General Disclaimer

One or more of the Following Statements may affect this Document

- This document has been reproduced from the best copy furnished by the organizational source. It is being released in the interest of making available as much information as possible.

- This document may contain data, which exceeds the sheet parameters. It was furnished in this condition by the organizational source and is the best copy available.

- This document may contain tone-on-tone or color graphs, charts and/or pictures, which have been reproduced in black and white.

- This document is paginated as submitted by the original source.

- Portions of this document are not fully legible due to the historical nature of some of the material. However, it is the best reproduction available from the original submission.

Produced by the NASA Center for Aerospace Information (CASI)

T 71-01249
NAS 12-2144

MEMORANDUM
RM-6132-NASA
FEBRUARY 1970

ECSS: AN EXTENDABLE COMPUTER SYSTEM SIMULATOR

N. R. Nielsen

FACILITY FORM 602
N70-29793 (ACCESSION NUMBER)
(THRU) 1
52 (PAGES)
(CODE) 08
CR-86403 (NASA CR OR TMX OR AD NUMBER)
(CATEGORY)

PREPARED FOR:
NATIONAL AERONAUTICS AND SPACE ADMINISTRATION

The RAND Corporation
SANTA MONICA • CALIFORNIA

MEMORANDUM
RM-6132-NASA
FEBRUARY 1970

ECSS: AN EXTENDABLE COMPUTER SYSTEM SIMULATOR

N. R. Nielsen

This research is sponsored by the National Aeronautics and Space Administration under Contract No. NAS 12-2144. This report does not necessarily represent the views of the National Aeronautics and Space Administration.

The RAND Corporation
1700 MAIN ST • SANTA MONICA • CALIFORNIA • 9040(

PRECEDING PAGE BLANK NOT FILMED.

PREFACE

This Memorandum describes the major design features of a special-purpose language for simulating computer systems. The language is being developed under a research contract with NASA (National Aeronautics and Space Administration) Electronic Research Center; the goal of the contract is to determine ways of providing better tools for the design and evaluation of computer systems. The Memorandum should be of interest to designers, analysts, and managers of computer-based systems, and to designers and users of simulation languages.

N. R. Nielsen, a Consultant to RAND's Computer Sciences Department, is Associate Director of the Computer Center at Stanford University.

PRECEDING PAGE BLANK NOT FILMED.

SUMMARY

This Memorandum contains both a review of current techniques for simulating computer systems and a description of the major design features of a special-purpose language for such simulations: ECSS: Extendable Computer System Simulator.

The Memorandum discusses the major design decisions leading to the current structure of the language, and presents some examples of its capabilities and use. One major design decision has been to make the language as "natural" as possible. The following example summarizes the ECSS approach by showing how a typical element of a computer simulation is described in ECSS and in other simulation languages. (SIMSCRIPT II is used here.)

ECSS:

```
SEND MESSAGE OF LENGTH 58 FROM PROCESSOR TO TERMINAL.A
VIA MESSAGE.PATH WITH PRIORITY 3
```

SIMSCRIPT II:

```
LET COMPLETION = 58 * TRANSMISSION.RATE (TERMINAL.A)
CALL TIME.OF.FREE.PATH (PROCESSOR, MESSAGE.PATH (*), 3)
  YIELDING EXTRA.WAIT
IF CURRENT.PRIORITY < 3
     LET EXTRA.WAIT = 0
     REGARDLESS
SCHEDULE A    MAKE.TRANSMISSION AT EXTRA.WAIT + TIME.V
SCHEDULE AN   END.TRANSMISSION AT EXTRA.WAIT + COMPLETION
                                    + TIME.V
```

Another major design decision has been to make it easy for users to extend the language beyond the ECSS facilities provided. Thus, if a user has difficulty describing a system with the standard ECSS facilities, he may easily write SIMSCRIPT II extensions to the ECSS part of the model. Having these features in the language should allow designers and

analysts to realize more efficient, flexible, and consistent computer systems through greater use of simulation both before and during the processes of system design and tuning.

CONTENTS

PREFACE iii

SUMMARY v

Section

I. INTRODUCTION 1
- Advantages of Simulating Computer Systems . 1
- Difficulties in Simulating Computer Systems 1
- Existing Simulators of Computer Systems ... 2

II. DESIGN GOALS OF ECSS 5
- Provide Facilities 5
- Minimize Housekeeping Requirements 5
- Provide Capabilities for Extending the Language 6
- Avoid Overstructuring 6

III. LANGUAGE SELECTION 8
- Rationale for Basing ECSS on an Existing Simulation Language 8
- Choice of Existing Simulation Language 9

IV. LANGUAGE DESIGN--OVERVIEW 11
- Binary Summary Deck 11
- Definition Section 11
- System Description 13
- Job Load Description 13
- User-Supplied SIMSCRIPT Programs 14
- Operation of the Translator 15

V. LANGUAGE DESIGN--KEY FEATURES 16
- Flow-Oriented Simulation Features 16
- Making Capabilities Transparent 17
- Overriding System Default Capabilities 17

VI. SYSTEM SPECIFICATION 19
- Specifying System Elements 19
- Categorizing System Elements 19
- Additional Information--Modifying Clauses . 22
- Labor-Saving Shortcuts 23

VII. LOAD DESCRIPTION 25
- Requirements 25
- Workload Description 25
- Extra Options and Extensions 26
- Branching and Looping Capabilities 27
- Assembler Capability 27

VIII. AN ILLUSTRATIVE APPLICATION OF ECSS 28
Configuration Specification 28
Job Specification 31

IX. CURRENT STATUS AND FUTURE DIRECTIONS 34
Testing 34
Possible Extensions 34

X. CONCLUSIONS 36

APPENDIX 37

REFERENCES 45

I. INTRODUCTION

ADVANTAGES OF SIMULATING COMPUTER SYSTEMS

Simulation techniques have long been recognized as a valuable means for investigating the behavior of complex systems. This has been particularly true for systems in the design process. The designer needs a tool to help him examine alternative approaches for evaluating a system's overall performance--especially when several new modules are to be integrated into that system. Each module may be well designed and well understood, but in the case of large systems, how well a collection of "good" modules will work together is frequently unclear.

These arguments are particularly true concerning computer systems--hardware systems, software systems, communications systems, and interconnected networks of systems. The problems encountered by designers and developers of such systems are now well known in the industry. Most computer-related periodicals published in recent years have mentioned: 1) delays in delivering software systems; 2) difficulties with system performance; or 3) incompatibilities among elements in a computer system. Clearly, these are areas in which simulation could greatly assist both the designer and the ultimate user. The designer needs to test and evaluate his design without having to build the system first. Also, the user needs to determine how the system will behave in a particular environment, how it should be configured to serve his needs best, and how changes in requirements would affect its performance.

DIFFICULTIES IN SIMULATING COMPUTER SYSTEMS

Many simulation studies of the aforementioned areas have been made [1], and the value of the approach has been demonstrated. However, most such studies reflect special cases.

Many simulations are unable to be completed adequately within the allotted time and resource budgets. Many others are not even attempted because of projected difficulties. Too often the requirement is for immediate information about a system. 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, the actual system would be constructed. In other cases, the expenditures required to develop the necessary simulation model would constitute a significant fraction of the total project budget.

