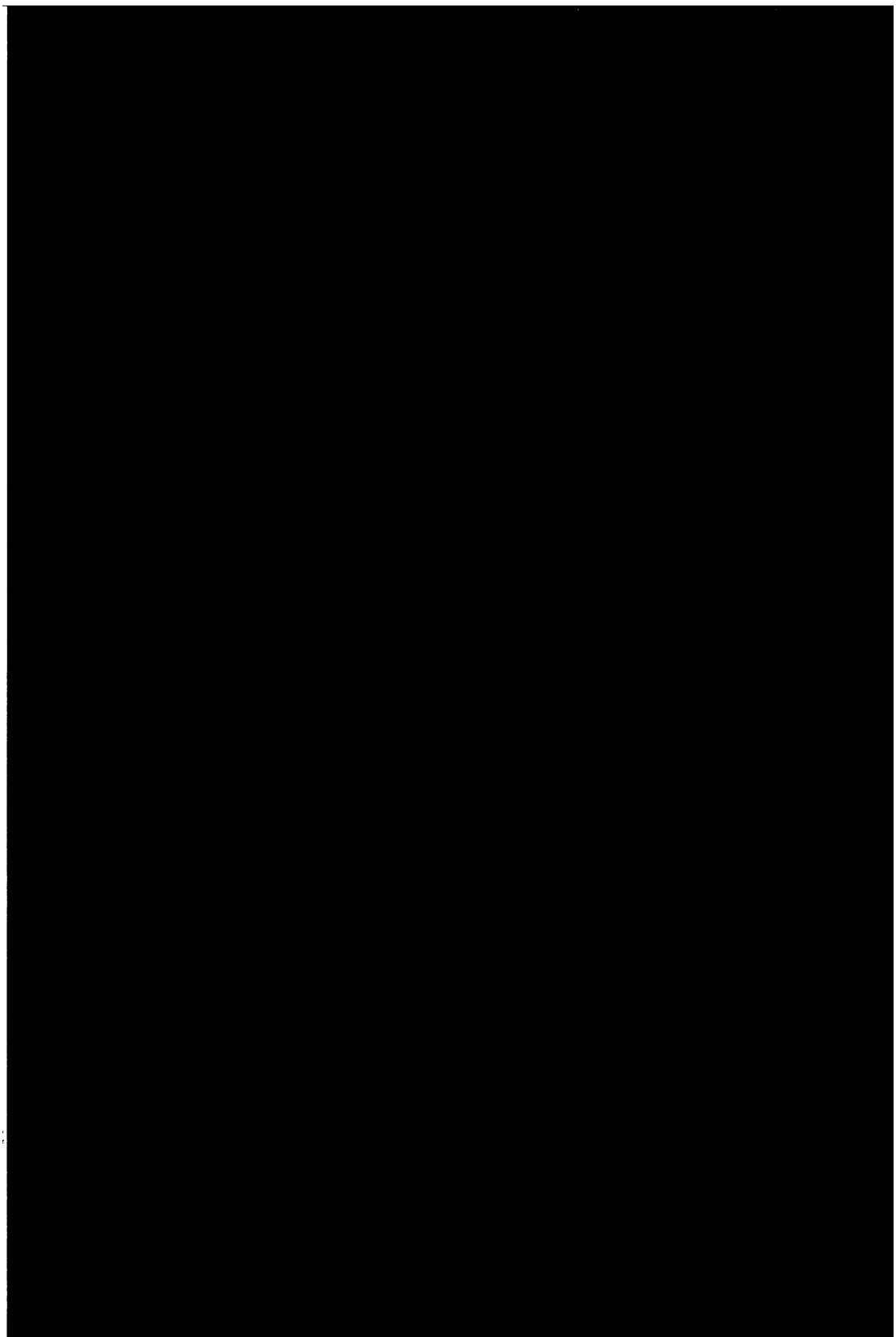


This blank page was inserted to preserve pagination.

INCREMENTAL SIMULATION ON A TIME-SHARED COMPUTER

by

MALCOLM MURRAY JONES

S. B., Massachusetts Institute of Technology
(1957)

S. M., Massachusetts Institute of Technology
(1958)

SUBMITTED IN PARTIAL FULFILLMENT OF THE

REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1967

Signature of Author . . . Malcolm M. Jones . . . .
Sloan School of Management, February 6, 1967

Certified by . . . . . Martin Greenberger . . . . . .
Thesis Supervisor

Accepted by . . . . . John M. Dyer . . . . . .
Chariman, Departmental Committee on Graduate Students

INCREMENTAL SIMULATION ON A TIME-SHARED COMPUTER

by

MALCOLM MURRAY JONES

Submitted to the Sloan School of Management on February 6, 1967, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

ABSTRACT

This thesis describes a system which allows simulation models to be built and tested incrementally. It is called OPS-4 and is specifically designed to operate in the environment of the Multics system. It represents a major expansion and improvement of the OPS-3 system implemented in CTSS and also includes many features adapted from other current simulation systems. The PL/1 language, augmented by many additional statements and new data objects, provides the basis for defining models in OPS-4. A list of desirable features for an incremental simulation system is presented and it is shown how OPS-4 incorporates these features, whereas other current simulation systems satisfy only some of them and are not suitable for use in a time-shared environment.

Some of the particular problems solved by OPS-4 are the implementation and identification of many data bases associated with one procedure, the achievement of apparent simultaneity of execution of many procedures, the use of multiple processes for achieving asynchronous operation of the simulation system, and a combination interpreter and incremental compiler which allows both the data base and model structure to be changed and the model immediately executed without the need for complete recompilation.

OPS-4 includes extensive debugging and tracing features which are particularly adapted to the on-line, interactive environment provided by Multics. OPS-4 also makes extensive use of list structures, so the techniques of memory compacting to reduce unnecessary paging activity are described. Numerous methods for obtaining statistical measures of a model's performance and plotting its dynamic behavior are provided in OPS-4. The use of graphical displays for debugging and dynamically monitoring a model's performance are discussed.

A simplified model of page and segment fault handling in Multics illustrates some of the features OPS-4 provides to allow the user to continuously interact with a model during its construction, testing and running phases. It also illustrates how the user himself may portray portions of a model that are not yet defined.

Thesis Supervisor: Martin Greenberger
Title: Associate Professor of Management

ACKNOWLEDGEMENT

I sincerely appreciate the attention, interest and encouragement given to this research by my advisor, Prof. Martin Greenberger. He was a constant source of ideas and a demanding critic. Many of the concepts presented in this thesis originated during stimulating discussions with him.

I am grateful to Project MAC which provided the facilities and support for this thesis and a stimulating environment for computer oriented research. Numerous individuals at Project MAC, especially R.C. Daley, E.L. Glaser, R.M. Graham, F. Kamijo, J.H. Morris, Jr., D.N. Ness, J.H. Saltzer, and D.B. Wagner, provided information and acted as critics of embryonic ideas.

I thank Profs. F.J. Corbató and M. L. Minsky who, with Prof. Greenberger, served not only as members of my thesis committee, but also as members of a special committee which supervised my interdepartmental program in Computer Sciences. I am grateful to Prof. Corbató for initially bringing me in contact with digital computers during the time he was working with the Whirlwind computer.

I also appreciate the assistance of Miss Janet Cohen and Mrs. Jackie Robinson, who typed numerous drafts of this thesis. Their ability to decipher my hand-written manuscripts far exceeds the abilities of the best currently available pattern recognition programs! Special gratitude goes forth to my wife, Jill, who typed the final draft of this thesis and provided the necessary spiritual support so important during the completion of this thesis.

CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENT iii

1. SIMULATION OLD AND NEW 1

On-Line, Incremental Simulation 1
Events and Activities 3
The Simulation Process 3
Simulation Languages 4
SIMSCRIPT 5
GPSS 6
SOL 7
SIMULA 8
OPS-3 9
A New Time-Sharing Computer 10
A New Simulation Language 12
Implementation of OPS-4 15
Thesis Overview 16

2. SIMULATION IN MULTICS 18

Simulation Essentials 18
Limitations of Multics and PL/1 for Simulation 19
An Overview of Multics 22
The Relation of OPS-4 to the Shell 24
Handling of Quits and Interrupts 26
Modifications to the Shell 27
Variable Calling Sequences 29
Multiple Process in OPS-4 31
The Problem of Reproducibility 32
Controlling Parallelism 33
Special Asynchronous Processes in OPS-4 34
Summary 35

3. CONSTRUCTING A MODEL

A Model in OPS-4 36
Hierarchical Models 37
Three Types of Programs in OPS-4 38
Uncompiled Programs 39
User Portrayed Programs 40

An Example 41
Two Approaches to Modelling 43
Constructing a Model from the Top-Down 44
A Functional Model of Multics 45
Linkage, Segment and Page Faults in Multics 47
Modelling the User 52
Modelling Segment and Page Fault Handling 53
Modelling a Running Process 56
Modelling the Scheduling of Processes 57
The Complete Model 57
Building the Model 59
Testing the Model 60
Completing the Model 62
Lending Realism to the Model 63
The Model as a Designing Tool 64
Constructing a Model from the Bottom-Up 64
Adding Structure to the Model 65
Material Based Versus Machine Based Models 66
Activities and Events 67
Conditional Activities 68
The Information Feature 70
Summary 71

4. THE SIMULATION DATA BASE 72

Global and Local Variables 72
Implementation of PL/1 Storage Allocation in Multics 75
Restructuring the Data Base 76
Sets, Queues, and Tables 76
Contents of the Global Symbol Table 79
Manipulating the Global Symbol Table 79
Manipulating the OPS-4 Data Bases 80
Private Data Bases 81
Hierarchically Structured Data Bases 82
Debugging Considerations 83
Multiple Copies of Local Data Bases 84
Difficulties with the Multics Stack 85
Solution to the Stack Problem 87
Linking Activities 88
Initialization of the Data Base 90
Summary 91

5. CONTROL OF ACTIVITY SEQUENCING 92

The Need for Activity Sequencing Statements 92

The Agenda 94
Modifying the Agenda 95
Scheduling Activities 96
Scheduling Conditionally Executed Activities 98
Rescheduling Activities 100
Cancelling Activities 100
Interrupting and Resuming Activities 101
Implicit Modification of the Agenda 102
Delays and Waits 102
Self-Interruption and Self-Cancellation 103
States of Activities 103
The Agenda Scan 105
The Agenda Structure 106
The Structure of an Activity Definition Block 109
The Structure of the Main Entry 110
The Structure of an Event Entry 111
The Structure of an Interrupt Entry 112
Time Advancement 112
Continuous Models 113
Returning Control to the Agenda 114
Modifying the Agenda Entries 115
Specifying the Parameters of Activities 119
Specifying the Variables in Conditions 121
The Pros and Cons of Alternate Sequencing Schemes 121
Another Type of Conditional Scheduling 124
Priorities 126
Real Time Events 127
Executing Activities Simultaneously 128
Manipulating the Agenda 130
Explicit Versus Implicit Scheduling 130
Summary 133

6. RUNNING AND DEBUGGING A MODEL 134

The Model Development Phase 134
The Use of Interpretation 135
Two Levels of Trace Specifications 136
Monitoring the Flow of Control 137
Monitoring Simulated Time 138
Monitoring Statement Label References 138
Monitoring Activity Calls 139
Monitoring Modifications to the Agenda 140
Monitoring the Agenda 141
Monitoring Statement Execution 142
Monitoring Specific Variables 142

Monitoring Errors and Automatic Definition of Variables 143
Controlling the Specification of Tracing 144
User Defined Traces 145
Controlling the Execution of Individual Programs 146
Running the Entire Simulation 149
Interrupting A Simulation 150
Recapitulation 152
The On-Line Environment 154
Summary 156

7. COLLECTING STATISTICS 157

OPS-4 Versus GPSS, SIMSCRIPT and DYNAMO 157
Time Related Statistics 159
Basic Statistical Measures 161
Collecting and Displaying Distributions 162
An Example 165
Queue Statistics 165
Time Series Plots 166
Summary 168

8. MEMORY MANAGEMENT TECHNIQUES 169

List Processing in Multics 169
Managing the Agenda 170
Reordering the Agenda 171
    Agenda Reordering as an Asynchronous Process 172
    Deciding When to Reorder the Agenda 172
Deleting Activity Definition Blocks 173
The Structure of Sets and Queues 174
Managing Sets and Queues 176
    Automatically Initiating Garbage Collection 177
Summary 180

9. INTERPRETATION AND INCREMENTAL COMPILATION 181

OPS-4 Programs 181
The OPS-4 Symbol Table 183
Inferring Data Attributes 184
The Execution Segment 186
Creating the Execution Segment 187
Detecting the Editing of an OPS-4 Program 193
Incremental Editing of OPS-4 Programs 194
Correcting Execution Errors 195
Detecting Changes in the Symbol Table 197

Checking Trace Specifications 198
Compiling an OPS-4 Program 199
Summary 201

10. GRAPHICAL DISPLAYS IN SIMULATION 203

Economic Considerations 203
A New Display Terminal 204
Advantages of Soft Copy Output 204
Disadvantages of Soft Copy Output 205
Centralized Reproduction Facilities 206
Producing Plots 207
Text Editing 208
Data Editing 211
Dynamic Displays 212
Implementation Techniques 214
Summary 216

11. SUMMARY 218

OPS-4 in Retrospect 218
OPS-4 Versus OPS-3 221
Conclusion 223

APPENDIX 225

BIBLIOGRAPHY 233

BIOGRAPHICAL NOTE 240

FIGURES

Figure 1. Functional Diagram of Multics 46
Figure 2. Functional Diagram of Linkage, Segment and Page Fault Handling in Multics 48
Figure 3. Segment and Page Fault Handling Model 55

TABLES

1. Activity State Transitions .105

Chapter 1

SIMULATION OLD AND NEW

This Chapter discusses the nature of simulation and tells how the environment of a time-shared computer allows a new type of simulation system to be implemented. It shows how the current simulation languages are not appropriate for this environment and then specifies what features should be available in an on-line simulation system.

On-Line, Incremental Simulation

The essence of simulation is imitation, or role-playing. One entity, the device performing the simulation, is made to assume the nature of another entity--the phenomenon being simulated. Simulation differs from direct experimentation because the phenomenon under study is usually not a part of the simulation.

An on-line simulation system allows both the user and the simulation device to cooperate and share the task of performing the simulation. It does this by providing facilities for the user to interact with the simulation device so that they may both play active roles in the simulation process as it is occurring. Thus, the user may perform some of the simulation functions himself and the simulation device perform the remaining ones. Alternately, the user may act only as a monitor and observe, verify and record data or modify and redirect the simulation when it strays erroneously from the desired path.[1,2,3] An on-line simulation system also allows the actual phenomenon being simulated to become a part of the simulation.

On-line simulation is not new. Many people have been simulating on-line with analog computers for years. Simulations which involve physical models are often conducted with the user on-line. Both management gaming and war gaming are limited forms of on-line simulations. However, the on-line, interactive use of a digital computer to build, modify, test and run simulation incrementally is new.

Advances in hardware and soft ware technology have made this possible. The cost of producing electronic components has decreased to the point that a user can now afford to have his own digital computer (e.g., he can buy a PDP-8 S for approximately $10,000). He can observe, and participate in a simulation by manipulating his model directly from the computer console just as he could with an analog computer. With large scale computers the same on-line interaction is also possible. The technique called time-sharing allows one large computer to dynamically reallocate its resources so that users sitting at remote consoles attached to this large central computer feel as if they have a computer of moderate capacity all to themselves.[4, 5] With these advances it is now possible to provide any user who wishes to simulate using a digital computer the same or greater degree of involvement in the simulation process as that obtained by a user accustomed to simulating with an analog computer.

It is with this environment in mind - the interactive mode of using either a small digital computer, or a large time-shared one - that the term on-line simulation is used in this thesis. The term incremental simulation is defined to mean the building, testing, and validating of a model piece by piece. This has always been the recommended method, but difficult to effect in a batch processing environment. On-line simulation systems now make incremental simulation easily realizable.

Events and Activities

The terms, event and activity are used frequently in the literature and throughout this thesis.[6, 7] An event is some action which changes the state of the simulation by modifying the simulation data base, and/or scheduling or cancelling the execution of other events in the system. An important aspect of an event is that it occurs at a specific point in simulated time and is instantaneous. An activity is a sequence of related events which are separated by specified intervals of simulated time. Thus, an activity exists over a period of simulated time.

The Simulation Process

Simulation is sometimes characterized as a three stage process. First, a descriptive model of the phenomenon is constructed. Then the model is tested. Finally, the completed model is exercised and, by inference, conclusions are drawn about the behavior of the real phenomenon being studied.

In reality no sharp line should exist between the first two stages. Ideally, the model building and testing stages are repeated many times and constantly interact with each other. In some instances, there is no formal final running stage, since by the time the model is fully debugged the user has obtained such a clear understanding of the phenomenon under study that it is not necessary to exercise the model any further.

The goal of an incremental simulation system is to completely remove the distinction between the building and testing phases, by allowing the user to interact continuously with his model. He should be able to experiment with his model, either in whole or in part, at any point during its development. He should be able to change any portion of his model at any time and immediately test the effect of these changes. To allow this flexibility an appropriate language in which a user may easily specify his model and a simulation system that allows complete interaction with the model and the ability to easily restructure it must be provided. Neither is sufficient by itself.

Simulation Languages

When designing a simulation language, the environment in which it operates - the simulation system - must be constantly kept in mind. For example, a language designed for an on-line environment should include facilities for allowing the user to directly communicate with a model as it is running. The type of debugging facilities provided in an on-line

simulation system may take advantage of this communication and therefore substantially differ from those provided in an off-line simulation system. The language should reflect this difference of environment. Present simulation languages designed for use in the off-line batch processing environment are unsuitable for on-line simulation. The following discussion outlines the deficiencies for on-line use of four prominent simulation languages.

SIMSCRIPT

SIMSCRIPT is one of the most widely used simulation languages.[8] Without substantial modifications it would be a poor on-line language. Why?

1. Complicated, fixed-field forms are used for data specification and data initialization. They would have to be replaced by more flexible substitutes, such as the English-line declarative statements proposed for SIMSCRIPT II.[9]
2. There is a total lack of debugging facilities in SIMSCRIPT - a handicap even to off-line users.
3. The user cannot easily access and modify the master scheduling system - known as the events list.
4. The user cannot make a small change in the model structure and see the result without recompiling and reloading the program
5. There is no method for specifying the conditional execution of events. Also, it is not possible to schedule events relative to

(e.g. before or after) previously scheduled events.

6. The facilities for collecting and producing statistical measures of a model's performance are very limited.
7. Because of SIMSCRIPT's event, rather than activity, orientation it is often necessary to pass as parameters excessive amounts of local data from one event to another.

On the plus side the set manipulation facilities in SIMSCRIPT are very powerful. Also, the fact that FORTRAN is a subset of SIMSCRIPT is an important asset.

GPSS

The GPSS language is another very popular simulation language.[10] A version of GPSS II modified for on-line use has been available on the M.I.T. time-sharing system (CTSS) since early 1964.[11] Although usage has been limited, more users find this augmented version of the GPSS language quite suitable for on-line use.

1. The debugging facilities are excellent.
2. The events chain, used for scheduling events, may be examined at any time.
3. Modifications to the model structure may be easily made either during a run, or between runs.
4. Any of the savex values and the comprehensive statistics automatically collected for blocks, queues, facilities, storages or tables may be printed at any time.

5. The user may input parameter or savex values from the console during a run.
6. Comments may be printed to identify numeric output, or indicate the flow of control within the model.

GPSS has several serious limitations, however, which made it infeasible for use as a general simulation language.

1. It is a very restricted language. No general algebraic facility is provided and arithmetic is restricted to integer mode. The types of entities in the language are very limited and the number of each type is fixed.
2. It is a closed language - it cannot communicate with subroutines written in other languages (the HELP block is not considered an adequate mechanism since GPSS users must know FAP and the internal layout of the GPSS system to make use of it).
3. It cannot be used for large models since the number of entities, such as facilities, storage, queues, savexes, etc. is limited by the restricted core space available.
4. No mnemonics can be used to label entities. All labels are numeric not symbolic.
5. Data specification and initialization are very restricted.
6. The events chain cannot be modified by the user from the console.

SOL

The SOL language retains most of the good features of GPSS and corrects many of its shortcomings.[12,13] Since it is an extended version of

GPSS embedded in ALGOL it overcomes most of the disadvantages listed above. However, before corrections to a model can be tested in SOL it is necessary to recompile the program, since SOL employs a partially compiled, partially interpretive system to execute simulations. SOL also does not allow the user to inspect the event chains. It does, however, permit conditional delays with its WAIT UNTIL statement. Also, SOL provides a simple statement which allows the user to generate random variates. The chief problem with SOL is that it is not commercially available, so it is difficult to objectively evaluate it.

SIMULA

The SIMULA language is the best simulation language currently available.[14,15] It contains ALGOL as a subset, has a flexible means of scheduling events and activities (called processes), allows the user to define any type and number of entities, the debugging and tracing facilities (which were not in the original version and have just recently been specified) appear to be quite good, and running time is reported to be excellent.[16,17] However, the present version of SIMULA is also not suited for on-line use.

1. Because SIMULA requires compilation of a model before it can be executed, to make changes in the model is a time consuming process - no matter how fast the compiler.
2. SIMULA does not allow a user to conditionally activate processes, a feature which is very powerful.

3. By adopting ALGOL as its base language SIMULA has made the same mistake as COBOL and the original FORTRAN I system - a model in SIMULA consists of one big program, all of which has to be recompiled whenever a change is made in any part of it.
4. At present the only aid SIMULA provides for collecting statistics is an automatic histogram plotting routine.
5. SIMULA has no provision for saving or restoring the status of a model during the simulation.

OPS-3

In contrast to these four languages the simulation system available in OPS-3 was specifically designed for on-line use.[18] (It is the only simulation system to my knowledge, that was designed for on-line use.) The following features of OPS-3 are particularly significant:

1. Specification of data structures and initialization of data in OPS-3 is done dynamically and is easily specified.
2. Complete or partial reinitialization of the model is simple and completely under the user's control.
3. It is easy to modify the model structure at any time and no compilation or reloading of programs is required.
4. The debugging and tracing facilities are comprehensive, flexible and easy to use.
5. The scheduling mechanism - called the Agenda - may be examined and/or modified by the user at any time.

6. A general algebraic language with implicit array operations is available, although the control statements in it are limited.

7. Communication with subroutines written in any language is simple and allows the basic features of OPS-3 to be easily expanded or shaped to the user's tastes.

Like GPSS, OPS-3 is largely an interpretive language, (the MADKOP feature to allow interpretively executed routines to be compiled is incomplete), and consequently running time is very long. The present OPS-3 suffers from a lack of statistics collecting routines; it has a limited variety of data entities; and it seriously limits the amount of core space available for both program and data. Also, the syntax used in some of the language statements is sometimes awkward and inconsistent.*

A New Time-Sharing Computer

The new time-sharing system called Multics which is now being implemented on a GE 645 at Project MAC will relieve many of the limitations of CTSS.[19-25] More than just an upgrading of CTSS, it incorporates a new philosophy of memory addressing. In Multics the logical

---

*Many of these deficiencies could be corrected and the missing features added since OPS-3 is a modularly constructed system. However, the lack of core space available in CTSS prevents adding any new features to the present implementation of the language. The basic OPS-3 system uses 24k of the 32k of core available to a user. The SCHED, DRAW and TAB operators used in simulations require almost another 2k of core. Thus, the user is left with only 6.8k of core space for his own programs and data storage.

sections of a program are specified as segments. A program may consist of up to 2^{18} segments and each segment may contain a maximum of 2^{18} words. To allow this size program in a physical core memory which is limited to a maximum of 16 million words a technique called paging is employed. Each logical segment is subdivided into pages 64 or 1024 words long. The Multics system keeps in core memory only those pages of each of the segments of a user's program that are actively being used at each particular point in the computation. When a new page is needed it will be automatically obtained by Multics and placed in core. When physical core memory is full, Multics will automatically remove pages that have not been recently referenced.

The segmentation scheme also allows users to share programs if they are written as pure procedures. Thus, if several people wished to use the same program simultaneously in Multics, only their data segments need be unique. The instruction segments need be represented only once (all programs are written as pure procedures). In contrast, if 3 people are concurrently using the OPS-3 system in CTSS, 3 separate copies of OPS-3 are maintained by CTSS. Multics also allows user programs to use procedure segments of the supervisor. This will simplify the writing of sophisticated user systems, since many parts of Multics may be used in place of redundant programs written by the user. Also, Multics allows a user to specify parallel processing of procedures, a feature which could be very useful in simulations. Many detailed refer-

ences to the implementation of Multics will be made throughout this thesis. This information is still undergoing revision and clarification and is not at present publicly available.

A New Simulation Language

This thesis defines a language and system for incremental simulation specifically designed to operate in a time-shared environment. It is referred to as OPS-4 throughout this thesis. It borrows concepts from all of the five languages described above and several others in addition.[26-30] The following is a summary of its important features.*

1. A subset of the PL/1 language is the basic language of OPS-4 and provides a general algebraic and data handling facility.[32]
2. The world view of OPS-4 is not narrowly restricted. That is, a user may express his model naturally in OPS-4 whether it is a material or flow oriented model, or a machine or entity oriented model, or a model combining both views.[33, 34] Also, both the activity and event orientations for describing models are available.[6, 7]
3. OPS-4 is specifically designed to encourage a user to build a model incrementally and test the partial model before all the pieces have been completed.

---

*Some of these items may also be found in a list of desirable features for general simulation languages prepared by Teichrow and Lubin.[31]

4. It includes statements to specify the generation of random deviates from the popular distributions.
5. Special data types known as sets, queues and tables are available in OPS-4 in addition to the normal data types of PL/1. There is no limitation on the number or size of any data types.
6. Communication among program elements and variables in a model constructed in OPS-4 is controllable, but not restricted.
7. Restructuring of the data base is simple and requires no recompilation of procedures being run in a debugging mode (the normal mode of execution).
8. It is possible to save the status of a model at any point and at any time by executing a simple statement.
9. Restoring the model to a previously saved state is equally simple. Thus, roll backs to previous points in simulated time are easily effected.
10. It is easy at any point during the simulation to reinitialize partly or completely both the simulation system's and user's data base and reset system time so that the simulation may be restarted or a series of simulation runs may be easily executed.
11. The user has flexible controls to specify the exact order in which events are executed during simulation.

12. No important part of the simulation system is hidden from the user. He has direct access to and the ability to modify every element of the simulation from his console.
13. Extensive and easily used debugging and tracing features are available.
14. It is easy to modify the logical structure of a model and quickly try the new modifications, without recompilation.
15. Flexible means for specifying the starting and stopping points or duration of the simulation run are provided.
16. It is possible to independently test individual components of the model, even if they are embedded in larger modules.
17. The user can interrupt a model at any point during the execution phase and redirect its path, examine and change the values of variables and then immediately continue the simulation from the point of interruption.
18. In situations where unusual circumstances arise, such as the backwards movement of time, the user is given the benefit of doubt and the simulation is not interrupted. However, a flag is set which may be interrogated by the program.
19. Facilities for collecting statistics are comprehensive, optional and easy to specify.
20. Usually, only the structure and mode of initial data inputs in procedures need be declared by the user. The structure

and mode of most data objects resulting from a computation is inferred by the rules of the computation.

21. Immediate on-line diagnostic explanations in graduated detail and verbosity are available to the user when an error is detected during the running of the model.

22. It is possible to run debugged portions of a model at full speed since they may be compiled programs. Interpretive techniques are used when necessary only for sections of programs not yet checked out.

Implementation of OPS-4

This thesis limits itself to a discussion of the design and proposed implementation of OPS-4 for Multics.* The implementation will be in two phases. First, the conceptual framework of the OPS-3 system will be used as a base and will provide the required general algebraic language. Over a year's experience with OPS-3 has shown it to be well adapted for on-line use. However, OPS-4 will use the syntax of PL/1 throughout and will be substantially restructured and enlarged so that the deficiencies of OPS-3 as implemented in CTSS will be eliminated.

The second phase will consist of the implementation of an on-line conversational version of a subset of PL/1. This is an ambitious goal

*It is expected that implementation will begin in the Fall of 1967.

and perforce must await more experience with developing compilers for PL/1. All programming of OPS-4 itself will be done in a restricted version of PL/1 being used at Project MAC for all systems programming. Experience using PL/1 to implement Multics has shown it to be a language well suited to the task of programming systems which require complex data structures.

Since OPS-4 will be programmed in PL/1 it should be possible to transfer it to other machines, in particular the IBM System/360 model 67, without a major recoding effort. Only the machine dependent modules and those Multics modules which are used directly would have to be changed.

OPS-4 could also be implemented in a variety of other environments, such as small stand-alone computer systems, or as an adjunct to other conversational computer languages. However, the design might differ significantly from the one that is presented in this thesis.

Thesis Overview

This Chapter has outlined the characteristics of OPS-4 and described the environment in which it will be implemented. Chapter 2 discusses the basic methodology of implementing OPS-4 in Multics. Chapter 3 describes, with the aid of an example of segment and page fault handling in Multics, the options a user has in building a model and discusses items 1-4 listed above. Chapter 4 discusses the structure of the simulation data base, covering items 5 through 9.

