EXPERIENCE WITH THE EXTENDABLE COMPUTER SYSTEM SIMULATOR

Donald W. Kosy
The Rand Corporation
Santa Monica, California

Abstract

A prototype version of ECSS has been implemented to aid in constructing simulation models of computer systems. A specialized language is used to describe hardware, software and system load and a service-routine package handles many of the bookkeeping details of model control. The full power of SIMSCRIPT II is also available for extending ECSS capabilities. Some weaknesses in the provided facilities have been noticed, however, which has led to a few general conclusions about the design of computer system simulators.

I. INTRODUCTION

The Extendable Computer System Simulator (ECSS) is a prototype language which has been designed and implemented to investigate ways of making the simulation of complex computer systems a less formidable task. Nielsen[1] demonstrates the need for a new language, free from the problems of using general-purpose languages and the drawbacks of existing computer system simulators. The approach taken has been to attempt to provide a convenient and natural means of describing computer system characteristics and computing processes while allowing the flexibility and power of a general-purpose simulation language.

Experience with several small models, written to test the initial version of the simulator, has indicated the soundness of the approach. In most cases, the models have been quickly and easily constructed. However, situations have also been found for which the language is not as useful as it could be, revealing shortcomings in both the breadth and versatility of the facilities provided.

This report states the current capabilities of ECSS, discusses its strengths and weaknesses, and prescribes some directions for further work. First, we review the concepts of ECSS and use a simple example to illustrate its powers. We then outline a number of cases where the current version of the language is not as helpful as it could be. Finally, we present several general design principles for computer system modeling languages that have become clearer in retrospect.

II. REVIEW OF ECSS

Three elements of ECSS are important in describing a computer system to be simulated: 1) the System Description section; 2) the Load Description section; and 3) the Service Routines.

System Description

The System Description characterizes those system elements that are considered static in ECSS. Declarations in this section specify the number of various types of hardware, the names of these devices and names of groups of devices, their capacities and capabilities, their interconnection, and the possible "software execution time overhead" incurred by operation of some devices in performing certain operations.

All static system elements in ECSS are called *devices*. A processor, disk drive, or terminal is modeled by specifying a device to have a particular set of characteristics. All devices are a collection of up to four capabilities, as shown in Fig. 1. Different device characteristics result from using different combinations of capabilities. We call

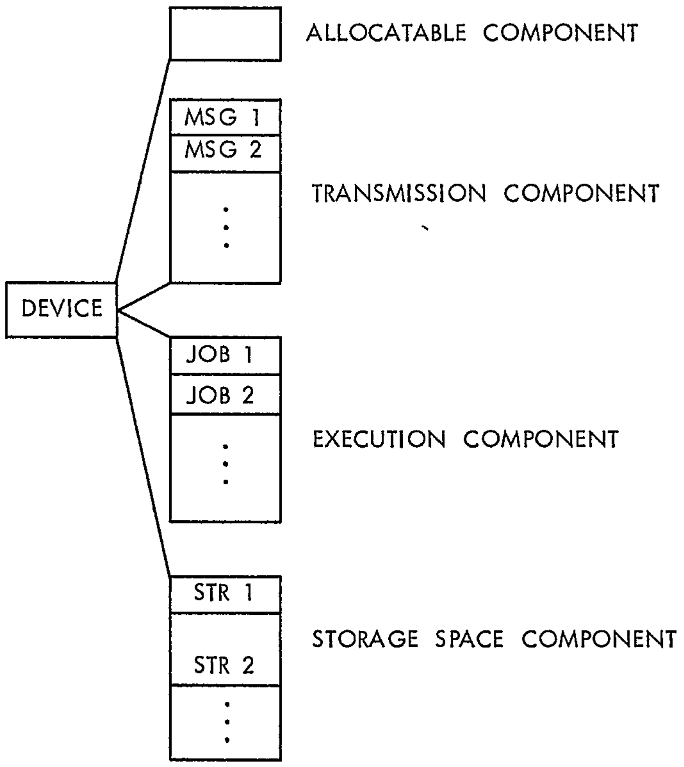

Fig. 1--ECSS Device Components

a device's capabilities its *components*. For some hardware models, the conjunction of several components may be meaningful, such as defining a central processor to execute instructions, transmit data, and have core storage; but often, only one component will be important, e.g., in defining a disk memory as only a quantity of storage space. The user is not restricted in his choice of components for modeling real-world devices.

