ECSS: AN EXTENDABLE COMPUTER SYSTEM SIMULATOR*

Norman R. Nielsen

Stanford University and The RAND Corporation

Stanford, California Santa Monica, California

Abstract

An important aid in the construction of a complex computing system or network is a simulation model for analyzing and evaluating various design approaches. Unfortunately, the programming and debugging effort associated with the development of such a simulator all too often renders the project impractical. Accordingly, ECSS has been designed and specifically tailored to surmount these difficulties. Further, it is built as an extension of Simscript II, so that the full power of a general purpose simulation language is available to the user for unique features or special requirements.

1. INTRODUCTION

The value of simulation techniques has long been recognized as a means for investigating the behavior of complex or otherwise incompletely understood systems. This has been particularly true in the case of systems in the design process. The designer needs a tool to assist him in evaluating some of the alternative approaches which he might employ, in evaluating a system's overall performance when several new modules are to be integrated into that system. Each module may be well designed and well understood, but it is often unclear in the case of large systems how well a collection of these "good" modules will work together. All too often it is a matter of "hope for the best" and "wait and see". Thus, the need for and value of simulation studies.

The foregoing arguments are particularly true with respect to computer systems - hardware systems, software systems, communications systems, interconnected networks of systems, etc. The problems encountered by designers and developers of such systems are by now well known to all in the industry. Hardly a computer related periodical has been published in recent years without some mention of delays in delivering software systems, of difficulties with system performance, etc. Clearly, this is an area where simulation could be of great assistance - both to the designer of the system as well as to the ultimate user. As mentioned above the designer needs to test and evaluate his design without actually having to build the system first. The user needs to determine how his system will behave in his particular environment, how it should be configured to best serve his needs, how changes in his requirements might affect its performance, etc.

The aforementioned are areas where simulation could be of great assistance. A number of simulation

* This work has been supported by the RAND Corporation and by the National Aeronautics and Space Administration.

studies of this nature have been made (1); the value of the approach has been demonstrated. However, for the most part such studies reflect special cases. Many simulations are unable to be completed adequately within the alloted time and resource budgets. Many others are not even attempted because of projected difficulties. All too often the requirement is for additional information about a system "now". The designer or user does not have two years in which to develop a special simulation model to make the necessary analyses. In that period of time, for better or worse, the actual system would be constructed. In other cases the expenditures required for the development of the necessary simulation model would constitute a significant fraction of the total project budget.

Thus, although the value of simulation is recognized and although there is often a desire on the part of the project to make simulation analyses, the required resources often make the construction and use of a simulation model impractical. What is needed, then, is a modeling capability specifically tailored for the simulation of computer systems. This would permit system models to be developed much more rapidly with far fewer resources than is possible today. It is this need that led to the development of ECSS.

Now a number of packages or simulators have been developed to assist users in evaluating the performance of alternative computer systems or configurations under various types of loading. However, these simulators do not provide the flexibility and capability required for many types of computer simulation work. This is evidenced by the number of computer system simulation studies that have been made with models developed specifically for those investigations.

For example, consider SCERT (Systems and Computers Evaluation and Review Technique) (2). This package attempts to provide a very general model that can be adapted to a user's requirements. Parameters are set to reflect the performance capabilities of the hardware/software systems being studied and to reflect the user's work load. However, this is the user's only point of contact with the simulation. If the model does not contain a parameter to reflect a particular function or to adjust some activity, there is little or nothing that the user can do to overcome this difficulty. Thus, while SCERT can be of value in analyzing many straight-forward uses of existing systems, it is of little use in studying more complex systems or systems which are in the design stage.

As another example consider S3 (System and Software Simulator) (3). This package does not attempt to adapt a single model to serve each need; rather it seeks to create a model tailored to each need. Thus, the user is required to describe the capabilities of the system under investigation, the manner in which the software operates, and the behavior of the various jobs which will be processed by the system. Essentially, there is a "language" in which to make these descriptions. S3 processes the statements in the language to form a simulation model. The difficulty lies in the fact that the user must work within the context of a limited number of statement types. The statements are fairly specific and limited in purpose. If the user needs to describe something for which there is no statement type (e.g. communication between two computers), he is locked-out. There is no way to use several other statements to gain the same end.

2. PURPOSE OF ECSS

The purpose or design goal of ECSS is really many fold. However, to summarize briefly, the intent is to overcome many of the problems discussed above with respect to the simulation of computer systems. This general statement can be broken down into a number of more specific goals.

Since the overall aim is to improve the ease and speed with which one can develop a model of a computer system, it is desired that ECSS enable simple models as well as parts of more complex models to be built merely by requesting the necessary facilities and by providing appropriate input data. Now clearly it is impossible to develop a capability which would enable any user to reflect any type of system in this fashion. However, a subset of commonly used facilities and capabilities can be provided in this manner without the necessity of a substantial amount of programming on the part of the user.

An important goal stems from the need to reduce the programming and debugging necessary to reflect a system with some unique or specialized features. This need was the motivating factor behind the development of ECSS, since it is situations such as this in which simulation can be so valuable and yet so difficult to perform. Because of the particular or unique aspects of the system to be modeled, it is not possible to eliminate completely the need for programming on the part of the user. However, the amount of programming required can be greatly reduced by providing some higher level capabilities designed specifically for use in modeling computer systems. Further, facilities can be provided to look after many of the flag setting, counter checking, and pointer adjusting types of activities that are found in a simulation, thereby eliminating some of the tediousness from the programming chore and removing the source of many of the bugs which are so time-consuming to track down. Accordingly, these capabilities and facilities are among the features of ECSS.