Chapter 5 discusses event sequencing, and how the user may control the sequencing. It therefore discusses items 11 and 12. Item 12 also pertains to Chapter 6 which describes the methods available to an on-line user to control the running and debugging of a model. Since giving the user this ability to exercise direct control over the running of a model is one of the main benefits of an on-line versus an off-line simulation system, items 12 through 18 are included in this discussion in Chapter 6. Chapter 7 covers the topic of collecting statistics and thus discusses item 19. Chapter 8 describes the use of list processing techniques in a simulation language. The last items, numbers 20-22, are treated in Chapter 9 which discusses implementation techniques. Chapter 10 is a presentation of the use of graphical display devices as an on-line communication media. Finally, Chapter 11 concludes this thesis with a summary of all the important issues of incremental simulation in Multics, and lists possible implementation difficulties.

Chapter 2

SIMULATION IN MULTICS

This Chapter lists the basic elements of a simulation system and describes how OPS-4 is embedded in Multics. The two chief issues discussed are the decision of where the responsibility rests for executing OPS-4 statements and the use of multicple processes by the OPS-4 simulation system.

Simulation Essentials

Chapter 1 listed many features that are desirable for an incremental simulation system which operates in a time-shared environment. Some of these features are absolutely necessary, and are found in all simulation systems. Others are conveniences that are peculiar to the on-line environment. It is essential that every simulation system contain the following elements:

1. A mechanism for describing and manipulating the simulation data base. The data base may be partitioned in many ways, but the simplest division is between global data, which is accessible to all activities, and local data, which is accessible to only one activity. Many activities may be represented by the same procedure and differ only in their data base characteristics.
2. An activity sequencing mechanism. Simulation activities are different from normal subroutines since they conceptually operate in parallel. The standard subroutine call is not

flexible enough to allow for the unpredictable flow of control between activities. Also, the transfer of control from one activity to another is a "sideways" transfer, not the standard hierarchical up or down transfer of control.

3. Special facilities for running and debugging a model. Many simulation programs have no fixed termination point. They can be run for variable periods of time. Special features are necessary for specifying the starting and stopping conditions of a simulation model. Also, because of the unpredictable order of activity sequencing, special debugging facilities are necessary for simulation systems.

4. Facilities for collecting statistical measures of a model's performance. Simulation models are constructed so that the user may learn something about a particular phenomenon. Many times statistical measures are helpful to assess the functioning of the model in different environments.

Chapters 4, 5, 6, and 7 discuss each of these simulation features in detail, describing their specific implementations in the OPS-4 system.

Limitations of Multics and PL/1 for Simulation

Multics is faced with many of the same requirements as a simulation system. It must successfully direct the execution of many processes, some dependent upon each other's actions, other independent of the other activities going on. Many of these processes may be operated in parallel

with each other. Also, Multics must maintain a global, or system wide, data base which records the location, status and description of every process and data segment, both active and inactive within the system. Multics must also continuously monitor the state of the system so that users may be properly billed for the resources they utilize. One is naturally led to ask, "Can't the facilities available in Multics be used directly to provide the basis of a simulation system?" Unfortunately the answer is no, for the following reasons:

1. Multics' scheduling system was not designed to be flexible enough for a general simulation language.
   A. Multics relies only on a computed priority for organizing its ready list; i.e. it is not possible for the scheduling process to use any process' attributes, such as scheduled execution time, of processes already scheduled on the ready list.
   B. The conditional scheduling mechanism is very limited. The Block-Wakeup system requires that Wakeup specifically know who to wake up. This is fine for the 'Wait for Event' type conditional of Pl/1 but inadequate for the general 'Wait until A=B or C>D' type conditional allowed in OPS-4.
2. Multics' scheduling system maintains and manipulates far more information than is necessary just for scheduling simulation

activities. Specifically, the majority of the information maintained by the traffic controller in the Active Process Table is needed only for Multics and is extraneous for a general simulation system.

3. Multics does not automatically implement a standard lock mechanism which guarantees that user processes which share a common data base do not get in each other's way. It is left to the user to implement whatever mechanisms are necessary for maintaining and protecting shared data bases. The standard lock mechanism used by Multics is available but each user must implement it himself as necessary.

A fourth point is the present lack of specification of the interprocess control and communication facilities in Multics. (The initial version of Multics will not allow the user to have more than one working process, although Multics itself will use multiple processes.)

A casual reading of the PL/1 manual might lead one to conclude that the PL/1 language is suitable as a simulation language.[32] However, study will show that it is not complete enough for simulation. The features for directing tasking are limited and, at the present time, not clear on such crucial matters as the sharing or independence of data bases by dependent tasks. Also, the data types in PL/1 are not as extensive as those needed in a simulation language (recall item 5

in Chapter 1). Furthermore, the sequencing methods available in PL/1 are not flexible enough. This does not mean that an incremental simulation system must be built from scratch. Indeed, as already stated in Chapter 1, Multics and PL/1 provide a strong foundation and are the basis of the OPS-4 system.

An Overview of Multics

One of the central design features of Multics is that a user may have several processes working for him simultaneously.* Indeed, after a user has completed the Login procedure he has 3 processes automatically established for him. These are:

1. The overseer process which handles certain resource management tasks, such as Login and Logout which must be done reliably.
2. The device management process which is directly in charge of the particular console device he is using, and delivers input strings to the third process.

---

*A process is basically a program in execution. The tangible evidence of a process is a processor stateword (a set of machine conditions) and an associated two dimensional address space (a core image.) The address space of a process, defined by a Descriptor Segment, determines the region of accessibility of the processor, both in execution of instructions and in obtaining data. A dynamic linking mechanism allows the process to change the contents and extent of its own address space, but this does not alter the fundamental view of a process as the execution of a program contained in the address space.

3. A working process which may be executing either a specific Multics command, a user program, or, if the user is at command level a procedure called the Shell.

The specific working process is dependent upon the environment that is automatically provided by Multics. This environment is determined by administrative authority and by the user himself. For example, a user may indicate, by appropriate Multics commands, that he wishes to have special versions of certain supervisor modules: e.g. a different scheduling mechanism, which always puts him at the top of the ready list, or a different typewriter management module which writes a verbatum copy of all typewriter input and output as file on secondary storage, or perhaps a private version of the PL/1 compiler which recognizes French rather than English keywords. If he has been allowed such freedom by the system administration this flexibility is possible. Conversely, the administrative authorities might decree that only certain modules are to be used and only certain commands may be accessed by this user, thus restricting him to a subset of Multics' available facilities.* This feature could be used to restrict the access of a student to specific programs, or to allow the Shell to be replaced by the OPS command. Thus, it would be possible to have a user automatically enter OPS-4 as soon as he completed the Login procedure.

---

*This is currently done in a limited fashion in CTSS by an access vector which allows various system programmers to execute privileged commands not available to normal users.

The Relation of OPS-4 to the Shell

An important issue to be decided is whether or not OPS-4 should be a single command in Multics and operate as a subsystem of Multics much as OPS-3 now does in CTSS, or whether each statement in OPS-4 such as,

Set A = 3

should be a separate command and the OPS-4 system viewed as an extension of the Multics command language. At the present time, it appears that both possibilities are available and that the distinction between the two may be academic.*

Implementing OPS-4 as a separate command in Multics is perhaps the safest route to follow. OPS-4 would then be able to exercise complete control over whatever it was asked to do. It could adopt unusual conventions for punctuations, provide echoing of input lines, and allow standard commands to be abbreviated (as does the '.' command in CTSS), etc. It would also be held accountable for all data bases that it created during its execution so that when a user left OPS the segment housekeeping module would be able to discard all segments that were not specifically declared to be saved. The standard OPS-4 procedures could also be bound together in one executable segment.

---

*This statement and the following discussion are purposely vague since many of the details of Multics which might influence a decision one way or another are not yet specified.

If OPS-4 were implemented as a separate series of commands in Multics the commands would be interpeted by the Shell procedure. Each command would be a separate executable segment so that both the segment and entry names need not be specified. Special provisions would have to be made in each command, to have them reference a common data segment. By providing a different Shell than the standard one the user could have the same freedom to use command name abbreviation or synonyms, invoke different parameter conventions, etc. proposed in the first alternative above. However, one difficulty which might arise would be to decide at what point to delete extraneous data bases created by the individual commands. A special command could be provided for this purpose, but the system could not count on the user executing it. If he did forget, the user might end up with several strange files appearing in his directory. Of course, this might be used to advantage by programmers knowing tricky things to do with these files. However, if providing access to these special data bases were really important to the sophisticated user, the same facility should be provided in the first alternative proposed above.

At the time of this writing, it appears that both of these alternatives could be programmed so that if they were both available a user would not be able to distinguish which version he was using. Indeed, since Multics allows a user to substitute any procedure for the Shell these two alternatives are really equivalent. The only possible minor difference between

one command and several commands might be in the flexibility and method provided to handle interrupts and quits initiated by the user from his console. In CTSS there is a considerable difference.

Handling of Quits and Interrrupts

For example, consider the following situation in CTSS.[4] Command X is sensitive to receiving interrupts - i.e. a special designated procedure is invoked by X when it receives an interrupt signal. If command X is executed directly from the console as a command any interrupts are received directly by X and cause the desired action. In addition, a quit signal will suspend the execution of X and return the user to command level. Suppose X is now a program available within the OPS-3 command. Assume both OPS-3 and program X wish to be able to receive interrupt signals. If an interrupt is received while X is running X acts on it directly in identical fashion as if X were a command. Likewise, a quit signal suspends the execution of X. It also, however, suspends the execution of OPS-3 as well. As long as X is in control, and without making special arrangements with X to forward interrupts, it is impossible to suspend the execution of X and return to a specified spot within OPS-3. That is, there is not a way for the user in CTSS to push or pop the interrupt stack from the console. It can only be done by programs called SAVBRK and SETBRK. This is a serious limitation on the use of the present OPS-3 system for executing programs which make use of the interrupt facility themselves.

In Multics, instead of the use of physical buttons and special signals to the supervisor to indicate interrupt and quit signals the Overseer process receives the quit signals from the user. Although this has not yet been specified, it is reasonable to assume that the interrupt signals will also be handled by the overseer. Since Multics allows the user to execute any procedure directly from the console, just by giving its name, it would be possible to terminate the execution of the current program and initiate the execution of any program, something that is impossible in CTSS. Thus a distinct interrupt signal may not be necessary in Multics. Instead, an interrupt procedure which saves the status of the interrupted procedure and calls a specified procedure might be provided.

Modifications to the Shell

The key to implementing OPS-4 as a series of separate, but coordinated commands is to provide a new Shell process. This new Shell would include all the facilities of the present Shell and provide the following additional features.

1. It would maintain the names of the segments which constitute the current data base and their associated symbol tables and provide these names as parameters in the command's calling sequence to all commands which wished to access the data base.
2. It would also maintain the segment name of the master simulation scheduling system known as the Agenda. This name would have to be passed to all commands which manipulate the

Agenda. (Alternately, the Shell might establish a small data base with a fixed name, such as Shell-data, which would contain the names of the current global data bases, symbol tables, Agenda and any other variable that commands wished to access. Any commands could then reference this segment to determine the working names of the other segments.)

3. The Shell's mechanism for calling commands would have to be expanded. Experience with the OPS-3 system has shown that there are two basic types of calling sequences - those whose number and type of parameters are fixed and those both whose number and type of parameters are subject to change. LISP also makes such a distinction, calling the former group of programs EXPR and the latter group FEXPR - denoting expressions and flexible expression respectively.[35]

The present Shell mechanism allows only fixed calling sequences since it is oriented toward the PL/1 restriction of a fixed number and type of parameters. However, there is a simple mechanism which is an adaptation of the GETP and the READSQ mechanism provided in OPS-3 which does not violate the PL/1 restriction of only fixed length and type of calling sequence.[18] It requires that the Shell recognize, by examining either the linkage segment, the symbol table segment, or the procedure segment of the command, that the command it is about to call has either as fixed (also referred to as standard) calling sequence, or a variable calling sequence.

If the calling sequence is fixed, it proceeds as the normal Shell would and does any necessary parameter mode conversions based on the parameter declarations in the symbol table for the command. However, if the calling sequence is specified as variable the Shell delivers the entire parameter string, exactly as it was typed, except for the leading command name and first succeeding blank, but including the standard terminating character 'NL' in place of either the terminal ';' or 'NL' character, to the command as one parameter of type varying character string. Thus, as far as PL/1 restrictions are concerned, the command has only one parameter, a variable length character string. It is then up to the command to process this character string any way it chooses and discover with what parameters it actually was called.

Variable Calling Sequences

For example the command,

Set A = B + C + (D - E)/F

would naturally call a standard algebraic parse routine such as is available in Multics debugging package. Alternately, the Schedule command which has such options as

Schedule X after Y

Schedule X at Sys-time + 36

Schedule X when A = 16 or C>3.5

might call a procedure very similar to GETP in OPS-3.*

---

*see bottom of page 30.

This procedure would have a number of entry points. The first call to this procedure from within the command would deliver the parameter string, a list of break characters to be used to parse the parameter string, and initialize the scan pointer. Subsequent calls to this special procedure might specify the mode in which the parameters are to be delivered to the command and the name or names of the dummy variables which are to receive the parameter values. Additional calls might back-up the scan, re-initialize it, invoke a new set of break characters, or ask for the return of the remainder of the unscanned parameter string so that it could be processed by a different routine, such as the standard algebraic parser.

It is also possible to conceive of more complicated scanning routines, such as those employed in CL II or COMIT which allow parameter selection and conversion to be based on the relation of a parameter with specified neighboring parameters.[36-38] However, experience with OPS-3 has shown that the sequential scan method outlined above is quite powerful and adequate for most calling sequences.

For example, consider the Shedule examples just presented. X and Y are names of activities, Sys-time + 36 is a numeric expression, A = 16

---

*GETP is a special routine used to parse the parameter string of a procedure subject to instructions delivered to GETP by the procedure itself. It is described on pages 96 to 99 of the OPS-3 Manual.[18]

or C>3.5 is a conditional expression, and after, at and when are keywords in the PL/1 sense. The meaning of these three Schedule statements is,

1. Schedule the activity named X immediately after the entry for the activity named Y already on the Agenda.
2. Schedule the activity named X to occur at the value of simulated time equal to the current value of simulated time plus 36.
3. Schedule the activity named X to occur when the Boolean expression A=16 or C>3.5 is true.

The first example has three parameters, each a literal of type character string. The second also has 3 parameters, the first 2 are literals of type character string and the last is of type float. The third example has 3 logical parameters each of type character string but the last is actually a string of 7 physical parameters of mixed type. The parameter conversion method just outlined allows the Schedule routine to retrieve its first two parameters as literal character strings, and then test the second parameter to determine in what mode to request the last parameter, or parameters.

Multiple Processes in OPS-4

Another important design decision relates to the freedom the user is given to specify asynchronous or synchronous processes in OPS-4. One of the major concerns of all current simulation systems is how to imitate simultaneity of events on a single processor computer. It would

appear that Multics offers a solution to this problem. It does, but not a complete one. The traffic controller maps the actual hardware of the GE 645 (which is limited to 8 processors) into an indefinitely large number of pseudo-processors each capable of running one process at a time. Conceptually, the user may regard the pseudo-processors as operating in parallel with each other. In actuality, the amount of simultaneity is limited by the number of physical processors being used. Therefore, it will always be possible for a user's model to create more simultaneous events than there are actual processors available to execute the events in parallel. Thus, sequencing rules will still be as important as they are in current simulation languages.

The Problem of Reproducibility

For example, consider a queing model having several servers and separate queues for each server. The server processes and the arrival process are all conceptually occurring simultaneously. When an arrival occurs, it enters the queue which is the shortest. Hypothesize that there are two queues both of the same length and shorter then the queues for any of the other servers. Assume that these two servers finish serving both their requests simultaneously and that at the exact same instant a new arrival occurs. In which queue will the arrival be placed? The answer can certainly depend on the order in which the two server process and one arrival process actually are executed on the computer.

This element of irreproducibility offers both new problems and new possibilities. During the debugging stage it is a serious handicap. Reproducibility is essential if bugs are to be easily recreated so that they may eventually be eliminated. Conversely, once a model has been substantially debugged the opportunity of actually observing the results of simultaneously interacting processes may add insight to the understanding of the model - especially if the element of non-reproducibility results in different model performance. It is analagous to reseeding the random number generator, and seeing a different sequence of activities.

Controlling Parallelism

This suggests that the user should be able to control whether simulation activities are executed sequentially or whether some of the activities are actually executed simultaneously. To provide this control OPS-4 will add a new attribute to activity declarations. The user may explicitly declare each activity (e.g. PL/1 procedure) to be either of type Sequential or Simultaneous. The default type will be Sequential. In addition, a global declaration of Sequential or Simultaneous may be invoked to cover all activities. However, the local declaration in each procedure will always take precedence over the global declaration. Thus, certain activities may be executed sequentially or simultaneously with other activities, independent of whether all the other activities are being executed sequentially or simultaneously. The exact mechanism for

effecting simultaneous execution of activities is discussed in Chapter 5 which describes the activity sequencing mechanism. If the user declared activities to be Simultaneous, he will have to program locks on the appropriate data bases, since OPS-4 will not do so in its initial implementation.

Special Asynchronous Processes in OPS-4

OPS-4 will make use of the multi-processing capabilities of Multics when they are available for some peripheral processes which do not affect the execution of the central simulation. These are;

1. User communication and asynchronous interaction with the simulation to allow game playing and the like.[39]
2. Asynchronous debugging monitors which allow the simulation to continue as trace results are simultaneously collected.
3. Statistics collection and processing.
4. Memory compacting (or garbage collection) of list structures that have diffused throughout many pages of memory.
5. Asynchronous probes of the simulation data base with carefully designed inputs.

Each of these processes is essentially independent of the main simulation process. However, appropriate interlocks will have to be programmed to insure correct execution of all processes.

Items 1, 2 and 5 are discussed in Chapter 6 which discusses running and debugging a model and Chapter 10 which describes the use of graphical

displays in simulation. Item 3 is described in Chapter 7 which describes methods of statistics collection. Item 4 is discussed in Chapter 8 which covers list processing techniques.

Summary

What has been discussed in this Chapter can be summarized briefly:

1. OPS-4 can be implemented in Multics as a single command having many subcommands or, by modification to the Shell process, as a series of separate but coordinated commands. However, since the Shell can be replaced by any user procedure these two alternatives are equivalent.
2. The simulation system itself in OPS-4 will generally involve only one process in Multics, although the user may specify more if he wishes. However, multiple processes will be used to accomplish several important functions which are conceptually independent of the main simulation process. In the initial version of OPS-4, however, there will be only one working process because of this restriction in the initial version of Multics.

Chapter 3

CONSTRUCTING A SIMULATION MODEL

The interactive features of OPS-4 described in this chapter allow a user to start building a model on the computer at a very early stage. The computer may then be used to help clarify and expand the formulation of the model from the very outset. The user is encouraged to build his model modularly so that it may easily be expanded in simple incremental steps. He may start at the inside and work out exploring the interactions of specific detailed functions, or else he may specify the entire structure in gross fashion and add detail as his understanding of the problem grows. Facilities are provided in OPS-4 to allow unstructured problems to be described in three levels of specificity. A simple model of segment and page fault handling in Multics is used to illustrate these features.

The ability to perform data gathering and data analysis, side by side with model formulation, testing and validation allows the user to easily explore the relations between his data and model structure.[2, 3, 40, 41] This may lead to a healthy cross-fertilization of ideas. It is one of the principal reasons why it is important to have a comprehensive general algebraic language included as part of a general simulation language.

A Model in OPS-4

The overall structure of an OPS-4 simulation model is rather simple, but quite different than the structure of a GPSS, SOL or SIMULA program.[10, 12-15] It is more akin to SIMSCRIPT, as it is organized around the concept of independent, separately compiled activities, which are written as external procedures.[8] Each activity has its own local data base, and may share data with other, but not all other activities. In addition, there is a global data base which is available to all activities. Individual activities may be hierarchically structured using the features available to PL/1. In addition, groups of activities may form a hierarchical structure. The concept of block structure and scope of

variables is extended to cover such structures of activities. (This is discussed in Chapter 4.)

Hierarchical Models

When studying a complex problem it is often helpful to subdivide the problem into parts. Each part may be further subdivided, and their parts subdivided again, ad infinitum, until a level of detail is reached that can be easily described and analyzed in simple terms. This the technique that humans appear to use in solving difficult problems and has been mimicked with fair success by various computer programs attempting to demonstrate intelligent behavior.[42, 43] In fact, most complex computer programs are written in this hierarchical manner.[44]

An interesting question is, "What is the route taken to write these hierarchically structured programs?" Is the whole hierarchical structure specified a priori and programmed starting at the detailed level, or does the structure grow in detail in parallel with the programming of increasingly detailed blocks? Historically, complicated programs such as the initial versions of the CTSS and Multics supervisors have been programmed using the former approach.[45] However, these systems have continued to evolve gradually over time by adding hierarchical detail. It often turns out that many of the most interesting and important problems are at the higher levels of the program structure and must await testing until the majority of the basic programs have been written.

When errors do arise at this level they may be very costly to correct, for they may require substantial restructuring of subcomponents of the system.

Simulation is often heralded as a solution to this problem. The suggestion is made that a simulation model can abstract the important high level interactions and focus on them, ignoring the detailed problems at lower levels. When the overall structure is completed then the details can be added. To do this successfully requires that a model be constructed hierarchically. However, to exercise such models in conventional simulation systems requires that all the key pieces be specified and assembled before a run can be made. OPS-4 offers a new approach to this problem.

Three Types of Programs in OPS-4

The OPS-4 system provides the user with three different modes of flexibility for specifying his model.

1. He may write an OPS-4 program, compile it, and then execute it.
2. He may write an OPS-4 program and execute it directly with no intermediate compilation phase.
3. He need not write a program at all, but may execute any previously written system or user procedures, of either type 1 or 2, directly from the console.

Were it not for the idiosyncrasies of computers, which make them

unable to converse directly in languages natural to the user, the first mode above would not need to exist. The second is considerably more natural for a user than the first. Both require a program - e.g. a specific sequence of actions - be formally stated and therefore they are fixed specifications. The third mode does not have this restriction. The user is not required to plan a sequence of actions in advance. He can improvise.

To distinguish between these three modes of specifications the following terms are used throughout the thesis. The first is referred to as a compiled program. The second is called an uncompiled OPS-4 program. The third is known as a user portrayed program. All three are allowed to be intermixed in a simulation constructed in the OPS-4 system.

Uncompiled Programs

Most programmers have grown to accept the necessity of compiling a program before it can be executed. However, compilation is not a natural function included in specifying a model, and a user should not be constantly forced to think about it after every iteration of change to his model. Compilation is related to the efficiency of processor utilization. Therefore, the user should view compilation merely as a means of more effectively using a scarce resource, not as a function necessary to allow execution of a program. Even at this level, however, the user must weigh the benefits of decreased execution speed of a compiled pro-

gram, versus the time taken to compile the program. During the debugging and program testing phase it is not unusual for the compilation time to exceed by two or three orders of magnitude the program execution time. Thus, methods for executing uncompiled programs that are 10-20, or even 100 times slower than the execution of compiled programs meet a very definite need.* Of course, an alternate approach is to attempt to shorten the compilation time. But this usually results in an unfortunate increase in execution time, because of the cruder programs produced by the hasty compilation. (This is the approach taken in the design of the MAD compiler.)[47]

User Portrayed Programs

One of the attributes of a time-sharing system that has been often praised is the feature that a program may communicate with the user and ask for help. It has also been stated that a time-sharing system relieves the programmer of the need to write programs for contingencies that may rarely occur. This is true. In OPS-4, the user portrayed program is used to provide both these options.

During the course of the construction of a model, any modules which the user realizes must be included in the model for completeness

---

*Experiments performed by J.H. Morris, Jr. with the OPS-3 system showed that interpretive execution of OPS-3 programs ranged from 25-100 times slower than execution of the same program after it was translated to a machine program using the MAD-KOP translator. The OPS-3 system is quite inefficient, and the methods outlined in Chapter 9 for executing OPS-4 programs should lower this figure considerably.[46]

and accuracy, but which he is not interested in describing, or modules which he does not know how to completely describe at that particular time, may be defined as user portrayed programs. This allows the model description to be logically complete, but does not force the user to switch from his main area of interest to consider something of lesser importance. He is only required to be specific about the functions of the module when it is actually needed. At that time the environment of the situation is established and it may be helpful in suggesting what is the proper formulation of the program.

An alternate use of the user portrayed program is to allow the user, or anyone else, to participate directly in the simulation as it is running. This feature, together with the ability in Multics to direct output to, or receive input from several terminals allows OPS-4 to be conveniently used to specify interactive gaming models.

An Example

Suppose we wished to study the operation of the segmentation and paging mechanism in Multics. Recall that segmentation is a technique implemented by software and hardware which divides each program into pieces called segments and required that only those segments referred to within the program in the most recent interval of time actually reside in core memory.[19-21] Furthermore, segments are divided into pages. Pages are of fixed size and the hardware addressing allows them to reside anywhere in memory. Thus, a segment need not occupy consecutive

blocks of core memory although to the user it may appear as if it does. (The user need not know anything about pages.) A segmentation and paging system seek to increase the utilization of memory, which is a scarce resource, so that more programs may share the memory simultaneously. It also allows a computer to accommodate programs that are larger than physical core memory.

Some of the questions about the segmentation and paging system which we might be interesting in answering are:

1. How many programs can we execute (from start to finish) in a given period of time with segmentation and paging as opposed to sans segmentation and paging?
2. How long do various classes of programs take to execute in a system that uses segmentation and paging compared to one that doesn't?
3. What is the total amount of system overhead in a system that uses segmentation and paging compared to one that doesn't?
4. How does changing the page size affect the answers to these three questions?*

To answer these questions we start to build a model. In fact,

---

*Obviously a page size larger than the largest segment in the system, or limited to the physical memory size - whichever is smaller - is like no paging at all.

since we are asking for comparisons, we must really consider two models - one that uses the swapping technique of CTSS, and the other that employs the segmentation and paging concepts of Multics. The CTSS system has already been analyzed by Scherr so we need not do it over again.[48] In addition, he has also collected the necessary data on programs size, the lengths of running time, the duration of pauses between successive requests for program execution, etc.

We realize that there is a strong positive correlation between the data Scherr collected and the CTSS system. In Multics the distribution of program lengths may be quite different because the number of active segments and pages changes dynamically and also, because of the possibility of sharing segments. However, in a paging system there is no limit on memory size as there is in CTSS, so the total size of all segments in a program may tend to be larger in Multics than those in CTSS. The user interaction rate is known to be positively correlated to the program completion rate. If the system completes programs rapidly, the user tends to submit more programs to be run. Keeping these facts in mind we decide that although we are interested in a comparative study we may have to establish multiple reference points even though they may not be realistic in both systems being studied.

Two Approaches to Modelling

How do we start to build a model in OPS-4? We can start either at the detailed level of the individual modules and build upward until we

have constructed all the modules that are necessary to completely specify the model, or we can start with a very abstract, simplified model and add detail as we find it necessary. The first approach might be likened to that of building a computer. The individual circuits are specified, they are combined into logical building blocks, the building blocks are combined into functional units, and the functional units are combined to complete the whole computer. The second approach might be likened to building a house. The outer structure is completed first and then the detail is added. Both approaches have merit and both approaches can be used in OPS-4.

Constructing a Model from the Top-Down

This second approach to building the model of segmentation and paging in Multics might lead us to conceptually think of Multics and the user in terms of a simple queing model. Users make requests, which enter a queue, and Multics examines the queue periodically and serves the requests. Although this is a correct description it is not very enlightening and certainly doesn't provide any information to help us answer questions about segmentation and paging in Multics. More detail must be added. Modelling the user as a simple arrival process with associated attributes of running time, program size, etc., may be adequate initially. However, the server side of the model needs to be more detailed.

A Functional Model of Multics

We might model Multics in more detail by specifying the major modules of which it is constructed. For example, the hard core supervisor logically consists of just 3 functions,

a) Memory management - the allocation of memory among competing processes, which is the job of the basic file system.

b) Processor management - the allocation of processors among processes and inter-process control and communication, which is handled by the traffic controller.

c) Secondary storage management - the details of addressing and manipulating segments stored on the drum, disk,tapes, etc. which is handled by the basic file system.

Communication with users at remote terminals is handled by subroutine calls to the I/O system which is not a basic part of the hard core supervisor, but is closely interrelated with it.