Thus, although the value of making simulation analyses is recognized and although projects often desire to employ this tool, the resources required often prohibit such an undertaking. The need, then, is for a modeling capability specifically tailored for the simulation of computer systems, a capability that would permit system models to be developed much more rapidly with fewer resources than is possible today. This need has led to the development of ECSS (Extendable Computer System Simulator).

EXISTING SIMULATORS OF COMPUTER SYSTEMS

Many 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 inadequacy is evidenced by the number of computer-system simulation studies that have been performed with models developed specifically for those investigations.

The following three studies, each covering a different type of computer system, illustrate this point: 1) Katz developed a model, using a general-purpose simulation language, to analyze System/360 batch-processing computer systems [2]; 2) Rehmann and Gangwere performed a simulation

analysis of a time-sharing system [3], constructing a simulation model specifically tailored for their analysis; and 3) Merikallio and Holland used a simulation model, developed without the use of a computer simulation package, to investigate the performance of an on-line, real-time air traffic control system [4].

SCERT (Systems and Computers Evaluation and Review Technique) [5] is one example of a computer simulation package. This package attempts to provide a 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, the user can do little to overcome this difficulty. Thus, SCERT can be valuable in analyzing many straightforward uses of existing systems, but it is of little use in studying more complex systems or systems in the design stage.

Another example, S3 (System and Software Simulator) [6], creates a model tailored to each need. Thus, the user is required to describe: 1) the capabilities of the system under investigation; 2) the manner in which the software operates; and 3) the behavior of the various jobs that the system will process. Essentially, a "language" exists in which to make these descriptions. S3 processes the statements in the language to form a simulation model. Problems arise because the user must work within the context of a limited number of statement types that are fairly specific and limited in purpose. He cannot describe something for which no statement type exists (e.g., communication between two computers). The use of other statements will not produce the same result.

The work of Zurcher and Randell illustrates yet a different approach [7]. They have developed a language that

enables the user to construct more effective models of complex systems. Essentially, their language permits the user to describe his system on several levels. Only when results are needed concerning a lower or more detailed level of behavior in one of the system's operational areas is reference made to the next level of the model; i.e., one may employ different levels of treatment for different portions of the model. Thus, greater accuracy is provided while computational requirements are reduced. This technique, however, is applicable to all large system simulations; computer systems illustrate only one possible area of application. At present, no report has been made describing language features designed to assist with only the modeling of computer systems.

II. DESIGN GOALS OF ECSS

The purposes--or design goals--of ECSS (Extendable Computer System Simulator) are many. Briefly, the intent is to solve many of the problems that arise in the simulation of computer systems. This general statement can be broken into a number of more specific goals.

PROVIDE FACILITIES

Since the overall aim is to improve the ease and the speed with which one can develop a model of a computer system, ECSS should 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. To develop a capability that would enable any user to reflect any type of system in this manner is clearly impossible. However, a subset of commonly used facilities and capabilities can be provided in this way so that the user does not have a substantial amount of programming.

MINIMIZE HOUSEKEEPING REQUIREMENTS

Another important goal is to reduce the programming and debugging necessary for reflecting a system with unique or specialized features. The need for this reduction has been the motivating factor behind the development of ECSS, since such situations make simulation extremely valuable and yet so difficult to perform. Because of the unique aspects of the system to be modeled, the need for programming by the user cannot be eliminated completely. 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 handle many of the flag-setting, counter-checking, and pointer-adjusting activities

that are found in a simulation, thereby eliminating some of the tedious detail from the programming chore and removing the source of many of the program bugs that are so time-consuming to eliminate. Accordingly, these capabilities and facilities are among the features of ECSS.

PROVIDE CAPABILITIES FOR EXTENDING THE LANGUAGE

Although the intent is to provide a general framework for modeling computer systems, this framework is not rigid. That is, if the user cannot 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 acquire the needed capabilities.

This is true both in the development of a simulation model and in the specification of the data for that model; i.e., in specifying the model, 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 in the development of a particular model. Also, in specifying the data for his model, the user should be able to use the higher-level facilities available. However, he still needs a general-purpose capability to specify characteristics or job behaviors in terms other than those provided. Thus, ECSS can be used to the extent desired, from 0 percent to 100 percent.

AVOID OVERSTRUCTURING

Another goal of ECSS is less apparent on the surface. Although a framework for computer modeling is desired, it should not dictate the structure or organization of the user's model. If the user's model design is forced into

a particular channel, much of its effectiveness is lost. However, the facilities and capabilities of ECSS are designed to be of assistance when the user desires to work either with a flow-oriented model or with an event-driven simulation, and when he wishes either to take an overall look at a system or to investigate only one particular subsection in minute detail. A single model may contain both characteristics.

III. LANGUAGE SELECTION

RATIONALE FOR BASING ECSS ON AN EXISTING SIMULATION LANGUAGE

The design goals mentioned above require that ECSS provide general-purpose capabilities; i.e., 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 that 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 many good languages with general-purpose capabilities existed, any resources devoted to the development of such capabilities would have been impractical. Second, requiring a user to learn an entirely new language to use ECSS was considered a serious drawback.

Although relying upon existing languages would clearly affect 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 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 reimplementation of well-known language features.

The prototype version of ECSS has been developed as a translator. The language extensions are thus translated into the host language as are any service routines supplied for a simulation model. The standard system for the host language takes care of, for example, compiling, and loading. Not only does this procedure reduce the effort required to implement ECSS, but it also preserves the host language system as "standard." Thus, no ECSS modifications have to be installed in any updates or new versions of the host system.

CHOICE OF EXISTING SIMULATION LANGUAGES

The need for powerful, general-purpose programming capabilities as well as simulation capabilities narrowed the host language choice considerably. It particularly eliminated such a language as GPSS [8], which has limited general-purpose programming capabilities. The only real contenders were SIMSCRIPT and SIMULA [9]. Both languages had most of the desirable features, and either would have been satisfactory for the implementation of ECSS. The selection of SIMSCRIPT over SIMULA was made primarily on the basis of the availability of the equipment to which the project had access.

SIMULA offers slightly more capability than does SIMSCRIPT. Particularly, the SIMULA concept of a "process," which is both an "event subroutine" and an "entity" in the simulation, is an extremely powerful feature. This capability enables SIMULA to handle flow-type problems effectively. Thus, the choice of SIMSCRIPT as the host language for ECSS necessitated the development of some SIMSCRIPT extensions to provide a simular capability (see Sec. V, p. 16). However, the differences between SIMSCRIPT and SIMULA are not so significant as to outweigh the value of local availability for this application.

SIMSCRIPT II [10] was chosen because it offers additional features and improvements that the earlier versions of SIMSCRIPT (the original version of SIMSCRIPT [11] and SIMSCRIPT 1.5 [12] do not. It provides the user with a much improved method of specifying the various entities and attributes that will be used. Storage allocation is also improved, as are the general-purpose capabilities; also the ability to make recursive subroutine calls is helpful. The power of SIMSCRIPT II is sufficient not only for the user's requirements but also for the system's purposes, permitting the entire system to be developed and used within the context of the same language. Thus, the user of ECSS

can work in SIMSCRIPT II; the language features of ECSS are constructed as extensions to SIMSCRIPT II; and the ECSS translator is written in SIMSCRIPT II.†

However, the choice of SIMSCRIPT II as a base language has dictated many of the characteristics of the ECSS translator. For example, SIMSCRIPT II is available only in a batch-processing version; therefore, ECSS could not have any interactive capabilities. Similarly, the compilation and execution efficiency for models constructed in ECSS will be strongly influenced by the compilation speeds of SIMSCRIPT II and by the quality of the object code emitted by SIMSCRIPT II.