Although the intent is to provide a general framework for modeling computer systems, this framework is not rigid. That is, if the user can not find the particular capability he needs (which will often be the case in this type of simulation), he is able to go outside the framework to the extent necessary in order to provide for himself the needed capabilities.

This is true both in the development of a simulation model and in the specification of the data for that model. That is to say, in specifying the model itself the user should be able to take advantage of the higher level constructs available to him. However, this should not preclude him from using a general purpose programming capability to accomplish certain functions for which no constructs are provided. Further, he should be able to specify his own constructs which would be advantageous or useful in the development of a particular model. Likewise, in specifying the data for his model, the user should be able to take advantage of the higher level facilities available. However, he still needs a general purpose capability to be able to specify characteristics or job behaviors in terms other than those provided. Thus, in one sense, ECSS is a helping hand that can be called upon to the extent desired, from 0% to 100%.

A further goal of ECSS is somewhat less apparent on the surface. Although it is desired to provide a framework for computer modeling, this framework should not dictate the structure or organization of the user's model. Much of the value of the assistance provided to the user would be lost if his model design were thereby forced into a particular channel. There are many examples of simulation studies which, although useful, were seriously hampered in their efficacy by the requirement that the model be structured along particular lines. The facilities and capabilities of ECSS are such that they can be of assistance whether the user desires to work with a flow oriented model or an event driven simulation, whether he wishes to take a gross overall look at a system or to investigate one particular subsection in minute detail.

3. LANGUAGE SELECTION

The design goals mentioned above for ECSS require that it provide general purpose capabilities. That is, it must be a general purpose programming language as well as a language with specific capabilities for the simulation of computer systems. The most effective way to accomplish this was to design ECSS as an extension to an existing language which had the necessary general purpose capabilities. First, the emphasis of the project was on the simulation of computer systems, not on the design of general purpose languages. Since there are many good languages in existence with general purpose capabilities, any resources devoted to the development of such capabilities would in a sense be wasted from the point of view of the project. Second, the need for a user to learn what would amount to an entirely new language in order to use ECSS would be a serious drawback.

Following the same line of reasoning it was also desirable to build ECSS as an extension to an existing simulation language. Although simulations of computer systems need a number of language features which are specific to the modeling of computer systems, such simulations still require many of the basic features provided by simulation languages. Thus, using the same type of arguments as above, it was desirable to avoid having to develop many of the basic simulation capabilities that are found in simulation languages.

In essence, then, ECSS was to be developed as an extension to an existing simulation language which had fairly powerful general purpose programming capabilities. Although this reliance upon existing languages would clearly impact the speed, efficiency, and user interface of ECSS, these considerations were of secondary importance. The purpose of the project was to develop a prototype version of ECSS in order to demonstrate the effectiveness of this approach to the simulation of complex computer systems. The emphasis was upon the new conceptual features of ECSS, not upon the re-implementation of well known language features. Should ECSS subsequently become popular but the host language prove to be a poor choice, it would then be possible to go back and implement a new version of ECSS which would be suitably independent of existing languages.

In keeping with this line of reasoning, the prototype version of ECSS was developed as a translator. The language extensions are thus translated into the host language as are any service routines that are supplied for a simulation model. The standard system for the host language then takes care of compiling, loading, etc. Not only does this procedure reduce the effort required to implement ECSS, but it also preserves the host language system as "standard." There are thus no ECSS modifications which have to be installed in any updates or new versions of the host system.

The need for powerful, general purpose programming capabilities as well as simulation capabilities narrowed the host language choice considerably. In fact the only real contenders were Simula (4) and Simscript. Both of these languages have most of the desirable features, and either would have been satisfactory for the implementation of ECSS. Thus, the selection of Simscript over Simula was made primarily on the basis of availability on the equipment to which the project had access.

The selection among the various versions of Simscript was also fairly straightforward. Simscript II (5) was chosen because of a number of additional features and improvements which it offers over the other earlier versions of Simscript. It provides the user with a much improved method of specifying the various entities and attributes which are to be used. Storage allocation is also improved as are the general purpose capabilities. The ability to make recursive subroutine calls is very helpful. The power of Simscript II is sufficient not only for the user's requirements but also for system's purposes, permitting the entire system to be developed and used within the context of the same language. The user of ECSS can work in Simscript II, the language features of ECSS are constructed as extensions to Simscript II, and the translator itself is written in Simscript II (with one exception - there is one capability that Simscript II does not have, and it was necessary to write a two instruction assembly language subroutine to remedy this deficiency).

4. DESIGN OVERVIEW

Before discussing some of the key features of ECSS, a broad overview will be given to illustrate the design and use of this tool. A schematic of the operation of the translator is shown in Figure 1. The input to ECSS consists of five parts: a "binary" summary deck, a definition section, a system description, a job load description, and any Simscript routines or data which the user himself has developed.