Figure 1 shows an overall functional diagram of Multics. Basically, there are three levels. The lowest level is concerned with hardware management. It transforms the actual hardware of the machine into a number of pseudo-processors, each with its own segmented memory and symbolically accessed files.[25] This level provides what has been called an extended machine.[49, 50] The next level up is concerned with resource management. Here the extended machine is allocated among

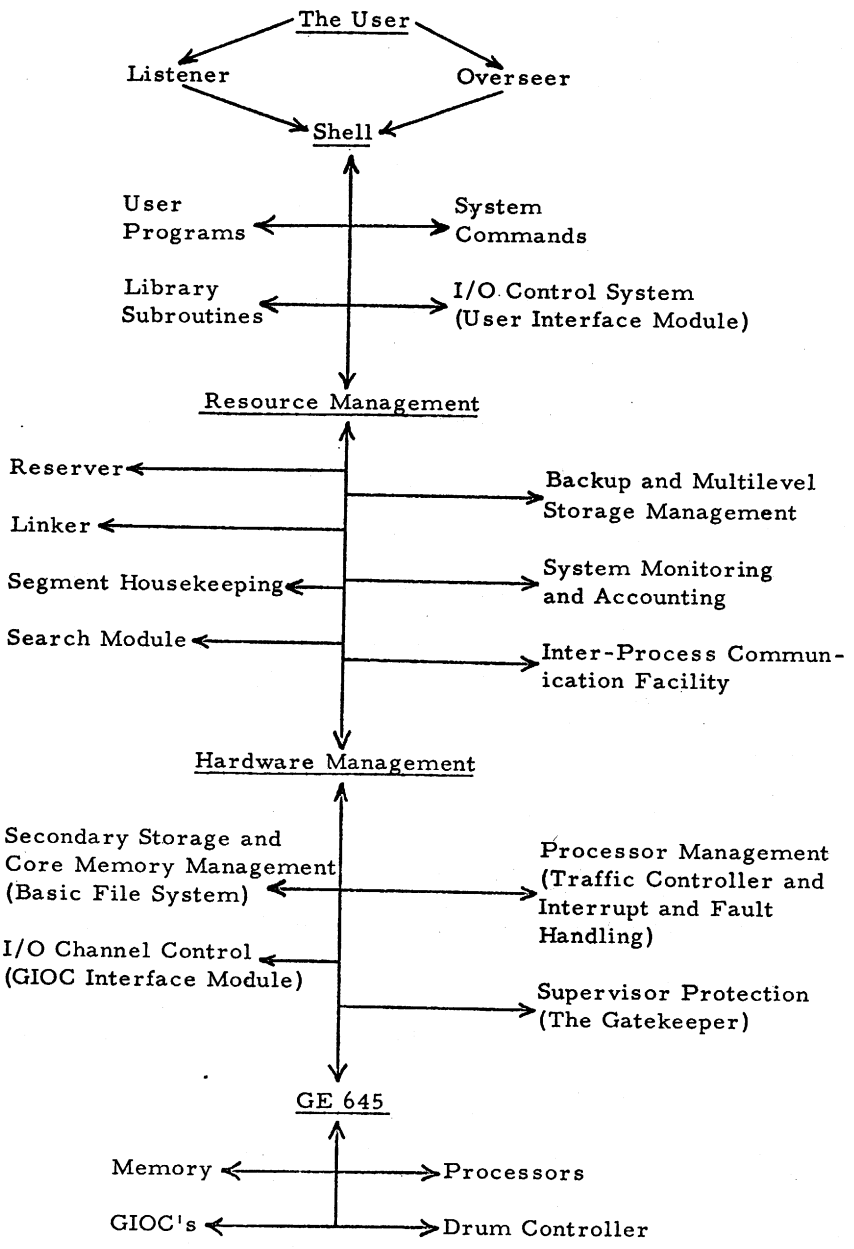


Figure 1. Functional Diagram of Multics

the various users and numerous administrative services and functions are performed. Finally, at the top level is the user, and the programs he may directly execute.

It is apparent that the operation of the basic file system is what we are primarily interested in modelling. However, we cannot completely neglect the other functions, since the basic file system calls the I/O system when it needs a page transported to or from secondary storage and the basic file system may be called by the traffic controller when it is necessary to switch processes.

Figure 2 shows a diagram of the linkage, segment and fault handling mechanism in Multics. The solid lines represent flow of control, with arrows designating direction. The circles indicate data bases and the dashed lines show what modules access the data bases.

Linkage, Segment and Page Faults in Multics

To clarify Figure 2, let us review exactly how linkage faults, missing segment faults and missing page faults occur in Multics. All segments - which can be either programs or data - are referred to be name, rather than by their physical location in either the memory or secondary storage (Only the basic file system knows the physical locations of a segment). When a process refers to the segment for the first time, the linker gains control through a linkage fault. (All symbolic references to segments are replaced by a linkage fault which is inserted by the PL/1 compiler when it translates a procedure into machine language.) The

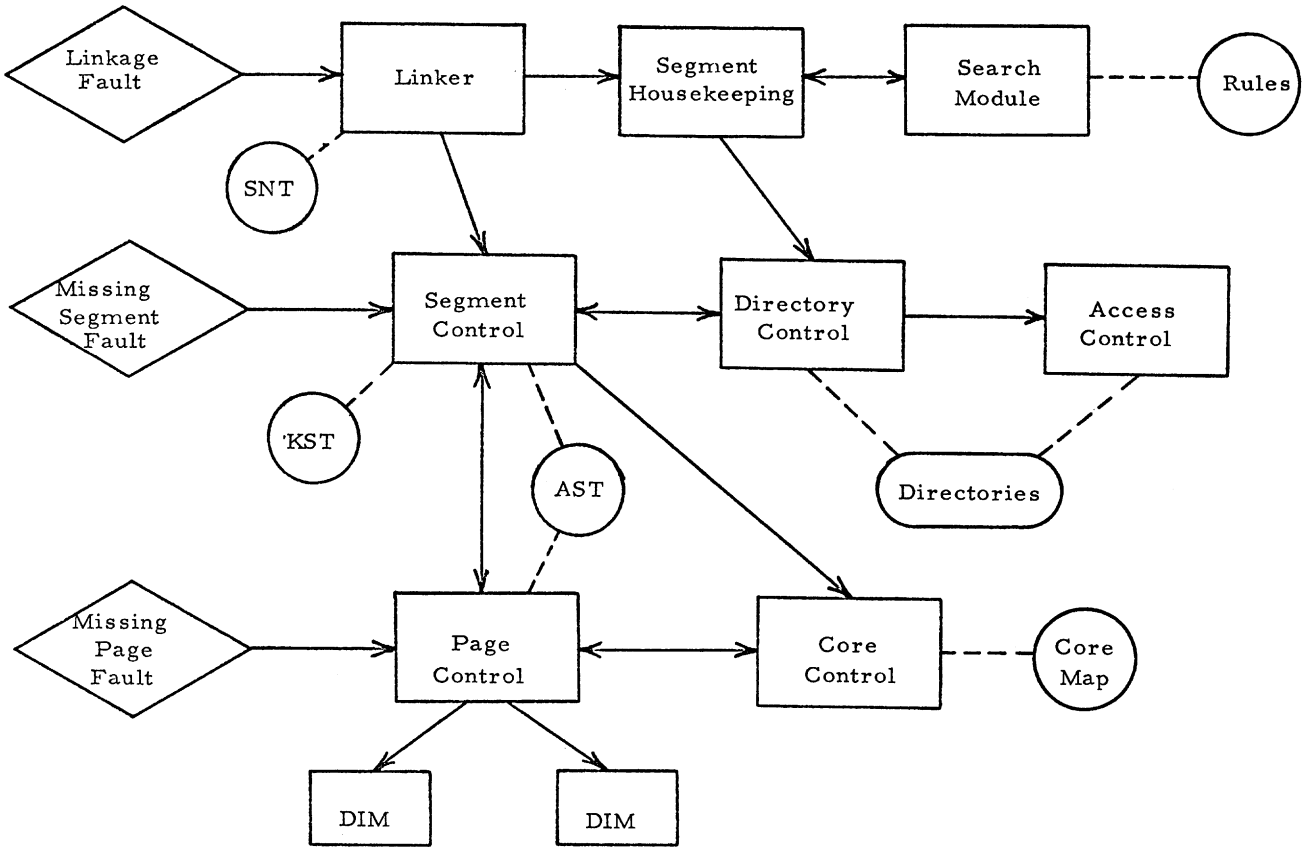


Figure 2. Functional Diagram of Linkage, Segment and Page Fault Handling in Multics

linker consults its segment name table (SNT) to see if it knows about the named segment. If it doesn't, it calls segment housekeeping to look for the segment. Segment Housekeeping calls the search module for advice on how to locate the segment. Segment Housekeeping then calls directory control to manipulate the directories and locate the segment. (This may require several recursive calls to the search module and directory control.) When the segment is located control passes back to the linker which enters the segment in the SNT. (The segment is now known to the process, but not yet loaded.) Now that the segment is located, or if it was found in the SNT originally, the linker calls segment control asking it to establish the segment. Segment control then calls access control to determine if the user has access rights to the segment. If he does, segment control assigns a segment number for the segment, creates a new entry for it in the known segment table (KST), stores the descriptor control bits, which includes the missing segment bit, in the descriptor segment, and returns the segment number to the linker. No part of the segment has yet been brought into core, but it is now directly addressable because it has a segment number.

As the process tries to make a reference to the segment a missing segment fault takes it directly back to segment control. Segment control finds the unique identifier for the segment from the KST and searches the active segment table (AST) a system-wide table, for a segment with this unique identifier. If it doesn't find it, it creates an entry in AST by

calling directory control to get its physical location. Segment control then places the segment description in the descriptor segment and calls page control to read in a page of the segment. Page control first creates a page table for the segment and returns to segment control. (Creating the page table may require that a page of core be removed to make room for the page table). The entries in this page table are filled with missing page bits. Segment control returns to the original process. The requested segment has not yet been brought into core memory, but not it has a page table.

As the original process tries further to complete its reference to the segment a missing page fault takes it directly back to the page control. Page control locates on secondary storage the specific page being referenced and finally brings it into memory by calling the disc or drum device interface module (DIM) which in turns calls the I/O system. Loading this page in core may require the removal of some other page belonging to this, or another process. To remove a page may require copying it out onto secondary storage if it is not a "pure" page. Thus, page control calls core control before the new page is loaded to see if there is room for the new page. Core control consults the "core map" to decide which page or pages of core, if any, should be removed. This decision is based on the frequency of usage of pages in core. If core control decides to remove one or more pages it calls page control recursively to remove the page(s). Page control calls the device inter-

face module (DIM) which queues the request. DIM in turn calls the I/O system to actually move the page(s).

Since the loading of the requested page, and possible prior copying of another page or pages out onto secondary storage takes time, the process generating the page fault is blocked until the page actually arrives in core. When the page arrives, the process is awakened and finally the reference to the page is completed. Future references to this page will not be subjected to such a torturous routing, unless the page goes unused for a long time, so that it is removed from core by the core control module. If so, its missing-page-bit is set on in the page table. The page is then brought back in when it is again referenced by repeating the steps just described in the previous paragraph. This same procedure is also followed when any other page in the segment is referenced. It is also possible that the segment may be removed because of inactivity. When the segment is later referenced, a missing segment fault, rather than a linkage fault, occurs and takes control directly to segment control. The sequence of steps in the previous two paragraphs are then repeated.

We may not want to model all of these modules, but we can begin to see what modules must be represented in a model in order to answer the questions posed earlier. For example, we can eliminate the function of the DIM and the I/O system by modelling it by a simple delay. This delay is drawn from an exponential distribution to which a fixed constant

is added. This assumes that most segments that are referenced are readily available, but that some, such as ones located on long access time devices such as tapes, require considerably longer to access.

Modelling the User

The user model referred to earlier needs to be complicated slightly to include the description of different segment types. We might classify all user segments as belonging to one of three classes.

a) sequential

b) cyclic

c) random

A sequential segment, an example of which might be a sequentially accessed data file, is always referenced in ascending consecutive locations. Thus, new pages will be needed at fixed intervals and old pages will not be referenced again. A cyclic segment, an example of which might be an executable program, loops through a fixed number of pages. A random segment, an example of which might be list-structured segment, generates random requests among a fixed number of pages.

When the attributes of a user are generated the mixture of these three types of segments will be specified as parameters of the user. Thus, the parameters of the user are:

1) Processor time for the request

2) Total number of segments in the request

3) Total length of all segments in the request

4) Pause time before the following request is executed

5) Number of sequential segments

6) Number of cyclic segments

7) Number of random segments

8) Average length of sequential segments

9) Average length of cyclic segments

10) Average length of random segments

The number of segments specified by parameters 5, 6, and 7 determine parameter 2. The sum of the products of parameters 5 times 8, 6 times 9 and 7 times 10 equals parameter 3. All segments of the same type are assumed to be the same length. (This is certainly not realistic, but simplifies the model) A request specifies the continuous usage of the processor, except for paging activity, and timer runouts caused by the scheduling policy. In this simple model there is no concept of multiple interactions per request. The data collected by Sherr corresponding to parameters 1, 3 and 4 are .88 seconds, 6,300 words, and 35.2 seconds respectively.[48]

After generating the above 10 parameters the request is entered in to a workqueue. When the request is completed the user delays the value of pause time. Then a new set of user characteristics is generated for the next request and the process just described is repeated.

Modelling Segment and Page Fault Handling

We can eliminate from the server model some of the modules shown

in Figure 2. In particular, we can ignore the problems of the linker, segment housekeeping, the search module, access control, and directory control in a simple model by making no distinction between the first reference to a segment and all later references. Thus, when a missing segment fault occurs control will always go to segment control which will check to see if there is an entry for the segment in the AST. If not, it will delay, make the entry in AST and call page control. Page control will check for the presence of a page table. If it is missing it will call core control. Core control will examine the core map to see if there is space. If there isn't space it will select a page to be removed and delay. It will then return to page control which will set the missing page fault bits on and return. When a missing page fault occurs, control will enter page control which will call core control again for space and return, after a possible delay, to page control which will reset the page fault bit and return to the user process.

A number of further simplifications can be made initially. We need not keep an actual AST, but can answer the question about whether the segment is active or not by a random draw from a specified probability distribution which we create rather arbitrarily. Likewise, we need not have a page table or a core map either initially. The questions regarding these data bases can initially be modelled by other probability distributions. Furthermore, after a segment fault, we will automatically enter the page fault routine. Figure 3 shows a flow chart for this simplified

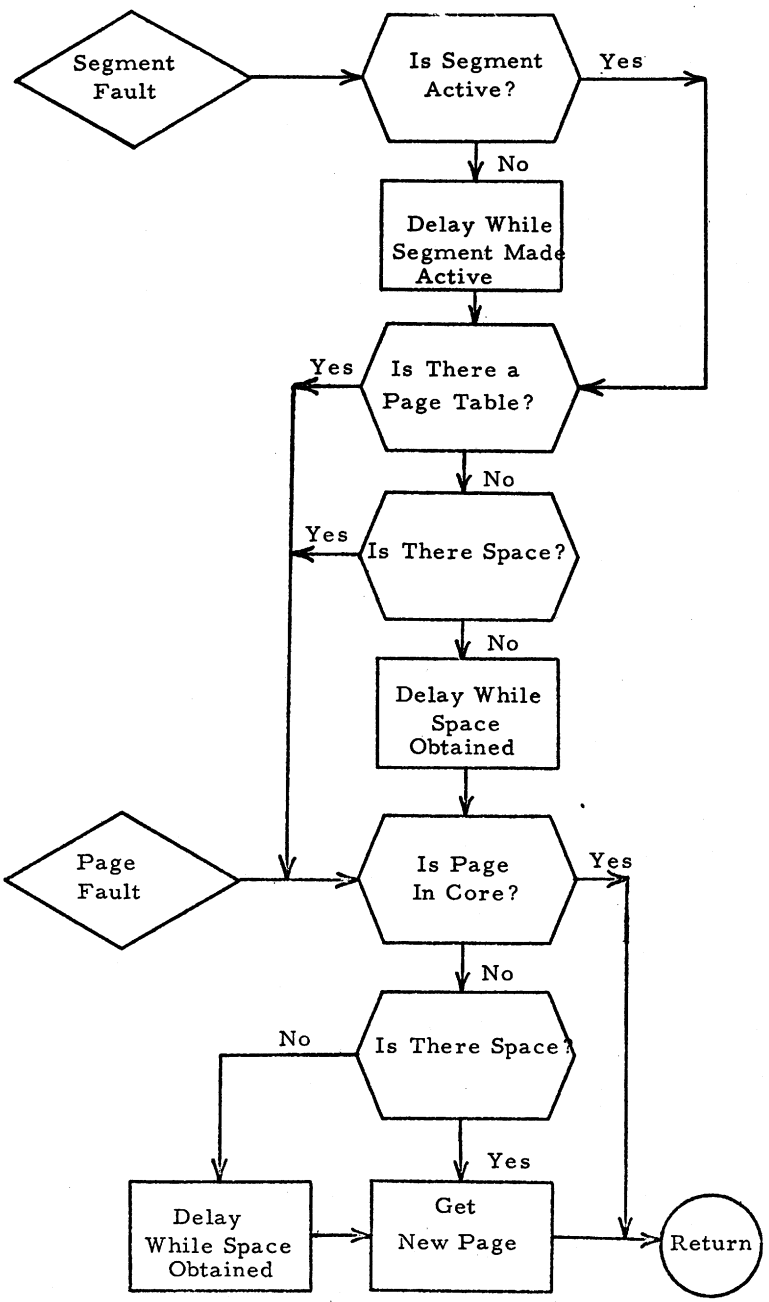


Figure 3. Segment and Page Fault Handling Model

model.

Certainly this model is inaccurate, but its forte is that it is so simple that it can be easily tested. When we are sure that it is functioning correctly, then we can replace the probability distributions by the specified data bases and gain more realism.

This is one of the advantages of constructing a hierarchical model on-line. We are encouraged to make trivial models and test them, before plunging into a mass of detail, because it takes little effort to do so and can give us insight immediately. If we were constructing the model off-line we would most likely pass over this simple model because it is so trivial.

Modelling a running process

This model is still not complete since there is no mechanism to generate the missing segment or page faults. We need to model a process in execution. Using the user characteristics about types of segments we can build a very crude model using probability distributions again.

First we draw a number which specifies what type of segment reference is to be made, sequential, cyclic or random. Using this number, and the user parameter which specifies the number of segments of this type in the process, we determine if a missing segment fault is to occur. If the answer is yes we call the segment fault routine. If the answer is no then we draw from a distribution specified by the segment type number and the average length of the segment the delay interval before a missing

page fault occurs in this segment. Before actually delaying the specified interval, we check to see if the time allocated by the scheduler for running this process will elapse before the delay is completed. If the answer is no, we delay. After the delay the page fault entry is called and the user cycle is restarted. If the process is to terminate, first we reduce the delay to the termination time and then delay. After the delay we set a switch indicating that a page fault is due and save the remaining amount of time left befoe the page fault. Then we return to the scheduler.

Modelling the Scheduling of Processes

The model of the scheduler may also be very simple. If the CTSS scheduling routine is used, it removes the new user processes from the work queue and enters them into the scheduler's queues. If a simple round robin schedule is used, the work queue may serve as the scheduling queue. In any event, the scheduler selects a user to run and calls the execution routine specifying which user is to run and for how long. (No pre-emption of a running user is allowed).

The Complete Model

This simple model now consists of the following parts:

1. An activity for generating user process characteristics called New-User.
2. An activity for scheduling the execution of user processes called Scheduler.

3. An activity for executing user processes and generating missing segment and page faults called Execute-User.
4. An activity for handling missing segment and page faults, which has two entries called Segment-Fault and page-Fault.

The New-User activity only generates user characteristics and waits until it receives a signal that the user is done running. The Scheduler is an asynchronous activity and gains control whenever a running program becomes blocked. It selects a user to be run and calls Execute-User with the particular user characteristics and the selected quantum. When the quantum is exhausted, the routine Execute-User returns to the Scheduler. When the total request time is exhausted control goes automatically to the New-User routine. Execute-User delays for an interval to represent the elapse of processor time and then calls either Segment-Fault or Page-Fault. When a delay occurs in either of these activities the current user is blocked and the Scheduler gains control again. It selects a new user to be run. When the segment or page fault delay expires control goes back to the Segment-Fault or Page-Fault routine which unblocks the particular user and puts him on the work queue. However, since another user is now running nothing further happens.

There is no provision in this model for originating and terminating users. The original generation of the users in the system can be done directly from the console or by an initialization routine which will generate a fixed number of distinct users. Termination can be ignored.

As the model becomes more sophisticated a user arrival routine can easily be added and the routine for generating user characteristics can be modified to include the possibility the termination of a user.

Building the Model

How might we start out to build this model? What activities should we write, and what ones should we portray ourselves? The New-User activity is simple so it can be written as an uncompiled OPS-4 program. A round robin scheduler is also easy to write. The activity for handling the missing segment and page faults has already been described and can also be written on compiled OPS-4 program. However, the Execute-User activity has not been thoroughly described and may profit by a certain amount of experimentation to see what might be a good algorithm for generating missing segment and missing page faults based on the user characteristics we have described. Thus, we decide to portray it ourselves.

To inform OPS-4 about the existence of this activity, all that is necessary to do is execute the PL/1 declaration statement,

```
Declare Execute-User Procedure User
```

When, during the execution of a model, control flows to this user portrayed activity Execute-User, the name of this activity and all parameters associated with it will be displayed on the console and the execution of the simulation will be suspended. We are then free to do anything we want.

Testing the Model

The three uncompiled OPS-4 programs can easily be tested by calling them directly from the console and executing them line by line to see if they work correctly. With the New-User activity we might also want to examine the collection of user characteristics to see if they are being generated correctly. The Scheduler needs little analysis since it just removes the top item from the work queue and schedules it to become active by calling Execute-User. The activity to process missing segment and page faults is the most complex. To verify that it is working correctly it will be necessary to examine the Agenda before and after executing it and also it will be necessary to see if it is manipulating the queue of blocked processes correctly. This will require displaying both the Agenda and the queue of blocked processes. When these three programs appear to be functioning satisfactorily we can put them all together.

First we might schedule two or three calls to New User to represent two or three distinct users. Then we would set some trace options so we could monitor the flow of control from one activity to another. We might also want to monitor the movement of simulated time and monitor any changes to the Agenda. Then we would start the model running, perhaps specifying that it should stop after a certain period of time or after so many activities have been executed. The first item on the Agenda - a call to New-User - would be executed, a new set of user characteristics

would be generated and the activity would wait until the user request was completly served. The next one or two calls to New-User would also be handled the same way and then suddenly an error message "No eligible activities" would be printed and the simulation would stop. What has happened?

We view the Agenda and see the suspended New-User activities but nothing else. Then we realize our mistake. Of course, we forgot to initiate the Scheduler activity! Since it is an asynchronous activity and only gets control when a process is blocked it must be initially called by us.

We reinitialize the Agenda, this time putting a call to the Scheduler after any of the calls to New-User. Now the first user gets scheduled and the Execute-User routine is called. Since we are portraying Execute-User ourselves the simulation system now requests that we perform its function. We examine the user characteristics for this particular user and decide to delay for a specified period of time. When this delay expires (which will be immediately since no other activity intervenes) we will then generate a call to either Segment-Fault or Page-Fault.

After executing the model this way for a while we may discover some errors that need to be corrected in the three uncompiled OPS-4 programs. If so, we simply edit the programs to make the corrections and immediately start to execute them again. No intermediate compil-

ation is necessary.

Completing the Model

The continual demands made upon us to portray the Execute-User activity may grow burdensome, and we also may now feel prepared to formalize the process that we have been doing in an ad hoc manner. So we write an uncompiled OPS-4 program, test it individually, and then include it as a part of the whole model. If everything works we're now prepared to start thinking about how to answer the original questions which inspired the construction of this model.

We realize that we need to collect some statistics. Specifically, we need to know 1) the total number of user requests completed in a given period of time, 2) the total elapsed time from start to finish of a user request, and 3) the percent of total time that was spent accomplishing supervisory functions as opposed to executing user programs.

The first is easily obtained by just adding one to a counter each time a request is completed. The second item is also easily obtained since the OPS-4 system automatically records the simulated time at which a new activity first becomes active. Thus, all we need do is add a few statements to the end to the New User routine to subtract the current time from the beginning time of the activity and store it for further processing, tabulate it in a frequency distribution or send it to an output device. The third item is also obtained without difficulty. If just requires accumulating all the delay times in the scheduling routine and

Segment-Fault and Page-Fault routine. At the end of the simulation this time is divided by the total simulated time to give a percentage. After making these few modifications we might decide to compile these 4 activities to speed up the execution of the simulation.

Lending Realism to the Model

This model's results will be realistic only in so far as the algorithm for generating missing segment and page faults and the modelling of the core map approximates the real situation. We could do what Scherr did, and collect a large number of statistics about user program characteristics and missing segment and page faults, or we could complicate the model substantially by adding detail and keeping t rack of what segments and pages are active both for each user and also system-wide - i.e. construct an individual and overall core map. The latter is equivalent to writing the equivalent pieces of the basic file system.

This raises the interesting question of why not include pieces of the basic file system as part of the model? One problem is the protection of the basic file system. It is part of the hard core supervisor and does not allow direct calls to be made to it from a user level. If this barrier couldn't be removed it might be possible to get signals from the basic file system regarding missing segment and page fault handling which would be used to drive the rest of the model. That is, the Execute-User activity would be replaced by a routine which monitored the basic

file system and collected the statistics regarding delays associated with missing segment and page faults. Also, in place of generating user characteristics, several real programs could be used.

The Model as a Design Tool

Even if it were not initially possible to incorporate actual pieces of the basic file system into the model or to monitor the actions of the basic file system, this primitive model could act as a crude design tool. First of all, it could show under what extreme loads the percent of overhead becomes very large or very small. If a realistic core map were included in the model it could be used to investigate alternate policies for removing pages from core. A preliminary estimate about the balance between numbers of users and the memory, processor balance could be obtained. Guidelines could be established about what might be the ratio between processor time and elapsed time for a request, etc.

Constructing a Model from the Bottom-Up

The bottom-up approach to modelling Multics discussed in the begining of this chapter might consist of modelling each module of the Basic File System separately, and then combining them hierarchically to form the complete model. In constructing these individual modules we might start experimenting with various formulations by executing statements directly from the console. When we felt we had a workable description we might choose to write it down as a program, execute it further and then compile it. Alternately, we might immediately start to write a

program and then test it, making repeated changes in the program specifications until it was accurate. The ability to modify and then immediately test OPS-4 programs without any intermediate compilation request is most important in this phase of model building.

Adding structure to the Model

When we felt we had sufficient number of the basic modules of basic file described we might wish to try combining them together. For example, we might like to check the interactions of the core control and page control modules. To do this we might follow the same route as before, first trying various combinations by hand, as it were, and then as we felt more sure of the way they interacted we could solidify the relationships by writing them down. During this stage, it is possible that some errors, or incompatibilities might become apparent in the definition of one or more of the basic modules. If that is the case we could easily modify the OPS-4 version of the program for one or more of the modules and test the changes, running some modules as OPS-4 programs, some as compiled programs, and perhaps specifying the interrelationships between the modules directly from the console. This process could continue until the model is completed. Along the way, we might find it beneficial to portray some of the modules, such as the search module ourselves. When the model is complete we would no doubt find it similar to the one constructed from the top, down. A possible difference might be that this latter module would contain more

detailed structure than the former.

Material Based Versus Machine Based Models

There are two points of view about the different facilities for constructing simulation models that this model of Multics illustrates. In this model each user process may be thought of as a different transaction flowing through the computer with its own distinct characteristics. The computer is viewed as a static object which operates on the transactions in identical fashion. This is what is referred to as a material based model - i.e. the items of material to be operated on are non-homogeneous, while the processing operation on all items is identical. However, it is also possible to view all user processes as homogeneous objects which simply generate segment and page faults. The computer is then required to distinguish between these two types of requests and react differently to them. This orientation is called machine based.

As we have just seen there is nothing inherent in OPS-4 which restricts a user to only one approach. Either one or both may be used. This freedom is provided by the availability of the rich data structures provided in PL/1, and the flexibility of the OPS-4 language for specifying the individual processing of transactions. The only disadvantage that this flexibility incurs is that it is not possible to easily provide automatic statistic gathering on pre-defined machine usage such as is available in GPSS, a material based languages, since there are no standard

machines defined in OPS-4. This disadvantage is outweighed by providing the flexibility to specify any machine or material types. Also, simplified statistic gathering facilities are available in OPS-4 and are discussed in Chapter 8.

Activities and Events

Some languages, such as SIMSCRIPT, require that a model be specified only in terms of separate events.[8] That is, if we were to model the action on on-line disk storage device in SIMSCRIPT we would have one event that specified the beginning of the disk seek, and which scheduled another event called the end of the disk seek. Such artificiality is not necessary in OPS-4. Instead a delay statement is used to specify the expiration of a certain amount of time before the activity representing the disk continues. This is much the same as one would do the the SOL language.[12,13] SOL, however, makes it difficult to schedule events directly. Everything is oriented toward the activity concept. All scheduling is done implicitly by the simulation system itself. In OPS-4 the approach of SIMSCRIPT for scheduling events directly is also available.

In OPS-4 an activity is described by writing a program which may define several events. For example, the activity New-User which generates user characteristics might be written as follows:

```
New User:   Procedure;
            Declare 1 user-char controlled,
                  2 processor-time float,
                  2 total-seg float,
                  2 total-length float,
                  2 Pause-time float,
                  2 Number-seq float,
                  2 Number-cyclic float,
                  2 Number-random float,
                  2 Length-seq float,
                  2 Length-cyclic float,
                  2 Length-random float;
            Allocate User-char;
Repeat:     Draw Processor-time from exponential 88.;
            Draw pause-time from exponential 35.2;
            .
            .
            .
            Draw length-cyclic from uniform 500 5000;
            Draw length-random from uniform 1000 20000;
            Enter User-char bottom work-queue;
            Wait Until processor-time = 0;
            Delay Pause-time;
            Total-time = Sys-time - btime;
            Display Total-time, User-char;
            Total-requests = Total-requests + 1;
            Go to Repeat;
            End New-User;
```

Conditional Activities

OPS-4 allows the execution of an activity or event to be dependent on some condition. Two different classes of conditions are distinguished. The first is identical to the event concept of PL/1 which is implemented in Multics with the Block and Wakeup modules in the Traffic Controller and the use of an Event Table. One activity may send a signal to any other interested events by the Set Event statement in PL/1 which declares that a specific named event has occurred. One or more other events

(or activities in OPS-4 parlance) may determine if a specific event has occurred by the statement Note Event. Also, one event may specifically Wait for another specified event to occur before it proceeds. This type of conditional activation of activities is limited in generality, but it is implemented in a very efficient manner in Multics.