Device characteristics are further specified by setting parameters of the components. Each type of component has different parameters according to its intended purpose. Some of these parameters are: 1) for Storage Space, the maximum amount; 2) for Execution, the instruction processing rate and maximum number of jobs that may be processed concurrently; 3) for Allocatable, whether a device may be allocated to more than one job simultaneously; and 4) for Transmission, the maximum data rate, number of simultaneous data streams that may be transmitted, delay per transmission, and others. There are also several parameters of interaction between components. Two of these are software overhead times, used to model the software execution necessary to initiate data transmission or device allocation, and degradation factors, used to account for Execution interference due to data transmission.

Besides characterizing individual devices, the System Description may define groups of devices. Groups are used for compact description (e.g., defining several devices having the same characteristics) and for automatic device selection at run time. Device selection is a feature of the built-in ECSS "operating system" that allows a group of devices to be put in a load command to indicate that the actual device picked to handle the command should be the one first able to, at any particular time during the run. The last function of the System Description is definition of certain data transmission paths within the system.

Load Description

The Load Description is used to describe the system's dynamic behavior. The load on a computer system is the work it must do where "work" means the utilization of device components for some amounts of time. Special programs in this section called "jobs" simulate the work of real application and control program processing by indicating sequences of hardware utilization commands.

Nearly any kind of deterministic or random effects may be included in a job to describe just which activities are to be done, for how long, and in what order. The sequence is controlled by the testing, branching, looping and other logical properties of the job. The simulated time at which a command is issued depends on the time necessary to do the work of the previous commands, and on any conditional delays within the job. A general example of the progress of simulated time in a job is presented in Fig. 2. One may think of a pointer moving through a job indicating which command is being processed. Note how this is reminiscent of the flow orientation of GPSS.

The logical behavior of jobs is provided by appropriate SIMSCRIPT II statements, e.g., the IF statement in Fig. 2. SIMSCRIPT and ECSS statements may be intermixed in job descriptions. In fact, the full capability of this general simulation language[2] is available in ECSS. New data structures may be created and used in the model, for example, to expand the description of the state-of-the-system beyond the condition of the devices.

Simulation time in processing ECSS commands results from two situations. First, the use of certain device components is explicitly time-consuming. Executing instructions as directed by an EXECUTE command, and data transmission as specified by a SEND/RECEIVE statement, always take time during which the appropriate component is busy. Second, some commands may or may not allow time to pass depending on the state-of-the-system. Conditional delays, indicated by WAIT or HOLD statements, are of this type, depending on whether the condition has come true at the time the statement is processed. Input may or may not have arrived in the job in Fig. 2 and hence the process pointer may or may not be held up at the HOLD statement. Requests for storage space or for device allocation, GET and ALLOCATE respectively, may cause delay because the device component is already being utilized, and the job must wait for it to become free. Execution or transmission commands may also involve this kind of delay if those components are busy. Priority ranked queues are associated with each component to keep track of pending requests. Marking the end of utilization of a device component, e.g., with a FREE space or a DEALLOCATE statement, usually does not hold up job processing.

Several other load commands take no time but serve to define the jobs and their initiation. These include JOB -- marking the beginning of a job, LAST -- marking the end, START -- commanding job processing, and INITIALLY START -- directing exogenous job-starting with a possible repetition interval to model system environment.

Service Routines

The Service Routines take care of the housekeeping details of internal model control. A collection of SIMSCRIPT II subprograms is used by ECSS to implement the actions specified by the Load Description. The event scheduling and process pointer management necessary to realize the flow orientation of jobs within a SIMSCRIPT II context are included. All the system-state updating associated with the interaction of jobs and devices, jobs and jobs, and devices and devices are incorporated in these subprograms.