---

†Because SIMSCRIPT II lacks one necessary capability, a two-instruction, assembly-language subroutine has been written to remedy the deficiency. This routine accesses the pointer to the base of the stack for SIMSCRIPT II's dynamic local variable storage.

IV. LANGUAGE DESIGN--OVERVIEW

The input to ECSS consists of five parts: 1) a "binary" summary deck; 2) a definition section; 3) a system description; 4) a job-load description; and 5) any SIMSCRIPT routines or data that the user has developed. Figure 1 is a schematic that shows how those elements are operated upon by ECSS and by SIMSCRIPT II to develop a simulator.

BINARY SUMMARY DECK

The summary deck is primarily an economy measure, providing the user with the ability to make certain types of small changes without retranslating the input. When a particular model is first input to ECSS, no summary deck exists. However, the output of that translation includes, for example, a deck indicating the contents of the summary tables that ECSS uses in constructing simulation models. These tables contain such information as system configuration and device capabilities. 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 inputs only this summary deck followed by the source statements for the desired changes. He is not required to retranslate the entire system description. If the user desires to make more substantive changes, he must retranslate the altered routines and modify any affected system routines. (These are provided by ECSS to perform some of the special capabilities in the object simulation program.) However, retranslation of the system description would still not be necessary since the summary deck permits the re-creation of the translator's tables as they were at the completion of the previous translation.

DEFINITION SECTION

The second part of the input data permits the user to define any special terms and units that are relevant to the

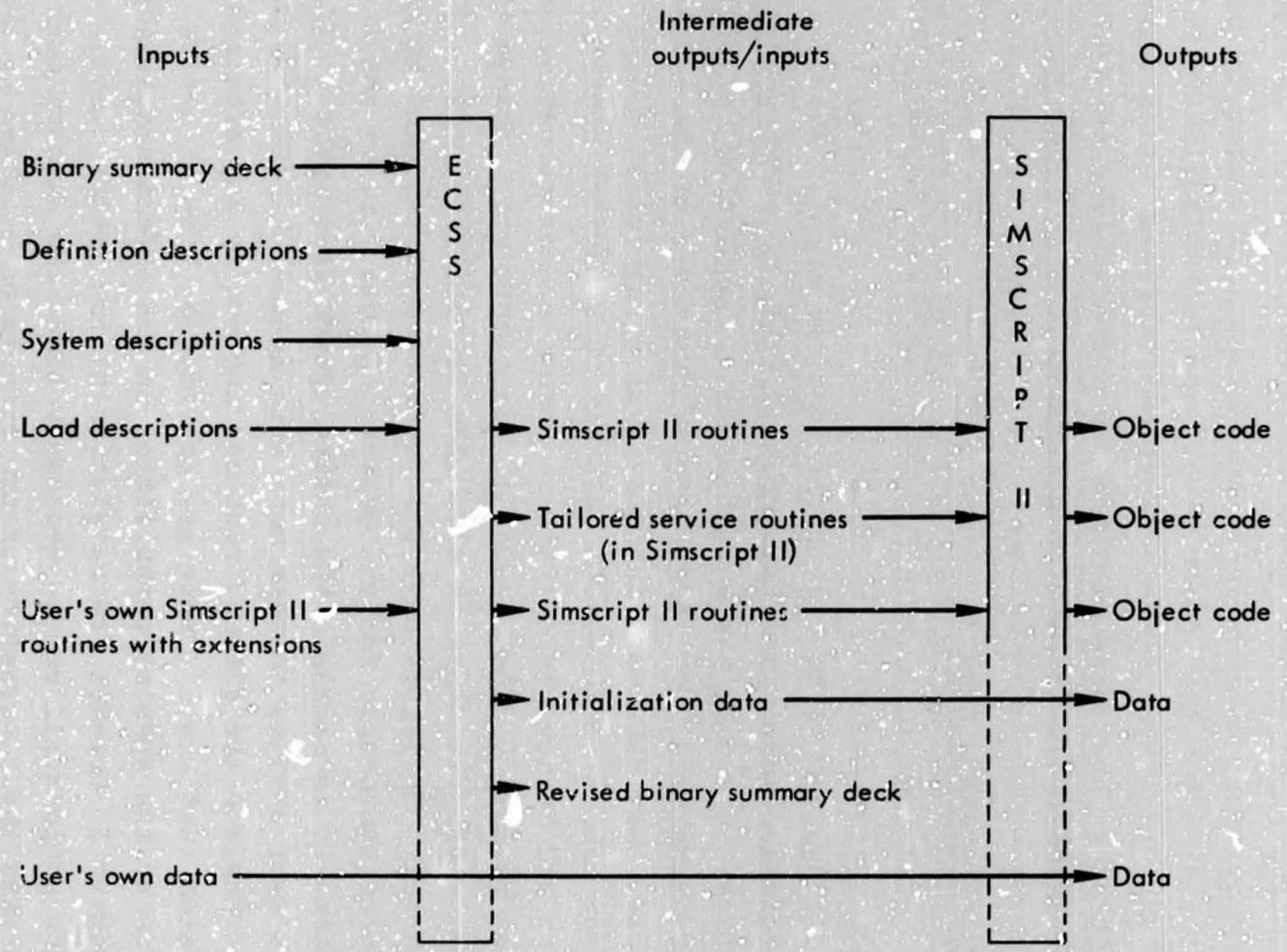


Fig. 1--Schematic of ECSS Translation

model being constructed. Although the translator recognizes the relationship between such units as seconds and microseconds, the relationship between words and pages or words and little.pages (or any other term a user might like to employ) cannot 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 ns long. These terms could then be used throughout the simulation input, permitting a greater degree of readability and specification convenience.

The definitions also can indicate the names of and the data for table look-up functions. The capabilities of SIMSCRIPT II are not comprehensive in this area, and this feature of ECSS enables the user to set up these frequently used functions quickly and conveniently. Additional commands or operation types may be defined if the user wishes to supplement the standard ones that are provided for describing the job load to be simulated (see Sec. VII, p. 27).

SYSTEM DESCRIPTION

The third part of the input data permits the user to specify the hardware configuration and the capabilities of the system to be modeled. Provision is made for a modest amount of software description; e.g., I/O processing or handling times can be indicated. In keeping with the SIMSCRIPT II philosophy, these specifications are in free form. The user is not required to check boxes or to put certain information in specific columns on the input card. Section VI contains a further discussion of the specification statements.

JOB LOAD DESCRIPTION

The fourth part of the input data permits the user to describe the behavior or characteristics of the jobs that

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. These descriptions also can provide information relative to the operation of the software system (e.g., queuing priorities). Section VII contains a further discussion of these capabilities.