The second type of conditional execution of activities allows the user to specify any relation between global and local variables as the triggering statement. For example, the condition 'processor-time = 0' was specified as controlling when the activity New-User was to be continued. This type of scheduling is extremely comprehensive, and subsumes the first type of scheduling, since Set Event could be replaced by setting a switch the name of which would be specified in a Wait statement. However, because of its generality, the testing of these conditional statements must be done continusously. Therefore, the execution speed of a simulation model having many conditional statements of this latter type is quite slow. In the Multics model the page control module could use the event type of conditional scheduling just described to alert it when the transporting of a page to and from core memory was completed. For example, it might state,

Wait Signal

where Signal is the name an event defined by the I/O module.

The Information Feature

Many times during the course of constructing a model, the user may need to be refreshed about the exact details of how a particular feature or statement in OPS-4 is used. OPS-4 will have available an on-line information system which will describe how to use the features of the OPS-4 system in sections of graduated detail. A similar system has proved to be a very useful and important feature in the OPS-3 system. There will be a few differences, however, in this feature, called 'Info,' in OPS-4, which are dictated by experience using the Guide operator which supplies information in OPS-3.

The basic difference is that an Info request will only supply information about one subject at a time. When it reaches the last section pertinent to the specified subject it will automatically return to normal execution. If there are one or more sections to a specific subject a carriage return automatically continues with the next section. If anything else is typed Info returns to normal execution and delivers the typed line to be executed by OPS-4. Furthermore, in addition to being able to ask for a specific section pertaining to a subject, a user may specify that he wishes to see all the sections pertinent to a subject.

The Info system will be implemented differently than the Guide feature in OPS-3. It will allow the user to type, in natural English, his requests for information about OPS-4. A program similar in nature to ELIZA, will scan for key words in the sentence and determine what information the user desires.[51] A hash-coded index of all subjects will

be stored in <OPS.Index>.* This index will contain the address of the beginning of the first section of each subject in <OPS.Information>. The format of <OPS.Information> will be a tree-structure so that each subject may have several sections, each of which have several subsections, etc. Facilities will be provided, as an integral part of OPS-4, to allow authorized users to revise the information contained in <OPS.Information> and create a new <OPS.Index>. This authorization will be implemented as part of the access control information stored in the directory branch of these two segments.

Summary

This chapter has described the various alternatives and features available to a user for constructing complex simulation models. It has particularly discussed the importance of hierarchically structured models. Using the segmentation and paging mechanism of Multics as an example, it has shown how a model may be constructed from the top, down, and from the bottom, up. The three types of programs that OPS-4 recognizes have been illustrated in the example. The richness of the OPS-4 language in providing both the machine based and material based orientations to modelling, and also the flexibility of two types of conditional execution of activities has also been discussed. Finally, the on-line information system available in OPS-4 has been described.

---

*The use of triangular brackets is the standard convention adopted to denote a segment in Multics. Thus <a> means the segment named "a".

Chapter 4

THE SIMULATION DATA BASE

The structure of the data base is an important aspect of every simulation. The particular way a user chooses to organize his data base strongly ineracts with the way he constructs a model. The structuring of the model and the structuring of the data base must be done in concert. One should not dominate the other, and both must be easily subject to change.

This chapter describes the concept of global and local variables and discusses how they are allocated among various segments by PL/1 and OPS-4. The ability to easily restructure both the global and local data bases is described as a particularly important feature of the OPS-4 system. The various types of data objects available in PL/1 are reviewed and the additional ones provided in the OPS-4 system are described. The special provisions OPS-4 makes for manipulating the global data base and its associated symbol table are also discussed. The concept of hierarchically related data bases is discussed. Finally, the solution OPS-4 adopts for allowing multiple copies of local data bases belonging to one procedure to co-exist and be uniquely identified is described.

Global and Local Variables

A simulation model almost always contains global data - data available to every activity in the simulation - and also local data which is available only to the activity that declares it. The global data establishes the environment in which the model operates. Every activity may freely modify the global data base without restriction. Local data, however, is only known to the activity that declares it, and can be used by other activities only if it is explicitly passed to them as parameters in their calling sequences. In this respect a simulation model is no different than any other collection of interacting programs. OPS-4 also allows one activity to manipulate the local data base of another activity

by explicitly linking to the data base. This is described at the end of this chapter.

The model of segmentation and paging in Multics discussed earlier almost completely ignores the problem of data bases. Almost every data base is eliminated to simplify the model. There are two types of data bases necessary in this model - a global data base for the directories, AST and Core Map and a data base local to each process group (a user in our model) for the SNT and KST. The external attribute in PL/1 declarations allow us to define the former as global data objects so that they may be available for use by all activities in our model.[32] The SNT and KST will not be defined with the external attribute and hence will be local to the individual activities.

Since activities are independently written procedures PL/1 requires that the declarations of all global (external) variables be repeated in every activity that refers to them. This is not necessary in OPS-4. It maintains a special global symbol table. If a user executes any PL/1 declaration statements directly from the console the declared symbols are automatically entered into this global symbol table irrespective of whether the external attribute was specified in the declaration. That is, the user is assumed to be operating at the global level unless he specifies otherwise. Alternately, any PL/1 declaration statements with the external attribute which appear within an OPS-4 program always add the definition of the symbols to the global symbol table when they are

executed.

The OPS-4 symbol table search mechanism always checks the global symbol table for a symbol definition if it is not defined in the local symbol table of the activity. Thus, in OPS-4 global symbols only have to be defined once, either directly by the user from the console, or by one activity. If a symbol is defined more than once the latest definition always superceeds the former definition. This allows a user to easily restructure or change the attributes of any variable at any time by giving just one declaration statement.

When an OPS-4 program is compiled, the special OPS-4 to PL/1 translator discussed in Chapter 9 automatically creates declaration statements with the external attribute for all symbols referred to in the activity that are not defined in the local symbol table, but are defined in the global symbol table.

If a user wishes to write his program initially in PL/1 he will, of course, have to declare all the global variables explicitly. The use of the 'insert file' feature recognized by the language translators available in Multics simplifies this task. It allows a user to create a file that contains all the declarations for common variables used by several programs. In place of writing these declarations in each program the user merely writes INSERT FILE XX, where XX is the first name of the file containing the declarations of common variables.

Local variables are handled normally as in standard PL/1 programs.

They are known only to the activity in which they are declared and allocated by that program. If several instances of an activity exist simultaneously during the simulation, each one will have its own local data base. The problems of identifying and keeping these local data bases separate from one another are discussed later in this chapter. These problems are unique to simulation models and distinguish them from other collections of interacting programs. They are, however, similar to the problems Multics encounters in allowing a pure procedure to be simultaneously executed by several different processes.

Implementation of PL/1 Storage Allocation in Multics

Because of the variety of data types and modes of storage allocation available in PL/1 it may be useful to review here how data storage is allocated by the PL/1 compiler in Multics. There are 3 modes of storage allocation in PL/1 - static, automatic, and controlled. Automatic is the default mode. It is used to provide the dynamic storage mechanism required by the block structure of PL/1. Static is similar to the normal FORTRAN type of allocation. Storage is allocated at the beginning of the program and never de-allocated. Controlled storage is allocated by the user with explicit allocate and free statements.

In Multics there are several standard segments used for storage allocation. They are <stat_>, <free_>, <stack>, and the procedure segment itself, <proc>. All static variables whether external or internal variables are placed in <stat_>. All controlled variables are placed in

<free_>. Also, all based variables, e.g. those that are referred to by a pointer variable are placed in <free_>. All local (internal) variables of automatic mode are placed in the <stack>. Any constants are placed, along with the instructions, in the procedure segment. OPS-4 follows these PL/1 rules in part and also refers to additional data segments, which are described later.

Restructuring the Data Base

One of the major features of the OPS-4 system is that a user may restructure his data base at any time by just executing new declaration statements. All OPS-4 programs will adapt to the change automatically (Chapter 9 discusses how this is done). Only compiled programs need be modified. This allows the user to concentrate on his model and ignore many of the mechanical details.

For example, in the Multics model, we might go through many iterations of changes in the data structure of the KST, AST and core map before we decided on the proper structure. We could add or subtrace elements and easily incorporate the new data manipulation statements in the model without any recompilation, provided all the activities were not yet compiled.

Sets, Queues, and Tables

To make the OPS-4 system more powerful and useful for simulations, three new types of data objects are added to those already available in PL/1. The first data object is a set. A set may contain any data object

as a member, e.g., scalars, arrays, strings, structures, etc., and even other sets. The implementation of sets in OPS-4 uses the SLIP type of list structure.[52] The declaration of a symbol as a set serves only to establish the set head and a pointer to it. All sets are allocated storage in a special segment, <OPS.lists>. The decision to do this rather than place sets in <free_>, is so that OPS-4 can control garbage collection of this segment. (Chapter 8 discusses the structure of sets and garbage collection.) With the addition of this new type of data object OPS-4 also expands the statements in PL/1 to include facilities for entering or removing elements at either the top or the bottom of a set, and before, after and in place of a specific member of the set. Statements are also provided to search sets in a forward or backward direction, and to detect the beginning or end of a set so that searching may be terminated. These statements are listed in the Appendix and are patterned after those available in SIMULA.[14,15]

The second new data object in OPS-4 is a queue. Queues are very similar to sets. In fact, the set manipulation statement may be used directly on queues. The difference between sets and queues is that OPS-4 provides some statements for monitoring queue sizes, computing the elapsed time of items in queues, etc. (Chapter 7 discusses this in more detail). Thus, a user will use a queue in deference to a set only because he desires to collect some statistical measures of queue usage. The implementation of queues in OPS-4 is similar to that of sets, and all

queues are also allocated storage in OPS. lists .

The table is the third type of data object added to PL/1 by OPS-4. It is provided to allow the user to easily collect distributions and is similar to the table features available in GPSS, SOL and OPS-3.[10, 12, 13, 18] The declaration of a table includes the lower and upper limits of the table range, and also the cell interval. Storage for tables is allocated in <free_> if it is declared to be external or in a special local segment if it is internal.

A tabulate statement is provided to allow a user to update the table with a specified count, or an implied count of one. Also, a variety of display statements are provided to allow the user to print the table in tabular form, or plot it either as a density or cumulative distribution, and as a bar graph, or broken-line graph. These display statements allow the user to specify the range of the table entries to be displayed and also vary the cell size so that the definition of individual cells may be magnified, or groups of contiguous cells may be aggregated. This flexibility is extremely important when the range of the table is large and the user wishes to see a condensed version of the entrie table, or else an expanded view of just a subset of the entire table. Standard default attributes are defined if the user is not specific in his output requests. These display statements may be executed at any point during the simulation and provide an excellent debugging tool. (See Chapter 7 for details.)

Contents of the Global Symbol Table

The global symbol table contains definitions of many other entities besides the global variables accessable to all activities. It is divided into permanent and transient sections. The permanent section contains a list of entries for every standard statement in the OPS-4 language giving their segment and entry names. This section of the global symbol table may be modified only by authorized people, and is in the special read-only segment, <OPS. statements>.

The transient portion of the global symbol table is in the segment, <OPS. symbol>. It contains the following types of entries:

a) Definitions of all global variables.

b) The names of all known OPS-4 programs that have not yet been compiled.

c) The names of all user portrayed programs.

The user may delete any of the entries in the transient symbol tabl e at any time. The OPS-4 system automatically deletes the name of an OPS-4 program when it is compiled by means of the OPS-4 to PL/1 translator. The names of user portrayed programs must be deleted explicitly by the user.

Manipulating the Global Symbol Table

The global symbol table is usually an important element of any simulation model. Therefore, special facilities are available for manipulating it directly within the OPS-4 system. The user may do the following:

1. He may uniquely name the current symbol table. (This can also be done by the rename command of Multics).
2. He may specify that a particular segment is to replace but not destroy the current symbol table. (This can be done by two rename commands in Multics.)
3. He may initiate and name a new symbol table which replaces but does not destroy the old one.
4. He may initialize the current symbol table, e.g. delete all the definitions. (This is equivalent to starting a new symbol table having the same name as the old one.)
5. He may append a segment to the current symbol table. (Note, since the symbol tables are hash coded, this is more than just a simple concatenation of the segments).
6. He may obtain a listing of the contents of the current symbol table.

All of these above options refer only to the transient portion of the symbol table. All symbol tables in OPS-4 are never deleted unless the user does so himself.

Manipulating the OPS-4 Data Bases

A user needs to have similar facilities for manipulating the data bases created by the OPS-4 system. In particular, the ability to uniquely name the current data bases, substitute different data bases for the current ones, and clear the current data bases are all frequently needed

functions when a user is experimenting with several different data sets belonging to the same model, or when a user is switching between several models. Also, during the stages when the user is incrementally building his model it is important to be able to combine various fragmental data bases.

Although most of these functions can be accomplished with Multics commands, the fact that there are several segments which taken together constitute the simulation data base makes it unnecessarily burdensome to require that he do so. Having special OPS-4 statements which simultaneously manipulate all the data bases that are pertinent to a given model is a considerable convenience.

Private Data Bases

Two or more activities may wish to jointly access data, but yet not make it generally available. Thus, they need their own private semi-global data in addition to their local data bases and the general global data base. The PL/1 external declaration attribute is adequate for describing this type of arrangement in compiled routines. Only those routines containing the external declarations will have knowledge of the existence of these private global variables. Thus, various routines may have different sets of external declarations. The intersection of the sets of declarations will define the data that is common to both routines. This is similar to the named common feature of FORTRAN IV.[53] Thus, although all external variables are stored in the same data segments,

namely <stat> and <free_>, they are not generally available.

However, in OPS-4 programs, as opposed to compiled programs, such protection does not exist. Any variable declared in the global symbol table is available to any other OPS-4 program. To limit access to only specifically declared programs an additional attribute called access is added to the standard PL/1 attributes allowed in a declaration statement. This access attribute may be followed by a list of programs that may be granted access to the declared variable. Conversely, a no-access attribute denies access to these named programs and allows all others access. These lists are stored in the global symbol table, and checked by OPS-4 before it allows any OPS-4 program to reference the variable (See Chapter 9 for details). When OPS-4 programs are compiled the OPS-4 to PL/1 translator generates external declarations for all global variables having no access restrictions and those variables which specifically allow access to the program being compiled.

The user, when executing statements directly from the console, is always allowed access to all variables. If this were not so, he could only debug programs by being within them - a very unsatisfactory restriction.

Hierarchically Structured Data Bases

The Multics debugging package allows symbol tables to be hierarchically structured. Thus, it follows that the corresponding data bases will be related to each other in a hierarchical manner. This allows the block structure concept to be dynamically extended during execution to

independently written and/or compiled routines. Although it does not appear that the PL/1 compiler will make use of this facility, OPS-4 programs may do so. Any variables declared in hierarchically superior OPS-4 programs will be known to the inferior OPS-4 programs, because symbol table linkages will be created dynamically as control flows from the superior to the inferior routine, and all symbol table searches in OPS-4 always start with the local symbol table and progress to the global symbol table.

Debugging Considerations

This dynamic hierarchical structure does present an interesting anomaly. When an OPS-4 program is executed independent of its superiors, e.g. when it is called directly by the user from the console, it will not know about any superior symbol tables except the global symbol table. Thus, symbols defined in higher level programs will be undefined. This may be a desirable feature. If, on the other hand, it is not, the user has the facility provided by the Multics debugging package to link manually the symbol tables together himself.

This hierarchical structure of data bases and symbol tables does make debugging more difficult. The user at the console is most naturally considered to be the most superior level. Thus, all variables defined in anything but the global symbol table are not directly accessible by him. However, the Multics debugging package has introduced special features to allow a user to reference any variable relative to the segment that

declared it. Thus, a?b means the symbol a within segment b. OPS-4 allows this type of referencing, and offers an extension. A user may specify a path to a particular lower level program, much as he specifies a path through various directories in the Multics file system. Specification of this path automatically establishes the symbol table linkages so that all variables defined in superior blocks will be known to the specified program. Then he may test and probe within the program and achieve identical results as if control had flowed to the program normally. A simple statement such as top will automatically return him to the top level again, and unlink the symbol tables.

Multiple Copies of Local Data Bases

Several instances of a simulation activity may exist simultaneously during a simulation. For example, in the Multics model there will be many different users being simulated simultaneously by just one set of programs. (All OPS-4 programs are pure procedures - i.e. they do not modify themselves.) Each of these users has certain data, such as the description of his programs characteristic and the Known Segment Table (KST) which must be kept separate from the other users. These data bases are local to the programs that manipulate them. Therefore, the problem is how to allow replications of the local data of an OPS-4 program.

The dynamic storage allocation features of PL/1 which allocate new storage each time a procedure is called provide a partial solution.

Each time a new call is made to a program in Multics it normally saves all the calling information in the stack and allocates new storage to contain the local data. As explained previously some of this data normally saves all the calling information in the stack and allocates new storage to contain the local data. As explained previously some of this data normally goes into <stack> and some of it into <free_>. Upon return to the original caller the local data areas are de-allocated. The only type of storage allocation which could cause difficulty is static. Allocation of storage for any variables declared to have the static attribute is done only the first time the procedure is entered. This causes no problems for static external variables, since they belong to the global data base, and there is only one instance of global data. However, local data with static storage allocation will not allow each replication of the data to be distinct. Instead, the latest version of the data will use the same storage as any former versions as thus obliterate the previous data. In many situations this may be the desired intent of the user. However, if it is not, it should be brought to his attention as a possible source of difficulty. Therefore, OPS-4 will flag any declarations of local data having the static attribute. Also, any pointers to allocated areas which normally are allocated within <stat_> will be flagged.

Difficulties with the Multics Stack

The normal Call, Save and Return macros used in Multics manipulate the stack as a simple push-down stack. That is, when a Call and

Save are executed it puts information indicating the calling routine on the top of the stack and pushes down one level all the information previously placed in the stack. When the corresponding Return is executed it takes the information pointed to, which is usually at the top of the stack, removes it, thus re-defining the top of the stack and uses it to determine where control should return. In recursive calls of procedures this mechanism operates correctly, since returns are executed in the same order as the calls are made, e.g. LIFO. However, simulation activities do not behave so properly.

Consider the following example. In the Multics model, the activity which simulates the handling of segment and page faults is called many different times. Each call may represent a different user and therefore must be kept distinct from any other calls. This segment and page fault handling routine simulates the time it takes to access secondary storage by a delay of a random amount of time. Assume user A calls the routine, which we will call Fault, at time 1200 and draws a delay time of 20 units. This delay interrupts the execution of Fault and control passes to the central simulation system. At time 1205 user B calls Fault and draws a delay time of 25 units. B will therefore return from Fault at 1230 whereas A will have already returned from Fault at 1220. Thus, A returns before B does. This causes trouble in a normal push down stack. When B calls Fault it puts information on top of the information A already put in the stack.

When A returns before B, it destroys B's information when it pops the stack.

The OPS-3 system avoided this problem by laboriously emptying the stack and saving it, along with all the local variables every time a DELAY or WAIT statement was executed in OPS-4. When it came time to return to the activity, the stack was reloaded and the local variables were reset to their previous values. This was a time consuming operation, but was dictated in part by the absence of dynamic storage allocation in the MAD language, the chief programming language available for users in CTSS to write their simulation models.

Solution to the Stack Problem

What is really needed, is a multiple stack arrangement, so that each call can be kept separate. Although Multics provides facilities for the user to redefine the stack by a simple supervisor call, the new stack is chained to the old stack, and thus is not independent. This is done because Multics wants to always keep track of the flow of control between programs, so that it can bail a user out if his program goes out of control. Separate processes do have separate stacks, but OPS-4 must work initially using just one process. OPS-4 makes use of the controlled mode rather then the automatic mode of storage allocation to provide a solution.

It implements its own multiple stack mechanism by allocating each instance of a procedure's local data base in a different segment. The

conventions for naming segments are described in the next section. All local variables are stored in this special segment. The pointer to the segment (e.g. the segment number) is maintained in a special definition block created by OPS-4 for each activity. This definition block also contains other important information about the activity and it is described in detail in Chapter 5. When a Delay or Wait statement is executed in an activity, the contents of the normal Multics stack is copied into this special segment. When the activity is continued at a later time this segment is accessed and is used to reload the Multics stack.

The reason for copying the stack is compatibility. Unless all procedures are processed by the OPS-4 to PL/1 translator, or unless a special PL/1 translator is available for OPS-4 users, it is impossible to guarantee that all storage is allocated in these special local data base segments. Thus it is necessary to perform the time consuming operation of copying the Multics stack into the local data base segment when an activity is interrupted. It allows normal PL/1 routines to use the automatic storage allocation features, and not confound the simulation system. If all routines are written in OPS-4 the stack will not contain any data, but only the flow of control information.

Linking Activities

One activity may wish to examine the local variables of another activity but not transfer control to it. For example, we might wish to

simulate a scheduling program in Multics which needed to examine information contained within each process to decide which process to schedule next. (Note: this is not the scheduling method used in Multics.) This is rather unusual request, since each activity's local variables are purposely not readily available to other activities. However, its implementation is straight forward.

The principal problem is identifying the specific instance of an activity whose local variables are to be made available to another activity. To simplify this problem, OPS-4 allows each activity to be uniquely named. However, it does not require that activities be uniquely named. For example, to uniquely name each instance of the activity which generates user characteristics we might say,

Schedule New-User named Sam

We could have also said just,

Schedule New-User

or,

Schedule New-User named Joe

That is, when creating a new activity we have the opportunity of uniquely naming that instance of the activity. If no unique name is assigned, the generic name, in this case, New-User followed by a unique serial number is assigned by OPS-4 automatically. These activity names are pointer variables and are created dynamically by the Schedule statement. They point ot the definition block for the activity. All definition blocks

are stored in the special segment <OPS.Activities>. The structure of definition blocks is described in Chapter 5. The unique activity name is also used as the name of the segment which contains the local data of the activity.

Specifying the specific name of an activity within another activity causes the local symbol table for the generic activity and the specific named local data base segment to be linked to the symbol table of the current activity. All the local variables of the linked activity may then be symbolically accessed using their variable names defined within the activity, and manipulated as if control had flowed to the activity. After the original activity is finished manipulating the variables of the linked activity it unlinks itself. This linking and unlinking mechanism is a direct adaptation of the connect feature in SIMULA.[14,15] It also has a parallel with the link that a user may make between files in different directories in Multics.

Initialization of the Data Base

Since the global data base may be manipulated by all activities, it is very easy to have one special activity which initializes it before the simulation begins. It is also possible to initialize the data base, assuming it was created previously, by using one of the data base manipulation statements discussed earlier in this Chapter.

Summary

This Chapter has introduced the concept of a global data base and its associated global symbol table and described the features for manipulating them which are particularly important in an on-line simulation model. It has also discussed how the local data bases of activities may be hierarchically structured and the unusual debugging problems that this presents. The augmentations of the PL/1 data types by the inclusion of sets, queues and tables is described. Finally a solution to the problem, unique to a simulation system, for manipulating and identifying the multiple instances of activities is presented.

Chapter 5

CONTROL OF ACTIVITY SEQUENCING

One of the unique features of a general simulation system is a mechanism for controlling the sequencing of activities in a simulation - e.g. managing the flow of control between sections of the simulation program. This Chapter introduces the concept of an Agenda, which is an ordered list of activities and shows how it is used to determine the sequencing of activities. It discusses the statements available in OPS-4 to add, modify or delete entries on the Agenda. The various different states of a simulation activity are discussed and it is shown how the statements in OPS-4 may change the state of an activity.

The Need for Activity Sequencing Statements

Normal collections of interacting programs find the subroutine call mechanism adequate for managing the flow of control between the parts of the program (e.g. subroutines). Even complicated heuristic programs do not require a special calling mechanism, although the flow of control is certainly far from predictable in these programs.[42, 43] What is different about simulation programs? Why is the subroutine calling mechanism not adequate for them?

It is because a simulation model is not just one program, but several programs operating in parallel. Each activity in a simulation model is conceptually executing in parallel with the other activities in the model. Since even new multi-processor computers may not have as many processors as there may be activities in a model, it is necessary to have some means of simulating the simultaneous execution of many activities on a one or n-processor computer, where n is less than

the number of activities in the model.

Thus, simulation systems need to have a mechanism for allowing separate programs to be run sequentially but appear to be running in parallel. A mechanism is needed for allowing these programs (e.g. activities) to transfer control among themselves, in a completely unpredictable manner. This is necessary because simulation programs may contain stochastic elements in them which determine when they want to run and for how long. Thus, it is impossible in general to predict when one program may wish to run, and how long its execution will take. If both of these factors were fixed it would be possible to specify a sequence of calls from one program to the next program and the standard subroutine calling mechanism would suffice.

If simulation activities had only the mechanism of the conventional subroutine calls available to them for transferring control, each activity would have to call every other activity when it was finished executing to see if any of them wanted to start executing. If more than one activity did it would have to resolve conflicts and assign priorities.

A slightly more sophisticated plan might utilize a special subroutine which tests the activities rather than require each activity to do its own testing. Then each activity would call the special sequencing routine when it wished to transfer control to another activity. Even this proposal has severe limitations. It requires that this special subroutine be aware of the presence of all activities in the model. Also, it

is very inefficient since this sequencing routine must check each activity every time it is necessary to transfer control from one activity to the next. Also, it is very difficult to implement any sort of a priority scheme, other than a fixed one which specifies the order in which activities are to be tested for execution. However, polling schemes similar to this were used in the early simulation programs.[26]

The Agenda

An obvious solution to this problem is to introduce the concept of an intermediary which does not need to know about all activities in the model, but only those that wish to be executed. When activity A wished to transfer to activity B, it would tell the intermediary about this desire, rather than transfer directly to B itself. The intermediary might then complete the transfer to routine B, or knowing that something else should be done before the transfer to routine B, it might interject a transfer to several other routines before it transferred control to B. In fact, the possibility exists that one of the intermediate routines might cancel the transfer to B and thus negate the original intent of routine A.

The simplest form of an intermediary is a queue. The names of activities wishing to be executed are entered in the queue. A central dispatcher examines the queue after each activity finishes execution and transfers control to the next activity. This technique augmented by a clock for internal timing is used as a simple scheduling mechanism in

many real-time programs, and in Multics itself.

Simple queues are limited to either a FIFO or LIFO policy. A more general queue discipline is necessary. The Agenda is the name given to this more flexible queue in the OPS system. The Agenda is an ordered list of activities and is implemented as a special type of set. (Its exact structure is described later in this Chapter). The order of entries on the Agenda indicates the order in which activities should be executed. However, because OPS-4 allows the execution of some activities to be conditionally specified, the actual order of execution may differ from the order of the entries on the Agenda. The Agenda is only used to route the flow of control between activities. The normal subroutine call is used for indicating the flow of control within an activity.

Modifying the Agenda

There are only three types of changes that can be made to the Agenda. A new entry may be added, an existing entry may be deleted, or an existing entry may be modified. Modifications to entries on the Agenda may cause them to be refiled in a different location.

Because the Agenda is really just a special set, all the set manipulation features are appropriate for the Agenda. That is, entries may be placed at the top or the bottom of the Agenda, immediately before or after an existing entry, or in place of an entry already on the Agenda. Likewise the entry at the top or bottom of the Agenda, before or after a

specific entry on the Agenda, or with a specific name may be deleted. However, because the specific structure of the Agenda is quite different from the structure of sets in OPS-4, special statements are used to modify the Agenda. These statements may be grouped according to the three categories of addition, deletion or modification.

Scheduling Activities

New entries are explicitly created by Schedule statements. A Schedule statement is used when one activity wishes to specify the execution of another activity. The Schedule statement is analagous to both the Create and Cause statements in SIMSCRIPT, and acts as a deferred subroutine call. The Schedule statement gives the generic name of an activity and may optionally define a specific name for the activity. It also defines the parameters of the activity and specifies the position on the Agenda where an entry is placed which defines the call to the activity. The position is specified as being at the top or bottom of the Agenda, or before, after or in place of an existing entry on the Agenda. For example, to explicitly schedule the execution of the activity which generated user-characteristics in the Multics model described in Chapter 3, we could write either

```
Schedule New-User after Finish
```

or

```
Schedule New-User named Sam after Finish
```

In both cases a call to the activity New-User is placed on the Agenda

after the first entry which calls an activity identified as Finish. (No parameters are included since the activity New-User has no parameters.) In the latter case this instance of the activity New-User is to be identified as Sam and a pointer variable named Sam is dynamically created. In the first case the name of the activity is New-User, the same as the generic name. This ability to specifically name each instance of an activity is the method provided in OPS-4 to identify individual activities. This name provides a way of identifying the activity when another activity wishes to refer to it. It also allows the local data bases of activities to be manipulated and filed in sets. The set manipulation statements provide a limited form of list processing capability in OPS-4.

One of the unique features of the Agenda is that each entry on the Agenda contains a special time attribute. If the user desires, he may insert entries on the Agenda at a position determined by a specific value of this trime attribute. For example, to continue the above example, we might have specified

Schedule New-User named Sam at 1225

This would insert an entry on the Agenda containing a call to the activity New User (with the specific name of this activity being Sam) at a position immediately following all previous entries with a time attribute of 1225 or less. This is a very useful scheduling option, and some simulation systems have relied on it as their only method for scheduling activities.[8]

Having available only this single type of scheduling statement restricts these simulation systems to be solely dependent on time as the forcing function which drives the simulation. In many, discrete-event simulations this is not a limitation. However, in many continuous simulation models time is just an incidental variable which is periodically updated. A general purpose simulation system must not be restricted to scheduling based only on time.