Moreover this package of routines supplies a number of such operating-system-like functions as resource allocation by priority, I/O interrupt handling, device selection from groups, and queueing, dequeueing and retrial of device requests. These capabilities allow

Simulated time at beginning of command processing		Job description	Comments
t_0		JOB TO SHOW·TIMING	
t_0		Command 1	(takes time t_1)
$t_0 + t_1$		IF VARIABLE > 0, GO TO NEXT ELSE	
VARIABLE $\leq$ 0	VARIABLE > 0		
$t_0 + t_1$	—	Command 2	(takes time t_2)
$\sum_{i=0}^{2} t_i$	—	Command 3	(takes time t_3)
$\sum_{i=0}^{3} t_i$	$t_0 + t_1$	'NEXT'	(takes no time)
$\sum_{i=0}^{3} t_i$	$t_0 + t_1$	HOLD UNTIL INPUT ARRIVES	(conditional delay — assume input arrives at $t_{in} > t_0 + t_1$)
t_m	t_{in}	Command 4	(takes time t_4)
$t_m + t_4$	$t_{in} + t_4$	Command 5	(takes time t_5)
$t_m + t_4 + t_5$	$t_{in} + t_4 + t_5$	LAST	
where $t_m = \max\left(\sum_{i=0}^{3} t_i, t_{in}\right)$			

Fig. 2--ECSS Job Processing

multiprogramming, multiprocessing, real-time processing, and conversational transmissions to be easily specified for a model. The ECSS user automatically gets these capabilities when defining the dynamics of his model.*

III. AN EXAMPLE OF THE ECSS APPROACH

To show how the ECSS sections fit together, we present a simplified model of a multiprogrammed batch processing system. Figure 3 shows the system hardware configuration. A processor connects through three I/O ports to a cardreader and two controllers which in turn are connected to four disks. The system's load consists of disk file updating routines, a sequence of which are entered by means of the cardreader. In addition, some simulated software is included in the load to schedule jobs according to their space requirement, and to handle disk allocation.

*This review has of necessity been brief. For a complete description of ECSS see Kosy[3].

Figure 4 lists the model's entire simulation program. Some comments appear in parenthesis; each statement has been numbered; and SIMSCRIPT statements are marked with an (S). Besides the system and load descriptions mentioned above, the listing shows a preamble, a definition-description section, some events, and the MAIN routine. The preamble (lines 1-11) defines global variables and other data structures to be used in the model. The definition section (12-15) is an additional ECSS element that allows relation of user-defined terms to basic ECSS terms (with possible conversion factors) for use in the system and load descriptions. The event OBSERVATION (24-29) in this program is used to view the system every five seconds and print out a record of its activity. QUITSIM (line 30) halts the simulation. Finally, the MAIN routine (31-35) indicates when to stop and when to make the first observation. It then directs the simulator to commence.

Model Operation

The ECSS job READER (lines 37-45) models units of work submitted to the system at exponentially distributed times, with a mean of 12 seconds. The work is characterized by its