USER-SUPPLIED SIMSCRIPT PROGRAMS

Regular SIMSCRIPT II programs written by the user constitute the fifth part of the input data. The only action taken on this input is to translate any ECSS extensions into standard SIMSCRIPT II before passing everything to the compiler. These programs can have a variety of purposes, including the provision of capabilities that are not offered by ECSS. Although one of the project's goals has been to provide features that are commonly used in this type of simulation work, no attempt has been made to provide all of the features that might be required in developing a particular simulation model. These programs might also be used to modify some of the service routines that are provided by ECSS. The user may modify anything in the object simulation model; he is not forced to use, for example, existing algorithms, or conventions. This step, too, attempts to give the user as much capability as possible without, at the same time, locking him in to a particular structure.

As an illustration, ECSS provides the mechanisms for handling the paging operations of some time-sharing system simulations. However, anticipating how the user might like to control the paging is impossible. Therefore, the user must write a new control program. ECSS provides exits to

the user's "monitor" at appropriate points. The user must then code, for example, only the tests that determine when to initiate a paging operation, and the pages to be transferred. A call issued to the paging mechanism at the appropriate time will handle the remaining details.

OPERATION OF THE TRANSLATOR

The processing of the various inputs by the translator generates several outputs. The previously mentioned binary summary deck is one of the translation's by-products. An initialization data deck used with the object simulation program is another important output. This deck provides flexibility similar to that provided by the summary deck. If the user wishes to change his configuration or to adjust other parameters that he has specified, no change is required in the object program. Only the initialization data must be altered. Thus, the user is not required to retranslate all of the system description to make this type of change, nor is he required to recompile any of the SIMSCRIPT II routines. However, more extensive changes would necessitate recompilation of the affected routines. The other outputs of the translator consist of: 1) the translated descriptions and routines; 2) the service routines provided by ECSS; and 3) the user's own programs and data.

V. LANGUAGE DESIGN--KEY FEATURES[†]

FLOW-ORIENTED SIMULATION FEATURES

One desirable feature of a simulation package--and one that is missing, unfortunately, from SIMSCRIPT II--is the ability to handle flow-oriented problems. At times it is beneficial to be able to reflect a system using this orientation rather than having to convert everything to an event-driven orientation; i.e., one should be able to employ the GPSS type of world view. Accordingly, two extensions have been added to SIMSCRIPT to handle this type of problem.

The first extension is the HOLD UNTIL statement, which interrupts processing at that point in the routine until the specified condition is satisfied. Processing will then resume with the succeeding statement. This capability is particularly important in the processing (execution) of behavior descriptions for the simulated jobs. Since SIMSCRIPT II routines are recursive, a HOLD will not prevent subsequent uses of a routine. Only the specific instance that 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 change in the interim.

A companion extension is the WHEN TRUE statement. This delimits a range of statements that 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 also be executed.

[†]A summary of the ECSS vocabulary and syntax is given in the Appendix (p. 37).

In the case of the HOLD and the WHEN statements, the user is not faced with a 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 adds greatly to the user's capabilities, but this flexibility is provided at the expense of execution time. Because of the efficiency problem, ECSS has been designed to treat a particular set of conditions more effectively. Conceptually, however, no distinction exists between the user-supplied and the translator-provided functions.

MAKING CAPABILITIES TRANSPARENT

ECSS has been designed to provide the user with as great a range of prepared capabilities as possible. However, flexibility in using these capabilities also has been important. Accordingly, if a user does not need or use a particular capability, his model is not penalized. Consider the following example.

Within ECSS are capabilities for handling CPU performance degradation due to CPU-I/O memory-referenced interference. However, if a user is performing a fairly high-level analysis (e.g., the allocation of disk storage space over the course of a day), making memory-interference calculations at every step would severely degrade the performance of the model, while contributing only marginally to its applicability. If unused, this capability will be made totally transparent to the final program.

In the event that an unused capability is subsequently required in the model, the user must retranslate a large portion of the program.

OVERRIDING SYSTEM DEFAULT CAPABILITIES

Another design feature 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 totally override these by providing his own queue-handling algorithms; or he may override them selectively by providing his own algorithms only for certain devices.

VI. SYSTEM SPECIFICATION

Another key aspect of ECSS is the convenient manner in which the user can describe the system that he wishes to simulate. The description consists of a number of statements that are arranged by the user to specify the desired system.

SPECIFYING SYSTEM ELEMENTS

The basic description statement is the SPECIFY statement, which enables 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 that the user might choose to define and use. 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). Also, a subclass name can be assigned 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.

CATEGORIZING SYSTEM ELEMENTS

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, which consists of all units that were specified together and were given a subclass

†In this and succeeding examples, the variables or names that the user provides are shown in lower-case letters.

name (e.g., the 3 60kb tape.drives specified above). The next level is the class, which consists of all units having the same class name. For example,

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

would define a class called tape.drives, having five devices in it. There would be two subclasses; one having three devices in it, the other having two devices.

Also, a SUBCLASS statement exists 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 add:

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

A device can belong to only one class, but to any number of subclasses. However, for the sake of efficiency, 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, other pools, 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. Figure 2 shows a schematic of the classification hierarchy. In simulating job behavior on a system, reference can be made to any of the four categories, depending on the level of specificity desired. If a transmission is to be made to a particular terminal, the device name for that terminal can be used. If, however, the intent is to create a file, the name of a class of disks or a pool of direct-access devices could be given, and the system would select a particular device from the indicated group. A POOL statement,

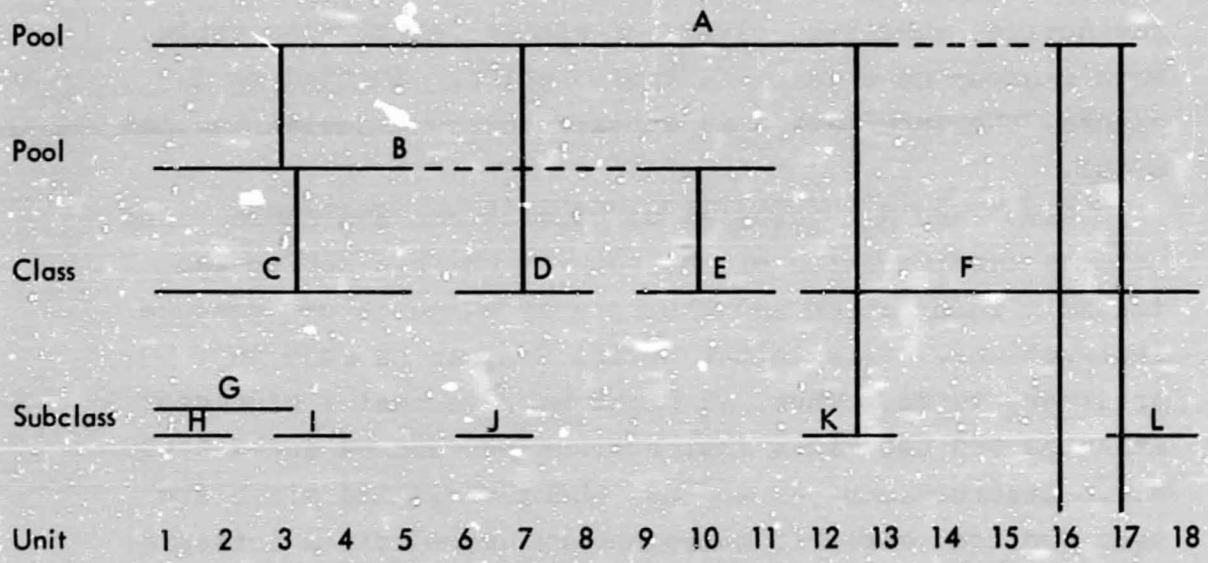


Fig. 2--Schematic of Classification Hierarchy

similar to the SUBCLASS statement, is available for the creation of pools.