Scheduling Conditionally Executed Activities

In addition to scheduling options which specify absolute or relative positions or which use a special attribute to determine the position of entries on the Agenda, there is an important option which is position independent. In OPS-4 it is possible to specify that an activity is to be executed when a specified condition occurs. Although the term is somewhat inaccurate, this type of scheduling is called conditional scheduling. To return to the example again, we could have specified,

Schedule New-User when X = B

meaning, schedule the execution of the activity New-User to occur when the condition X = B is true. Alternately, we could have specified the condition for the execution of New-User as being dependent upon the completion of some specific event or events. This latter feature is just an extension of the 'Wait for Event' statement in PL/1.[32]

These two types of conditional scheduling statements are very powerful and greatly simplify the user's task in structuring simulations in which

the execution order of activities is subject to many constraints. However, since the first type of condition specified is so general, it is impossible to implement it is any efficient manner using software alone. What is really needed is a hardware trap to alert the system to changes in the values of the specified variables.

To guarantee that the activity is executed as soon as the specified condition becomes true, OPS-4 must re-check the validity of all general conditions after the execution of every event in the simulation. This is a time-consuming operation and thus users are wise not to specify general conditional scheduling as an excuse for laziness, but only when absolutely necessary. On the other hand, the conditional statement which specifies a particular event is quite efficiently implemented. It is implemented in a manner similar to the basic method Multics uses for controlling the execution of processes. Thus, in place of specifying the condition A = O, or A = 1, it would be more efficient to define an Event A which is associated with the value of the variable A being either 1 or 0, and specify the completion of Event A as the condition.

The conditional and unconditional scheduling options available in OPS-4 may be combined to allow a user to specify both a condition and either a position on the Agenda or an explicit time as being the determining factors which influence when an activity may be executed. This means that the condition for activities scheduled at a future time do not have to be tested until that time becomes the current time.

Rescheduling Activities

Sometimes it is necessary to reschedule or reposition the entries representing calls to activities already existing on the Agenda. For example, in the Multics model, the scheduling module must be able to reschedule a user to run again if he does not complete his request after the first service. This is done by the Reschedule statement which gives the specific name of an activity. It has all the options of the Schedule statement. This means it is possible to change activities that were conditionally scheduled to being unconditionally scheduled and vice versa. It is also possible to change the parameters of an activity with a Reschedule statement, although this may alternately be done by using the connect feature discussed in Chapter 4. The Reschedule statement may refer to the time attribute of the entry being moved and thus easily reschedule an activity by adding or subtracting a fixed time to the previously scheduled time without actually knowing the previously scheduled time of the activity.

Cancelling Activities

On occassion, it is necessary to cancel the scheduled execution of an activity. This is done by removing its entry from the Agenda. As in the Reschedule statement, activities are identified by giving their specific name rather than their generic name. However, cancelled activities may still be referred to, because their definition block still exists.

Interrupting and Resuming Activities

An activity must be interrupted when a circumstance arises in the simulation which dictates that the activity cannot continue in execution. For example, when the user pushes the quit button he causes the current process in execution to be interrupted. Or when a higher priority activity wishes to use some facility which is already in use, the activity using the facility must be interrupted. Interrupting is different from rescheduling, because at the time an activity is interrupted it is not possible to specify when it may be resumed. Interrupting should also not be confused with cancelling an activity. A cancelled activity disappears from the Agenda, whereas an interrupted activity is just set off to the side. However, it is possible that interrupted activities may never be resumed and thus effectively become equivalent to cancelled activities.

The activity to be interrupted is specified by name and removed from the active part of the Agenda. When the specified activity is later resumed the interval of simulated time during which it has been interrupted is added to the value of the system time attribute and this new value of the system time attribute is used to compute the position of the entry for the activity in the Agenda.

If a Reschedule, Cancel, Interrupt, or Resume statement specifies the name of an activity, for which there is no entry on the Agenda, a flag is set indicating a possible error and the simulation continues with

no modification being made to the Agenda.

Implicit Modification of the Agenda

All of these five methods of modifying the Agenda require that the user explicitly execute them either from the console or by incorporating them in the definitions of activities. Also, all of these statements refer to a specific named activity which is normally an activity other than the activity in which they appear. Thus, these five statements might be termed external scheduling statements. There are corresponding statements in OPS-4 which allow the user to modify the state of the activity in which they occur, and by doing so implicitly modify the entry on the Agenda for the current activity.

Delays and Waits

The simplest form of implicit scheduling statement is the Delay statement. It specifies an interval of time during which the execution of the current activity is suspended. When the interval expires, the activity is automatically resumed. The operation of the Delay statement is equivalent to an unconditional Reschedule statement.

The Wait statement specifies a condition rather than an interval of time as the factor which determines when the current activity is continued. As long as the condition is false, the activity remains interrupted. When the condition is satisfied the activity is resumed. The operation of the Wait statement is equivalent to a Reschedule statement that specifies a condition rather than an interval of time. Both forms of

conditions, the specific named event, and the general variable relationship may be specified in a Wait statement.

It is also possible to combine the Delay and Wait statements so as to specify a specific interval of time and a condition, both of which must be satisfied before the activity is resumed. The possibility of specifying either a specific interval of time, or a condition is discussed later in this Chapter.

Self-Interruption and Self-Cancellation

An activity may want to interrupt itself and not specify how it is to be resumed. This is accomplished by executing the Interrupt statement with no arguments. Unless some external activity resumes the activity it will never be restarted. It is also possible for an activity to cancel itself. There is no explicit statement for this in OPS-4. Activities are automatically cancelled - e.g. their entries on the Agenda are removed - whenever they execute a Return, Exit or End statement. A Continue statement is similar to the Return, Exit and End statement, except that it does not change the Agenda entry for the activity in any way.

States of Activities

An activity may be in only one of six states at any given time during its existence in an OPS-4 model. The names of these states and their definitions are as follows:

1. Active - The activity is currently being executed on a processor.

2. Unconditionally Scheduled - Only the passage of time is necessary before the activity will become active.
3. Conditionally Scheduled - As soon as the specified condition is true the activity will become active.
4. Conditionally Scheduled at a Future Time - Once the indicated time has elapsed and as soon thereafter as the specified condition is true, the activity will become active.
5. Interrupted - The activity can not become active until it is referenced by a Resume statement.
6. Inactive - The activity has not been created, it has been cancelled or has terminated naturally by executing its final 'End' statement. However, it still may have a local data base and be referred to by another activity.

Any activity which is not in one of these six states is unknown to the OPS-4 system. An activity is any of these states may be linked to by another activity. (Recall discussion at end of Chapter 4). The following table shows how the various sequencing statements alter the state of an activity. The current state of any activity may be determined by the special OPS-4 function 'State' which requires the name of the activity as its only argument. If the activity is inactive it returns a zero, otherwise it returns an integer from 1 to 5. If the activity is inactive, but linked to by another activity it returns the value 6.

Sequencing Statement	Original State	New State
External		
Schedule	6	2, 3, or 4
Reschedule	2, 3, 4, 5 or 6	2, 3, or 4
Cancel	2, 3, 4, or 5	6
Interrupt	2, 3, or 4	5
Resume	5	2, 3, or 4
Direct Subroutine Call	any state	1
Internal		
Delay	1	2
Wait	1	3
Delay and Wait	1	4
Interrupt	1	5
Continue	1	state before becoming active
Return, Exit, End	1	6

Table 1. Activity State Transitions

It is interesting to note that it is not directly possible for the external sequencing statements to cause an activity already entered on the Agenda to become active. This guarantees that the Agenda will always be in control. However, the statements, Reschedule Z top; Delay O, are equivalent to a direct call to the activity Z. The direct call causes an activity to be activated at its currently defined re-activation point.

The Agenda Scan

Normally, the Agenda determines the next activity to become active. It does this in a simple manner. Unless the activity releasing control has specified otherwise, the scan of the Agenda to determine the next activity to become active is started at the top of the Agenda. The first entry is

examined. It may be one of the two types corresponding to the three states numbered 2, 3, or 4 described previously. If it is an unconditional entry - e.g. the activity it specifies is unconditionally scheduled (state two) - then the specified activity is immediately made active. If it is a conditional entry - e.g. the activity it specifies is conditionally scheduled (state two or three) - then the condition is tested. If the condition is true, the activity is immediately made active. If the condition is not true this entry is passed by and the next entry is tested.

Because of the presence of conditional entries on the Agenda, several entries may be tested before an eligible activity is found. If the entire Agenda is scanned and no eligible activity is found, an error is reported to the user.

The conditional entries on the Agenda represent only those activities that were scheduled with the general variable condition. Activities scheduled with the specific event conditions are placed in a different position on the Agenda, and are not checked by the Agenda scan.

The Agenda Structure

The full Agenda is really a tree structure. The main branch is the one just described. A separate branch exists for each event that is specified in a specific event condition. For example, if the user specified

Wait for Event A and B

an entry pointing to the main entry for the activity in which this statement

occurred would be placed on the branches for the events A and B. The two entries in the branches A and B would be chained together and a count field would be set to two indicating both events needed to be satisfied. If instead of requiring that both the Events A and B occur before the activity continues the user specified A or B, the entries on the Agenda would be the same, except the count field would be set to 1, indicating that either of the events could cause the activity to be made active. If the user does not explicitly state either 'and' or 'or' the default case always assumed to be 'and'. The count option available in PL/1 may also be used.

For example,

Wait for Event A, B, C, D, E, (3)

specifies that when any three of the events A, B, C, D, or E are complete the activity is to continue. This is implemented by storing the specified count rather than the total number of named events in the count field of the entries in each event branch.

In all three of these cases when one event is completed its corresponding entry is removed from its event branch and the counts in all the entries on the other event branches are decreased by one. When any count reaches zero, all the remaining entries are removed, and the main entry for the specified activity is inserted on the main branch of the Agenda.

The PL/1 statement,

Set-Event (event name)

is used to define the completion of an event. It checks for the existence of the named event branch and makes all the modifications necessary to that branch and any entries on other branches which may link to the entries on the original branch. If all the event conditions for the activity is satisfied the specified activity is merged into the main branch of the Agenda.

All interrupted activities are also kept on a separate branch of the Agenda. When an activity is interrupted it is removed from the main branch and an entry containing the time of interruption and a pointer to the entry for the interrupted activity is placed in the interrupt branch of the Agenda.

The Agenda is thus a list of lists. The first sublist is the main branch, the second the interrupt branch, and the third is a pointer to a hash-coded table which contains pointers to each of the specific event branches. Since the number of sublists is fixed at 3, the pointers to these sublists are stored in a 3 element vector. The first element contains a pointer to the beginning of the main branch. The second element contains a pointer to the beginning of the interrupt branch, and the third contains a pointer to the base of a hash-coded table. Each entry in the hash-coded table is one word long and contains two pointers, one to the name of the event and the other to the event branch. This table is rehashed when it is too long or too short.

The reason for including these event branches and interrupt branch as part of the Agenda, rather than using the Event Table provided by Multics and a separate interrupt list is mostly a matter of centralization. The user should be able to obtain all the information about the states of activities just by examining the Agenda. He should not have to look in three different places.

The Structure of an Activity Definition Block

Each activity that has been scheduled in the OPS-4 system always has a certain amount of identifying data associated with it. This is called its definition block. The specific activity name pointer, discussed in Chapter 4, always points to the definition block for an activity. The definition block of each activity is 12 words long and contains the following information: (The length of each component is indicated parenthetically.)

1. A pointer to the entry for the activity on the Agenda. This field is zero if the activity is inactive. (18 bits)
2. A code which identifies the current state of the activity. (3 bits)
3. A count which indicates the number of activities currently linked to this activity and also indicates Set membership (15 bits)
4. A time attribute which defines the simulated time when the activity first was unconditionally scheduled or when it first

became active if it was conditionally scheduled, called BT. (1 word)

5. A time attribute which defines when the activity last finished execution, called LT. (1 word)
6. A pointer to the segment containing the local data for the activity. (This segment contains the specific name of the activity.) This is an ITS pair. (2 words)
7. A pointer to the entry point in the activity where control will flow when the activity is made active. This is an ITS pair, since it points directly to a segment. (2 words)
8. A pointer to the symbol table for the activity (which contains the generic name of the activity). This is also an ITS pair, since it points directly to a segment. (2 words)
9. A back pointer to the symbol table of the activity which created this entry. This is also an ITS pair, since it points directly to a segment. (2 words)
10. A pointer to the parameters of the activity stored in <free_>. (18 bits)
11. Unused. (18 bits)

The Structure of the Main Entry

Each entry on the main branch is 3 words long and contains the following information:

1. A pointer to the proceeding entry on the main branch. (18 bits)

2. A pointer to the following word on the main branch. (18 bits)
3. The execution time attribute for the activity, abbreviated ET. (1 word)
4. A pointer to the general variable condition, which itself is stored in <free_>, if the activity is scheduled with a general variable condition, or a zero field if the activity is unconditionally scheduled or is inactive. (18 bits)
5. A pointer to the definition block for the activity. (18 bits)

The Structure of an Event Entry

Each entry in an event branch is linked to the other entries on the same branch and to other branches if multiple events have been specified by a specific event condition. An entry contains the following information:

1. A pointer to the preceeding entry for another activity in this branch. (18 bits)
2. A pointer to the following entry for another activity in this branch. (18 bits)
3. A pointer to the entry for this activity in the preceeding event branch or a zero field if only one event was listed. (18 bits)
4. A pointer to the entry for this activity in the following event branch, or a zero field if only one event was listed. (18 bits)
5. A count field. If the count is initially 0 it indicates an 'or' condition between events. If the count is greater than zero it indicates an 'and' condition between events. (18 bits)

6. A pointer to the definition block for the activity. (18 bits)

The total length of each event entry is three words.

The Structure of an Interrupt Entry

An interrupt entry is also 3 words long. It contains the following information:

1. A pointer to the preceeding entry in the interrupt branch. (18 bits)
2. A pointer to the following entry in the interrupt branch. (18 bits)
3. The value of simulated time when the activity was interrupted. (1 word)
4. A pointer to the definition block for the interrupted activity. (18 bits)
5. Unused. (18 bits)

Time Advancement

The Agenda scan mechanism is used to advance simulated time. The rules for advancing time are as follows:

1. If the entry for the activity selected to be activated is unconditional the system time is set to the value of the ET for this entry. If this results in a backward movement of simulated time a flag is set.
2. If the entry for the activity selected to be activated is

conditional and the current value of simulated time exceeds or is equal to the ET of this entry the value of system time is not modified. If the current value of simulated time is less than the ET of this entry then the value of simulated time is set to the ET for this entry.

The user is also free to advance time himself by executing an assignment statement of the form,

Set sys-time = expression

This is useful in simulations where all activities are scheduled relative to each other, rather than scheduled at specific times. It then provides the only means for advancing the clock.

Continuous Models

Many econometric models or the type of continuous feedback models which the DYNAMO system is designed to simulate are of the type just mentioned.[54] These models require only the cyclic solution of a series of difference equations and the increase of system time by a fixed DT in between each cycle. Introducing the concept of permanent entries - entries which can never be deleted - allows such models to be easily handled in OPS-4.

An Agenda for this type of model would be fixed and consist only of permanent entries. Each activity would return to the next entry on the Agenda, by means of the 'Continue next' sequencing statement, rather than to the top of the Agenda, as the normal rule. The last entry on the

Agenda would call an activity which would update system time by the specified DT and then return to the top of the Agenda by executing an unmodified Continue statement. This would start the cycle all over again. Thus, a generalized discrete event simulation system may be used to model continuous systems with ease. The reverse, however, is not true.

Returning Control to the Agenda

Once an activity becomes active it can not be de-activated by the simulation system. Only the eight internal sequencing statements listed in Table 1 may cause an active activity to enter an inactive state. The external sequencing statements do not alter the state of the current activity or interrupt its actual execution.

Recall the definitions of activity and event given in Chapter 1. An activity that is active is executing one event within the activity. Unless the user specifically changes the value of system time, an event occurs at a single instant of simulated time. Each event may define an interaction point of that activity with other activities in the system (i.e. a point where control may be transferred). There are no interaction points within an event, unless the user specifically provides one. For example, inserting a delay of length zero within an event will force an interaction point without causing simulated time to advance. All events in an activity are concluded with one of the 8 internal sequencing statements. The value of simulated time will be automatically increased by

the simulation system between the execution of successive events.

When an activity becomes inactive by virtue of executing one of the internal sequencing statements, control automatically returns to the Agenda system. However, before an activity returns control to the Agenda, it may specify whether the Agenda scan is to be continued from the current entry, or whether the scan is to return to the top of the Agenda. If no specification is made the default case is to return to the top of the Agenda and restart the scan. By normally starting at the top of the Agenda, after the execution of each activity, it is possible to test the general conditional entries and see if any of the conditions have changed. If there are many conditional entries this can be a time consuming operation. The user can skip this rescan of the conditional entries by returning to the next entry on the Agenda.

Modifying the Agenda Entries

All the sequencing statements modify one or more of the components of the Agenda entry to which they refer and change the identification code of the activity pointer. They may also move an entry from one branch of the Agenda to another.

The Schedule statement always creates a new definition block for an activity, and also creates an entry which is inserted in the specified position on the Agenda, either on the main branch, or in a specific event branch. If a specific event is specified, the name or names of the events are looked up in the hash table, added to the table if necessary, and the

appropriate branch entries created in addition to the definition block. When an activity is first inserted on the main branch the three time attributes ET, BT and LT are always initialized with the same value of simulated time. If the schedule option specified 'before,' 'after', or 'in place of' an existing entry on the Agenda the value of simulated time used for initialization of the ET, BT and LT of the new entry is the ET of the specified activity, provided it is scheduled unconditionally. If the specified activity is scheduled conditionally, the current value of simulated time is used instead to initialize the ET, BT and LT of the new entry. If the schedule option is 'top', 'bottom' or just a condition, the time used for initialization of ET, BT and LT is the current value of simulated time. If the schedule option specifies a particular time, that time is used to initialize the ET, BT and LT of the new entry. The schedule statement also fills in all the other components of the definition block and allocates space for the parameters and the condition, if any, in <free_>.

The Reschedule statement may modify the position of the entry on the Agenda. However, only if a Reschedule statement specifically specifies a new time is the ET of the rescheduled entry changed. When it is necessary to move an entry, the forward and backward pointers of the entry and the pointers for its two old neighbors and its two new neighbors are changed appropriately. A Reschedule statement may also change the condition pointer of the entry by adding or removing a condition and

allocating or deallocating the space for the condition in <free_>. If the condition being added or deleted specifies a specific event rather than a relation between variables, the entry may be moved to or from a specific event branch and the main branch. The Reschedule statement may also affect cancelled activities.

The Cancel statement removes the specified entry from the Agenda by changing the pointers of the two adjacent entries. However, the definition block for the activity is not destroyed. The Interrupt statement only moves the entry from its position on the main branch to the interrupt branch. This requires the changing of the pointers of the adjacent entries on the main branch and inserting the entry at the end of the interrupt branch.

The Resume statement is similar to the Reschedule statement, except it refers only to items on the interrupt branch. The specified activity is re-inserted in either the main branch or the specific event branch depending on the options specified. Unless the Resume statement specifically specifies a new execution time for the activity, the interval of time the activity was interrupted is computed by subtracting the time stored in the interrupt branch entry from the current value of simulated time and then added to the old ET of the activity to define the new ET for the activity.

If one activity calls another activity directly the Agenda entry is not modified, since control never passes to the Agenda.

All the interval sequencing statements except for Return, End, and Exit, always do three things:

1. They modify the LT of an entry by reseting it to the current value of system time.
2. They redefine the entry point for the activity.
3. They modify the mode which identifies the state of the activity. They may also deallocate space for the activity's parameters and set the pointer to zero if it was not already zero.

A Delay statement also modifies the ET of the entry and recomputes its position on the Agenda and makes the changes in the 4 pointers to effect its movement. It also sets the condition pointer to zero, and deallocates space in <free_> if it was non-zero previously.

A Wait statement updates the ET for the entry to the current system time, sets the condition pointer for the entry, allocates space for the condition in <free_> and stores the condition there. It then moves the entry either to a new position on the main branch or on an event branch.

A Wait and Delay statement combines these two actions. It computes a new ET by using the specified time, sets the condition pointer, allocates space in <free_> and stores the condition. It also moves the entry to the appropriate position on either the main branch or the event branch.

The internal Interrupt statement operates similarly to the external

interrupt statement and in addition resets the condition pointer to zero and deallocates the space for the condition in <free_>. The Continue statement makes no additional changes to the activity entry. The Return, Exit and End statements change the pointers of the adjacent entries so as to effectively remove the current entry from the Agenda.

The current values of the three time attributes for an entry, ET, BT and LT may each be obtained by using standard functions in OPS-4 described in Chapter 7. But they cannot be modified directly by the user.

Specifying the Parameters of Activities

As we have seen the scheduling of a simulation activity is different than a normal subroutine call. One result of this difference is that there may be a significant delay between the time a simulation activity is scheduled and the time it is actually executed; (there is usually no delay between the time a normal subroutine is called and the time it is executed.) Thus, when scheduling an activity there must be a mechanism for indicating whether the values of the parameters at the time the activity is scheduled or their values at the time the scheduled activity is executed should be delivered to the called activity.

This is related to the call-by-name, call-by-value parameter option offered in ALGOL.[55] If the name of a data object is specified as the parameter of a scheduled activity, then the value of the data object is not obtained until the activity is executed. Alternatively, if the

value of a data object is specified as the parameter of a scheduled activity, then the value at the time the activity is scheduled is used when the activity is executed. If parameters are of the type value, rather than name, then storage must be allocated immediately for the parameters. Thus, if an activity is to be scheduled with the value of an array, the storage for the array is allocated at the time the activity is scheduled, rather than when the activity is executed. Two special functions Current-Value and Later-Value are used in OPS-4 to distinguish between the current value of a parameter at the time it is first mentioned and the later value of a parameter at the time the activity is executed.

Experience with the OPS-3 system has indicated that most parameters of an activity are the value type.[18] Thus, when an activity is scheduled, storage for the parameters is allocated in <free_>, the values of the parameters at the time the activity is scheduled are stored in <free_> and a pointer to these parameters is defined in the entry for that activity on the Agenda. If instead, the name of a parameter is desired, the user must specifically indicate that this is his intent when the activity is scheduled by using the Later-Value function. For example,

Schedule X at 32 with 15 'A' Later-Value (New) Z

would associate the literal parameters 15 and A, the later value of the variable New and the current value of the variable Z with the entry for X on the Agenda.

Specifying the Variables in Conditions

An analogous problem is the interpretation of the variables in a general condition for a conditional entry on the Agenda. Since the condition is specified at the time the activity is scheduled on the Agenda either the current value or names of the variables specified in the condition should be appended to the entry for the activity on the Agenda. Unlike the parameters of an activity, the variables in a conditioned expression are usually specified by name rather than Value. This is easy to understand, since if all the variables in a condition were specified with their current values the truth value of the condition would never change, thus negating the whole purpose of the conditional entry. Therefore, when a user schedules a conditional entry on the Agenda he must specifically indicate those variables in the conditional expression whose current values are to be used rather than their later values. Conditional entries generated implicitly by a Wait statement may contain variables local to the activity. Since local variables of an activity will not change value while the activity is not active, unless the activity is linked to, the current value of any local variables specified in a Wait statement are always used, unless the user directs otherwise by using the Later-Value function.

The Pros and Cons of Alternate Sequencing Schemes

The Agenda mechanism of OPS-4 is not the only possible sequencing system that can be used in a simulation. It is possible to use a

sequencing system that does not maintain an ordered list, and requires a search to select the next activity to become active. This type of sequencing system has some advantages compared to the ordered list of the Agenda for certain situations. Let us list the pros and cons of these two types of sequencing systems.

A. Advantages of the Agenda Mechanism

1. Selecting the next activity to become active requires only a minimum amount of searching of the Agenda.
2. Having an ordered list allows activities to be scheduled relative to activities already on the Agenda - e.g. the 'before', 'after' and 'in place of' options.
3. Displaying the Agenda is a meaningful and very helpful debugging technique, since the order of entries indicates the possible future flow of control in the simulation.
4. It is easy to create a dummy Agenda to test a specific interaction pattern between activities.
5. Since the Agenda is always ordered, there is no ambiguity in determining in what order activities should be executed.
6. A list-structured Agenda requires no physical movement of entries, only the changing of pointers.

B. Disadvantages of the Agenda Mechanism

1. The scheduling or rescheduling of an activity may be time

consuming, since the Agenda may need to be searched to locate the proper new position for the activity entry. (Only the 'before,' 'after,' 'top,' and 'bottom' and 'in place of' options require no search.)

2. The position of an entry on the Agenda is determined at the time an activity is scheduled. Unless an activity is conditionally scheduled, this means that the order of execution of events is determined well before the event is executed and cannot be affected by events which occurred later.

C. Advantages of an unordered list

1. The scheduling of activities never requires any search.
2. Rescheduling activities requires only one search to locate the old entry.
3. It is not necessary to waste time and space for manipulating and storing list pointers.

D. Disadvantages of an unordered list

1. All the eligible activities must be searched, before the next activity to become active can be selected.
2. The problem of resolving conflicts between two or more simultaneously eligible activities must be decided by the scheduling system or through the use of priority schemes.

The fact that OPS-4 activities may be scheduled both conditionally

and unconditionally really allows both sequencing systems to be used advantageously. If a user does not wish to specify any ordering to the execution of activities all activities may be scheduled conditionally. This complicates the task of the scheduling system when it must determine the next eligible activity, but simplifies the user's job. However, if the user specifies an exact or relative order of executing activities the problem of selecting the next eligible activity disappears. The fact that OPS-4 also allows activities to be named, and that the name of an activity always points directly or indirectly to activity entry on the Agenda makes it possible to locate a specific activity without a search.

The OPS-4 Agenda mechanism is a major restructuring of the OPS-3 Agenda although it is conceptually quite similar.[18] The scheduling system of GPSS II and SOL are also similar, but not as flexible since they have limited external sequencing statements.[10,12,13] SIMULA does not allow conditional scheduling in its full generality.[14,15] In SIMSCRIPT the user can only schedule by specifying the time of an event.[8] CSL, GSP, and MILITRAN all rely on the scheduling activities conditionally and maintain no ordered list of scheduled activities.[26-28]

Another Type of Conditional Scheduling

OPS-4 allows activities to be scheduled conditionally, or conditionally with a specific time specification. It might be desirable to be able to specify a third type of scheduling which would be either a condition or a specific time specification. For example, a user might want to state,

Schedule 2Z at 1200 or when P = Q

or

Delay 20 or until P = Q

The difficulty with implementing this type of conditional scheduling is that the entry for the activity really should be on the Agenda in two places. One should be a conditional entry with an ET equal to the current time. The second entry should be an unconditional entry with an ET of the specified time. The two entries should be linked, so that if one is selected as being eligible for execution, the other is automatically deleted.

An alternate technique would be to insert only one entry on the Agenda. It would have an ET equal to the specified time, but it would be inserted in a position determined as if it had an ET equal to the current time. The Agenda scan would have to be complicated so that the entry would be treated as a simple conditional entry if system time was less than the ET. However, if a different entry elected as being eligible was an unconditional one with an ET equal to or greater than the ET of this special entry, then this special entry would be executed as an unconditional entry instead of the selected one.

Both these suggested implementation techniques complicate the Agenda scan mechanism. The desired result can already be accomplished by including system time as one of the variables in a general variable conditional. For example,

Schedule ZZ when sys-time = 1200 or P = Q

can achieve the same result. However, because the Agenda mechanism advances time in unequal and unpredictable intervals care must be exercised when the system variable time is used within a condition of a sequencing statement. Consider the following situation. An unconditional entry for the activity R with an ET of 1205 immediately follows this entry for the activity ZZ just scheduled and the current time is 1150. The entry for activity ZZ is examined and P does not equal Q, nor is time equal to 1200, so the next entry is examined. This entry is unconditional, so system time is advanced to 1205 and the activity R is executed. It is now impossible for the system variable time to equal 1200 and satisfy the conditional relation of the entry for activity ZZ. It is possible that this was the result desired. If, however, the intent was to execute the activity ZZ at a time no later than 1200 a dummy unconditional activity with an ET of 1200 must be scheduled to prevent the system variable time from leaping over 1200. This dummy activity will serve only to advance the system variable time to 1200 so that the conditional entry for activity ZZ will be satisfied.

Priorities

The Agenda does not recognize any priorities among scheduled activities. The order of entries on the Agenda is the only factor which governs the order in which activities are executed. Thus, priority only has meaning when activities are scheduled.

The general scheduling rule in OPS-4 is FIFO. That is, if an entry is being inserted based on its computed ET, then it is normally inserted after all entries with the same or lesser values of ET. This holds for both conditionally and unconditionally scheduled activities.

If a user wants to envoke a LIFO scheduling rule he may do so by attaching the modifying adjective 'first' to the scheduling statements. This will then have the effect of inserting an entry filed by ET following all entries with lesser ET's but in front of all entries with the same ET.

Since the user has so many other scheduling options it does not seem necessary to implement any numerical priority system such as GPSS II allows.[10] Also, the fact that system time is a floating point variable rather than integer allows the user to directly implement a priority system by varying the scheduled time of 'simultaneous' activities by small increments or decrements.