The summary deck is primarily an economy measure, providing the user with the ability to make certain types of small changes without the necessity of retranslating all of the input again. The first time that a particular model is input to ECSS there is, of course, no summary deck. However, the output of that translation includes, among other things, a deck indicating the contents of the summary tables which ECSS uses in constructing simulation models. These tables contain such information as system configuration, device capabilities, etc. Should a user subsequently wish to make a minor change in his model (e.g. change the number of disk units or alter the transmission rate of a channel), he need input only this summary deck followed by the source statements for the desired changes. It is not necessary to retranslate the entire system description. The making of more substantive changes, of course, necessitates the retranslation of the altered routines as well as the modification of any affected system routines. (These are provided by ECSS to perform some of the special capabilities in the object simulation program.) However, it would still not be necessary to retranslate the system description, since the summary deck essentially permits the recreation of the translator's tables as they stood at the completion of the previous translation.

The second part of the input data permits the user to define any special terms and units that are relevant for the particular model being constructed. Although the translator recognizes the relationship between such units as seconds and microseconds, the relationship between words and pages or words

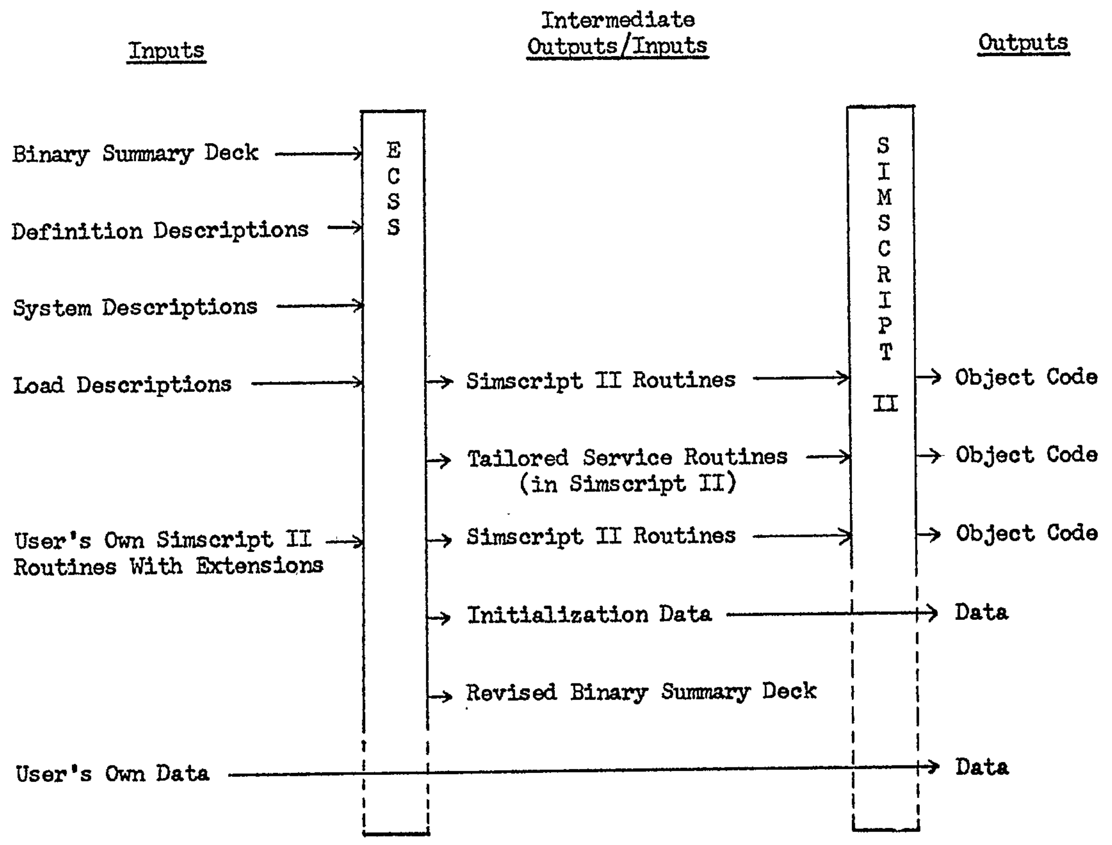

Figure 1

Schematic of ECSS Translation

and little.pages (or any other term a user might like to employ) is something that can not be fixed immutably for all users. Hence, the user is given an opportunity to define any such non-standard terms that he might wish to employ in developing a simulation. He might choose, for example, to define a spool.buffer as being 100 words in size and a clock.pulse as being 500 nanoseconds long. These terms could then be used throughout the simulation input, permitting a greater degree of readability and specification convenience.

The definitions can also indicate the names of and the data for table look-up functions. The capabilities of Simscript II are not as comprehensive in this area, and this feature enables the user to quickly and conveniently set up these frequently used functions. Finally, additional commands or operation types may be defined if the user wishes to supplement the standard ones provided for describing the job load to be simulated (see Section 7).

The third part of the input data permits the user to specify the hardware configuration and capabilities of the system to be modeled. Provision is made for a modest amount of software description. In keeping with the Simscript II philosophy, these specifications are in free form. There is no requirement to check boxes or to put certain information in specific columns on the input card. Section 6 contains a further discussion of the specification statements.