ADDITIONAL INFORMATION--MODIFYING CLAUSES

In addition to the specification and the logical grouping of devices forming the system to be simulated, a number of modifying clauses 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.

Seven broad categories of clauses are available. One 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, where mix.a and mix.b are user-provided variable names having user-defined interpretations. The descriptions of job processing can then be stated in terms of mix.a and mix.b instructions (or other terms) 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 modifying job descriptions.

A second clause type indicates the other devices to which a device or a group of devices is connected; i.e., a disk could be connected to a control unit, and the control unit could be connected to several disks and a channel.

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 the responsibility of allocating the peripheral units, but this need not be true. The user also can 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 either a constant or a function. In the latter case, the performance or the requirements can be made to vary with the state of the device or with the other activities in the system.

One clause indicates the amount of execution and/or transmission time absorbed by a device during the processing of a message transmission. Another clause indicates the other devices whose activity 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 interruptible and to indicate the capacity of a device; e.g., space, number of users, number of simultaneous messages, maximum transmission rate, and cumulative transmission rate for all simultaneous messages. Thus, much information can be provided about the performance of the various devices on the system and about their interrelationships.

LABOR-SAVING SHORTCUTS

To specify all of the preceding information about each device would be tedious, even with the use of pools and classes. Therefore, two additional modifiers can be appended to the SPECIFY statement. First is the "TO BE THE SAME AS name" modifier, where name is some device, class, or pool that has already been specified. This enables all 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 characteristics of the specified device or group to be like those of another device or group, except for changes indicated by the clauses immediately following the phrase.

In this way, the user can indicate the configuration of the system to be simulated and can describe the capabilities and interrelationships of this configuration. The input follows the SIMSCRIPT II philosophy as it is free form with a minimum of punctuation. The user is not required, for example, to indicate characteristics in any particular order, or to place information in particular sections of cards.

VII. LOAD DESCRIPTION

One key factor in any computer-system simulation is the manner in which the work load is to be represented. The user should be given a great deal of capability as well as flexibility in this area. He must be able to reflect adequately those characteristics of the job load that are important for his particular system model. At the same time, the specification procedure must be easy to use.

REQUIREMENTS

A number of commands or instructions are needed for specifying the characteristics and behavior of the work load. Also, the repertoire must provide testing, branching, looping, and subroutine capabilities. These features are important if one is to control the size of job descriptions. 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."

WORKLOAD DESCRIPTION

ECSS attempts to meet these requirements in several ways. Thirteen special commands have been designed to assist in the description of the work load to be simulated; these commands can be divided into three groups. The first group consists of five resource-allocation commands: ALLOCATE and DEALLOCATE deal respectively with the allocation (reservation, dedication) and deallocation of devices (e.g., tape drives and printers) to particular jobs; GET and FREE deal with the acquisition and release of space on devices (e.g., buffer space and disk space); FILE, the fifth command, is concerned with the placement of program

and data files in the simulated system. The user may specify devices when using these commands, or he may use class or pool names. In the 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 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.) Although STEP and START STEP are analogous to the JOB and START JOB commands, they 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, or peripheral processor). The WAIT command deals with suspending 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, or all of that job's subjobs have been completed).

EXTRA OPTIONS AND EXTENSIONS

Most of the commands have a number of options and parameters, providing a wide capability range. The 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.
```

Also, a facility exists that allows a user to augment these commands with those of his own choosing. This capability has been provided since it would be impossible to foresee all possible commands that a user might wish to employ.

BRANCHING AND LOOPING CAPABILITIES

The requirements for branching, looping, and other capabilities could have been met by introducing more commands into the description language. However, SIMSCRIPT II includes these features, so such an effort would have been redundant. Accordingly, the whole SIMSCRIPT II language can be used in the load descriptions, thus providing all of the required capabilities. Considering 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 a large array of tools for describing the work loads to be simulated.

ASSEMBLER CAPABILITY

The SIMSCRIPT II compiler meets the requirement for a description "assembler." The description commands are translated into SIMSCRIPT II, and 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" that would also handle all of the SIMSCRIPT II statements. Such a trade-off has obvious advantages for a pilot developmental project.

VIII. AN ILLUSTRATIVE APPLICATION OF ECSS

To give the reader a better understanding of how ECSS can be used to model a computer system, the following small information-retrieval system is simulated as an example.† The system consists of one CPU, three random-access disks, and twenty inquiry terminals that allow users to query the system. The system refers to the data files on disk, and obtains information about the user's identity. It then provides the desired data and updates a master file with, for example, information about the requestor, and the type of information necessary to satisfy his request. The model has deliberately been simplified for illustrative purposes. Thus, its readily apparent limitations should not be considered indicative of ECSS limitations.

CONFIGURATION SPECIFICATION

Figure 3 shows the ECSS code necessary for describing the hardware system. The first three sections specify the hardware in the configuration. For simplicity, no consideration has been given to I/O channels, or disk-control units. However, these could easily be added in a manner analogous to that shown. For the same reason, a fixed 75 ms per disk I/O operation is used to reflect seek times and rotational delays. Also the CPU message-processing time of 500 μsec could easily be made dependent upon the type of message being handled. Since all users employ the same retrieval program, no explicit mention is made of memory capacity. Instead, a constraint of 10 users is employed. As this example shows, considerations not required for a particular model can readily be ignored.

---

†The reader should also compare the ECSS descriptions in Figs. 3 and 4 with the equivalent descriptions of the system were it to be simulated in FORTRAN, PL/1, or even SIMSCRIPT.

```
SYSTEM DESCRIPTION
SPECIFY 1 PUBLIC PROCESSOR, EXECUTES 100 MIX.A.INSTRUCTION PER MILLISECOND,
                            EXECUTES 150000 MIX.B.INSTRUCTIONS PER SECOND,
                            CONNECTS TO TERMINALS, DISKS,
                            HAS CAPACITY OF 10 EXECUTION USERS,
                            ABSORBS 500 MICROSECONDS PER MESSAGE,
                            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
```

Fig. 3--Configuration Specification for the Information Retrieval System

```
LOAD DESCRIPTION

JOB USER.QUERY

     LET REPETITIONS = UNIFORM.F(2.0,6.0,1)

     LET THIS.UNIT = .ACTUAL.EXECUTING.DEVICE(.JOB)

     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 AS A
               RESPONSE TO QUERY.JOB WITH RESPONSE FROM QUERY.JOB WAITING
               HERE FOR RESPONSE

          WAIT 10 SECONDS

     LOOP

LAST

JOB QUERY (UNIT, JOB, REPETITIONS)