Real-Time Events

Multics allows a user to define 'real-time' events, e.g. events that are to occur at some specified time of day (not simulated time). For example, a user may request that a certain program be run at 12 noon. Alternately, the event can be defined relative to the current time. Thus, a user might specify that a certain event was to occur in five minutes.

These real-time events are implemented in Multics by using the Calendar Clock, and placing entries in a special Calendar Clock Wake-Time Table. This table can be viewed as a real-time Agenda, in which

entries can only be scheduled by explicitly giving their execution time.

The existence of this feature in Multics means that a user could construct simulations that have real time features in them or run a simulation that is synchronized with real time. Alternatively, the user might use these real-time events as special debugging aids or snapshot traces.

This suggests numerous possibilities for using OPS-4 to execute various management gaming models which require interaction with the players at specific points in time. Also, the possibility of incorporating various physical devices as parts of the simulation suggests itself.

Executing Activities Simultaneously

Chapter 2 introduced a new procedure modifier which may be used to specify whether an activity may be executed simultaneously with another activity on a multi-processor computer. In practice, the structure of activities is such that they only are actually executed at specific points in simulation time. Thus, the degree of simultaneity is limited. However, since the Agenda is an ordered list it provides the key to determining whether two or more activities may be executed simultaneously. There are two situations where simultaneous execution is appropriate. If two or more conditionally scheduled activities are simultaneously eligible for execution they may be executed simultaneously. If two or more unconditionally scheduled activities have the same ET they may be scheduled simultaneously.

When an eligible entry is located on the Agenda, the procedure attributes for the activity are checked by consulting the symbol table for the activity. If the simultaneous attribute is specified and the next eligible entry meets the eligibility requirements specified above, then its procedure attribute is also checked. If it specifies simultaneous execution, then the first activity may be started. Before the second activity is started, the test for a third eligible activity is performed. This goes on until there are no more eligible activities, or else the simultaneous attribute is off for one of the activities. At this point the last activity having the simultaneous attribute is started into execution.

A count of the number of simultaneously executing activities is kept, and each time an activity finishes execution the count is decremented by one. Also, when each activity finishes execution a test is performed to see if any conditionally scheduled or new unconditionally scheduled activities with the current ET have become eligible. If so, and if they have the simultaneous procedure attribute they are initiated immediately. Only when the count of the number of simultaneously executing activity reaches zero, can the next unconditionally scheduled activity at a different ET be executed.

It is difficult to predict how much simultaneity may actually exist in a simulation. Experience with OPS-3 (which did not allow simultaneity) indicates that most unconditional entries have different values of ET. Therefore, it seems that the major benefit might come through the sim-

multaneous execution of several conditionally scheduled activities. However, the necessity to lock data bases such as the global data base may effectively limit the amount of true simultaneity in a simulation.

Manipulating the Agenda

Because the Agenda plays such a key role in the OPS-4 simulations it is important that the users be able to manipulate it freely. At any time during the course of a simulation the entire Agenda or any specified portion of it may be displayed for the user to analyze. This has proven to be a very powerful debugging technique in OPS-3. The user may also modify the Agenda directly from the console by executing any of the external or internal sequencing statements.

The Agenda is a separate segment in OPS-4 called 'OPS.Agenda' and the user may replace it at any time with any other segment having the same structure as the Agenda. Thus, he may easily switch between various states in the simulation by replacing the Agenda. This switching can be accomplished dynamically within any activity and allows simulations within simulations to be easily effected. With this flexibility it is important that backward movement of time not be interpreted as an error, but just flagged as an unusual event, since an inner simulation might consist of a forecast which is used by the master simulation.

Explicit vs. Implicit Scheduling

The variety of scheduling statements in OPS-4 exceeds what is available in any of the current simulation languages.[31] It is probable that a

user will not make use of all of them in one model and indeed, different models may not require the use of all of them. They have been provided, however, so that a user may have a choice, and not be forced into using a scheduling method that seems awkward to him.

A user not experienced in constructing simulation models often is confused about how to control the execution of several asynchronous activities so as to make them appear to execute simultaneously. He knows that the different activities communicate and interact with each other, yet he may not program any direct communication or direct transfers of control between activities. Simulation systems which implicitly schedule the flow of control between activities and keep the scheduling mechanism hidden from the user, such as GPSS and SOL often are the most baffling to the novice.[10, 12, 13] Languages such as SIMSCRIPT which require the user to explicitly schedule the flow of control between activities sometimes seem more natural.[8] A programmer is used to thinking only in terms of direct subroutine calls. The concept of an intermediary (the Agenda in OPS-4) which actually makes the subroutine may be difficult to grasp.

Experience with the OPS-3 simulation system, which provides a method for explicitly scheduling simulation activities similar to SIMSCRIPT and also contains an implicit scheduling system similar to GPSS and SOL has shown that when the scheduling system is brought out into the open and easily examined, users have little difficulty understanding how the

scheduling mechanism works.[18] The on-line environment is used to great advantage when the user is allowed continuously to monitor the contents of the Agenda. Seeing the entire Agenda gives a much clearer understanding of the interaction between the activities in a model than just the tracing of the flow control from one activity to the next.

When a user schedules activitis explicitly he has to consider the interaction between the activities in the model and their implications. This is why the user is allowed to be very explicit in dictating the position of the entry for the activity on the Agenda in OPS-4. When the scheduling of activities is done implicitly, the user has almost no control over the scheduling system and may not understand all the implications of the scheduling decision. A user that is forced to consider the cause and effect relationships among activities in the model, will be more likely to detect errors in logic than when these effects are hidden from him and he is forced to rely on statistical measures to judge the model's performance.

There is often a trade - off between the amount of detail to which the user is exposed, or forced to consider, and the understanding he gains of his model's performance. In OPS-4 a user can examine every detail of the scheduling system. On the other hand, he can also ignore these details and rely on aggregated statistical measures. The former is important during the early stages of structuring a model. The latter is useful for comparing the performance of alternate models that are

substantially debugged.

Summary

This chapter has discussed how the sequencing of activities is accomplished in a simulation system, and has shown why it is more complicated than in a normal non-simulation program. The thirteen external and internal sequencing statements available in OPS-4 have been described and the six states of an activity defined. The Agenda mechanism has been explained and the detailed structure of the Agenda has been specified. The difference between calling an activity with the name or value of a parameter has been illustrated and the analagous problem for specifying the interpretation of variables in a conditional scheduling expression has been described. The method for simultaneously executing two or more activities in a n-processor computer system and also for allowing real-time activities has been discussed. This Chapter concludes with a discussion of explicit and implicit scheduling methodology.

Chapter 6

RUNNING AND DEBUGGING A MODEL

Two important aspects of any simulation system are its running efficiency and the facilities it offers for debugging a model. The former is usually overemphasized and the latter not emphasized enough. Unless a simulation model is used as a production tool running efficiency need not receive so much emphasis. The design, debugging and validation phases (validation should really be considered a part of the debugging phase, and should not be postponed until the model is substantially complete) of a complicated simulation model very often consume more computer time than the production phase. Certainly these three phases consume more elapsed time and require more concentrated attention of the model builder in a complicated model.

This Chapter describes the debugging and tracing options available in OPS-4 which are particularly adapted to an on-line environment. It also discusses the features available in OPS-4 which give the user flexible control over the execution of specified portions of a model.

The Model Development Phase

As was stated in Chapter 1, the goal of a simulation experiment is to gain some understanding of a model so that this understanding may be transferred to the real system that the model describes. Many researchers in the field of simulation have pointed out that by the time a model is completed and fully debugged they may have gained such a clear understanding of the model that it is not really necessary to exercise it further. The production phase serves only to confirm their qualitative understanding of the functioning of the model and give specific quantitative figures so that a detailed report may be published. Of course, the latter should not be deprecated, since one of the important aspects of scientific research is to record knowledge and understanding

so that others may thereby profit.

The OPS-4 system places major emphasis on the importance of the model development phase. OPS-4 is designed so that all aspects of the model are out in the open and subject to detailed scrutiny. The user may probe, examine, trace and modify any portion of the model during its development phase. This enhances his ability to comprehend complex models.

The Use of Interpretation

Interpretive techniques are used to provide the flexibility necessary for easily changing the model. This does result in an unavoidable amount of overhead. But the techniques described in Chapter 9 minimize it as much as possible. However, by using interpretive techniques, OPS-4 allows many more types of tracing than what is possible if normal compiled techniques are used.[56]

OPS-4 provides the two modes of simulation - one for debugging, the other for production. Only the debugging mode involves interpretation. It is not necessary to make any changes in a model to go from one mode to the other. All that needs to be done is to compile the procedures that constitute the model. However, the tracing and control statements described in this Chapter are only effective in uncompiled programs.

If a model is to be used as a production tool, it may be necessary to rewrite sections of the model after it has been debugged so that run-

ning efficiency may be improved. Also, the user may desire to eliminate some unimportant aspects of the model to increase running speed. In addition, if a production model does not require any interaction with the user, it may be run as a batch processing job, rather than on a time-sharing system. This is possible, because the basic core of OPS-4 is a collection of procedures that may be called by any PL/1 program.

Two Levels of Trace Specifications

In OPS-4 all tracing options may be specified at two distinct levels. The user may specify that tracing is to be in effect globally or he may specifically tag individual statements that are to be traced. OPS-4 maintains a set of global trace switches, one switch for each of the trace options which may be specified in the OPS-4 system. These switches may be set (reset) by execution of the Trace (No-Trace) statement. However, since the Trace and No-Trace statements may be executed within any procedure the global effect of the trace settings may easily be localized.

In addition to this global injunction of tracing the user may point to a specific statement in any procedure and specifically turn on or off any of the trace options by setting local switches stored with each statement. This is done using the incremental editor described in Chapter 9. These individual statement trace settings take precedence over the global ones. They must be individually reset in the same manner as they were set, except that it is possible to clear all the settings for one

statement at once.

Monitoring the Flow of Control

Since the flow of control from one activity to the next is often influenced by stochastic elements in a simulation model, the ability to monitor this flow of control is an important debugging aid. It is also helpful to be able to monitor the flow of control between procedures within any activity. (Recall that an activity in OPS-4 may include many procedures which call each other with normal subroutine call statements.) Associated with this is the ability to monitor the values of parameters that are being transferred from one activity or procedure to another.

The OPS-3 system provides this monitoring facility for transfers between KOP's and the STRACE command of CTSS provides the same facility for transfers between BSS subroutines.[4,18] The monitoring of all activity and procedure calls in OPS-4 is accomplished by having the call routine check the status of the call trace switch before completing the call. If the call switch is on the name of the activity and/or procedure called is displayed. (Recall that OPS-4 allows activity names to be different from procedure names associated with the activity.) The names of all the parameters, or literal values, will also be displayed if the parameter option was additionally specified.

Monitoring Simulated Time

The movement of the simulated clock in a discrete, event-oriented simulation is also an important feature. Sometimes the model builder is interested in knowing only when the clock changes. In other cases, however, he may be interested in the fact that the clock doesn't change in going from one event to another.

There are two ways to monitor time in OPS-4. The first is to specify the tracing of system time. Whenever the value of system time changes, the old time, new time, and activity causing the time change are displayed. The second is to specify the tracing of the execution time, ET, associated with every activity. Whenever control flows from one activity to the next, the ET of the new activity is displayed. The name of the activity is not displayed, however. It may be obtained in numerous other ways.

The ET of a conditional activity is not equivalent to the execution time of the activity since it indicates when the activity was last scheduled for execution. Therefore, displaying the ET alone might be confusing. To avoid such confusion, all ET's of conditional activities are identified by appending the letter C to the ET when they are displayed.

Monitoring Statement Label References

Statement labels are usually included in programs because they are necessary for indicating the targets points of branching statements. Often, statements labels have mnemonic significance. Therefore, displaying

the names of all statement labels as they are reached during the execution of a program provides any excellent record of the flow of control within a procedure. OPS-4 stores the statements labels alongside the executable statements. (See Chapter 9 for details). Therefore, it is easy to display a statement label when this tracing option is specified.

Since all procedure entry points have statement labels, this trace option would appear to duplicate the procedure name trace previously described. However, there is a subtle difference. Statements labels, can not be traced in compiled procedures. This includes the entry names of the procedures. However, if the compiled procedure is called by an uncompiled program, the call statement - which gives the entry name - can be traced.

Monitoring Activity Calls

Many times it is desirable to trace more than just the name and/or ET of any activity. It may be important to see the parameters of the activity, the name of the entry point within the activity and the condition associated with all conditional entries. This is called an activity trace. It is similar to the procedure call trace. The two important differences are, 1) it applies only to activity calls, not all procedure calls, and, 2) it always displays the parameters, the entry point name and the condition, if appropriate, associated with an activity entry.

The procedure call, statement label, and activity trace provide

three different means of displaying the names of entry points in programs. If all three trace switches were simultaneously turned on it woud be redundant to display the entry name three times. This problem is solved by testing the trace switches in a specific order. The most comprehensive traces are tested first and the specific traces are tested last. Thus, if the activity trace were on, the procedure call and statement label trace switches would not be tested.

Monitoring Modifications to the Agenda

The scheduling, rescheduling, interrupting, resuming and cancelling of activities on the Agenda is also an important fact of interest to the model builder. When any changes are made to the Agenda it is important to know the value of system time, the name of the activity that is modifying the Agenda and the modification itself. Entries are placed on the Agenda by the activity sequencing statements described in Chapter 5. Changes made by all of the sequencing statements, just the external or internal statements, or changes made by specific sequencing statements may all be monitored. For example,

Trace Entries

would display the changes made by any of the sequencing statements, while

Trace Delay Entries

would display the changes made only by Delay statements. In both cases the value of system time and the activity making the modification would be displayed also.

Monitoring the Agenda

Another important tracing feature is to be able to follow the scan of the Agenda when the next eligible entry is being selected. Specifying

Trace Agenda

will display the name of the activity, its condition if any, and the parameters of the activity for every entry that is examined during the scan, including the entry selected as the next eligible one. This is a particularly helpful debugging technique when the sequencing of events is not going as planned.

It is always possible to display the entire Agenda or any portion of it at any time, either from within activity, or from the console, by issuing the instruction

Display Agenda

The display of the entire Agenda helps to provide the model builder with an overall view of the future course of the simulation. The Agenda scan trace just focuses on the current activity. Experience with OPS-3 has shown that both of these are very effective debugging aids.

The Agenda itself may be used to implement a wide variety of debugging techniques. Normally, the Agenda contains only calls to activities. However, it is possible to place any legal statements in the OPS-4 language on the Agenda. DO LOOP's, GO TO's, Input and Output statements, etc. may all be placed on the Agenda. In fact, placing Trace and No-Trace statements on the Agenda is a very effective way of controlling

the duration of tracing in a simulation. These statements may be inserted or deleted by using the Schedule or Cancel Statements.

Monitoring Statement Execution

Within an activity it is possible to monitor every statement that is executed, i.e. before a statement is executed the entire statement is displayed. This is the most thorough method available for monitoring the flow of control in a program. Although it generates voluminous output, it is particularly effective during the early stages of model development, and is also useful when bugs cannot be detected by other more selective methods. Associated with this is the ability to display the results of statement execution. This trace only has meaning for those statements that have a definable result, e.g. Set, Draw, if and Repeat statements.

Monitoring Specific Variables

The monitoring of all references to or changes in specific variables made by any statement in OPS-4 is a particularly effective debugging tool. Special features, which are discussed in Chapter 9, have been designed to make its implementation as efficient as possible. Whenever a variable is referenced, as a parameter of any OPS-4 statement, the name of the variable and its current value is displayed. Variables receiving new values have the previous values displayed before they are destroyed, since the result trace just described may be used to display the new values.

The tracing of array variables must be used judiciously. The entire array will be displayed when the entire array is referenced by the assignment statement. This may be the user's intention. Or he may have only been thinking about monitoring specific elements within an array. It is not possible to specify just a single cell within an array to be traced when the entire array is referenced. However, when only portions of an array are referenced or modified, only the referenced portion is displayed. Identifying subscript specifications are always displayed in addition to the values themselves.

Monitoring Errors and Automatic Definition of Variables

There are three more trace options that are also available. The first,

Trace Error

prints out a full error diagnostic comment whenever an error is detected during the execution of a program. If the error trace is off, only a short cryptic error comment is given. In either case, when an error is detected the simulation stops. The second option,

Trace Flag

prints out a full comment about any unusual occurrence, such as moving the clock backwards, or cancelling an event that isn't entered on the Agenda, etc. that the system detects. If the flag trace is off no comment is printed, but an internal switch, which may be interrogated by the program, is set. In either case the simulation continues. The last

option,

Trace Define

prints a comment specifying the name and type of a new result variable that is automatically defined. If this trace is off, no comment is printed.

Controlling the Specification of Tracing

The power of these tracing facilities lies not just in their availability, but in the flexible manner in which they may be used. (Their initial settings are determined by the OPS-4 system and consultation with the user option feature of Multics.) Limited forms of tracing have been available since the early days of computers. Indeed, the IBM 704 had a special hardware feature to allow tracing of every transfer of control within a program. It was rarely used! The trouble with this feature was that it usually flooded the programer with information. The same complaint can be voiced about the TRACE, UNTRACE feature of GPSS II.[10] The difficulty with both these trace features lies in the detachment of the user from his program. The user operating in a batch processing mode is forced to guess when to start tracing and when to stop. Since the penalty for guessing wrong - wasting a run and having to wait for the next shot at the machine - is very high he usually errs on the side of specifying too large a tracing range and the result is an overwhelming volume of output.

The user at an on-line console is not faced with this dilemma.

First of all, if he guesses wrong he can immediately try again. More important, however, is the fact the guessing can be greatly eliminated. Like a hunter tracking his prey, the user can slowly converge on the problem area by watching the trail of the simulation or by making periodic tests along the way. When he senses that the region of the program containing a bug is imminent he may specify any type of tracing that is needed. If it turns out to be a false alarm, he may quickly turn off the tracing. When the bug is eventually caught he may immediately abort the run and set about the task of correcting the error. Alternately, he can replace the erroneous results with a correct ones and proceed with the simulation, hoping to catch further bugs along the way or see if the remainder of the model is correctly structured.

When multiple user processes are available in Multics, the setting and resetting of the trace options, and the probing and modification of the simulation data base may be carried out asynchronously with the execution of the main simulation process. Also, using similar facilities the user may be able to execute any specified process in parallel with the main simulation. In particular, asynchronous input and output processes will be initiated by the Shell procedure or its counterpart in OPS-4.

User Defined Traces

The standard OPS-4 tracing features just described provide graduated levels of information, and allow the model builder to remain detached and receive occassional progress reports, or to become fully im-

mersed in the model and observe the minute details of its execution. In addition to using the built-in trace features, the user is free to specify his own debugging monitors. This is easily done by writing special procedures which may be called from any point in the program, or by editing the program and inserting any statements that are appropriate. Combining standard PL/1 If statements with Trace statements allows selective control over the effectiveness of the tracing options. The fact that OPS-4 programs do not require compilation after programs have been modified makes the frequent editing of programs to insert or remove trace statements a very convenient technique.

Controlling the Execution of Individual Programs

The important aspects of debugging a model are establishing a specific environment, selectively executing specific portions of the model in the environment and observing the model's performance. The use of the normal input, and assignment statements, as well as the features described in Chapter 4 for manipulating data bases allows the user to easily establish any environment for the model. The settings of the trace options may also be regarded as an extension of the environment. They, in conjunction with the output statements and statistical processing statements described in Chapter 7, provide the monitoring facilities for observing the model's performance.

The facilities for selectively executing programs in OPS-4 fall into two categories:

1. The execution of an entire procedure, or series of procedures.
2. The execution of delimited portions of a procedure.

Accomplishing the first requires no special provisions. In OPS-4 any procedure may be executed directly by the user just by giving its name and parameter values - which may be variables defined in the currently defined symbol table or literal values. When the procedure executes a Return, Exit, or final End statement, control is returned to the user. This feature in itself makes on-line debugging very flexible. The same procedure may be executed many times with different parameter values, and the results of these executions quickly compared.

The execution of specified portions of a procedure requires some additional control information. The two chief items necessary are the specification of the starting and stopping points within the procedure. The starting point may be given explicitly in terms of a statement label or an ordinal statement number (statements are not explicitly numbered in OPS-4 programs.) If no starting point is specified the indicated entry point is always taken as the starting point. A special Execute statement is used to specify this control information.

In addition to the specification of a specific statement label or ordinal statement number as the terminal point within a procedure, a count of the number of statements to be executed may be specified. If

no count is specified, a count of 1 is assumed. This provides a simple way of single stepping through a program. (In OPS-3 the convention adopted was that after the first statement was executed a carriage return meant execute the next statement). When a count is specified as the terminal condition, it is decreased by 1 for every statement executed. Calls to other procedures are counted as only 1 statement execution.

All three of these stopping definitions may be modified by appending a repetition modifier. This causes the execution of the procedure to be terminated after the stopping point has been passed the specified number of times. This feature is particularly useful when program loops are being debugged. The following examples show some of the various ways of using the Execute statement.

Execute XYZ from NEXT 2 times

Execute XYZ to LOOPZ

Execute XYZ from ABLE to line 12 3 times

Execute XYZ from ABLE next 5 lines

Parameter specification is not necessary unless execution starts at an entry point. In that case the parameters are listed right after the entry point name. For example,

Execute ABC 3.6 4.1 to LOOP3 2 times

Only uncompiled OPS-4 programs may be executed in this manner. The Execute statement checks to see if the specified program is defined in

the global symbol table. If it isn't, it defines it.

Running the Entire Simulation

Controlling the execution of an entire simulation model also requires the specification of a starting and a stopping point. The Start statement is used to specify the name of the first procedure to be executed in the simulation. Usually this procedure, initializes the data base and schedules the execution of the initial activities on the Agenda. Normally, this initial procedure has no parameters. If it does they follow the name of the procedure as normal.

The stopping point may be specified four different ways.

1. By specifying the clock time at which the simulation should stop.

2. By specifying either a general variable condition or a specific event condition which will stop the simulation when it becomes true.

3. By specifying the number of activities that are to be executed before the simulation stops.

4. By inserting specific Exit statements at any location within the program.

The first three methods of controlling the duration of the simulation are mutually exclusive - only one may be in effect at a time. The last one also allows the user to specify multiple stopping points, any one of which may terminate the simulation. The Exit statements are inserted in the program by the user with the normal editing command.

The first three control options also automatically cancel themselves, whereas the Exit statements must be specifically deleted. All four of

these methods of controlling the duration of the simulation preserve the status of the simulation so that it may be saved on the disk for a later console session, or so that the simulation may be continued after any probing or inspecting of variables, validation analyses, or changes in the model structure have been made.

The first three of these stopping conditions are specified by appending a phase beginning with the word stop to the Start statement, as illustrated below.

Start Initialize stop at 1600

Start Initialize stop when Event X

Start Initialize stop after 100

The first two options actually result in a special activity being inserted on the Agenda by the simulation system. The pointer to this activity is stored by the system, and updated each time a new stoping specification is given. The third option is implemented by a special count register, which may be modified by the simulation during its execution. In fact, it is possible to respecify any of these three options as individual stop statements from within any activity.

Interrupting a Simulation

In each of these above methods, the termination of the simulation must be planned in advance. There is one additional technique which requires no advance planning. It may be used at any time and at any point within the simulation. By pushing a special "quit" button on the console

the simulation is stopped dead in its tracks no matter what it was doing. The exact machine instruction it was next going to execute is remembered. However, because this is such an unplanned and abrupt interruption it does not often occur at a logical halting spot. For example, the simulation might be right in the middle of a procedure that was recomputing the elements of an array and had only finished 3 of the 5 rows. Or, the simulation might have been interrupted while it was reading or writing information on the disk. Therefore, although it is possible, it is not always meaningful to save on the disk the status of a simulation that has been interrupted in this way and then attempt to restart it from a different point at a latter time. In fact, to resume the simulation, so that it continues with the very next instruction it was about to execute when it was interrupted, requires a slightly different technique than resuming a simulation that was halted by one of the 4 stopping options. However, if the user decides that it is not necessary to resume at the exact point of interruption, but he does wish to continue, any of the normal methods of continuing the simulation specified above may be used.

The normal method for continuing a simulation that has halted is to respecify the starting and stopping points by using the Start statement just described. However, to continue a simulation that has been interrupted from its exact point of interruption the Resume statement is used. If no stopping phrase is appended to the Resume statement the previous stopping conditions will apply. However, the user may specify any stopping condition, in the same manner as with the Start statement, and

these will superceed the previous specification.

One of the most important uses of this interrupt feature is to make unprogrammed inspections or changes in the data base and also set or reset the trace options as the simulation progresses. This interrupt feature may also be used in interrupt a simulation that has gone astray or appears to be in a loop. In addition, it is also a useful way of monitoring the progress of a simulation that is giving no output. Finally, if it is necessary to arbitrarily interrupt a simulation because of some unexpected event before it reaches its normal halting spot, this interrupt feature allows the exact status of the model to be saved and then resumed at a later date. Never is the user forced to wastefully retrace his steps unless he specifically desires to do so.

Recapitulation

With the ability to arbitrarily set initial conditions, specify any starting point and with these various methods of stopping a simulation, the user has a great deal of flexibility in controlling the running of a model. Alternate simulation strategies may be compared by restarting the simulation from a given point with different decision rules of data values. The user may easily construct arbitrary sequences of events on the Agenda and quickly see the results of executing part or all of the sequence. In addition, he may execute a single activity directly from the console and supply any parameter values he wishes. This easily allows him to test activities with many sets of data values and observe

the results immediately. Such freedom of control allows the user quickly to answer the question, "What if ..." with, "Well, that's interesting!" by trying his hypothesized situation when it occurs to him and immediately seeing the result. Being able to quickly and easily restructure a portion of a model is also an important aspect of debugging in OPS-4.

As was mentioned in Chapters 4 and 5, the OPS-4 system has been specifically designed so that no information regarding the present or past status of the simulation is carelessly discarded. The data base of activities that have terminated naturally, or been specifically cancelled may still be probed since they still exist, unless the activities have been rescheduled in the mean time using the same name. By using the segment addressing techniques of the Multics debugging system, any variables in any local data base may be manipulated. This policy of not discarding information is an important feature, since when an error has occurred it allows the user to look backward at the history of the simulation and attempt to see what caused the error.

At any time, either dynamically within an event, or through a specific instruction issued from the console when the simulation is halted, it is possible to save the complete status of the simulation data base on the disk. This is a particularly attractive feature, either in the debugging phase, or during the running of the model when the user wishes to test alternative strategies. It means he can halt the simulation, save

the status of the simulation (i.e. its data base) on the disk, and then let the simulation continue until a bug is found, or an alternative policy is exercised. He can then stop the model and easily restore it to its previous state, including the automatic resetting of the system time, by simply recalling the previous status from the disk. This facility also means that between actual sessions at the console separated by minutes, hours, days or weeks, no work is lost, nothing has to be repeated. Providing the user can remember what he was doing, he can continue from one console session to the next as if there has been no interruption at all.*

The On-Line Environment

The previous discussion brings up a very important point about working with a simulation model at an on-line console. Because of the responsiveness of an interactive simulation system, the tracing facilities, the degrees of control a user may exercise over the running of the model, the ability to interject himself directly in the model and the important fact that he may devote full attention to a model for several hours without interruption; it is possible to become completely immersed in the model - a rapport is established between the researcher and his model that is impossible in a batch processing environment. This is probably the most important contribution an on-line, interactive

*This assumes that both the basic OPS-4 system and Multics itself have not been significantly modified during the interum period.

simulation system can offer a mature researcher who has a difficult problem he is striving to understand through the technique of modelling. It is a new experience for people who have been trying to cope with the sporadic 1 hour, 4 hour, 8 hour, or 24 hour or longer turnaround times in a batch processing environment. Now a researcher can effectively have a digital computer to himself and work for long periods without interruption just as the engineer with an analog computer, and the chemist in his laboratory may be able to devote his full attention to the solution of a problem, or the performance of an experiment. Furthermore, because of the versatility of the computer, the availability of a rich collection of statistical data analysis routines right at hand, if it is necessary to analyze some preliminary results before continuing further this may be done quickly and easily without letting the researcher's major train of thought grow stale.[40]

This means that validation of the model may be done incrementally, just as the model is built incrementally. Each piece may be tested at the time the user is most interested in that particular aspect of the model. The model may be continually re-validated as it grows. There may be continual interation between the data collection, data analysis, model building, and validation phases of a large scale simulation experiment. This removes the possibility of validation being ignored completely, or the more common occurrence of the validation being performed many months after the detailed portions of the model have been locked deep within the structure of a large simulation model.

Summary

This Chapter has described the large variety of tracing options in OPS-4 that are available to the user to help him debug a model. It has also discussed the special control features that are necessary in an on-line environment to allow the user to execute selected portions of the model. These features, together with the facilities for interrupting a model, saving its status on the disk and later resuming its execution allow the user to validate the model in small segments by easily trying numerous alternatives and examining the results. By allowing the on-line user to become fully immersed in the workings of his model, the model development phase takes on a new level of importance.