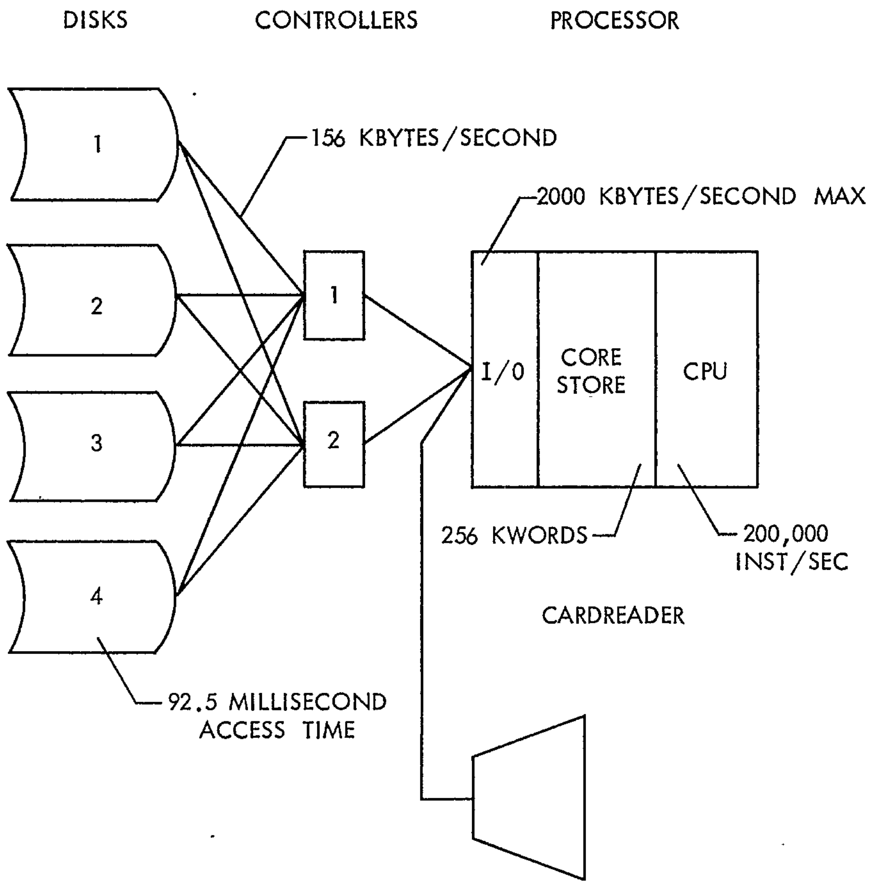

Fig. 3--Multiprogrammed Batch System

space requirement, selected from an exponential distribution with a mean of 104 (KWORDS), and a duration parameter, represented as the number of simulated processing cycles, which is read in from a data card. Work to be done is placed in a queue, JOB.QUEUE, and ranked on its space requirement from high to low. SIMSCRIPT II provides the definition and manipulation statements for this queue.

The prototype for the work to be done is the APPLICATION job (64-74), which runs on the processor. First, one disk is designated as the input source and one of the disks is selected (by the built-in ECSS operating system) as the output unit by the ALLOCATE statements. This task is considered to require 50 msec of overhead time per allocation (from statement 17), and hence the processor's execution component is busy for 100 msec as well as the allocatable component of the disks being set. Because the disks are PUBLIC, more than one job can use them simultaneously.

Next, the APPLICATION job begins the simulated reading of blocks of data, processing of the data, and writing of an output record (68-70). The RECEIVE statement picks a free data path from the INPUT.FILE disk, through one of the controllers, to one of the I/O ports of the processor. The job automatically waits for a free path if none is available. Transmission components of these devices are then tied up for the duration of the disk access time (92.5 ms.), plus the time for transmission of 800 bytes at 156 kilobytes/sec (disk speed). APPLICATION job processing is suspended for that time, as directed by the WAITING clause. The processor execution component is then busy simulating the execution time for 10 sets of 400 instructions (SUBROUTINES) at 200,000 instructions/sec. Finally, the SEND statement goes through a similar procedure as the RECEIVE to model writing the output record. The job's last action is to reset the allocation status of the disks it used, i.e., free them, which takes no simulated time in this model.

