable of handling a variety of measurements and conditions of operation. A software device is therefore indicated. To avoid the expense of an additional processor, the measurement software will be multiprogrammed with the system being measured.
2. It should introduce little artifact.
3. It should record all information of interest. The complete specification of the information of interest will not be achieved until the entire Meta-system, including the system modifications to be evaluated, is specified.
4. It should be amenable to flexible off-line analysis (i.e., information must be detailed).
5. It should be flexible, so that the same general approach could accommodate new and more specific areas of investigation.

The choice of the measurement that meets all of these constraints is the *event trace.* This will be more precisely defined later. For the moment, the event trace is a record of the important occurrences in a computer systems operation. These occurrences are interrupts, activations of particular hardware devices within the system, or

* The work reported here was partially supported by the Information Systems Branch of the Office of Naval Research (Contract N0014-67-A-0216-0014).

calls on significant subroutines. The event trace is recorded at various points in the operating system by writing a small amount of data specifying the nature of the event, and the time at which the event occurs, into a buffer area in main memory. The buffer is written to external storage whenever it becomes full.

If the events constituting the event trace are properly defined, the event trace may be an extensively detailed record of the system's operation. It is, however, raw data, and will be processed by the evaluation function of the Meta-system.

The evaluation function of the Meta-system

Ultimately, the evaluation of the computer system will be provided by a system designer's response to the parameters of system performance obtained by measurement. And his response will depend upon the options available to him—which are determined by economic and political considerations. In order for the Meta-system to complete the evaluation of a system, it must include the system designer. Henceforth, 'evaluation function' will refer to the automatic part of evaluation—the processing of measurement data to be more helpful to the system designer.

The following possibilities for the evaluation function of the Meta-system are apparent:

1. If the event trace is a record of the time of the beginnings and terminations of each interaction of the user tasks, the evaluation function may condense this data to obtain the response time distributions for the user tasks.
2. If the event trace is a record of the activations and deactivations of the hardware devices, the evaluation function may condense this data to create a record of the utilization factor of each device.

In either of these cases, the actual evaluation will be obtained by comparing the condensed data with some standard. Since the purpose of the evaulation is to determine the modifications that will improve the system's performance, the standard for evaluation should be the same data taken from variant versions of the system, especially modified versions in which performance might be improved.

The evaluation function should satisfy these two requirements: (1) It should enable the system designers to identify potential performance improvements to the system, and (2) it should indicate the performance of the modified systems without the expense of performing the modifications.

The Meta-system described thus far has the form indicated in Figure 1. The evaluation function contains a 'trial modification' loop in which measurements of variant versions of the system are obtained. Another possibility for the evaluation function suggests itself:

3. The evaluation function may be a simulation model of the system, or modified versions of the system. Condensed data, such as (1) and (2) above, may be obtained from the model.

Selection of the simulation model as the evaluation function imposes special requirements on the measurement function. Consider the measurements of one system that are useful for simulating another system. The measurements must not be the final results, such as the utilization factors of various hardware components, since these will be obtained from the simulation model. Rather, they will be measurements that can be interpreted by the simulation model—*frequencies* of occurrence of the operations, for example. The simulation model may then allocate a different time interval for each operation, and different utilization factors will be obtained. New resource allocation techniques, as well as other system algorithms that influence the resource allocation, may be investigated in the simulation model.

The measurements taken from the operational system will therefore be measurements of the user task *demand* for various system resources, which is related to the *allocation* of the resources through the operation of the system.