Chapter 7

COLLECTING STATISTICS

Collecting statistics in a simulation model poses problems for both the user and the simulation system. Often the user does not think about collecting statistics until after he has written major portions of a model. Then, when he starts to consider what measures of performance are necessary, he sometimes finds that substantial changes must be made to either the simulation data base or the model structure. This is an annoying situation. Of course, the user can be blamed for not thoroughly thinking through at the outset the statistical measures necessary to evaluate his model. But, he shouldn't have to. An incremental simulation system should encourage the researcher who has a nascent idea to use the system from the outset to help develop the idea to maturity. The simulation system must be flexible enough so that as the user's ideas unfold he need only make changes in the simulation model that are proportional to changes in his conceptual model. However, it should not do a lot of unnecessary work. If a user substantially revises the method for simulating a particular aspect of the model he may have to substantially change his program. On the other hand, the addition of one of two variables for collecting statistics, and the insertion of a few statements to process them should not require substantial programming effort. This concept applied to OPS-4 dictates that the basic simulation system should not be burdened with the task of automatically processing statistics. However, the system should make the basic data items available to the user and provide convenient ways of manipulating them. Three types of statistics are available in OPS-4.

1. Reports concerning the state of the simulation at a specific point in time.
2. Summary statistics, relating to data accumulated through time.
3. Time series statistics which describe the behavior of the simulation over a period of time.

OPS-4 Versus GPSS, SIMSCRIPT and DYNAMO

The GPSS family of simulation languages automatically provide a plentiful assortment of statistics.[10] Many of them are always of interest, a few of them are almost never of interest and the rest may or may not be important depending on the type of model being simulated. The auto-

matic collection and processing of these statistics is one of the reasons simulations in GPSS run slowly. Unfortunately, if the user desires some statistical measures that have not been provided by GPSS, it is difficult, or impossible to obtain them. (GPSS does allow the user to communicate with subroutines through the HELP block, but this is not recognized as an acceptable solution to the problem.) Also, it is impossible to associate descriptive text with any of the statistics GPSS provides, so that interpretation of the results by anyone other than the programmer is difficult.

The SIMSCRIPT language adopts the opposite approach.[8] No statistics are automatically calculated. Only an ACCUMULATE statement to sum integrals over time and a COMPUTE statement which automatically computes, sums, means, and standard deviations of variables are provided. A sophisticated report generation feature is available, however, so that annotated and readable reports may be produced quite simply. Also, since SIMSCRIPT is an extension of the FORTRAN language, the user may easily calculate any statistical measures he needs. But the user is required to do almost all the work in collecting, storing and processing the raw data for his statistical measures.

The DYNAMO system is particularly noted for its excellent time series plots.[54] Any simulation variables may be automatically plotted at fixed time intervals and fixed ranges are specified for each variable. Seeing a graphical display of values of variables over a period of time

adds a dimension that is not captured by summary statistics.

A middle of the road approach is adopted for OPS-4. It automatically collects and retains certain information which is of general interest. It also aids the user in obtaining measures that depend on system operation which would normally be difficult for him to obtain by himself. Simplified statements for many of the commonly performed operations are provided. A complete algebraic language is available in PL/1, and a simplified report preparation and labelling system is also available through the picture attributes and formatting facilities of PL/1.[32] Procedures discussed briefly in Chapter 4 for tabulating and displaying distributions and collecting and computing queue statistics are also available. The following paragraphs discuss these features in more detail.

Time Related Statistics

In a discrete, event oriented simulation, the user is often interested in statistical measures that have to do with simulated time. For example, such questions as, "What was the average processing time?", "What was the average waiting time?" or "What was the average machine utilization?" seem naturally to occur. Since an event oriented simulation system is driven by time, the simulation system is required to keep track of at least the execution time (ET) of an unconditionally scheduled activity. With a conditionally scheduled activity no event time is available since it is unpredictable. However, with conditionally scheduled events the natural time-related statistic is how long the event waited for its execution

to occur. With only a trivial amount of extra labor, a simulation system can help provide this answer by recording the simulated time at which the conditional activity was scheduled. The delay interval can be computed by subtracting this scheduled time (ET) from the current time when the activity is executed. Only one extra storage cell is required by the simulation system for each conditional entry. This is a small price to pay in exchange for being able to obtain the waiting time so easily. In actuality, since the structure of all entries in OPS-4 - conditional or unconditional - are identical the space is already available.

The amount of simulated time between events represents the consumption of time by some activity and this interval is of importance in computing the utilization of facilities. It is obtained in OPS-4 by allocating a second additional cell of storage per entry on the Agenda. This cell contains the execution time of the last event (LT) that was executed in the activity. Thus, the interval of simulated time between any events may be easily computed by subtracting LT from ET. Finally, in an activity oriented system, where activities consist of sequences of events, it is often desirable to know the total activity duration. This is provided, by a third additional cell per entry on the Agenda, which records the first scheduled execution time for the activity (BT). Each of these three raw datums are available through system defined functions, etime, ltime, btime, respectively. In addition, the waiting time of a conditionally executed activity is obtained by the function wtime; the delay interval between

the current event and the preceeding event in an activity is provided by the function dtime; and the total elapsed time for the activity may be obtained by the function atime. Although, each of these functions usually has no calling argument, implying the current active activity, it is possible to obtain these statistics for any entry on the Agenda by calling the functions with a specific activity name.

Basic Statistical Measures

To make it simple for the user to collect the data to calculate the basic statistical measures of means, variances and standard deviations an Accumulate statement similar to the one in SIMSCRIPT is provided. For example, executing the statement

Accumulate number sum and sumq of X in XNUM XS and XSQ

will automatically increment the variable XNUM by one and the variables XS and XSQ with the value and squared value of the variable or expression X, respectively, thus computing the number of items, sum of X and the sum of X^2. If only one or two of the sums are wanted the other names may be omitted. Since either a simple variable, a function result, or a complicated expression may be accumulated, the user may easily answer the three questions posed earlier in this chapter.

As a further aid, however, a Compute statement is available to compute the mean, variance and standard deviation, from the number, sum and sum of squares of any variable. Thus,

Compute mean MX var VX st-dv SDX from XNUm SX and XSQ

will compute these useful statistics for the variable X accumulated in the Accumulate statement illustrated above. An alternate form of the Compute statement which accepts a vector of raw datums is,

Compute mean MZ and var VZ of Z

This computes the mean and variance of the elements stored in the vector Z starting with Z(1). If only the first n element of Z are valid datums where n is less than the length of the vector, Z(O) should contain n.

Collecting and Displaying Distributions

In many models a complete distribution of data values is required in addition to the mean, variance and standard deviation of the data. For the user to tabulate the distribution himself would require a substantial amount of programming. Three operations are provided in OPS-4 to simplify this task. The first defines a table by giving the name of the table, lower limit, cell interval, and upper limit. For example,

Declare T table 10 5 100

defines T to be a table with discrete cells 10-15, 15-20, 20-25,...,90-95, and 95-100. In addition to these 18 cells, an underflow cell for values below 10 and an overflow cell for values of 100 are also automatically provided.

To record entries in a table the tabulate operator is available.

Tabulate X in T weight W

uses X as an index to determine in which cell of the table T it should add the weight W. If a weight W is not specified, a weight of one is assumed.

If X falls exactly on a cell interval it always goes into the upper cell. Thus, if X were 25, the weight W would be added into the cell 25-30, not the cell 20-25. However, if X were 100 it would go in the last cell 95-100, not in the overflow cell. Finally,

Display T

produces a tabular listing of the table, giving the number of entries, total sum of all the entries, the mean, standard deviation and the individual cell subtotals, cell percentages and cumulative percentages listed by cells.

It is also possible to display just a portion of a table and aggregate cells. For example.

Display T from 30 to 75 cell 15

will list just the five cells 30 and below, 30-45, 45-60, 60-75, and 75 and over, as well as the standard summary statistics. If a new cell width smaller than the original width is specified a flag is set and the defined cell width is used instead.

In addition, distributions may be plotted either as bar graphs or broken-line graphs. Also, either the distribution or its cumulative may be plotted. Subsections of the entire distribution may be specified and the user is allowed to specify aggregation of cells and designate the actual physical space to be devoted to each cell. For example,

Plot T bar from 25 to 80 cell 5/20

specifies that the interval 25 to 80, inclusive, of the table T is to be

plotted as a bar graph and that a cell interval of 5 is to be used but actually occupy 20 units of space on the resulting graph. If the product of the specified physical cell width times the computed number of cells exceeds the available space, a flag is set and the largest allowable physical cell width is used.

The height of the graph is also limited by the amount of available space. The ordinal divisions are computed by subtracting the minimum aggregated cell count, including underflow and overflow cells, and the dividing this into the available space. The vexing problem of adjusting these varied divisions to asthetically pleasing numbers (e.g. .5 rather than .489 or .516) is ignored, since the user also is allowed to specify the ordinal scale by giving a minimum value, interval and maximum if he wishes. The plot routine will visually flag any cells which fall outside these limits.

Alternately,

Plot Cum of T line

will plot the cumulative distribution of the entire table T as a broken line graph. The width of each cell is automatically computed by dividing the available space by the number of cells, including one for underflow and one for overflow. Vertical scaling of cumulative graphs provide no problem. The available space is uniformly scaled (using pleasing intervals) from 0 to 1.

An Example

These routines form a very powerful package. For example, in the Multics model the waiting time distribution for the user requests waiting for processing could be tabulated in a table called WT by specifying,

Tabulate wtime in WT

every time the activity for executing users was activated. Likewise, if the variable Q represented the queue size of activities waiting to be run the state probability distribution QD, describing the queue length could be obtained by executing the statement,

Tabulate Q in QD weight Sys-time-Otime

every time just before the size of Q is incremented or decremented. The variable Otime would also be updated at the same time so that it represented the last time the Q was modified. If only the average length of the queue was desired instead of the complete distribution this could be obtained by executing the statement

Accumulate number and sum of Q*(Sys-time-Otime) in QNUM and QSUM

in place of the Tabulate statement.

Queue Statistics

Because statistics on queues are so often desired in a simulation, special operations for automatically accumulating queue statistics are provided in OPS-4. Every time a new element is inserted in or removed

from a queue (recall a queue is a special data object provided in OPS-4) by the normal set manipulation statements, all the queue statistics are automatically updated. These include the maximum contents, average contents, current contents and total number of entries in the queue. Also, the average, maximum and minimum duration of each element in the queue including the current ones, and the percent of the time that the queue is empty are computed. The current contents, average contents, maximum contents and total number of entries of any queue may be obtained directly by the user at any time by calling the functions sizeq, avgq, maxq and totalq with the name of the queue. The time related statistics for queues are obtained by the functions avg-time, max-time, min-time and zero-time. They may all be obtained simultaneously by using the Display statement. The duration of a particular element in a specific queue may be obtained by the function qtime which requires the queue name and element name as arguments. The detailed structure of queues is described in Chapter.8.

Time Series Plots

A facility similar to that offered in DYNAMO will be available in OPS-4.[54] A user may declare any variable to have an additional attribute called plot. Associated with this attribute is the specification of a minimum and maximum scale value and a plot character (used to identify it from other variables plotted simultaneously). If no minimum or maximum is specified the standard range of zero to 100 is assumed. Also,

if no plot character is designated the first letter of the variable name will be used. OPS-4 will maintain a list of all variables having this plot attribute. During the Agenda scan if an unconditional activity is about to be made active and it results in the change of system time, all the variables with the plot attributes will be plotted. In the normal mode of usage, activation of conditional activities or uncondtional activities having the same ET will not affect plotting.

The rules for plotting may be revised by the user at any time, however. By making calls to the routine Change-plot he may direct that a plot occur before:

1. The activation of every activity
2. The activation of all unconditional activities
3. The activation of all conditional activities
4. The normal mode

Since individual variables may be redeclared dynamically the plot attribute may be selectively removed and reinstated during the simulation and the maximum and minimum ranges modified and the plot character changed.

The normal rule for indicating the time axis is to space one division for each unit of time. Since time changes discretely, the resulting plot may have many apparent plateaus. The time axis is automatically labelled at fixed intervals. Also, if one of the non-standard modes of plotting is in effect and the time axis is not increased, successive plots

will be plotted at the same position, possibly resulting in multiple values from some variables.

The special routine Change-plot, described above, may be called to change any of these three facts; i. e. the number of units of time per division may be modified. (fractional values indicate amplification of the time axis), the labelling interval may be modified, and plots which occur at identical values of system time may be plotted as distinct frames. When multiple processes are available in Multics this plot feature will be implemented as an asynchronous process.

Summary

OPS-4 has followed a middle course with regard to simulation statistics. No statistics are automatically generated and displayed as a by-product of an OPS-4 simulation run. However, many of the important basic datum are automatically provided by OPS-4 and may be obtained by the user through standard OPS-4 functions. This allows the basic simulation to be uncluttered and run relatively fast. However, when it is necessary to collect a distribution, or obtain queue statistics the user does not have to write the detailed programming required. A few statements inserted in the right place in the program will do most of the work for him.

Chapter 8

MEMORY MANAGEMENT TECHNIQUES

List processing techniques provide a method for solving problems in which both the size and number of data objects are unpredictable and constantly changing. This situation is common to simulation. OPS-4 uses a list structured Agenda, and also provides the set and queue data objects which are list structures. A particular problem with list structures is the management of memory. Either memory management is continuous - i.e. a list of available space is constantly updated or else it is sporadic - i.e. garbage collection is performed whenever space is exhausted. This Chapter discusses the policies used in OPS-4 which are especially designed for the Multics environment.

List Processing in Multics

The Multics environment provides a mixed blessing for the implementor of list processing systems. On one hand its paging system allows list structures to be larger than physical memory. This is a significant achievement since almost every major program that uses list processing techniques has been contrained by the amount of directly accessible memory.[57] On the other hand, memory references in list processing programs are most unpredictable. A series of consecutive references to a list structure may each reference a different page. Since only a few pages of a single user's list structured program will be residing in core memory simultaneously, it is probable that an unusual amount of paging activity may be initiated by list processing programs. The memory management policies of OPS-4 try to keep lists ordered in such a way as to reduce the probability of out of page references.

Managing the Agenda

Chapter 5 described the Agenda structure. Recall that is is a tree structure with 3 branches. All the entries on the main branch, interrupt branch and specific event branches are 3 words long. All entries use the double pointer system of SLIP to reference the preceeding and following entries.[52] The specific event branches have a second set of pointers which link them to entries on other branches.

To simplify management of the Agenda, the three branches and the hash-coded table for the conditional event branch are stored in the segment <OPS. Agenda>. This is one of the most frequently referenced referenced segments in the OPS-4 system. A minimum of one reference is made each time an activity is made active. (Here reference is used in the logical sense and may imply many actual memory references.) Because of the possibilities of conditional entries on the Agenda many references to <OPS. Agenda> may be made before an eligible entry is selected. All the external and internal sequencing statements also reference the Agenda. Thus, it is imperative that the Agenda be efficiently organized to limit page faults.

Fortunately, the successive references to the Agenda by the Agenda scan are very predictable. The less frequent references to the Agenda by the sequencing statements are not predictable. Therefore, the Agenda entries will be ordered to coincide with the order in which they are referenced by the Agenda scan. This means that all the entries on the main branch will follow each other consecutively and will be ordered by their ET.

The interrupt branch comes first in <OPS. Agenda>. The specific event entries and their hash table follow the interrupt branch, but precede the main branch. This leaves room at the end of the main branch for new entries. When new space is needed for a main entry, an interrupt, or an event entry it will be obtained from the current physical end of the Agenda. (A special cell always contains the address of the first unused space in <OPS. Agenda>.)

As new entries are inserted on the main branch and entries are moved to and from the specific event and interrupt branches, the physical order of the entries will no longer coincide with the Agenda scan order. Also, some entries may represent inactive activities.

Reordering the Agenda

To restore order to the Agenda, a special reordering process is initiated. It does the following:

1. Traverses the interrupt branch and copies it into the beginning of a new segment <OPS. New-Agenda> in the order that the entries are referenced.
2. Traverses the specific event branches, by going through the hash table, copies them into <OPS. New-Agenda> following the interrupt branch and recomputes the addresses of the branch entries in the hash table. If the hash-coded table needs rehashing, because it is too large or too small, it is done at this time.

3. Traverses the main branch of the Agenda and copies all the active entries into <OPS.New-Agenda> in the order that the entries are referenced.
4. Deletes <OPS.Agenda> and renames <OPS.New-Agenda> to <OPS.Agenda>.

Agenda Reordering as an Asynchronous Process

When multiple user processes are available in Multics the Agenda reordering will be done in parallel with the normal simulation. Certain procedures will have to be established to inhibit modification to the Agenda while it is being copied. The reordering processes will lock the Agenda to prevent any activity from modifying it. This may cause a process to become blocked. When steps 1 through 3 of the reordering are finished the reordering process will change the current activity pointer to <OPS. New-Agenda>. It will then proceed with step 4 and finally unlock the Agenda. Any blocked processes may then proceed.

Deciding When to Reorder the Agenda

It would be foolish to reorder the Agenda continuously because reordering it is time consuming. Also, if the Agenda is small and always fits within one page, it is not necessary to reorder it. In addition, even if the entire Agenda does occupy several pages, if the frequently referenced portion is on one page ther is no need to reorder it. The only reason reordering is necessary is to eliminate unnecessary out-of-page references. The fact that there may be holes in <OPS.Agenda> because

of entries for inactive activities is not a serious concern. It is expected that the need to unscramble the order of the Agenda will always occur well before the space in <OPS. Agenda>is exhausted.

The Agenda scan operation will provide the necessary clue as to when it is time to reorder the Agenda. This is the only operation that is penalized if the Agenda is unordered. All other references to the Agenda have no predictable pattern to them. Therefore, every n^{th} scan of the Agenda, when n is a parameter that is self-adjusting or may be explicitly set and/or modified by the user, the Agenda scan mechanism will check the pointer address to see if they reference different pages. This is done by monitoring bits 11, 12, 13, etc. of each address. If the number of distinct page references divided by the total number of references made to the Agenda exceeds some threshold, which may be set and/or modified by user, and the number of Agenda references made during the Agenda scan was above some minimum number (such as 3 or 4), where this number may also be set and/or modified by the user, the Agenda reordering process will be automatically invoked. The user will also be directly able to invoke the reordering process by giving the statement Reorder-Agenda at any time.

Deleting Activity Definition Blocks

Activity definition blocks are never deleted from <OPS. Activities> when they become inactive. Therefore, it is possible, although not probable, that their sum of the active and inactive definition blocks may

exceed the size of <OPS. Activities>. If this happens the normal rule is to notify the user and ask for advice. In all likelihood the simulation has gone out of control and generated too many activities. However, it is possible that numerous inactive definition blocks might be taking up too much space. If this is the case, the user may direct that all these inactive entries be deleted. (If he wishes to save specific ones this can be done by temporarily rescheduling the specific activities, issuing the deletion order, and then cancelling these activities.) It is also possible for the user to direct that all inactive entries are to be automatically deleted when they become inactive and are not linked to by any other activity. This can be done with safety when the simulation is debugged and references to inactive activities are not contemplated.

The Structure of Sets and Queues

Both sets and queues are structured using the double pointer system of SLIP.[52] Each entry in a set consists of two words. The first word contains the two 18 bit backward and forward pointers which link this entry to the preceeding and following entries in the set. The second word contains an 18 bit pointer to the name of the element or name of another set entered in this set and a special 18 bit code identifying whether this entry represents an individual element or the head of another set. The first word in the head of a set also contains the standard forward and backward pointers. The second word of the set head contains

the 18 bit code identifying it as a set head, and an 18 bit field which contains a count of the number of elements in this set. This set length is not necessary but is provided for the user's convenience. The set length may be obtained by the function Set-size.

Each entry in a queue consists of 3 words. The first word contains the backward and forward pointers. The second word contains an 18 bit pointer to the element in the queue, and an 18 bit zero field. The third word contains the value of simulated time when the entry was placed in the queue. The head of a queue consists of 8 words structured as follows:

1. Forward and backward pointers.
2. A special code identifying it as the head of the queue and a count of the total number of entries which have been, or currently are members of the queue.
3. Two 18 bit fields which contain a count of the current number of entries in the queue and the maximum number of entries which existed in the queue at any one time.
4. The value of simulated time when the queue was created.
5. The value of simulated time when the queue was last modified - i.e. either a new entry was added or an old one deleted.
6. The maximum duration of an entry in the queue, including the current ones.

7. The minimum duration of any entry that is no longer in the queue.
8. The sum of the duration of all entries in the queue, including the present ones.

All these items are necessary to compute the queue statistics described in Chapter 7.

Every time a new set or queue is created as a result of executing one of the declaration statements, the set head or queue head is created. A pointer to this head is stored on a simple 1 word per entry list maintained by the OPS-4 system. This entry contains two pointers. One points to the next element on the list and the other points to the set or queue head. The use of this list is described in the next section.

Managing Sets and Queues

All sets and queues used in OPS-4 are stored in <OPS. lists>. This is done so that OPS-4 itself may exercise complete control over memory management of this segment. Rather than using a free storage list and introducing reference counts into the set and queue heads, a garbage collection scheme is employed when the space in the segment is exhausted. This decision is based on research which shows that if the ratio of memory actually used to memory available is small, the time saved by not returning discarded words to the free storage list is greater than the time spend reorganizing memory.[58] It is expected that this condition will hold in OPS-4 since the length of all sets and queues should

be substantially less than the limit of a segment (2^{18} words).

The garbage collection scheme will copy all the sets and queues into a segment <OPS.New-lists>, delete <OPS.lists> and rename <OPS.New-lists> to <OPS.lists>. It uses the special list of set and queue heads which is stored in <OPS.lists> to access all sets and queues. The second word of both set and queue heads contain an identifying code which the copying system uses to distinguish whether it is copying a set or a queue. Even if sets and queues are empty they are not deleted. Only the elements of sets and queues that are no longer referenced will be deleted during the copying stage. If a set contains other sets as members, these are copied as part of the original set, and these sub-sets are marked as having been moved so that they are not recopied. The special list of set and queue heads is copied after all the sets and queues have been copied. OPS-4 itself needs to remember only the address of this special list and the address of the beginning of ususued memory at the end of <OPS.lists>.

Automatically Initiating Garbage Collection

Normally the garbage collection operation will only be performed when the end of <OPS.> lists has been reached. However, the user may specifically envoke the garbage collection operation any time he wishes by executing the statement Purge-Lists. A third possibility that also exists is to have it envoked automatically. Although space in <OPS.lists> may be far from exhausted, frequent modifications of both sets and queues

may result in them being spread over many pages. References to either the top or bottom of sets and queues will cause no difficulty. However, if sets and queues are being continusously searched, this may result in a considerable amount of overhead. To reduce this overhead, the garbage collection operation may be performed. The question is when should it be initiated by the system?

What is needed is some measure of the efficiency of accessing sets and queues. One measure might be one minus the inverse of the number of consecutive memory references between memory references which require a page reference: i.e.

$$1 - \frac{1}{MRPP}$$

where MRPP stands for memory references per page. This measures only dynamic memory references, and ignores any memory locations occupied by elements which are not actively referred to. In the worst case, where each element referenced was on a different page, this measure would be zero. Assuming a page length of 1024, and a non-cyclic set (two words per element), in the best case this measure would be

$$1 - \frac{2}{1024} = .9990$$

Unfortunately, without a hardware modification or an interpretive system to monitor all memory references, this measure can not be calculated.

If it is assumed that most references to sets and queues require them to be searched then the ratio of the current length of all sets and

queues to the total number of words in <OPS.lists> that have been allocated, provides an indication of the dispersal of these sets and queues through memory and may be used to infer the probability of out of page references. The actual length of each set and queue is stored in the headers and thus by traversing the special list of sets and queues this total length can be determined at any time. Alternately, an additional cell could be monitored by OPS-4 which would always contain this sum. The assumption about searching may not be valid with queues since they tend to be LIFO or FIFO structured. However, sets may be more frequently referenced on the basis of specific element membership than simply a reference to the top or bottom element of a set.

It is also possible to obtain from the Multics accounting system an indirect measure of paging activity per unit of time. The Core Residence Meter will indicate how many word-seconds a user is charged for and the Core-Usage Meter will measure the processor time. Dividing the first by the second gives the amount of memory a user is charged for. Because of the method of charging for pages jointly shared by several users this is not an accurate measure. Also, this includes all the pages in a user's process, not just those in <OPS.lists>.

It is possible, however, that combining this measure of total core space with the previous ratio of allocated to actually used space in <OPS.lists>, that a useful indication of when to automatically envoke garbage collection will be more accurate than either one taken alone. Not until

Multics is running will it be possible to tell. It is also reasonable to provide both of these measures to the user and let him use them as indications of when to perform a garbage collection if it is not done automatically.

Summary

This Chapter has discussed the problem of memory management of list structures in the Multics environment. Specifically, a method for reordering the Agenda to reduce the unnecessary out of page references caused by the scrambled order of the Agenda and a garbage collection scheme for cleaning out discarded set and queue elements in the segment <OPS.lists> are described. A technique for automatically invoking the reordering of <OPS.Agenda> is described. A similar technique for automatically initiating garbage collection in <OPS.lists> is also described.

Chapter 9

INTERPRETATION AND INCREMENTAL COMPILATION

Most of the specific implementation problems and techniques employed in the design of OPS-4 have been discussed in previous chapters when they were pertinent to the topic being covered. However, the methods employed to create and execute the on-line version of PL/1 and its additional features which constitute an OPS-4 program have been passed over. This chapter describes these implementation methods. A combined interpretive, incremental statement by statement compilation approach is used. The use of an inter-line interpreter which executes either compiled statements or interpreted statements allows programs to be edited and immediately executed and also provides full tracing flexibility at only a moderate cost in overhead.

OPS-4 Programs

An OPS-4 simulation program may be created just like any other program in Multics by using the normal editing command of Multics. The program is an ASCII character file having the class name OPS. It may be printed, punched, etc., with the normal Multics commands and even created off-line and loaded onto secondary storage.

The alternate of having a special editing command which is used only to write OPS-4 programs and which performs immediate syntax checking does not seem to be that desirable. Being able to quickly and easily correct errors when they are detected during execution is far more important and is provided by the interpretive execution system described below.

An OPS-4 program obeys most of the conventions of PL/1 and will look very similar to a standard PL/1 program.[32] The major difference in syntax of an OPS-4 program is the requirement to prefix the

word Set, followed by a space to all assignment statements and the ability to write the name of any procedure or OPS-4 program at the beginning of a statement without preceeding it with the word Call. For example, instead of writing the PL/1 statements,

```
A = X + Y
Call Invert (A, B)
```

the OPS-4 statements would be written as,

```
Set A = X + Y
Invert (A, B)
```

Although it may appear to be annoying to have to prefix all assignment statements with the word Set followed by a space, the user soon becomes accustomed to it. The use of the word Set standardizes the languages so that every statement begins with a procedure name. This allows any procedure to be a statement in the language without having to prefix these statements by the word Call.

The additions to the PL/1 language supplied by OPS-4 are the expansion of allowable data types to include sets, queues, and tables, the procedure attributes simultaneous and sequential, the plot and access attributes associated with any variable and the numerous additional statements to manipulate the new data types, schedule events, and collect statistics.

Since any procedure followed by any arbitrary parameter string (recall the discussion in Chapter 2) may be written as a statement in an OPS-4 program, the many standard additional statements in OPS-4 are

implemented just by writing procedures which perform the specified actions. Thus, the OPS-4 language is open-ended and may be easily modified and adapted to suit the tastes of any user.

When a user interpretively executes an OPS-4 program named X, the ASCII file X.OPS is opened and a minimum of four new segments are created or referenced. One segment is created for the execution version of the program. Another segment is created for the symbol table of the program. A third segment created is the standard linkage segment for the program used for all external references. Finally, one or more of the standard Multics data segments may be referenced and some new ones created to hold the local data objects defined in the program. The possibility of the local data base of a procedure being fragmented into more than one segment causes no difficulty. It arises solely as a convenient way of implementing the various data objects and modes of storage allocation which PL/1 allows and was discussed in Chapter 4.

The OPS-4 Symbol Table

The format of all OPS-4 symbol tables will closely resemble the format of the standard PL/1 symbol table so that all the standard symbol table searching routines in the Multics debugging package may be used. The only difference is the addition of new attribute types. In particular, this compatibility implies that the OPS-4 symbol table will be tree-structured and hash coded so that searching for symbols will obey the

scope restrictions of PL/1 and also be very rapid. All the information about the location of variables will always be relative to a fixed base. The techniques adopted by Multics for linking separate symbol tables may be used to link together the symbol tables of separately written procedures independent of whether they were written as OPS-4 programs or compiled programs.

An OPS-4 symbol table is created dynamically as its companion OPS-4 program is executed for the first time. As the PL/1 declaration statements are executed they cause the declared information to be added to the symbol table. If any attributes for identifiers are left unspecified, their default values are entered into the symbol table. Two special attributes called time and trace are always automatically defined for each identifier. The trace attribute is initially set to off. The time attribute is the number of microseconds since 0000 GMT, January 1, 1901, and is provided automatically by the Calendar Clock of the GE 645 hardware. The use of these attributes will be described later in this chapter.

The linkage segment is also created simultaneously with the symbol table and contains entries for all identifiers having the external attribute.

Inferring Data Attributes

OPS-4 uses a scheme for inferring the attributes of data objects that result from a computation. This means that it is necessary to de-

clare only the attributes of the actual "inputs" to a procedure - i.e. the procedure parameters plus any other variables which are used as given in the procedure, but not declared in a superior block.