The fourth part of the input data permits the user to describe the behavior or characteristics of the jobs which will be "executed" on the simulated system. In addition to a number of statement types designed specifically for this purpose, the user may employ as many operations of his own construction as he wishes. Further, he may employ the full capabilities of Simscript II as well as the ECSS extensions to that language. Thus, the user has a substantial amount of power for developing job behavior descriptions. Section 7 contains a further discussion of the capabilities in this area.

The fifth part of the input data consists of regular Simscript programs which the user has written. The only action taken on this input is to translate the ECSS extensions into standard Simscript before passing everything on to the compiler. The routines in this section can have a variety of purposes. The user might be providing capabilities which are not offered by ECSS. Although one of the goals of the project was to provide many of the features which are commonly used in this type of simulation work, there was no attempt to provide all of the features that might be required in developing a particular simulation model. The user might also be modifying some of the service routines which are provided by ECSS. Everything that is in the object simulation model is open for the user to modify as he might see fit; he is not forced to use existing algorithms, conventions, etc. This is another of the steps taken in an attempt to give the user as much capability as possible without at the same time locking him into a particular structure.

As an illustration, ECSS provides the mechanisms for handling the paging operations of some time-sharing system simulations. However, it is next to impossible to anticipate how the user might like to control the paging. Therefore, the user must write his own control program. ECSS provides exits to the user's "monitor" at appropriate points. The user then need code only the tests necessary to determine when to initiate a paging operation, which pages to transfer, etc. A call issued to the paging mechanism at the appropriate time will handle the rest of the details.

The processing of the various inputs by the translator generates several outputs. The binary summary deck (mentioned above) is one of the by-products of the translation. Another important output is an initialization data deck for use with the object simulation program. This deck provides a degree of flexibility similar to that provided by the summary deck. If the user wishes to change his configuration or to adjust other parameters which he has specified, no change is required in the object program. Only the initialization data need be altered. Thus, just as it is not necessary to retranslate all of the system description in order to make this type of change, it is not necessary to recompile any of the Simscript routines. More extensive changes would, of course, necessitate recompilation of the affected routines. The other outputs of the translator consist of the translated descriptions and routines, the service routines provided by ECSS, and the user's own programs and data.

5. DESIGN - KEY FEATURES

One of the desirable features of a simulation package, and one which unfortunately is missing from Simscript II, is the ability to handle flow oriented types of problems. There are times when it is beneficial to be able to reflect things using this orientation rather than having to convert everything to an event driven orientation.* Accordingly two extensions were added to Simscript to handle this type of problem.

The first is the HOLD UNTIL statement which essentially interrupts processing at that point in the routine until a specified condition is satisfied. Processing will then resume with the succeeding statement. This capability is of particular importance in the processing (execution) of the behavior descriptions for the simulated jobs. Since Simscript II routines are recursive, there is no problem with a HOLD tieing-up a routine and preventing other uses of it. Only the specific instance which caused the HOLD to be executed is held. Thus, a single routine may be suspended in several different places by several different requests at any given point in time. All values of important local variables are preserved for the duration of the hold. The values of global variables may, of course, change in the interim.

A companion extension is the WHEN TRUE statement. This delimits a range of immediately following statements which are to be executed at such time as an indicated condition is true. The remainder of the routine containing the WHEN is processed normally; its execution is not interrupted. However, as soon as the referenced condition is satisfied, the indicated range of statements will be executed.

In the case of both the HOLD and the WHEN statements the user is not faced with some predetermined set of conditions from which he must choose. He is free to write his own function, incorporating a test of any type and complexity as appropriate. This greatly adds to the user's capabilities, but it can become somewhat inefficient in terms of execution time. Accordingly, there is a particular set of conditions which can be dealt with more efficiently. However, conceptually there is no distinction between the user supplied and translator provided functions.

ECSS was designed to provide the user with as great a range of prepared capabilities as possible. On the other hand, flexibility in using these capabilities was also important. Accordingly, if a user does not need or use a particular capability, his model is not penalized by the fact that he could have used that capability. Consider the following example.

There are capabilities within ECSS for handling CPU performance degradation due to CPU-I/O memory reference interference. However, if a user is making a fairly high level analysis (e.g. the allocation of disk storage space over the course of a day), the necessity to make memory interference calculations at every step would severely degrade the performance of the model. Accordingly, capabilities which are not used are made totally transparent to the final program. There is no penalty for having had the opportunity to employ these unused capabilities. However, if such an unused capability is subsequently required in the model, it will be necessary to retranslate a goodly portion of the program. Flexibility is never free, but this is a relatively small price

* In other words, one should be able to employ the GPSS type of world view.

to pay.

Another design feature along this same line concerns the partial overriding of system default capabilities by the user. For example, the simulation model generated for a user generally has certain default queue handling procedures for each device in the simulated system. However, the user may override these totally, providing his own queue handling algorithms; or he may override them selectively, providing his own algorithms only for certain devices.

6. SYSTEM SPECIFICATION

Another key aspect of ECSS is the manner in which the user can describe the system which he wishes to simulate. The procedures are fairly powerful, but they still permit the user a degree of flexibility. The description consists of a number of statements which the user can arrange in a convenient manner in order to specify the desired system.