Next, it must be noted that a task's demand for system hardware resources cannot be represented independently of the system on which the task is run, for two reasons. First, because the hardware resources (as well as the other resources—macros, algorithms,

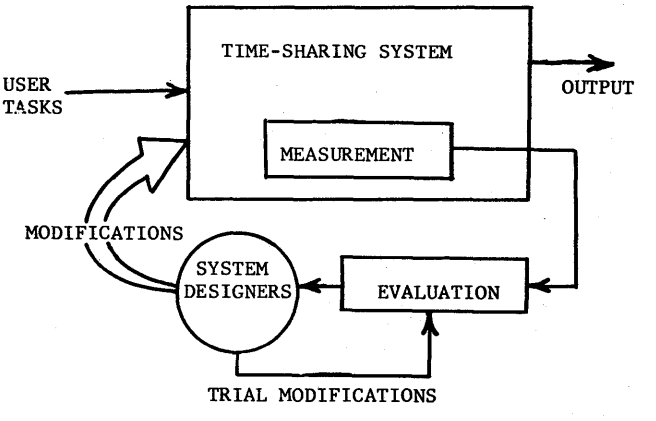


Figure 1—The Meta-system

tables, etc.) vary from system to system. Second, because it is the system as well as the user that generates the demand. Only at the highest level of demand specification—the level of machine-independent languages—is the demand purely due to the user. But even in this case, the demand for the execution of a program written in a high-level language cannot be correlated in a system-independent fashion, to the demand for hardware resources.

The result that is important to the theory of execution-simulation, is that it is possible to find representations of user task demand that are *relatively* independent of the system on which the task was run.

Relatively independent demand representations are representations which remain valid for a system that is within a specified class of modifications of the system from which the representations were taken. One concrete example may help make the concept of relative independence clearer. A task may be represented as a series of I/O operations. The number and frequency of the I/O operations are functions of the size of the data block that is involved in a single I/O operation. Then the sequence of demands for the I/O operations is a valid representation of the task's demand in every system which has the same block size in its external storage. It will remain valid even if the speed of the I/O device is changed, or if the configuration of the system or the scheduling of the device is altered, changing the wait time for the I/O operations.

It has been found possible to extract representations of user task demand for system resources from several different levels of operation. In each case, it is necessary to preprocess the event trace as taken from the system, before using the trace in the simulator. The preprocessor and the remainder of the Meta-system is described in the next section.

OVERVIEW OF THE META-SYSTEM

The Meta-system that was developed in detail was designed for operation on a hypothetized system which has the characteristics of three large time-sharing systems—TSS on IBM 360, TSOS on the RCA Spectra 70/46, and the Multics system on the GE 645. Specifically, it includes the features of recursive and reentrant operating system routines, demand paging, multiprogramming, multiple-I/O paths, multitasking and a Virtual Access Method. Reference to the details of the computer system in this overview of the Meta-system will be reference to the common, and commonly-known features of these systems.

An outline of the Meta-system is shown in Figure 2. The three parts of the Meta-system—the recording

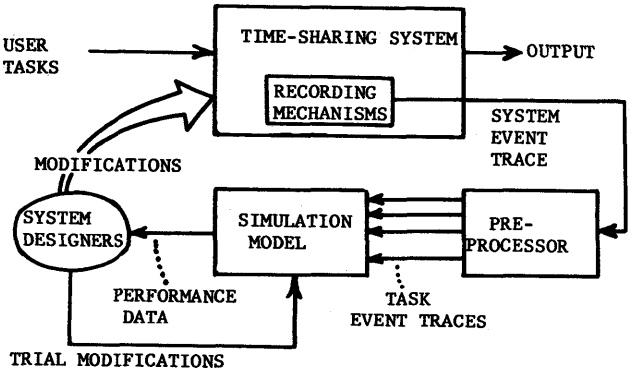


Figure 2—The developed Meta-system

mechanisms, the preprocessor, and the simulation model—are described in the following paragraphs:

The recording mechanisms

Measurement of the system's operation is performed by small open subroutines embedded within the operating system. The subroutines record the significant events in the system's operation. An event is composed of the following data items:

1. the *time*
2. an identification of the task (*task number*) for which the event occurred
3. the identification of the *type* of event
4. *data* associated with the event

A few examples of event types are the following: (In each case the *time* and *type* are recorded. *Task number* is recorded for each event type except 'idle.' The *data* field may or may not be recorded, depending upon the event type.)