A scheduling algorithm is modeled in the INITIATOR job (46-63). It starts APPLICATION jobs in the order determined by the JOB.QUEUE, i.e., large jobs first (if the queue is empty, an INITIATOR tries again in one second). Execution time for scheduling is modeled with the first EXECUTE statement. The GET statement reserves a contiguous block of the processor's storage space necessary to satisfy the space requirement. If not enough space is available, the INITIATOR job is suspended until there is. When space is reserved, another INITIATOR is started (for the next JOB.QUEUE work unit) and

```
 1.  (S) PREAMBLE (defines global variables and data structures)
 2.  (S)   DEFINE TOTAL.IN.TIME AS A REAL VARIABLE
 3.  (S)   DEFINE JOBS.STARTED, JOBS.COMPLETED, JOBS.IN.PROCESS AS INTEGER VARIABLES
 4.  (S) TEMPORARY ENTITIES
 5.  (S)   EVERY JOB.DESCRIP HAS A SPACE.REQMT, A LOOP.REQMT AND A TIME.IN AND BELONGS TO THE
             JOB.QUEUE
 6.  (S)   DEFINE SPACE.REQMT, TIME.IN AS REAL VARIABLES
 7.  (S)   DEFINE LOOP.REQMT AS AN INTEGER VARIABLE
 8.  (S)   THE SYSTEM OWNS A JOB.QUEUE
 9.  (S)   DEFINE JOB.QUEUE AS A SET RANKED BY HIGH SPACE.REQMT
10.  (S) EVENT NOTICES INCLUDE OBSERVATION,QUITSIM
11.  (S) END


12.      DEFINITION DESCRIPTION(incorporates user terminology)
13.        DEFINE UNITS KBYTES=1000 TRANSMISSION.UNITS,KWORDS=SPACE.UNIT
14.        DEFINE UNITS SUBROUTINES=400 INSTRUCTIONS
15.      END


16.      SYSTEM DESCRIPTION (defines number and characteristics of each class of devices)
17.        SPECIFY 1 PROCESSOR,
             EXECUTES 200 INSTRUCTIONS PER MILLISECOND (execution rate)
             TRANSMITS 2000 KBYTES PER SECOND (transmission rate)
             HAS CAPACITY OF 3 TRANSMISSION USERS (models 3 subchannels)
             CONNECTS TO CONTROLLERS
             DEGRADES PROCESSOR BY 10% PER 200 KBYTES/SECOND (models cycle-stealing)
             HAS CAPACITY OF 256 KWORDS (storage space)
             ALLOCATES DISKS IN 50 MS (models gross software effect)

18.        SPECIFY 2 CONTROLLERS, EACH
             HAS CAPACITY OF 1 TRANSMISSION USER
             CONNECTS TO DISKS,PROCESSOR

19.        SPECIFY 4 PUBLIC DISKS, EACH
             HAS CAPACITY OF 1 TRANSMISSION USER
             ABSORBS 92.5 MILLISECONDS PER MESSAGE (access time)
             TRANSMITS 156 KBYTES PER SECOND (disk transmission rate)
             CONNECTS TO CONTROLLERS

20.        SPECIFY 1 CARD.READER, CONNECTS TO PROCESSOR

21.        PATH OUTPATH IS PROCESSOR,CONTROLLERS, DISKS
22.        PATH INPATH IS DISKS,CONTROLLERS,PROCESSOR
23.      END


24.  (S) EVENT OBSERVATION SAVING THE EVENT NOTICE
25.  (S)   RESCHEDULE THIS OBSERVATION IN 5 UNITS (seconds)
26.  (S)   LET AVG.TURNAROUND=TOTAL.IN.TIME/JOBS.COMPLETED (calculate statistics)
27.  (S)   LET AVG.THRUPUT=JOBS.COMPLETED/TIME.V
28.  (S)   LIST TIME.V,JOBS.STARTED,JOBS.IN.PROCESS,JOBS.COMPLETED,AVG.TURNAROUND,AVG.THRUPUT
             (print out statistics)
29.  (S) END

30.  (S) EVENT QUITSIM  STOP END

31.  (S) MAIN
32.  (S)   SCHEDULE A QUITSIM AT 1000
33.  (S)   SCHEDULE AN OBSERVATION AT 5
34.  (S)   START SIMULATION
35.  (S) END
```

Fig. 4--Batch Processor Simulation

```
36.          LOAD DESCRIPTION (defines sequences of system utilization commands)
37.          JOB READER
38.    (S)     CREATE A JOB.DESCRIP
39.    (S)     LET SPACE.REQMT(JOB.DESCRIP)= EXPONENTIAL.F(104.0,1)
40.    (S)     READ LOOP.REQMT(JOB.DESCRIP)
41.    (S)     LET TIME.IN(JOB.DESCRIP)=TIME.V
42.    (S)     ADD 1 TO JOBS.STARTED
43.    (S)     FILE JOB.DESCRIP IN JOB.QUEUE
44.            START JOB READER IN EXPONENTIAL.F(12. ,1) SECONDS ON CARD.READER
45.            LAST

46.          JOB INITIATOR
47.    (S)     IF JOB.QUEUE IS EMPTY,
48.              START JOB INITIATOR ON PROCESSOR WITH PRIORITY 2 IN 1 SECOND
49.    (S)       RETURN
50.    (S)       ELSE
51.    (S)     REMOVE THE FIRST JOB.DESCRIP FROM JOB.QUEUE
52.    (S)     ADD 1 TO JOBS.IN.PROCESS
53.            EXECUTE 100 INSTRUCTIONS (space reservation processing)
54.            GET SPACE.REQMT (JOB.DESCRIP) CONTIGUOUS KWORDS FROM PROCESSOR
55.            START JOB INITIATOR ON PROCESSOR WITH PRIORITY 2
56.            EXECUTE 500 INSTRUCTIONS (job initiation processing)
57.            START JOB APPLICATION(LOOP.REQMT(JOB.DESCRIP)) ON PROCESSOR
                 WITH PRIORITY 1 WAITING HERE FOR COMPLETION
58.            FREE SPACE.REQMT(JOB.DESCRIP) KWORDS FROM PROCESSOR
59.    (S)     SUBTRACT 1 FROM JOBS.IN.PROCESS
60.    (S)     ADD 1 TO JOBS.COMPLETED
61.    (S)     ADD TIME.V-TIME.IN(JOB.DESCRIP) TO TOTAL.IN.TIME
62.    (S)     DESTROY THE JOB.DESCRIP
63.            LAST

64.          JOB APPLICATION (REQD.LOOPS)
65.            ALLOCATE DISKS# 1 AS INPUT.FILE
66.            ALLOCATE DISKS AS OUTPUT.FILE
67.    (S)     FOR L=1 TO REQD.LOOPS DO ...
68.              RECEIVE RECORD "BLOCK OF DATA" OF LENGTH 800 FROM INPUT.FILE VIA INPATH
                   WAITING HERE FOR COMPLETION (read data)
69.              EXECUTE 10 SUBROUTINES (process data)
70.              SEND RECORD OF LENGTH 160 TO OUTPUT.FILE VIA OUTPATH WAITING HERE
                   FOR COMPLETION (write data)
71.    (S)       LOOP
72.            DEALLOCATE INPUT.FILE
73.            DEALLOCATE OUTPUT.FILE
74.            LAST

75.          INITIALLY START READER ON CARD.READER (initialize the system)
76.          INITIALLY START INITIATOR ON PROCESSOR WITH PRIORITY 2
77.          END
```