The basic description statement is the SPECIFY statement, enabling the user to identify the basic devices to be included in his system. The translator recognizes such names as processor, channel, memory, control unit, and device as well as any other categories which the user might choose to employ. In addition, the user indicates whether the device is to be public (can be assigned for access or use by all users) or private (must be assigned exclusively to a single user). There is also the ability to assign a subclass name to the unit or units specified (see below). For example,*

```
SPECIFY 3 PRIVATE 60kb tape.drives
```

Tape.drives would be the user defined class name and 60kb would be the optional subclass name.

Devices can be categorized in four ways for convenient reference. The basic building block is the individual device or unit. Each such device can be referenced specifically by name if the user has provided one. The next level of aggregation is the subclass. This consists of all those units which were specified together and given a subclass name (e.g. the 3 tape drives specified above). The next higher level of aggregation is the class, which consists of all units having the same class name. Thus, for example,

```
SPECIFY 3 PRIVATE 60kb tape.drives
SPECIFY 2 PRIVATE 90kb tape.drives
```

would define a class called tape.drives having 5 devices in it. There would be two subclasses, one having 3 devices in it, the other having 2 devices.

There is also a SUBCLASS statement so that additional subclasses can be created. Thus, one could achieve the same effect as the previous example by stating

```
SPECIFY 5 PRIVATE tape.drives
FORM A SUBCLASS CALLED 60kb FROM tape.drives 1-3
FORM A SUBCLASS CALLED 90kb FROM tape.drives 4-5
```

One could go on to add

```
FORM A SUBCLASS CALLED 9.track FROM tape.drives 2-4
```

In other words, a device can belong to only one class but to any number of subclasses. However, each subclass must consist of a single series of sequential units.

The highest level of aggregation is the pool, which consists of any combination of devices, subclasses, and classes that the user wishes to specify. This category allows any desired grouping of devices and overcomes the limitation placed upon the membership of subclasses. Thus, in simulating job behavior on a system, reference can be made to any of these categories depending upon the level of specificity desired. If a transmission is to be made to a particular terminal, then the device name for that terminal can be used. If on the other hand the intent is to create a file, then the name of a class of disks or a pool of direct access devices

* In this and succeeding examples, the variables or names which the user provides are shown in lower case letters

could be given, and the system would select a particular device from the indicated group. There is a POOL statement, similar to the SUBCLASS statement, for the creation of pools.

In addition to the specification and logical grouping of the devices forming the system to be simulated, there are a number of modifying clauses which may be used to provide additional information about the various devices. These clauses may be appended to the specification statements or they may be used subsequently in the description, following the name of the device, subclass, class, or pool to which they apply. When a group name (e.g. a pool name) is modified by a clause, the information is applied to every device in that group.

There are seven broad categories of clauses available. One of these permits specification of the rate at which a device can transmit information and the rate at which it can execute instructions. This latter definition can be made in arbitrary terms. Thus it might be said that a processor executes 500,000 mix.a instructions per second and 350,000 mix.b instructions per second. The descriptions of job processing can then be stated in terms of mix.a and mix.b instructions (or whatever) and hence can be developed without reference to the system on which they are to be processed. Thus, the performance of a variety of systems could be simulated using the same job load without any modification of the job descriptions.

A second type of clause indicates the other devices to which a device or group of devices is connected. Thus a disk could be connected to a control unit, and the control unit could be connected to several disks and a channel, etc. Another clause indicates the other devices for which a group has responsibility in terms of allocation and deallocation. Normally one would specify that the processors have responsibility for allocating the peripheral units, but this need not be the case. The user can also specify the time required to make the allocation (deallocation) by the responsible device. As with most of the clauses, such parameters can be provided by means of either a constant or a function. In the latter case, the performance or requirements can be made to vary with the state of the device or with the other activities going on in the system.

A clause is available which indicates the amount of execution and/or transmission time absorbed by a device in connection with the processing of a message transmission. Another clause indicates the other devices whose operation is degraded by the operation of a particular unit.

Other types of clauses are available to indicate whether or not the operation of a device is interruptable and to indicate the capacity of a device in terms of space, number of users, number of simultaneous messages, maximum transmission rate, cumulative transmission rate for all simultaneous messages, etc. Thus, a substantial amount of information can be provided about the performance of the various devices on the system and about the inter-relationships among these devices.

To specify all of this information about each device can become rather tedious, even when use is made of pools and classes. Accordingly, there are two additional modifiers which can be appended to the SPECIFY statement. The first is the "TO BE THE SAME AS name" modifier, where name is some device, class, or pool which has already been specified. This enables all of the characteristics of the specified device or group to be set identically to those of the referenced device or group. The other modifier is the "TO BE THE SAME AS name EXCEPT" phrase. This also defines all the characteristics of the specified device or group to be like those of another device or group except for the changes indicated by the clauses immediately following.

In this way the user can indicate the configuration of the system to be simulated and can describe the capabilities and inter-relationships of this configuration. The input follows the Simscript II philosophy in that it is free form with a minimum of punctuation. There are no requirements that characteristics be indicated in any particular order, that information be placed in particular sections of cards, etc.

7. LOAD DESCRIPTION

One of the key factors in any computer system simulation is the manner in which the "jobs" or work load for the simulated system are to be represented. It is important to give the user a great deal of capability as well as a great deal of flexibility in this area. He must be able to reflect adequately whatever characteristics of the job load are important for his particular system model. At the same time the specification procedure must be easy to use.