> The 'on' event, signifying a task gaining control of the processor. No data is associated with this event.
>
> The 'idle' event indicating the beginning of a processor idle period. No task number or data need be recorded with this event.
>
> The 'I/O-req' event, which is recorded when a task (or a system routine) requests a physical I/O operation. The data associated with this event is a unique representation of the physical address (e.g., device, cylinder, track) involved in the operation. An 'I/O-req' event is not synonymous with the initiation of an I/O device because the system may delay the actual operation.

The page fault of 'pf' event, indicating the necessity for a demand paging operation. The data, in this case, is the virtual address of the page required.

The event of a request for a Logical I/O operation, or 'LI/O.' This event is recorded when a task calls on a system routine to perform a Logical I/O operation. The data associated with this event is the logical specification, of the record required (e.g., file name, record name). The data is, essentially, the input parameters to the LI/O routine.

Other events complete the specification of the system's operation. The set of all events that occur during the operation of the system, recorded in the order they occur, is known as the system event trace.

The preprocessor

The system event trace is first preprocessed before becoming the input to the simulation model. Both the preprocessor and the model are run off-line.

The preprocessor accepts the system event trace in mass storage as input. The preprocessor has two functions:

1. To decompose the system event trace into event traces representing the resource demand of each task.
2. To 'purify' the task event trace. Since the task event traces will become input to a simulator of part of the system, the effects of that part of the system in those traces should be removed before the traces are used as simulator input. The preprocessor does this. Several examples of 'system' influence (i.e., the system to be simulated) in the representation of 'system' demand, will be shown later.

The output of the preprocessor is a set of task event traces—one for each task that was active during the period the event recording mechanism was operating. The events in the task event traces are much like those of the system event trace except that:

(a) The task numbers are not recorded in the events, since each event in a trace is of the same task.
(b) The time of each event is adjusted to be relative to the operation of that task only.

An example of the second function of the preprocessor —removing system influence in the event trace—is as follows: One event in a system event trace is a call on a Logical I/O (LI/O) routine. The LI/O routine calls on a Physical I/O (PI/O) routine, and the Physical I/O event is recorded. This call on the PI/O routine is not due to the task, because the task specified its I/O demand at the Logical level. The Physical I/O call must be considered due to the system, and is removed by the preprocessor of the system event trace.

The simulator

The simulator accepts the task event traces as input. The simulation model includes the operation of the system, from the level at which the events in the trace are recorded, to the hardware. The simulator consists of:

(a) A Clockworks, which selects the next event from the task traces and increments the simulation time.
(b) An Event Analyzer: the analog of the interrupt analyzer in the actual system.
(c) The Event Response Routines: models of the operating system routines.
(d) The Hardware Section: representations of the system hardware devices.

The output of the simulation model is the data that allows evaluation of the system and isolation of areas of possible improvement. This data consists of:

1. Utilization factors of the various devices.
2. Response time characteristics for the task interactions.

The utilization data is recorded in the simulation model by summing the simulated operating and idle times of each hardware device. Response times are calculated by the difference between the simulator time at which the first 'on' event of the task is accepted by the model, and the simulator time at which the 'terminate' event is accepted.

This data, obtained from the model, may be compared with the same data taken directly from the operating systems.

Levels of Meta-system awareness of system operation

It is obvious that the level of detail of the simulation will depend upon the class of modifications that is being contemplated.

Since the event trace, after some preprocessing becomes the input to the simulation model, the definition of the events in the trace will depend upon the

extent of the simulation model. If the events in the trace are representations of some aspect of the original system's *operation* (such as the operation of a hardware device), and that aspect of the system is altered in the simulation model (i.e., the device characteristics are changed), then the event trace is irrelevant and useless to the simulation. To be useful, the events must rather be representations of the user tasks' requirements or *demand* for that aspect of the system's operation. The term 'system resource' which usually indicates the hardware devices of the system, may be extended to include any aspect of the system's operation that may be of interest—specifically, the system service macros, the scheduling routine, the loader, or a compiler. Therefore, the definition of the events in the trace are seen to depend upon the definition of system resource that is used for the specification of resource demand.