Fig. 4--Batch Processor Simulation (Continued)

some execution time, representing job initiation bookkeeping, is called for. Following that, an APPLICATION job is started on the processor, passing its duration as an argument. Upon completion, the INITIATOR proceeds to release the space for that unit of work (line 58) and terminates.

Several INITIATOR and APPLICATION jobs may be running concurrently on the processor in a multiprogrammed fashion. Contention for devices is resolved by priority (note that INITIATOR jobs with priority 2 will always get a device before an APPLICATION job), then by time of request. This is performed by the Service Routines. Scattered throughout the jobs are various SIMSCRIPT II statements that collect the data periodically reported by the OBSERVATION event.

Advantages of the ECSS Approach

1) *Natural Input Language.* Both ECSS and SIMSCRIPT II statements are quite English-like and their procedural format allows a clear and flexible design. The definition-description permits incorporation of user-defined terms for various dimensional units. Considerable freedom for mnemonic names of devices, jobs, variables, events, etc. is allowed.

2) *Provides Declarations and Commands for Common Computer System Operations.* Hardware elements are compactly defined and described in the system description, and a variety of utilization commands for describing loads are provided. Note that such things as requests for storage space, job starting, execution and data transmission require only single statements. The Service Routines assume much of the modeling burden by handling the details and the built-in operating system capabilities.

3) *Flexibility.* ECSS provides a variety of techniques for modeling systems. Both flow-oriented jobs and events can be used to

operate on the system state, depending on the modeler's preference. Jobs can model program behavior, e.g., APPLICATION running on the processor, or system input characteristics, as READER running on the cardreader, or arbitrary activities running on any appropriate device. The generality of devices, each having as many as four components, allows nearly any kind of equipment to be modeled. No particular structures are forced on the user.

Nor is any particular level of detail required of the model. The example focuses on scheduling software, in INITIATOR, including space reservation and release and necessary execution time, while I/O interrupt handling, transmission path selection, multiprogramming and other things are left up to built-in ECSS functions. In other models, these things may be of interest and could be modeled explicitly. Changes in detail are also readily incorporated into a given model.

4) *Extendability*. ECSS and SIMSCRIPT II statements are used in conjunction to build simulation models. Provision of all the data types and procedural statements of SIMSCRIPT allows the user to go beyond the primary ECSS capabilities for any purpose. Note how the JOB.QUEUE was included in our batch example. In addition, the user may extend the definition of a device, a job, a transmission or any other ECSS structure by appropriate SIMSCRIPT preamble statements that add attributes to these entities. Although not shown in the example, new commands consisting of combinations of SIMSCRIPT and ECSS statements may be defined as another means of extending ECSS capability.

5) *Modifiability*. For some simulation models, the user might like to change certain ECSS operations. In the above example, the user may not wish to wait for space to become available for the first APPLICATION job on the JOB.QUEUE, but would like to know if space is available or not. Such changes would require modification of one or more of the Service Routines. This is relatively easy because the Service Routines are both modular and written in SIMSCRIPT II. The source code for any of these routines is completely open to change. Of course this would require a fairly good knowledge of the internal working of ECSS, but the clarity of the SIMSCRIPT II code makes change much easier than if the user had to cope with assembly language.