     HOLD UNTIL INPUT ARRIVES THEN CONTINUE

     EXECUTE 2500 MIX.A.INSTRUCTIONS

     FOR I = 1 TO REPETITIONS - 1 DO

          SEND MESSAGE OF LENGTH NORMAL.F(15.0,3.0,1) TO UNIT VIA ANSWER.PATH
               AS A RESPONSE TO JOB WITH RESPONSE FROM JOB
               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(1500.0,125.0,1) 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(60.0,1) SECONDS
     AND EVERY EXPONENTIAL.F(300.0,1) SECONDS AFTER ARRIVAL
```

Fig. 4--Job Descriptions for the Information Retrieval System

One statement is used to establish a subclass of disks called MASTER.DISK. This subclass consists of a single disk that 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 that may be used.

JOB SPECIFICATION

Figure 4 shows the job descriptions for the information-retrieval application. The first job type, USER.QUERY, runs on a terminal and reflects user behavior at a terminal. This illustrates the power of the ECSS job descriptions that can be used to describe the behavior of any part of the system. Jobs need not be restricted to execution on CPUs. The second job type, QUERY, runs on the CPU and processes the requests for information.

These simplified job descriptions are analogous to the simplified system description presented in Fig. 3. Since the logical relationships in the tasks to be executed by the retrieval system are basically more complex, the descriptions in Fig. 4 do not seem as clear as the system description in Fig. 3.

The USER.QUERY job description is fairly straightforward. The first step calculates the number of interactions that will occur between the user and the information-retrieval program on the CPU. This number is generated from a uniform distribution between two and six. The next step starts a subjob; namely, the query program on the CPU. The user then interacts with the system, making a request, waiting for a reply, and then digesting the reply. If the appropriate number of interactions has not been completed, the "respond and wait" behavior is repeated; otherwise, the user is finished and leaves the terminal.

The QUERY job description is more complicated. The initiating job (i.e., the terminal job) passes three parameters (UNIT, JOB, REPETITIONS). These indicate respectively: 1) the identity of the terminal to be serviced; 2) the identity of the job on that terminal (since potentially two or more users could share a terminal); and 3) 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 initializing the program and preparing a query to the terminal user.

The next section of the description indicates the repetitive behavior of each interaction. The program sends a reply to the terminal, performs further processing, requests 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, additional processing is done; then the program must await a reply from the terminal user. After receiving this reply, more processing is done, and the program loops back to respond to the user again. When the appropriate number of iterations has been completed, the program provides the requested information to the user, updates the disk file with data concerning the information necessary to satisfy the user's request, and terminates.

Although in most cases the lengths of I/O messages 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. Also, 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.

All QUERY jobs on the CPU are initiated on the terminals by USER.QUERY jobs. The last statement in Fig. 4 provides

for the generation of terminal jobs. An exponential distribution with a mean of 60 sec determines the time at which the first job will start on each terminal. Thus, each terminal can become active initially at a different time. Further, each time a job arrives at a terminal, another exponential distribution will determine when the next job of that type will arrive. 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.

IX. CURRENT STATUS AND FUTURE DIRECTIONS

The ECSS specification and design package (language and service routines) is essentially complete; and 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 a maximum of 2000 lines of SIMSCRIPT II code. The size of each routine varies with the user's simulation requirements. Only the code for those facilities needed for a particular simulation is included in that simulation's service routines. Also, the use of a general package for simulating computer systems will adversely affect object-time execution efficiency.

TESTING

The resource requirements of ECSS are of secondary importance to the project. The primary goal is to reduce the manpower and the time requirements necessary for putting a computer system simulation in operation. Accordingly, measurements in these areas will indicate the success or failure of ECSS. Once ECSS is operational, two simulations will be undertaken with it. Unfortunately, available resources will not permit these simulation models to be duplicated in other languages by personnel having equivalent experience. Thus, the test will not be as conclusive as one might desire, from a statistical viewpoint. However, similar simulations have been developed previously, so the planned simulations should provide an interesting comparison to the value of the ECSS approach in practice.

POSSIBLE EXTENSIONS

The present version of ECSS is a pilot effort in approaching the computer-system simulation problem. Like any prototype, it has a number of weaknesses, the most serious

being the overemphasis on hardware capabilities. That is, commands and facilities in ECSS assist in modeling many types of hardware-related actions (e.g., selecting a path and transmitting a message to a device while considering device capabilities and constraints on that path). Although ECSS does have some software-related capabilities, it does not offer the user an array of software-related commands and facilities similar to that provided for the specification of the hardware system.† Generally, the user must resort to some SIMSCRIPT II programming to reflect software features of the simulated system.

Because of the nature of ECSS organization, providing additional features or capabilities is not difficult. 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 not an easy task. Further research in this area is necessary so that more programming tasks can be picked up by systems like ECSS instead of being handled by the builder of a simulation model.

---

†See Appendix (p. 37) for a summary of the ECSS vocabulary and syntax.

X. CONCLUSIONS

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

Use of the resulting package, ECSS, will relieve the programmer of much detail work and thus allow him to concentrate on the essential logic of the simulation. By reducing the time and resources required to develop and debug simulation models, it is hoped that a greater percentage of the future complex computer systems will be analyzed prior to construction. Not only would some costly failures be avoided, but some improved systems might result.

APPENDIX