Lastly, the parts of the system that are of interest, and considered to be resources, will be included in the simulation model.

Thus, it can be seen that all of the following are inter-related:

- the definition of system resource used to specify resource demand.
- the class of trial modifications
- the extent of the simulation model
- the definition of the events in the trace

In the course of the design of the Meta-system it became apparent that these four entities could be specified at several different levels, which could best be differentiated by calling them different levels of *Meta-system awareness of the system operation.*

At the lowest level of awareness, only changes in speed or configurations of the hardware devices are potential modifications, and only the hardware and the scheduler of the hardware devices need be included in the simulation model. Any program calling for a hardware operation will be considered a user program, and the user programs' demand for system resources is the demand for hardware operations. The events in the trace, in this case, will be occurrences of the requests for hardware operations.

At the highest level of Meta-system awareness—total awareness of system and user programs—any modification to the real system may be made to the simulation model, since the simulation will be total—and the model as complex as the entire system. The events in the trace will be defined in terms of instructions or commands written at the terminals, and the system resource defined as all of the programs that respond to these commands.

Between these two levels, several more practical levels

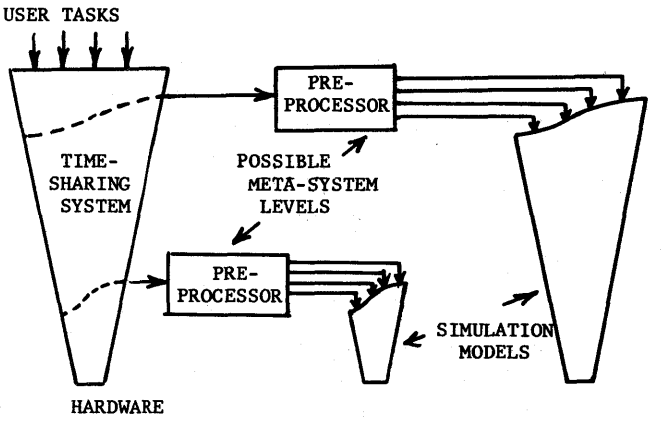


Figure 3—Conceptualization of Meta-system levels

of Meta-system awareness have been demonstrated in the detailed design of the Meta-system.

Meta-system levels of awareness are represented graphically in Figure 3. The representation of the time-sharing system in this figure is quite arbitrary. It roughly corresponds to the levels of logical complexity of the information-processing capabilities of the system, which are greatest for the parts of the system that directly communicate with the user, and least at the level of the hardware. Representations of system operation taken from one level are used as the input to a simulator of all parts of the system below that level, including the hardware. Several such levels of measuring and simulating the system are possible.

The design of any one particular Meta-system includes the determination of the Meta-system level. The factors determining the Meta-system level, listed above, must be selected to be mutually compatible.

DESIGN PROBLEMS OF THE META-SYSTEM

The analysis of the operation of time-sharing systems for purposes of implementing the Meta-system centered on isolating representations of user task demand for system resources that are independent of the allocation of the resources.

The representation of task demand taken from within the system are obtained by viewing the execution of any program above the Meta-system level to be due to the 'user,' even though the instructions being executed may have been coded by a system's programmer (as would occur during compilation of a user program) and the rest as the 'system.' The user task demand is given by the calls on the system functions.

These representations of task demand, however, are not easily separated from the operation of the system

(below Meta-system level). There are feedbacks from system to task: allocation of system resources to the task modifies the task demand for system resources. Most of the problems encountered in the design of the Meta-system are due to the system influence in the 'pure' or system-independent representations of user task demand. The system influence is removed, in each case, by one or a combination of the following techniques:

(a) Construction and placement of the event recording mechanisms in the system to either exclude the system influence, or include supplementary information so that it may be removed later.
(b) Removing the system influence in the preprocessor of the event trace.
(c) Carrying the system influence into the simulator, but designing the simulator to neutralize its effect.

The following paragraphs outline some of the problems encountered in defining, extracting, and using representations of resource demand, and the solutions to them. Other problems, relating to the efficiency and practicality of the technique are also discussed.