6) *Rerun Convenience*. Simulation models are usually rerun a number of times both during development (to debug, test, and validate the model) and in production (to investigate behavior under various conditions). ECSS has been constructed to avoid recompilation of the model for each run. First, it produces a summary deck of the static system description, which allows changing system parameters without reprocessing the system description. Second, the object decks representing the jobs, events, functions, and other routines written by the user are available for subsequent runs. Finally, the system and load are independent of each other with respect to system parameters (e.g., CPU speed, number of disks, etc.). This allows different systems and loads, which were not originally processed together, to be combined later as summary decks and object modules, again avoiding the overhead of recompilation.

IV. WEAKNESSES OF ECSS

What Is a "Weakness"?

ECSS is weak in some areas in the sense that certain capabilities are less convenient or less flexible than they ought to be. Although generous use of SIMSCRIPT II makes it possible to model nearly any system in ECSS, the user must still do more work than he should have to do for certain classes of operations. (Questions of efficiency of implementation are secondary at this stage of ECSS development and will not be discussed here.)

The difficulty in coping with weaknesses varies considerably. Least difficult is writing extra SIMSCRIPT II routines, not involving ECSS system variables, to include some operation not specifically in the ECSS language. Manipulating ECSS system variables outside the Service Routines is fairly difficult since a knowledge of the function and use of these variables by the ECSS system is required. Most difficult is changing the Service Routines themselves, which demands an intimate knowledge of their working and interaction. We will describe examples of some noticeable problems.

Neglected Statements and Capabilities

Certain features should be included in a computer system simulator which have not been incorporated in this version of ECSS. It is still difficult to formulate a complete set of these, but there are a few which have become apparent. For one, although all data are available to the user, no statistics are automatically collected or reported by ECSS. (Hence the need for the OBSERVATION event in the example simulation.) Such things as device utilization, average waiting time in queues, and average queue length are nearly always of interest in computer simulations. These statistics should be collected and perhaps produced as a summary report on demand. It would also be convenient to receive a summary of the static system simulated because the system description may be quite complex if hierarchical groups are used. Another desirable feature would be a built-in technique for stopping and restarting a model at any point, perhaps allowing parametric changes to account for initialization periods. There is, furthermore, very little consistency checking of the model at translation time or run time. ECSS should do more to point out self-contradictory or meaningless system descriptions.

Several common computer operations now must be modeled in SIMSCRIPT II and probably deserve specific ECSS statements to handle them. A statement to change job priorities during their run would contribute convenience and clarity to a model, particularly when simulating such operating system functions as task scheduling or interrupt masking. Another missing capability is data file placement on storage devices for file access modeling.

Although files may now be placed on specific devices, or allocated from groups of devices, it would be quite handy to have statements indicating file position on a device. This could ease calculation of variable access times for sequences of transmissions from different files on the same device. A third deficiency is lack of any built-in polling operations. Many applications require service requests to be handled differently from a priority-interrupt fashion, e.g., involving round-robin scheduling. The controllers could have polled the three I/O ports, in our example, rather than being interrupt-driven by the ports. Incorporation of arbitrary (or changing) servicing order would be eased by ECSS statements to implement it.

Hard-To-Get-At Mechanisms

Control of mechanisms invoked by declaration is sometimes desired in the load description. One instance is the degradation of instruction execution rate as declared by a DEGRADES clause in the System Description. Reductions in execution rate may occur for other reasons than transmission interference. Multiple CPU's contending for the same core memory will not generally run as fast as if only one CPU had access to the memory. The degradation mechanism could easily handle such cases if the user had more direct control over it when specifying jobs.

Software overhead is another delarative feature that could be more accessible. In the batch model, allocation takes 50 msec of overhead time, but a variety of other activities like job initiation, task switching, or core storage reservation may also take time that should be counted as overhead. Hence, the user should be able to indicate overhead time for a variety of operating system functions, or perhaps at any time he wishes, to realize greater utility of the overhead accumulation procedures.

Over-Automaticity

The user may also need greater control over the functions of load description statements. Sometimes these statements automatically do more than the user desires. GETing space in the INITIATOR job of the batch example not only finds and reserves space on the processor, but also waits for space, if enough is not available at request time. This holds up further processing of the INITIATOR job. Or, one may want to return to processing INITIATOR with a note indicating an unsuccessful request if insufficient space was available. That is, one wished only to use that part of GET that found the maximum contiguous amount of space left and compared it to the amount required, and not the part which queues unsuccessful requests for retrial.