This leads to a number of requirements. First, there must be a number of commands or instructions for use in specifying the characteristics and behavior of the work load. Second, the repertoire must provide testing, branching, looping, and subroutine capabilities. These features are quite important if one is to bring the size of job descriptions "under control." Third, references to other locations in the job description should be relative rather than absolute so that changes in the description can readily be made. Thus, an "assembler" is required for the description "language".

ECSS attempts to meet these requirements in several ways. First, there are thirteen special commands designed to assist in the description of the work load to be simulated. These commands break down into three groups. The first group consists of five resource allocation commands. ALLOCATE and DEALLOCATE are concerned respectively with the allocation (reservation, dedication) and deallocation of devices (e.g. tape drives, printers, etc.) to particular jobs. GET and FREE are concerned with the acquisition and release of space on devices (e.g. buffer space, disk space, etc.). The fifth command, FILE, is concerned with the placement of program and data files in the simulated system. The user may specify specific devices in using these commands, or he may use class or pool names. In these latter cases, an appropriate device will be selected from the group each time the command is executed.

The second group consists of five commands related to the execution of job load descriptions or sequences of commands. JOB indicates the beginning of a distinct new job entity or item of work and triggers the necessary internal bookkeeping and statistical routines. LAST indicates the completion of all of a job's processing steps. START JOB can be used to start a job initially, or it can be used by an executing job to start a subjob or subtask. It is possible to specify that particular types of jobs are to be created at specified intervals or after the occurrence of certain conditions. STEP and START STEP are analogous to the JOB and START JOB commands but refer to the start of a "subroutine" of commands or instructions.

The third group consists of three commands more directly concerned with the execution behavior of a simulated program. SEND is used to initiate the transmission of information or messages from one point in a system to another. The EXECUTE command is used to reflect execution requirements on a device (e.g. CPU, peripheral processor, etc.). The WAIT command is used in connection with the suspension of the processing of a job's description commands until certain conditions have been satisfied (e.g. all of that job's current I/O operations have been terminated, all of that job's subjobs have been completed, etc.).

Most of the commands have a number of options and parameters associated with them which provide a wide range of capability. The actual form of the commands follows the Simscript II philosophy and runs from a simple

```
ALLOCATE 2314.disk AS output.file
```

to a more complex

```
SEND MESSAGE OF LENGTH 58 FROM main.cpu
  TO user.terminal VIA terminal.path AS
  A RESPONSE TO terminal.job WITH PRIORITY 3
```

WAITING HERE for COMPLETION; WAITING AT checkpoint FOR RESPONSE

In addition there is a facility for the user to augment these commands with additional ones of his own choosing which would be of greater assistance in a particular application. This capability was provided since it would be impossible to foresee all of the possible commands which a user might at some time wish to employ.

The requirement for branching, looping, and other capabilities could well have been met by introducing still further commands into the description language. However, these features are already provided in Simscript II, so that such an effort would have been redundant. Accordingly, the whole of the Simscript II language is permitted to be used in the load descriptions, providing all of the required capabilities (and then some). Between the ECSS commands, the user's own commands, the general purpose capabilities of Simscript II, and the ECSS extensions to Simscript II, the user has quite an array of tools for describing the work loads to be simulated.

The requirement for a description "assembler" is met through the use of the Simscript II compiler. The description commands are translated into Simscript II, and then the entire description is processed by the regular compiler. Although this adds the extra step of translation, it eliminates the need to develop a description "assembler" which would also handle all of the Simscript II statements. Such a trade-off has obvious advantages for a pilot developmental project.

8. AN ILLUSTRATIVE APPLICATION

In order to provide the reader with an indication of how ECSS can be used to model a computer system, consider the following small information retrieval system as an example. This system consists of a single CPU, three random access disks, and 20 inquiry terminals. Users arrive at these terminals from time to time and query the system. The system makes reference to the data files on disk, obtains information about the user's identity, etc. The system then responds with the desired data and updates a master file with information about the requestor, the type of information necessary to satisfy his request, etc. The model has deliberately been simplified for illustrative purposes. Thus, its readily apparent limitations should not be taken as indicative of ECSS limitations.

Figure 2 shows the ECSS code necessary to describe the hardware system. The first three sections specify the hardware in the configuration. For simplicity no consideration has been given to I/O channels, disk control units, etc. However, these could easily be added in a manner analogous to that shown. For the same reason a fixed 75 milliseconds per disk I/O operation is used to reflect seek times, rotational delays, etc. Since all users employ the same retrieval program, no explicit mention is made of memory capacity. Rather, a constraint of 10 users is employed. Again, memory capacity, sizes of jobs, etc. could have been taken into consideration. However, as this example shows, considerations not required for a particular model can readily be ignored.

One statement is used to set up a subclass of the disks called MASTER.DISK. This subclass consists of a single disk which contains the master file for collecting data on the information necessary to respond to the terminal requests. The last lines in the system specification section merely indicate the communication paths in the system which may be used.