This Appendix describes the grammar of the ECSS language. The notation and format of "The SIMSCRIPT II Programming Language: Reference Manual," RM-5776-PR, by P. J. Kiviat and R. Villanueva has been used. Basic constructs of SIMSCRIPT II described in that Memorandum are also used, with the exception that strings and text may not be used in ECSS statements. The grammatical primitive "name" may, in the remainder of the Appendix, be modified with preceding words (e.g., "path name") to clarify certain statements. It is still interpreted grammatically as a "name."

This material is provided simply to give the reader a better understanding of the nature of ECSS. A more comprehensive user's manual will be provided later.

I. DEFINITION SECTION

Terms

time-units = ({ SECONDS / SEC } { MILLISECONDS / MS } { MICROSECONDS / US } { NANOSECONDS / NS })

The ECSS translator recognizes the above time units, the SIMSCRIPT II time units, and any units related to these by a DEFINE UNITS statement.

space-units = { *user name* / SPACE.UNIT }

Storage may be specified in this basic unit or in units related to it by a DEFINE UNITS statement.

unit-dimension = *user name*

Any name may function as a unit dimension when appropriately defined by a DEFINE UNITS statement.

transmission-units = { *user name* / TRANSMISSION.UNIT }

The amount of I/O to be transmitted may be specified in this basic unit or a unit related to it by the DEFINE UNITS statement.

transmission-rate = *transmission-units* [/ PER] *time-units*

The ratio of I/O transmission is specified with this unit.

Statements

1) DEFINITION DESCRIPTION

Marks the beginning of the definition section.

2) DEFINE UNITS $\left\{ [n] \left\{ \begin{matrix} \text{transmission-units} \\ \text{time-units} \\ \text{space-units} \\ \text{unit-dimensions} \end{matrix} \right\} = [n] \left\{ \begin{matrix} \text{transmission-units} \\ \text{time-units} \\ \text{space-units} \\ \text{unit-dimensions} \end{matrix} \right\} \right\}^{o}$

unit-dimension

Establishes relationships between units, or defines those names to be used as unit-dimensions unrelated to other units.

3) DEFINE JOB OPERATIONS TO BE (name)^{o}

Defines new commands to be recognized in the load-description section.

4) DEFINE [TABLE] FUNCTION name TO BE $\left\{ \begin{matrix} \text{CONTINUOUS} \\ \text{DISCRETE} \end{matrix} \right\}$ WITH n VALUES $\left\{ n \; n \right\}^{n}$

Calling this function with one argument returns a number from the table specified by the listed value pairs, which means independent value-dependent value. The function will interpolate if CONTINUOUS is specified.

5) END

Marks the end of the definition section.

II. SYSTEM DESCRIPTION SECTION

Terms

device group dg = $\left\{ \begin{matrix} \textit{device name} \\ \textit{subclass name} \\ \textit{class name} \\ \textit{pool name} \end{matrix} \right\}$

Device, subclass, class, and pool are levels of hardware aggregation. A pool consists of any combination of pools, classes, subclasses, and devices; a class consists of subclasses and devices; and a subclass consists of a combination of devices.

$clauses$ = $\left\{ \begin{matrix} \text{execution clause} \\ \text{transmission clause} \\ \text{connection clause} \\ \text{allocation clause} \\ \text{capacity clause} \\ \text{absorption clause} \\ \text{degradation clause} \\ \text{interruption clause} \end{matrix} \right\}$

These clauses specify hardware capabilities as follows:

1) execution clause =

$$\left\{ n \text{ unit-dimensions PER time-units} \right\}^{o}$$

Indicates processing rate of the simulated components.

2) transmission clause =

TRANSMITS n transmission-units PER time-units

Indicates transmission rate of the simulated components.

3) connection clause =

CONNECTS TO $\{dg\}^{o}$

Indicates what other devices or groups of devices are directly connected in the simulated system.

4) allocation clause =

$$\left\{ \begin{matrix} \text{ALLOCATES} \\ \text{DEALLOCATES} \end{matrix} \right\} \left\{ \left\{ \begin{matrix} \text{ALL} \\ dg \left[\text{IN} \left\{ \begin{matrix} n \\ \text{time-units} \end{matrix} \right. \right. \text{time-units ACCORDING TO } \textit{function name} \left. \right\} \right] \\ \text{ALL EXCEPT } \{dg\}^{o} \end{matrix} \right\}^{o} \right\}$$

Specifies hardware-device allocation.

5) capacity clause =

$$\text{HAS} \left\{ \begin{matrix} \text{CAPACITY} \\ \text{SIZE} \end{matrix} \right\} \text{OF} \left[\begin{matrix} n \\ \textit{function name} \end{matrix} \left\{ \begin{matrix} \text{TRANSMISSION USERS} \\ \text{EXECUTION USERS} \\ \textit{transmission-rate} \text{ CUMULATIVE} \\ \textit{transmission-rate} \text{ ABSOLUTE} \\ \textit{space-units} \end{matrix} \right\} \right]^{o}$$

Specifies various upper bounds on hardware.

6) absorption clause

$$\text{ABSORBS} \left\{ \begin{matrix} n \text{ time-units PER MESSAGE} \\ \text{time-units PER MESSAGE ACCORDING TO } \textit{function name} \end{matrix} \right\}$$

Specifies message-processing time.

7) degradation clause =

$$\text{DEGRADES}\left\{dg \quad \text{BY}\left\{\begin{array}{l} n \quad \% \quad \text{PER} \quad n \ \textit{transmission-rate} \\ \% \\ \text{PERCENTAGE} \end{array}\right\} \text{ACCORDING TO } \textit{function name}\right\}^{\circ}$$

Indicates other devices degraded by the operation of a particular unit.

8) interruption clause =

$$[\text{IS}]\ \text{INTERRUPTABLE WITH [THE] OPERATION}\left\{\begin{array}{l}\text{CONTINUING}\\ \text{RESTARTING}\end{array}\right\}$$

Indicates type of interrupt handling.

Statements

1) SYSTEM DESCRIPTION

Marks the beginning of the system description section.

2) SPECIFY n $\left[\begin{array}{l}\text{PUBLIC}\\ \text{PRIVATE}\end{array}\right]$ [SUBCLASS NAME]

$$\textit{class name}\left[\begin{array}{l}\text{TO BE THE SAME AS } \textit{name}\ \left[\text{EXCEPT}\quad \{\textit{clause}\}^{n}\right]\\ \text{, [EACH]}\ \{\textit{clause}\}^{n}\end{array}\right]$$

Identifies the basic devices to be included in the simulated system and their capabilities.

3) ORGANIZE A POOL CALLED *name* CONSISTING OF $\{dg\}^{\circ}$

Creates a pool from previously defined device groups.

4) FORM A SUBCLASS CALLED *name* FROM $\left\{\begin{array}{l}\textit{class name}\\ \textit{subclass name}\end{array}\right\}$ n - $\left\{\begin{array}{l}\text{LAST}\\ n\end{array}\right\}$

Creates a subclass from the indicated members of a class or a subclass.

5) $\left\{\begin{array}{l}\text{POOL}\\ \text{CLASS}\\ \text{SUBCLASS}\\ \text{DEVICE}\\ \text{EACH}\end{array}\right\}$ dg $\{\textit{clauses}\}^{n}$

Specifies hardware characteristics for a device group.

6) MAKE *name* $\left\{\begin{matrix}\text{EQ}\\\text{EQUIVALENT}\end{matrix}\right\}$ TO *class name* (*n*)

Creates a device name for a specific class number.

7) DELETE $(dg)^{o}$

Allows changes to a system specified by a binary summary deck.

8) PATH name IS $(dg)^{o}$

Creates a named sequence of connected device groups.

9) END

Terminates the system description section.

III. JOB LOAD DESCRIPTION SECTION

Terms

device function *df* =