Task identifications in the event trace

A basic function of a multiprogramming operating system is the scheduling of each task's use of system resources. Thus, the way in which the events of each task are intermingled in time is due solely to the influence of the system.

In another instance of system operation—specifically, the one provided by the simulator—one task may run faster or slower, relative to the performance of the others. The simulator must therefore view the representations of each task's demand separately.

The first way in which the representation of resource demand is purified, then, is to separate the resource demand of each task, into 'task event traces,' in a preprocessor of the simulator input. In order to do this, each event in the system event trace must be identified with a task.

The event recording mechanisms are therefore placed in positions within the operating system at which an identification of the task is known. This is no great obstacle to the implementation of the measurement portion of the Meta-system. In most cases, the Task Control Block for the task being operated on is immediately available to the operating system routine. When it is not, some unique representation of the task such as task number or TCB address, is always maintained by the system, and may be used as the task identification.

Some events are caused by the system only, and yet must be recorded in order to complete specification of task demand information. For example, the event of the processor beginning to idle need not be associated with any task, when it is recorded. Later, the 'idle' event will be used as a 'task relinquishes processor' event. The preprocessor of the event trace, having knowledge of which task is on the CP, will complete the specification of the event.

Representation of processor time requirements

A task's demand for the processor is given by the number and type of instruction executions it requires. Since it is neither practical nor necessary to count and simulate the execution of the individual instructions* a task's processor demand is taken to be the processor time required by the task.

The measurement of the time a task spends in the processing, however, is influenced by the amount of memory interference due to I/O operations into memory, taking place simultaneously with processing. Therefore, the task's processor demand will be defined to be the time the execution of the task would take if no memory interference were present. The system influence due to simultaneity is eliminated in the preprocessor of the event trace. The preprocessor calculates the 'pure' processing time, as follows: Let m be the memory speed (cycles per second) and c be the average fraction of memory cycles needed by the processor. Then with no memory interference, processing for a period of t seconds will spend ct seconds utilizing the memory.

Now suppose I/O operations taking k bytes per second are being performed in the background. The time for the processors use of memory will be expanded by a factor of $m/(m-k)$. The total time t' taken to perform the original t seconds of processing is:

$$t' = \left(1 - c + \frac{cm}{m-k}\right) t$$

The quantity $\{1-c+[cm/(m-k]\}$ will be called the memory interference factor f.

Each of the event traces taken from the system will contain the actual time taken on the processor, t'. But, in order to isolate the task requirement for processor time $- t -$, the trace must also contain an indication of the amount of I/O being performed simultaneously with processing, so that f may be known during each interval

* Modifications internal to the processing unit will not be considered here.

of processing time. Each event trace is constructed to contain some account of the I/O activity and the calculation of t from t' and f is performed in off-line processing of the event trace.

Specification of memory requirements

The specification of the hardware resource requirements made by the user programs—either directly or via a call on a system routine—are generally quite unambiguous. The specification of memory requirements is an exception.

A task's memory requirement is actually one word—instruction or data—at a time. For obvious reasons, memory must be allocated in larger units—in paging systems, one page or block at a time. The requirement for a page of memory—when the page is not allocated, will result in an unambiguous specification of demand: the page fault. But the system cannot know whether the demand still exists one memory cycle after the page has been allocated. Hence, the system itself specifies when pages should be de-allocated. It will generally do this by assigning a probabilistic value to the demand for the page and deallocating the block when either one of these conditions is met:

(a) When it becomes known that the page is no longer needed;
(b) When some other task has a demand for the memory block occupied by the page, which is greater than the probabilistic demand for the page;
(c) When it is known that the page will not be needed for a period, and it is likely that condition b) will be met before the end of the period.

If these deallocation judgments are made optimally, a page fault for a particular page will not occur soon after the deallocation of that page.

The record of page allocations and deallocations, then, is an inexact specification of the task's demand for memory: it shows a large degree of system influence. However, it is the only record of memory demand available without special hardware to monitor memory utilization. This example of system influence is not removed during preprocessing of the event trace, but is removed by the simulation model, and removed only when necessary.