Figure 3 shows the job descriptions for the information retrieval application. There are two job types. The first, USER.QUERY, runs on a terminal and reflects the behavior of the user or requestor at a terminal. This illustrates the power of the ECSS job descriptions which can be used to describe the behavior of any part of the system. Jobs need not be restricted to execution on CPU's. The second job type, QUERY, runs on the CPU and actually processes the requests for information.

```
SYSTEM DESCRIPTION
SPECIFY 1 PUBLIC PROCESSOR, EXECUTES 100 MIX.A.INSTRUCTIONS PER MILLISECOND,
                            EXECUTES 150000 MIX.B.INSTRUCTIONS PER SECOND,
                            CONNECTS TO TERMINALS, DISKS,
                            HAS CAPACITY OF 10 EXECUTION USERS,
                            IS INTERRUPTABLE WITH THE OPERATION CONTINUING
SPECIFY 20 PUBLIC TERMINALS, EACH CONNECTS TO PROCESSOR,
                             HAS CAPACITY OF 1 EXECUTION USER,
                             HAS CAPACITY OF 1 TRANSMISSION USER,
                             TRANSMITS 4 WORDS PER SECOND
SPECIFY 3 PUBLIC DISKS, EACH TRANSMITS 100000 WORDS PER SECOND,
                        CONNECTS TO PROCESSOR,
                        HAS CAPACITY OF 1 TRANSMISSION USER,
                        ABSORBS 75 MILLISECONDS PER MESSAGE
FORM A SUBCLASS CALLED MASTER.DISK FROM DISKS 1 - 1
PATH REQUEST.PATH IS TERMINALS, PROCESSOR
PATH ANSWER.PATH IS PROCESSOR, TERMINALS
PATH PATH.TO.DISK IS PROCESSOR, DISKS
PATH PATH.FROM.DISK IS DISKS, PROCESSOR
END
```

Figure 2 Configuration Specification for the Information Retireval System

```
LOAD DESCRIPTION

JOB USER.QUERY

     LET REPETITIONS = UNIFORM.F(2,6,1)

     START JOB QUERY (THIS.UNIT, JOB, REPETITIONS) CALLED QUERY.JOB ON
          PROCESSOR

     FOR I = 1 TO REPETITIONS DO

          SEND MESSAGE OF LENGTH 14 TO PROCESSOR VIA REQUEST.PATH WITH
               RESPONSE FROM QUERY.JOB WITH PRIORITY 6 WAITING HERE FOR RESPONSE

          WAIT 10 SECONDS

     LOOP

LAST

JOB QUERY GIVEN UNIT, JOB, REPETITIONS

     HOLD UNTIL INPUT.ARRIVES EQ 0, THEN CONTINUE

     EXECUTE 2500 MIX.A.INSTRUCTIONS

     FOR I = 1 TO REPETITIONS - 1 DO

          SEND MESSAGE OF LENGTH NORMAL.F(15,3,1) TO UNIT VIA ANSWER.PATH
               AS A RESPONSE TO JOB WITH RESPONSE FROM JOB WITH PRIORITY 2
               WAITING AT REPLY FOR RESPONSE

          EXECUTE 1200 MIX.B.INSTRUCTIONS

          RECEIVE MESSAGE OF LENGTH 30 FROM DISKS VIA PATH.FROM.DISK WAITING
               HERE FOR COMPLETION

          EXECUTE 3450 MIX.B.INSTRUCTIONS

          'REPLY'

          EXECUTE NORMAL.F(1500,125,1) MIX.A.INSTRUCTIONS

     LOOP

     SEND MESSAGE OF LENGTH 50 TO UNIT VIA ANSWER.PATH AS A RESPONSE TO JOB
          WITH PRIORITY 2

     SEND MESSAGE OF LENGTH 75 TO MASTER.DISK VIA PATH.TO.DISK WAITING HERE
          FOR COMPLETION

LAST

INITIALLY START USER.QUERY ON EACH TERMINALS AT EXPONENTIAL.F(60,1) SECONDS
     AND EVERY EXPONENTIAL.F(300,1) SECONDS AFTER ARRIVAL
```

Figure 3 Job Descriptions for the Information Retrieval System

The description of the USER.QUERY job is fairly straightforward. The first step calculates the number of interactions that will take place between the user and the information retrieval program on the CPU. This number is generated from a uniform distribution between 2 and 6. The next step starts a subjob, namely the query program on the CPU. The user then begins interacting with the system. He makes a request, waits for a reply from the system, and then digests that reply. If the appropriate number of interactions have not been completed, the "respond and wait" behavior is repeated; otherwise the user is finished and leaves the terminal.

The description of the QUERY job to run on the CPU is somewhat more complicated. The initiating job (i.e. the terminal job) passes three parameters (UNIT, JOB, REPETITIONS). These indicate respectively the identity of the terminal that is to be serviced, the identity of the job on that terminal which is to be serviced (since potentially two or more users could share a terminal), and the number of interactions that will be required. The HOLD statement indicates that the job is to wait until input is received from the terminal. Then 2500 Mix A instructions are to be executed for the purpose of initializing the program and preparing a query to the terminal user.

The next section of the description indicates the repetitive behavior for each interaction. The program sends a reply to the terminal, performs some further processing, requests some information from one of the disk files, and then waits for completion of the disk I/O operation. Upon receipt of the information from the disk, some additional processing is done; then the program must wait for a reply from the terminal user. Upon receipt of this reply some additional processing is done, and then the program loops back to respond to the user again, etc. Upon completion of the appropriate number of iterations, the program provides the requested information to the user, updates the disk file with data about the information which was necessary to satisfy the user's request, and terminates.