$$\text{FUNCTION } name \left\{ \text{OF } \left\{\text{DEVICE}\right\} s \left\{\begin{matrix}\text{TRANSMISSION}\\\text{EXECUTION}\\\text{USERS}\\\text{SPACE}\end{matrix}\right\} \right\}^{o}$$

A user-defined function may have quantities dependent on the status of various devices as arguments.

Statements

1) LOAD DESCRIPTION

Marks the beginning of the load-description section.

2) $\left\{\begin{matrix}\text{JOB}\\\text{STEP}\end{matrix}\right\} \left\{\begin{matrix}\text{TO}\\\text{FOR}\end{matrix}\right\} name \left[\begin{matrix}\left\{\begin{matrix}\text{THE}\\\text{THIS}\\\text{GIVING}\\\text{GIVEN}\end{matrix}\right\} (name)^{o}\\ ((name)^{o})\end{matrix}\right]$

Indicates the beginning of a job or a step to be run on the simulated system.

3) $\left\{\begin{matrix}\text{DATA}\\\text{PROGRAM}\end{matrix}\right\}$ FILE *name* LOCATED ON *dg*

Names a file to be located on a specific device or a device in the indicated device group.

4) ALLOCATE *dg* AS *name*

Picks a device called name to be used by the job or step.

5) $\left\{\begin{matrix}\text{SEND}\\\text{RECEIVE}\end{matrix}\right\}\left\{\begin{matrix}\text{MESSAGE}\\\text{RECORD}\end{matrix}\right\}$ OF REMOTE $\left\{\begin{matrix}d\\df\end{matrix}\right\}\left[\begin{matrix}\text{FROM}\left\{\begin{matrix}\text{HERE}\\d\\df\end{matrix}\right\}\\\text{TO}\left\{\begin{matrix}\text{HERE}\\d\\df\end{matrix}\right\}\end{matrix}\right]^{*}$ VIA *path name*

$$\left[\begin{matrix}\text{WITH RESPONSE FROM } d\\\text{AS A RESPONSE (TO } d\text{)}\\\text{WITH PRIORITY }\left\{\begin{matrix}d\\df\end{matrix}\right\}\\\left[\text{WAITING }\left[\begin{matrix}\text{AT label:}\\\text{HERE}\end{matrix}\right]\text{ FOR }\left\{\begin{matrix}\text{COMPLETION}\\\text{RESPONSE}\end{matrix}\right\}\right]\end{matrix}\right]^{*}$$

Specifies a message to be sent via a given path. Execution may be specified to resume at a given point after completion of the message or after a response to the message is received. Subscripted labels are not allowed.

6) WAIT FOR $\left\{\begin{matrix}\text{COMPLETION OF ALL }\left\{\begin{matrix}\text{RESPONSES}\\\text{I/O}\\\text{JOBS}\end{matrix}\right\}\\\left\{\begin{matrix}d\\df\end{matrix}\right\}\ \textit{time-units}\end{matrix}\right\}$

Processing should resume at this point when the indicated conditions are satisfied.

7) DEALLOCATE *name*

Frees a previously allocated device.

8) $\left\{\begin{matrix}\text{EXECUTE}\\\text{TIEUP}\end{matrix}\right\}\left\{\begin{matrix}d\\df\end{matrix}\right\}$ *unit-dimensions* $\left[\text{WITH PRIORITY }\left\{\begin{matrix}d\\df\end{matrix}\right\}\right]$

Defines an amount of processing to be done by the simulated device on which the job or step is executing.

9) GET { a / df / ANOTHER } [CONTIGUOUS] space-units [{ ON / OF / FROM } name / CALLED name]

Specifies an amount of primary or secondary storage to be used by the job or step.

10) FREE { a / df / ANOTHER } space-units [{ ON / OF / FROM } name / CALLED name]*

Releases storage space.

11) START STEP name [{ THE / THIS / GIVING / GIVEN } (s)* / ((s)*)] [YIELDING (name)*]

A step is a "subroutine" to be executed on the simulated system with no delay.

12) START JOB name [CALLED name] [{ THE / THIS / GIVING / GIVEN } (s)* / ((s)*)]

ON [EACH] { a / df } [WITH PRIORITY { a / df } / { IN / AT } { a / df } time-units / WAITING [AT label / HERE] FOR COMPLETION]*

Indicates that the job should be executed on the simulated system on the indicated device either immediately or when some conditions are met. The processing may be directed to continue at a given point after the job is completed.

13) INITIALLY START name [{ THE / THIS / GIVING / GIVEN } (s)* / ((s)*)] ON [EACH] { a / df }

[WITH PRIORITY { a / df } / { IN / AT } { a / df } time-units / AND EVERY { a / df } time-units AFTER { COMPLETION / ARRIVAL }]*

Specifies when a job is to be started, if it is not initiated by another job or step. Jobs may be started repetitively.

14) *user defined job operation name* [THE / THIS / GIVING / GIVEN] *(e)*^{o} [YIELDING *(name)*^{o}]

Calls a user-provided routine to be treated like a job operation such as EXECUTE or ALLOCATE.

15) LAST

Marks the end of a job or step.

16) END

Terminates load-description section.

IV. SIMSCRIPT EXTENSIONS

In addition to the special sections described above, regular SIMSCRIPT routines may be included in the simulation model. Several extensions to the standard SIMSCRIPT language are provided to allow development of a flow-oriented model, similar to a GPSS model, if desired. These extensions are described below.

1) WHEN [IT IS] TRUE THAT conditional function name [*(e)*^{o}]

EQ 0 [,] THEN DO [DOWN] TO *label*

When conditions specified in the function are met, control is passed to the statement list delimited by the label.

2) HOLD UNTIL { {INPUT / MESSAGE / RECORD} ARRIVES / *conditional function name* [*(e)*^{o}] EQ 0 } [,] [THEN CONTINUE]

Processing is interrupted at this point to be resumed when the condition is satisfied.

3) CONDITIONAL FUNCTION [TO / FOR] *name* [{THE / THIS / GIVEN / GIVING} *(name)*^{o} / *((name)*^{o}*)*]

Begins conditional function definition.

4) START *job name* [THE / THIS / GIVING / GIVEN] *(e)*^{o} / *((e)*^{o}*)*;] ON [EACH] {*e* / *df*} [FROM {*e* / *df*} / WITH PRIORITY {*e* / *df*} / {IN / AT} {*e* / *df*} *time-units*]

A job may be started by any user routine.

REFERENCES

1. Nielsen, N. R., "Computer Simulation of Computer System Performance," *Proc. of 22nd Natl. Conf, ACM*, Thompson Book Company, Washington, D.C., August 1967, pp. 581-590.

2. Katz, J. H., "An Experimental Model of System/360," *Comm. of ACM*, Vol. 10, November 1967, pp. 694-702.

3. Rehmann, S. L., and S. G. Gangwere, Jr., "A Simulation Study of Resource Management in a Time-Sharing System," *Proc. of 1968 FJCC*, Thompson Book Company, Washington, D.C., December 1968, pp. 1411-1430.

4. Merikallio, R. A., and F. C. Holland, "Simulation Design of a Multiprocessing System," *Proc. of 1968 FJCC*, Thompson Book Company, Washington, D.C., December 1968, pp. 1399-1410.

5. *SCERT: Systems and Computers Evaluation and Review Technique*, Comress, Inc., Washington, D.C., 1967.

6. *S3: System and Software Simulator*, Defense Document Clearinghouse, AD 679269, September 1967.

7. Zurcher, F. W., and B. Randell, "Iterative Multi-Level Modeling--A Methodology for Computer System Design;" *Proc. of IFIP Congress*, August 1968, pp. D138-D142.

8. *General Purpose Simulation System/360, User's Manual*, International Business Machines, Inc., Form H20-0326-2, 1968.

9. Dahl, O. J., and K. Nygaard, *SIMULA--A Language for Programming and Description of Discrete Event Systems*, Norwegian Computing Center, Oslo, 1966.

10. Kiviat, P. J., R. Villanueva, and H. M. Markowitz, *The SIMSCRIPT II Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1969. (Also, R-460-PR, The RAND Corporation, October 1968.)

11. Markowitz, H. M., B. Hausner, and H. W. Karr, *SIMSCRIPT: A Simulation Programming Language*, Prentice-Hall, Englewood Cliffs, N.J., 1963.

12. Karr, H. W., H. Kleine, and H. W. Markowitz, *SIMSCRIPT I.5*, California Analysis Centers, Inc., Santa Monica, Calif., 1966.