The simulator will, in general, handle memory allocation differently from the allocation shown in the event trace. If, during the simulation, the task trace shows a page fault for a page that the simulator has already allocated, the page fault event is simply skipped. On the other hand, if the simulation model deallocates a page when it was not deallocated in the real system, the simulator must impose a potential page fault on itself. It replaces the page fault by evaluating the page re-use time as a random variable. The specification of the random variable is made from the average value of the page re-use times of the previous and next re-use times for the page that are available in the event trace.

Entrances and exits to system routine

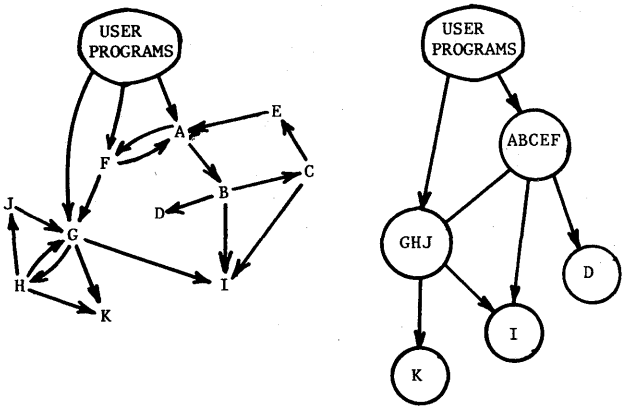
Higher levels of specification of user demand for system resources are demands for system functions. The events indicating these functions are recorded at the entrances to the routines performing the functions. The operating system is written as a set of recursive subroutines so that a call on one system routine may result in calls on several others. If the original call on a system routine is taken to be an indication of user program demand, then these secondary calls, which are not made directly by the user program, may not be considered user demand. The events corresponding to these calls are system-contributed data, and must be eliminated from the event trace. A method of distinguishing user program calls from system program calls is required.

In order to distinguish system program calls from user program calls on the system functions, both the entrances to and exits from the system routines are recorded as events. Off-line processing of the event traces will remove the secondary calls that occur between the entrance event and exit event of a particular routine.

In order to place these event recording mechanisms in the system, the entry and exit points of the system routine of interest must be identified. The identification of entry points is straightforward. The identification of the exit points of system routines is, in general, a difficult problem. Each transfer of the following types must be analyzed to determine whether it should be considered an exit from the subroutine:

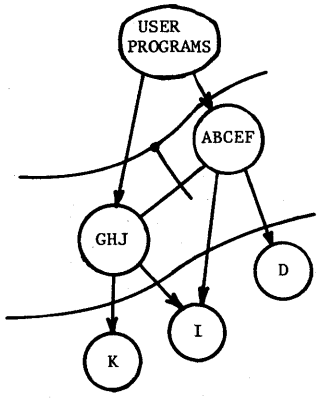
(a) Transfer to the return address provided by the standard subroutine call.
(b) Transfer to any address provided as a parameter to the subroutine.
(c) Transfer to an address taken from the pushdown stack of subroutine calls.
(d) A non-subroutine type transfer to another system function.

The methodology outlined in the design specifies at which of these transfers the event recording mechanism



a) SUBROUTINE STRUCTURE OF OPERATING SYSTEM

b) CLASSES OF SUBROUTINES



c) DEFINITION OF META-SYSTEM LEVELS

Figure 4—Meta-system levels due to subroutine structure

(in some cases, a conditional recording mechanism) should be placed.

Volume of data recorded: the concept of class of subroutines

As the representation of user resource demand is refined, it becomes more a specification of logical functions to be performed than a specification of physical operations.

Because of the subroutine structure of the operating system, it may be necessary to record many subfunctions of one logical function. Also, it is necessary to carry some representation of the physical operations even at logical level of user demand specifications. Hence, high-level specifications of user demand require more events than lower-level specifications.

In order to generalize the technique of recording the event trace at higher levels of demand specification, the number of events must be somewhat independent of the level of specification—it cannot increase indefinitely as the specification level increases.