Although in most cases the lengths of I/O messages are indicated by constants and the amounts of processing are indicated by constants, these data can also be supplied by functions. Thus, the query back to the terminal is specified as a normal distribution with a mean of 15 words and a standard deviation of 3. Likewise, at the bottom of the interaction loop the number of Mix A instructions is specified as a normal distribution with a mean of 1500 and a standard deviation of 125.

The QUERY jobs on the CPU are all initiated by USER.QUERY jobs on the terminals. The last statement in Figure 3 provides for the generation of the terminal jobs. An exponential distribution with a mean of 60 seconds is used to determine the time at which the first job will be started on each terminal. Thus, each terminal will initially become active at a different time. Further, each time a job arrives at a terminal, another exponential distribution will be used to determine when the next job of that type will arrive at that terminal. Thus, in this case, the arrival of new users will not be influenced by the number of users waiting for a terminal nor by the time taken to service any given user.

9. CURRENT STATUS AND PLANS

The specification and design of the ECSS package (language and service routines) is essentially complete; the implementation of the translator is currently underway. The service modules required at execution time to perform the various functions described above can consist of up to 2000 lines of Simscript II code. The actual size of each of the various routines is, of course, very much a function of the requirements of the user's simulation. As was discussed above, only the code for those facilities actually needed for a particular simulation is included in the service routines for that simulation.

Also, with regard to computing time resources, the use of a fairly general package for simulating computer systems will result in some disadvantage with respect to object time execution efficiency. However, the resource requirements of ECSS itself are of secondary importance to the project. The goal of the project is to reduce the manpower and time requirements necessary to get a computer system simulation up and running. Accordingly, it is measurements in these areas that will indicate the success or failure of ECSS. Once ECSS is operational, two fair size simulation efforts will be undertaken with it. Unfortunately, the resources available will not permit the duplication of the construction of these simulation models in other languages by personnel having equivalent experience, etc.. Thus, the test will not be as conclusive as might be desired from a statistical point of view. However, similar simulations have been developed previously, so the planned simulations should still provide an interesting comparison of the value of the ECSS approach in practice.

The present version of ECSS is a pilot effort designed to test the validity of an approach to the computer system simulation problem. Like any prototype it has a number of weaknesses. Probably the most serious is the overemphasis on hardware capabilities. That is, there are commands and facilities in ECSS to assist in modeling many types of hardware-related actions (e.g. selecting a path and transmitting a message to a device while taking into consideration device capabilities and constraints on that path). Although ECSS is not without software related capabilities, the user is generally not provided with the convenient means of use as he is with hardware related functions. For the most part it is necessary for the user to do some programming in order to reflect software features of the simulated system.

Given the organization of ECSS, it is not difficult to provide additional features or capabilities. The problem lies in determining what types of software system capabilities and options to include. For example, one might wish to provide a built-in paging capability, so that time-sharing systems using a page-turning strategy could easily be modeled. However, anticipating the variety of paging strategies that might be employed is a nontrivial task. Further research in this area is necessary, so that more of the programming tasks can be picked up by systems like ECSS rather than having to be handled by the builder of a simulation model.

10. SUMMARY

The building of a simulation model for a computer system or network is often a very time-consuming and expensive task. Yet, such an analysis tool is frequently needed in the design, installation, and operational phases of these complex systems. Accordingly, some extensions to an existing simulation language have been developed to assist in carrying out some of the activities that are common to many computer system simulations.

The resulting package, ECSS, is built around a translator which converts the special features and commands into Simscript II for compilation into object code. Like any general package designed to serve more than one purpose, there will be execution inefficiencies stemming from the use of ECSS. However, by removing much of the detail work from the hands of the programmer, he can concentrate on the essential logic of the simulation. By thus reducing the time and resources required to develop and debug simulation models, it is hoped that a greater percentage of the complex computer systems to be designed would be analyzed prior to their construction. Not only would some costly failures be avoided, but some improved systems might result.

REFERENCES

1. Nielsen, N.R., "Computer Simulation of Computer System Performance," Proceedings of the 22nd National Conference, ACM, Thompson Book Company, Washington, D.C., August 1967, pp. 581-590.

2. "SCERT: Systems and Computers Evaluation and Review Technique," Comress, 1967.

3. "S3: System and Software Simulator," Defense Document Clearinghouse, AD 679269, September 1967.

4. Dahl, O., Myhrhaug, B., and Nygaard, K., "SIMULA 67," Norwegian Computing Center, Oslo, 1968.

5. Kiviat, P.J., Villanueva, R., and Markowitz, H.M., The SIMSCRIPT II Programming Language, Prentice-Hall, Englewood Cliffs, N.J., 1969.

ABOUT THE AUTHOR

Norman Nielsen has been involved in the simulation of several large scale computing systems and has thus experienced some of the problems associated with the simulation of these types of systems. The lessons learned in the course of these investigations are reflected in the design of ECSS. Nielsen is currently serving as a consultant to the Computer Systems Analysis section of the RAND Corporation's Computer Sciences Department. At Stanford he holds a joint appointment as Assistant Professor of Operations and Systems Analysis in the Stanford Graduate School of Business and as Deputy Director of the Stanford Computation Center.