A similar problem also occurs when SENDing data. Although not shown in the batch example, it is possible to specify a message to be conveyed from one job to another by means of a data transmission. These messages are often used to activate further processing of jobs waiting for them. Instead of terminating and restarting the INITIATOR job in the batch model, it may have been more realistic to suspend and reactivate it by means of a clock interrupt. This involves only the signaling features of the SEND statement, all communication being within the processor, but there is no way of simply doing it without all the path selection and other transmission machinery. Several other cases could be mentioned which illustrate a need to use only part of the power of a load description statement.

Awkward Control Program Representation

Although the automatic operating system is a great help in some models, the distribution of control functions between the Service Routines and user-written jobs makes for a sometimes strained relationship between the model and the real system being simulated. In the batch example, the work schedule INITIATOR job interfaced fairly smoothly with the ECSS system. This may not be the case if different algorithms are desired for other kinds of queue handling, resource management, or other schemes for controlling the order of occurrence of internal model events. Low level changes in these operations usually require extensive changes both to the Service Routines and to the jobs representing the load's simulated software. Moreover, the code implementing the new operating-system functions is scattered over parts of several routines, thereby decreasing model clarity. It would be desirable to keep most or all of user-specified control program models in one place, either through a number of "operating-system-jobs" or some other unifying concept.

V. DESIGN CONSIDERATIONS FOR FURTHER ECSS DEVELOPMENT

Pointing out the weaknesses in something under development is usually tantamount to saying how it will be improved. Reflection on the above has further indicated some more general factors in designing computer system simulation languages. These may be used to anticipate future difficulties and correct them before they intrude into some new application of the language.

Declarative vs. Procedural Description

Although declarative specification of system features is most convenient, procedural specification is more flexible. Since one cannot anticipate all possible types of interaction, all interaction mechanisms should be accessible to procedural control so that the user can make the fullest use of provided capabilities. Declarative specifications should be retained, however, for their convenience, and perhaps even be expanded to further parameterize the built-in operating system (e.g., more opportunities for overhead time specification).

Statement Scope

Just as one cannot anticipate all possible interaction types, one cannot foresee all the combinations of ways to command the system components. Variable sophistication of detail requires greater user control over the actions of the load description statements. Powerful statements, incorporating a number of operations in a predetermined sequence are necessary to quickly develop coarse models. For more detailed models, each operation should be available separately. Control at a lower functional level allows the same Service Routine mechanisms to be put together in a greater number of ways without the chore of altering their structure. The inclusion of more optional clauses for statements may ease transitioning from coarse to fine levels of detail during model development.

Operating System Accessibility

Service Routine alteration could also be avoided by inclusion of a number of exit points to user routines. Such a "monitor" is included now but operates more as an observer, telling what the system did, not what it is about to do. Substituting an active monitor, with the power to skip some of the built-in operations on command, would provide more flexible use of the ECSS operating system features.

Automatic Summaries

A computer system should include automatic collection of statistics, and automatic output of statistical and other summaries, in addition to user access to all operation data. Certain values of interest are well enough known that statistics on them should be collected. Preformatted reports of these statistics would provide a convenient overview of model operation. Machine and man readable system structure summaries could further make the simulation clearer to the user and aid in rerunning the model. Trace-type output should also be available on demand (perhaps through the monitor) but undesired volumes of operational data should be avoided.

VI. SUMMARY

Use of the initial version of the Extendable Computer System Simulator has shown it to be a convenient and powerful analysis tool. Its provisions for describing both common computing system elements and operations eases much of the modeling burden, while its extendability and modifiability insure its suitability for uncommon applications. Certain ommissions and inflexibilities have been noticed, however, which has lead to some suggestions for improving its convenience. Implementation of these suggestions is proceeding.

REFERENCES

1. Nielsen, N. R., *ECSS: An Extendable Computer System Simulator,* The Rand Corporation, RM-6132-NASA, 1970.

2. Kiviat, P. J., R. Villanueva, and H. M. Markowitz, *The SIMSCRIPT II Programming Language,* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1968.

3. Kosy, D. W., *The Extendable Computer System Simulator Language Specification Manual,* The Rand Corporation, R-561, (in process).