The generalization of the definition of higher level event traces must be made in such a way that the events indicating the operation of routines that are always called as the consequence of higher-level routines, are not included in the higher level traces. This requires analysis of the set of routines making up the operating system to identify classes of subroutines, that have a partial ordering imposed upon them by the nature of their calls.

The classes of subroutines are defined as follows: Routine A is in a class greater than or equal to Routine B if A calls on B, either directly or through another routine. If, in addition, routine B calls on routine A, then A and B are in the same class.

As an example, applying the definition of class to the subroutine structure of Figure 4a, in which the subroutine calls are indicated by arrows, yields 5 classes, whose partial ordering is shown in Figure 4b. The user programs are always a class by themselves, and always the highest class, since they are never called by the system as subroutines. From the fact that the user programs call on routines A, F and G, the class of user programs call on the classes ABCEF and GHJ.

Once the classes of the operating system routines are established, as in Figure 4b the set of levels at which the resource demand of the system may be measured (the level of Meta-system awareness of system operation), may be selected. The level is represented by the interface between the classes of routines that are considered user programs (higher in the ordering) by the Meta-system, and the classes of routines that are considered part of the system (the lower part of the ordering). A level is chosen by simply drawing a line across the arrows representing the calls on the subroutine classes.

Only the entrances and exits to the subroutines of the classes that are adjacent to the Meta-system level of awareness need be recorded as events.* The routines that are called only from higher-level routines within the Meta-system awareness need not be recorded, even though they may have been recorded in a previous, lower-level Meta-system.

The set of levels of Meta-system awareness that may be chosen for the subroutine structure of the example is

* This does not imply that the entrances and exits to *every* subroutine of such a class must be recorded, because a finer analysis (e.g., the operating system structure of Figure 4a) may show that only several of the routines of a class are called from above the Meta-system level. The analysis by class is a first approximation to specify a set of routines that need not be recorded. It is true, however, that if one routine of a class is included within the model, then each routine of the class, and all classes below it, must be included in the model.

the following:

(a) any of the 7 subsets of {K, I, D}, the lowest classes (excluding the empty set)
(b) {GHJ, I} or {GHJ, I, D}
If the class GHJ is chosen, then the lower class, K, need not be recorded since it occurs only as a consequence of GHJ. The other lower class, I, must be recorded, since it is called from above GHJ.
(c) {ABCEF, GHJ}
The user programs call on this set of system program classes. All other classes result from calls on this set; therefore, calls on these other classes need not be recorded.

It must be remembered, however, that the Figure 4b is a structural representation of Meta-system classes, and therefore provides only an estimate of the number of events which will actually be recorded. The volume of recorded data will depend upon the frequency with which control passes through the Meta-system level. Also, the calls on the subroutine classes are recorded at the entrance to the subroutines. Thus, if the Meta-system level crosses one arrow leading into a class, the level will in fact cross the other arrows into that class as well, whether or not this is intended in the definition of the level. For example, in Figure 4c, two levels are shown. One, the higher level, is inefficient, since some of the calls on the GHJ class result from the previously-recorded ABCEF class. In this case, a call sequence from the user program to F to G is recorded twice. The lower Meta-system is efficient.

Modeling system routines

The simulation model will include the models of some of the system routines. In order to preserve the economy inherent in simulation (as opposed to implementation and testing) the models of these routines are simplified. Yet the important aspects of their operation—in particular, the decisions that ultimately result in hardware resource allocation—must be duplicated within the model.

Simplified versions of system routines have been developed in the design of the Meta-system. It has been estimated that 60 percent of the code in the executive of a multi-programmed operating system exists for the purpose of error checking. It is assumed in the Meta-system design that the paths resulting from the error checks are taken rather infrequently, therefore, they are not a great influence on the resource allocation process. These error checking paths are omitted from the versions of the system routines in the model. Likewise, security considerations, in file operations, do not determine the location or identification of a particular data item. It may be assumed that the frequency of file operations being blocked for security reasons is small enough not to influence the utilization data obtained in the simulation. Security data has been omitted from the model.

SUMMARY

This report has been a summarization of the concept of the Meta-system, and a review of the problems that are encountered in the design of such execution-simulation systems, rather than simply a recounting of the details of the design.

The completion of the design of the Meta-system and successful trial runs of the system (insofar as they are possible on an unimplemented system) provide strong evidence that the Meta-system is technologically feasible, and will be an aid in the development of a time-sharing system. The question of its economic feasibility still remains since the implementation will be a considerable task. However, simulation models are employed during the development of most new computer systems. Development of the simulation model to operate in the Meta-system at the outset of the design is probably a feasible approach, since the full benefit of the Meta-system will be obtained in the later stage of design, and it may then be given to the user to optimize the system for his particular usage.

ACKNOWLEDGMENTS

The author is considerably indebted to his dissertation supervisor, Dr. Noah S. Prywes, for his aid, advice and encouragement during the period of the research. Also, many thanks are due to Dr. David Hsiao, who generously contributed his insight, experience and enthusiasm.

REFERENCES

1 G M AMDAHL B BLAOUW
Architecture of IBM S/360
IBM Journal of Research and Development Vol 8 No 2 April 1964
2 P CALINGAERT
System performance evaluation: Survey and appraisal
CACM Vol 10 No 1 pp 12–18 January 1967
3 D J CAMPBELL W J HEFFNER
Measurement and analysis of large operating systems during system development
AFIPS Proc FJCC Vol 33 pp 903–914 1968

4 C T GIBSON
Time-sharing in the IBM System/360: Model 67
AFIPS Proc SJCC Vol 28 p 61 1966

5 R C DALEY J B DENNIS
Virtual memory processes, and sharing in MULTICS
CACM Vol 11 No 5 p 306 May 1968

6 P J DENNING
Equipment configuration in balanced computer systems
IEEE Transactions on Computers Vol C–18 No 11 pp 1008–1012 November 1969

7 J R DENNIS
Segmentation and the design of multiprogrammed computer systems
J ACM Vol 12 No 4 pp 589–602 October 1965

8 E W DIJKSTRA
Structure of THE multiprogramming system
CACM Vol 11 No 5 May 1968

9 G H FINE P V McISAAC
Simulation of a time-sharing system
Management Science Vol 12 No 6 pp B180–B194 February 1966

10 D FOX J L KESSLER
Experiment in software modeling
Proc AFIPS FJCC 1967

11 *IBM System/360 time sharing system, concepts and facilities*
IBM Document C28–2003–3

12 *IBM System/360 time sharing system, dynamic loader—Program logic manual*
IBM Document Y28–2031–0

13 *IBM System/360 time sharing system, resident supervisor—Program logic manual*
IBM Document Y28–2012–3

14 A S LETT W L KONIGSFORD
TSS/360—A time-shared operating system
AFIPS FJCC 1968 pp 16–28

15 R A MERIKALLIO
Simulation design of a multiprocessing system
AFIPS Proc FJCC Vol 33 Pt 2 1968

16 N R NIELSON
The analysis of general purpose computer time-sharing systems
Document 40–10–1 Stanford University Computation Center December 1966

17 G OPPENHEIMER N WEIZER
Resource management for a medium scale time sharing operating system
CACM Vol 11 No 5 p 313 May 1968

18 E L ORGANICK
A guide to multics for subsystem writers
Project MAC Doc March 1967

19 *70/46 processor reference manual*
RCA Corp Document 70–46–62 March 1968

20 *TSOS executive macros and command language reference manual*
RCA Document 70–00–615 June 1969

21 J H SALTZER J W GINTELL
The instrumentation of multics
CACM Vol 13 No 8 August 1970

22 F D SCHULMAN
Hardware measurement device for IBM System/360 time sharing evaluation
Proc ACM National Conference pp 103–109 1967

23 V A VYSSOTSKY F J CORBATO R M GRAHAM
Structure of the multics supervisor
Proceedings FJCC pp 203–212 1965