For example, the statements,

Set A = B*C

New CUSTOMER named JOE

Draw RATE from Normal (MN, STDV)

all result in the automatic definition of the data objects A, JOE and RATE, respectively, which need not be explicitly declared. The attributes of A are implied by the combination of the attributes of B and C and the multiplication operation. The attributes of JOE are identical to the attributes of CUSTOMER, since JOE is a new entity belonging to the class CUSTOMER. The attributes of RATE are derived from the operation of the Draw operator and the type of distribution specified which, in this instance, generates a floating point random sample from the normal population with mean MN and standard deviation STDV. Since the symbol table is always checked to see if all the attributes of the resultant variable are defined before the inferred attributes are defined, it is possible to override the inferred definition by declaring some, or all of the attributes of the resultant variable if the user desires. This scheme for inferring data attributes, has been used successfully in OPS-3 and found to be a very important feature of an on-line programming system. It relieves the programmer of much unnecessary effort, but still

allows him to be specific when necessary.

If the user wishes to write a procedure in which the attributes of the resultant data object influence the operation of the procedure, then the attributes of the resultant data object clearly must be thought of as inputs to the computation as well as outputs. If, when the procedure is executed, the attributes of the resultant data object are not defined, the parameter accessing mechanism will ask that the attributes be defined before it returns to the calling procedure. Note, that all statement labels are also treated as variables, and they are implicitly declared and entered into the symbol table as relative pointers within the execution segment.

The Execution Segment

The special execution segment referred to earlier is also created dynamically as the OPS-4 program is executed for the first time. The sole reason for creating this segment is execution efficiency. An OPS-4 program could be executed completely interpretively directly from the OPS-4 program segment containing the ASCII text as is done in the present OPS-3 system.* Interpretation allows a user to complete flexibility to make any modifications to the model structure at any time and have them immediately take effect. It is also quite simple to program the

*In OPS-3 the equivalent of the OPS-4 program segment is a special file called a KOP, which can only be written and read by the OPS-3 system.[18]

basic inter-statement sequencing for such an interpretive system. However, the execution of programs in this manner is very slow since any statements executed repetitively must be completely re-interpreted each time they are re-executed.

Analysis of the execution of OPS-3 KOP's has shown the inefficiency of the interpretive mode execution to range from almost none, for the event sequencing operations and array manipulation statements, to intolerable amounts, for simple scalar substitution statements such as,

Set X = 3, B = 516

and transfers of control such as,

Go to Next

Therefore, in OPS-4, the techniques of incremental compilation employed in other on-line languages will be used selectively.[59-62] Thus, it is necessary to create the special execution segment to contain the compiled code.

Creating the Execution Segment

When the user executes an uncompiled OPS-4 program either from within any procedure or directly from the console the interpreter checks to see if the execution segment for the program exists. If it does it immediately starts to execute it line by line as described later in this Chapter. If no execution segment exists one is created. The first item, which is entered in a special record at the beginning of the execution segment

is the current time. This uniquely identifies when this segment was created. As each statement in the OPS-4 program is read and executed an eight word entry is created for it in the execution segment containing the following fields:

(The length of each field is indicated parenthetically)

1. Forward and backward pointers to reference the previous and following entries in the execution segment. (1 word)
2. The name of the specified statement is converted to an absolute pointer to the specified segment name/entry name in one of the following ways: (In all cases the result is an ITS pair which occupies two words).
   a. The global symbol table is consulted to determine if the name is known as a standard statement in the OPS-4 language. If it is the entry OPS/name is entered.
   b. If the global symbol table reports that the name is that of an uncompiled OPS-4 program, then the entry OPS/CALL is entered in the record. This is the special call to an OPS-4 program, similar to the CALLK operator in the OPS-3 system. It indicates that the called procedure is to be executed interpretively.
   c. If the global symbol table reports that the name is a user defined procedure, then the entry OPS/USER is entered in the record. This is a special call, similar to OPS/CALL,

that indicates that the user will portray the procedure directly from the console.

d. If no entry is found for the statement name in the global symbol table, it is assumed that this is a compiled procedure defined by the user, or that it is a standard library procedure. In either case the Search Strategy Module is called to locate the segment and determine the entry point.*

e. If the Search Strategy Module can not find the specified segment name/entry name, the user will be asked for guidance. He may then specify the correct segment name/entry name, indicate that he is supplying an OPS-4 program name instead, or indicate that he will portray the execution of the specified procedure himself from the console.

3. A special time attribute which contains the time at which the symbol table was most recently modified. (52 bits)

4. The set of switches indicating what traces have been explicitly turned on for this statement. (10 bits)

5. The set of switches indicating what traces have been explicitly turned off for this statement. (10 bits)

---

*This assumes that user defined procedures named in OPS-4 programs are compiled one per segment. If this is not the case, the user can specify the segment name/entry name directly when he writes the OPS-4 program.

6. A pointer to the text of the statement which is contained in the OPS-4 program segment. (18 bits)
7. A pointer to the statement label definition in the symbol table, or a zero field if there is no label with this statement. (18 bits)
8. A pointer to the executable code if the statement is compiled, or a zero field if the statement is not compiled. (18 bits)
9. A switch indicating whether any of the parameters are defined in the global symbol table. Only global variables having no access qualification, or those that specify the name of this program may be referenced by statements in this program. (1 bit)
10. A count of the number of parameters in the statement, or a zero field if the statement analyzes its own parameter string. (17 bits)
11. A pointer to a variable length string of trace indicators one for each of the parameters of the statement. This field is zero if the parameter count is zero. (18 bits)
12. A pointer to a variable length string of pointers, one for each of the parameters of the statement. The field is zero if the parameter count is zero. (18 bits)

If any of the parameters in the string, other than a result parameter are undefined, the user is alerted and asked to define them before execution continues.

The details of the algorithm used to decide whether or not to compile a statement or execute it interpretively are not clear at this time. They must await final specification of each of the standard statements in the language. These specifications depend in turn on details of Multics which are not yet available. However, the following general guidelines will probably be followed.

1. All simple transfer statements (e.g. Go to's) will be compiled.
2. All Call statements will be compiled.
3. All statements which parse their own parameter strings will not be compiled.
4. Do Loop's will not be compiled, but executed interpretively so that automatic checking for subscript limits can be performed by OPS.
5. Simple Set statements, e.g. those with straight-forward indexing and/or only scalar arithmetic, will be compiled.
6. All standard input and output statements will be compiled as subroutine calls.
7. All non-standard statements not defined in the global symbol table will not be compiled.

The compiled code, parameter trace indicators and parameter pointers are all located in <free_> which is the segment Multics uses to store all variable length strings.

The general rules for pointers to the parameter attributes in the symbol table are clearer:

1. All compiled statements and/or statements with fixed parameter strings will have pointers to the parameter definitions in the symbol table. The code for these compiled statements will include the standard PL/1 dope and specifier information.
2. All statements which parse their own parameter strings and also are not compiled will not have pointers to parameter definitions.
3. Any parameters which are literals will cause special definitions to be created for them. Storage for literals is allocated in the execution segment.

After completing the creation of this entry in the execution segment, the statement is executed and this process repeated for the next statement in the OPS-4 program.

Because of the multiplicity of branches in a complicated program it is possible that some statements appearing in the OPS-4 program will not have been executed, and therefore not appear in the execution segment if the procedure outlined above is followed explicitly. Therefore, the following modification is necessary. When execution of a branch statement causes a reference to a statement label not yet defined in the symbol table and hence "below" the current location in the program, a search is started to locate the referenced label by reading through the rest of the program until it is found. If it is not found an error is reported to the user and execution suspended. The user may correct the error by de-

fining the label, or by changing the name of the referenced label to a known one. In either case the program then continues from the point determined by the referenced statement label. If the program statements that are being passed over are not already in the execution segment, they are read from the OPS-4 segment and translated as described above, but not executed. Begin and End statements are recognized since they cause a new branch to be referenced or created in the tree-structured symbol table. In addition, all statement labels encountered are defined in the symbol table. This is done to guarantee that all undefined statement labels encountered during execution are below the current program pointer. When the referenced statement label is located complete execution of program statements is resumed. This technique assures that all statements to which control may possibly flow during the execution of the program are included in the execution segment. Note, this also assumes that all declarations for each block must occur at the beginning of the block. Declarations intermixed with normal statements will become effective only when control flows through them and they are actually executed.

Detecting the Editing of an OPS-4 Program

If a user edits, or in any manner modifies the OPS-4 program segment, the time-last-modified attribute for the OPS-4 program kept in the user's directory by the file system will be automatically updated. The name of the program will also be redefined as an uncompiled OPS-4 pro-

gram in the global symbol table. When the user then asks that the modified program be executed OPS will note that the internal time attribute of the execution segment does not agree with the directory entry's time attribute for the OPS-4 program segment. It will then delete the execution segment, symbol table segment, linkage segment and any associated data segments and start afresh as if the OPS-4 program had just been initially created. Note - this means that it may be impossible to refer to any previous instances of the activities that have already been created using the former version of the program.

Why is such drastic action as this necessary? Isn't it possible to make the necessary changes in the appropriate segments and proceed? Yes, it is, but it is not simple. Since the editing of the OPS-4 program may be done using the standard Multics editing program or in an unknown variety of other methods there is no way to simultaneously make the corresponding changes in the symbol table, linkage segment, data segments and/or execution segment. Furthermore, the editing program does not mark which statements in the OPS-4 program segment have been modified. Thus, it would be necessary to run a comparison check on the entire OPS-4 segment against the execution segment, linkage segment symbol table segment and data segments. This is equivalent to regenerating these segments.

Incremental Editing of OPS-4 Programs

However, an alternate editing command will be provided by the OPS-4 system which will allow incremental changes to all the necessary segments.

The difference between this command and the normal Multics editing command is that it will be aware of the presence of the other segments beside the OPS-4 program. For example, new or modified declarations will result in the corresponding changes in the symbol table entires and possibly the linkage segment and updating of both the symbol table internal time attribute and the time attributes of the particular entries that were modified. When changes are made in any executable statements in the OPS-4 program the corresponding changes will be made in the records of the execution segments and possibly one or more data segments. Although the user may conceptually think that he is editing the OPS-4 program segment, in actuality, the execution segment is the key segment manipulated by the special editor since it contains pointers to the other segments.

This editor also must be used when the user wishes to set the trace switches attached to each statement. Executing the Trace or No-Trace statements within this editor will cause the specified on or off switches for the statement to be set.

Correcting Execution Errors

During the execution of an OPS-4 program, all errors detected by the system are immediately brought to the attention of the user. The OPS-3 philosophy for error diagnostics is followed. First, a terse comment followed by any meaningful variables and their values are displayed for the user's analysis. If this is not sufficient to explain the nature of

the error to the user, he may ask for clarification of the type of error or more information about where the error occurred and under what circumstances. He may do as much probing as necessary to determine the nature, cause, and extent of the error. To correct the cause of the error may require editing of the OPS-4 program.

If the incremental editor is used, execution may immediately continue after editing is completed. However, if any other editing technique is used it is necessary to start execution of the program over from the starting point. Before or after correcting the statements that caused the error, the user may want to execute several statements to repair the damage caused by the erroneous execution of the program. This may be done by executing the required statements directly from the console. To then return to the point in the program where the error was detected the user merely executes a Resume statement and execution starts at the beginning of the statement which was not previously completed. If the error was detected while control was passing between statements (except by the explicit transfers caused by branch statements) execution continues with the next statement to be executed. If the user wishes to start the program at a different point than where it stopped he may use the Execute statement described in Chapter 6. Of course, the user always has the option of ignoring the error and continuing the execution of the program from any arbitrary point.

Detecting Changes in the Symbol Table

Whenever the symbol table is modified, the time of modification replaces the internal time attribute at the beginning of the symbol table. In addition, the time attribute of each of the identifiers that were modified are updated. Thus, when the execution segment is used to execute the program one simple check is made before each statement is executed. The special time attribute for each statement is checked against the time attribute for the corresponding local symbol table and also the global symbol table, if it is referenced. If they match execution proceeds. If not, the time attribute for each variable in the parameter string is checked. If none of the variables have time attributes which exceed the time attribute for the statement, the statement time attribute is updated to the time attribute of the entire symbol table and execution proceeds. However, if one or more recently modified variables are encountered, then the appropriate changes are made in the statement and the statement's time attribute is then updated to the time attribute of the symbol table and execution proceeds. The advantage of this method is that is is not necessary to check every statement in the program each time a variable definition changes since each statement does its own checking before it is executed. Also, after the initial check of all variables in a statement, repeated executions of the same statement requires only a single equality check. This method also allows dynamic redefinition of variables during execution, a feature not available in the compiled PL/1 program.

Checking Trace Specifications

The inter-statement interpretive mode of execution provides an opportunity to check the trace switches. First, the two trace fields for the statement to be executed are checked. If the "on" field is non-zero it is "ored" to the system-wide trace settings. If the "off" field is non-zero it is complimented and "anded" to the result of the previous union operation. This result is used to specify what traces are to be performed. If both local trace fields are zero only the global settings are used. Conversely, if the global settings are all zero (off) then only the local settings are used. The statement label trace, statement name trace, and complete statement trace are easily checked in this manner.

The specific variable trace is handled differently. The trace attribute in the symbol table for symbol X is set and reset by the trace X, and no-trace X statements. Whenever this attribute is set or reset the time attribute for the identifier and the entire symbol table time attribute is also reset. This will cause every succeeding statement to be alerted about a change in the symbol table. As execution proceeds and the attributes for all parameters of each statement are checked for changes, the change in the trace attribute will be noted. It will be recorded in two ways. The variable trace bit in the "on" field for the statement and the trace bit for the specified parameter in the statement will be each set.

This method ensures that once the flurry of symbol table accessing

activity caused by a change in the symbol table subsides no further references to the symbol table are necessary. The individual statement contains the necessary information. The use of a special trace bit for the entire statement simplifies the testing to determine whether or not any of the parameters in a statement are currently being traced. Only if this trace bit is on is it then necessary to check the trace indicators for each parameter.

Compiling an OPS-4 Program

When the user is satisfied that his program is working satisfactorily he may wish to compile it into the standard format of a directly executable PL/1 program to increase the execution speed. A special OPS-4 to PL/1 translator (similar to the MADKOP translator of OPS-3) is provided for this purpose.[46] It does the following:

1. Strips off the word Set and the following blank from all assignment statements.
2. Prefixes the word Call followed by a blank to all procedure names that are not defined as statements in the PL/1 language.
3. Converts the parameter strings of all procedures which analyze their own parameter strings into a quoted literal character string.
4. Prefixes all statements that start with names of OPS-4 programs, as opposed to compiled procedures, by a Call to a special entry point named OPS/CALL and inserts the parameters

of the OPS-4 program as a quoted literal string. (This entry point performs the switch into the interpretive mode of execution used to execute OPS-4 programs. When the OPS-4 program executes its return statement the system leaves the interpretive mode of execution and returns to the calling procedure.)

5. Prefixes all statements that start with names of user defined programs, i.e. unwritten programs portrayed by the user at the console, by a Call to a special entry point named OPS/USER and inserts the parameters of the user defined program as a quoted literal character string. (This entry point is similar to the previous one, except that the user at the console is called. When he gives his return signal the system returns to the calling procedure.)
6. Generates declarations for all data objects that are undefined. This includes those data objects which OPS-4 defines using its scheme for inferring data object attributes and those variables declared in outer blocks. The translator may ask for help if it cannot find the symbol table defining these global variables. If any identifiers are defined in the global symbol table, their declarations will include the external attribute.
7. Ignores all Trace and No-Trace statements will may still be left in the program.

8. Removes from the global symbol table the definition of this program as as uncompiled OPS-4 program.

After the translation to PL/1 is finished the PL/1 compiler is called to compile the program.

To guarantee a correct translation experience with OPS-3 has shown that it is usually best to execute the compiled program immediately before translating it. This insures that all references to global symbols will be found. The compiled version and the OPS-4 version of a program should then produce identical results. The only noticable side effect should be a reduction in the execution time. Once an OPS-4 program has been compiled the compiled version is always used by the Search Strategy Module in place of the OPS-4 version when it is referenced by another program.

Summary

This chapter has discussed how individual OPS-4 programs may be created, executed and modified. It has also described what segments are created during the execution of an OPS-4 program. The structure of the execution segment is described in detail. The use of a special time attribute to act as an alarm whenever changes are made to the program or symbol table is presented as an efficient way to allow changes to be immediately acknowledged and acted upon, without requiring continuous time-consuming checking. It is mentioned how the scheme used to infer

the attributes of data objects resulting from a computation makes it unnecessary to declare most variables in an OPS-4 program. Also the on-line diagnostic system is described and the implementation of the tracing options is discussed. It is shown how the difficult problem of variable tracing is handled in straight-forward manner by the use of the special alarm feature. Finally, the method for translating OPS-4 into standard PL/1 programs is described.

Chapter 10

GRAPHICAL DISPLAYS IN SIMULATION

The production of graphical output is an important adjunct of most simulations. Many relationships can be grasped more quickly or understood more completely when presented graphically. In an on-line, interactive simulation system graphical output is particularly important. A user on-line does not want to stop his simulation, spend a significant portion of time manually plotting or tabulating some intermediate results, and then continue. An on-line simulation system should do this for him.

This Chapter discusses some of the current display devices and details and their limitations. It describes a new, low cost terminal being designed at Project MAC which could provide the basis of an ideal simulation terminal. The many types of graphical facilities that might be available in OPS-4 if this terminal were used are described.

Economic Considerations

CRT display devices may be divided into two categories - those that produce only textual output such as the GE 760, the Sanders 720, and the IBM 2260 and those that have line drawing capabilities and use a character generation device to display text such as the IBM 2250. The former are quite inexpensive, renting for under $100 a month, but because of their lack of graphical capability, they are not acceptable as a simulation console. They are usually limited to one size of letters and are best described as high speed electronic typewriters.

The latter group of devices would be adequate but their present cost is too high. The IBM 2250 with its associated 2840 control unit rests for over $2,000 a month. In contrast, the typewriter consoles such as the IBM 1050 or 2741 rent for approximately $150 a month, and

the teletype consoles models 33 and 35 rent for around $100 a month.

A New Display Terminal

Research now underway at Project MAC is aimed at producing a low-cost graphical console with the capabilities of the IBM 2250 but costing only approximately $5,000, or on a rental basis only a little more than $100 a month.[63] It is designed around a direct view storage tube so that it is not necessary to regenerate the picture 30 or 40 times a second to keep it from flickering. Thus, it can be remotely connected to a centralized computer by a telephone line with only a 2,000 bit per second data rate. Characters are generated using a 7 by 9 dot matrix and can be displayed at a rate of 200 per second - about 15 times as fast as the IBM 1050. Line drawings can be produced at the rate of 200 inches per second. The maximum number of characters that may be displayed is 4,000 - 50 lines with a maximum of 80 characters per line. This compares to a maximum of 960 characters for the IBM 2260. The size of the CRT is 10 x 12 inches. A complete picture of moderate complexity takes about 10 seconds to generate - the principal limitation being the bandwidth of the phone line, not the display hardware itself.

Advantages of Soft Copy Output

The term soft copy has been used to describe the presentations on CRT devices. The term hard copy generally refers to printed output. Devices producing soft copy have many advantages over typewriter

devices which produce only hard copy. The primary one is their ability to generate line drawings. Typewriter-like devices are naturally limited to crude graphical imitations using x's or other characters. Speed is another advantage. It is indeed frustrating to be forced to display significant amounts of textual information on a typewriter console at a maximum of 15 characters per second. To create graphs on a typewriter is even more frustrating since so much time is spent spacing over blank areas. The ability to modify portions of a display and quickly see the entire corrected version without having to recreate the entire picture is also a significant advantage of graphical display devices.

Disadvantages of Soft Copy Output

One of the present limitations of soft copy devices is their inability to produce hard copy. The majority of the time a complete transcript is not necessary. However, if the user occasionally wishes to look back and review prvious work it can not be done with just a scope output.* More important, however, if the user wishes to have a permanent copy of a particular display, at present his only resort is to take a Polaroid picture. This is unsatisfactory because of its size limitation. There is a real need for a graphical console that would allow the user to obtain within 10 or 20 seconds a hard copy reproduction of a display on demand. Both Xerox and IBM are reported to be working on such

*The display group at MAC has proposed that a small buffer might be used to hold 2 or 3 previous pictures.

device, but no information is presently publicly available. Another limitation of most CRT display devices is their refresh rate - i.e. the time it takes to display a new frame. Even devices like the 2250 require a few seconds to change pictures. This means that any rapidly varying dynamic data can not be effectively presented. A further problem is the bandwidth of the line between the computer and the display and the encoding of data to be sent over the line.

Centralized Reproduction Facilities

At present, the only type of devices that produce hard copy graphical output use film techniques. This, of necessity, requires a more elaborate and expensive mechanism which implies a centralized installation and long processing time. Stromberg Carlson has for 4 or 5 years marketed a microfilm recorder known as the SC-4020. (3M has just recently announced a similar device.) It is driven by a standard IBM magnetic tape unit and produces high quality graphical output on microfilm. Bell Telephone Laboratories have found it to be a very effective device and have used it extensively.[64] Stromberg Carlson has just announced a newer version called the SC-4060.[65] It is actually a self-contained device including a small digital computer. It may be directly connected to a larger computer or also be fed from magnetic tape, teletype, punch cards or punch paper tape. It offers 4 times the resolution of the older SC-4020 (i.e. 4,000 by 4,000 points) and also provides 4 sizes of alphanumerics. The announced rental is

just over $8,000 a month.

A company called Photomechanisms, located on Long Island, New York, also makes hard copy graphical output devices. However, they use rapid film processing techniques, thus hard copy is available almost immediately. Their main market has been the military. One of their products provides an 8 by 11 hardcopy print in 25-30 seconds after exposure. The cost of this device is approximately $30,000. The resolution quality is not as good as the SC-4020, but would be acceptable for most simulation graphical output.

Producing Plots

Display devices have many uses in OPS-4. The production of the plots described in Chapter 7 is one of the most obvious. Particularly during the model building stages, when relationships between variables may not be understood, a graphical representation may be very helpful. The facilities provided to allow a small area of a plot to described and magnified is particularly important. The production of on-line graphical presentations can be considered just a first approximation when the user is also on-line since he may immediately make any adjustments that are appropriate. For example, the problem of determining the separation between distinct points may be adjusted by the user. Also, the setting of minimum and maximum scale values may be done by the user, after viewing an initial plot, so that a few extreme data values do not cause the plot to have a range which extends

considerably beyond the area of interest. Once a satisfactory presentation is produced and appropriate labels are specified the user could request by a suitable statement that a hard copy version be produced on the centralized reproduction facility. The hard copy would not be immediately available if a device such as the SC-4060 were used, but could be picked up several hours later. Alternately, the central facility might mail the hard copy to the user. The on-line display console is used as a working tool to view preliminary results and to allow specially tailored plots to be created.

Text Editing

A CRT display is also a helpful device for editing programs. The ability to point to variables, words, or statements that are to be changed or deleted in a program is more powerful than the 'ED' and 'EDL' context editors that are available to a CTSS user seated at only a typewriter terminal. Also, the ability to delimit phrases, that are to be deleted or moved, by brackets rather than having to type the whole phrase is much easier for the user. The proposed low-cost display will have a cursor that can be positioned to the correct spot by a mechanical control device. Allowing the user to write directly on the CRT or similarly by supplying him with a RAND tablet device for writing does offer some advantages over the user of a typewriter having an augmented character set but introduces many new problems and increases the cost of the display console.*

One particular aspect of editing that is vexing with the ED and EDL editing commands is the problem of interchanging sections of text, e.g. the movement of a sentence or phrase appearing at one position in the text so that it appears as a duplicate, as a replacement for another section of text, or as an insert in another position in the text. This can be done quite easily with a display system by using the cursor to delimit the phrase to be moved or duplicated and then pointing to the new spot where the insertion or replacement should be made. With the present typewriter editing programs, the phase must be retyped in the new position where it is to appear.

The IBM DATATEXT editing system does simplify the problem of moving sections of text from one position in a file to another.[66] However, it does this at the expense of requiring the user to number each unit of text (a unit may be one or more sentences).

The QED editing program just introduced in CTSS allows text movement in a simpler manner. Using its context editing features, a user describes the first line of a section of text to be moved, and then specifies that all the lines up to and including a concluding line (which is specified by context or by giving a count) be written into one of 128 distinct buffers. He may then insert the contents of this buffer or any other at any position in the text. He may also write selected lines of

---

*The Rand tablet alone costs approximately $6,500.

text as separate files, and read these files and insert them at any point in the text. However, QED currently operates only on whole lines, not phrases. Also, pointing to the lines is simpler and less ambiguous than describing a line by context.

Using a display system to edit programs also allows a more global approach to editing. As much text as can be displayed on a scope may be examined at one time. The text may be rolled forward or backward, so that the continuity of the program may be preserved. The present typewriter editing systems are seriously deficient in this respect. If the user is making only single line changes, such as correcting the syntax errors detected by a compiler and specifically mentioned by line number on the diagnostic output, the present editing programs suffice. However, if the user is in the program development stage and is making important structural changes to the model, he must be able to see large sections of the program.

Today, a user in the program development stage requires frequent print-outs of his entire program so he can see the program as a unit. In fact, with only a typewriter for output, one of the most serious problems facing a user who is making numerous structural changes to a large program is to keep track of the current state of the program. It is necessary to see a large section of the program so that the effect of a change made in one part of a program on the other parts of the program may be studied. A scope display should alleviate this problem

considerably and thereby substantially reduce the necessity for frequently obtaining listings of the program.

Data Editing

A researcher who is constructing and testing a simulation model wants to edit his data as well as his model. This is where the use of a cursor is particularly desirable. He should be able to designate areas of curves or areas in a plot that look suspicious and request magnification of the area, or obtain a tabulation of the data values alongside the graphical output using a split screen approach. If the plot represents a function which he is specifying, rather than a plot of some empirical data, he might wish to actually redraw the shape of the curve so as to specify a new function.

Being able to easily generate graphical displays should significantly increase a user's understanding of his data, and thereby shorten the time span required to construct and debug data-based models. A user seated at such a display console will not have to spend considerable effort to perform statistical analyses to gain limited forms of description of the data such as the mean, median, mode, standard deviation, etc. Instead he quickly will be able to see the data and from many perspectives. He may still wish to obtain the popular statistical measures for comparative purposes, but he will not be limited to them.

Dynamic Displays

One of the major advantages of a graphical display device over a typewriter device is the ability to present changing information. This is something that can be done quite easily on a scope since usually only the modified area needs to be recreated whereas it is very inefficient to do this with a printing device since the entire presentation must be recreated. The dynamics of both the program and data can be presented, as a moving picture rather than a series of snap shots so that the user may interact with the simulation as he sees situations developing which require some action.

A simple example is the dynamic time series display of key variables, discussed in Chapter 7, which allows changes in data values to be observed as they occur. Another example would be to display the frequency distribution of the length of the work queue in the Multics model so that the distribution shape, and change in shape could be observed as the simulation progresses. Being able to see changes in distributions over time adds a new insight not available from end-of-run statistics. Also, a display of key variables could be particularly helpful in determining, during the initial stages of a simulation, when a steady state had been reached , so that the collection of statistical measures of performance could be started without introducing any abnormal biases caused by the starting conditions.

Particularly during the debugging stage, the ability to see the dynamic

of a model is most important since the dynamic interaction of the parts of a simulation program may be more complex and varied than a non-simulation program. In addition to being able to observe the relations among variables and the relations between variables and time, the user should be able to observe the dynamic relations of sections of a program.

One possibility would be to display a matrix of boxes, where each box represents an activity. The boxes could be sub-divided to represent the events within an activity. The box for the current activity and the sub-division for the current event could be highlighted or made to flash, if that is possible. In the corner of the screen the simulated time could be displayed. In the corner of each activity the status of the activity, inactive, scheduled unconditionally or scheduled conditionally, interrupted, etc., could be indicated. If an activity was scheduled more than once and in more than one way, individual counts could be displayed next to the activity status. With this type of a display the overall status of the simulation would be apparent, as well as the flow of control from one activity to another. The user could be given the ability to display only selected activities and selected events within activities.

Another possibility is to display a horizontal bar graph. Each bar would represent one activity. Time would be the horizontal axis and the length of the bars would indicate the last active period of the activity. Dotted lines could be used to extend the bars of activities that are scheduled to be reactivated at a known future time. Conditionally scheduled

activities could be indicated by extending the bars for the events with a shaded area which would automatically advance as the clock advanced. The bar representing the currently active activity could be pointed to by the cursor. All the activities or only selected ones specified by the user could be displayed in this way. This presentation might be helpful to illustrate the simultaneity of various activities.

Many of the tracing options discussed in Chapter 6 could be displayed individually or in combinations. For example, displaying the Agenda whenever it changed could be a very powerful debugging technique. The new or modified entry could be indicated by the cursor. Not only would a display presentation take less time to produce then a hard copy printout, but the fact that a cursor can be programmed to point to any item or area of the display makes it much easier for the user to spot any changes.

Implementation Techniques

To present these types of displays in a useful way, a simulation program might have to run considerably slower than its maximum execution rate because of the rapidity of executing events and the time taken to transmit and generate each new display picture, or else a sampling technique would have to be used. In a small stand-alone computer this could be accomplished by increasing the number of repetitions of a display frame before moving on to the next one. The obvious penalty is a proportional increase in computer time. In a time-sharing system, where

a user receives intermittent bursts of computation the solution could be different.

The simulation program would receive short bursts of computation separated by an inactive period which would be specified by the program, so that the display rate could by controlled by increasing or decreasing the length of the inactive period of the program. Each short burst of computation would allow one event to be executed and the display updated. The program would then enter an inactive state for some predetermined fraction of multiple of a second. Since a storage tube is used the display would not fade and would not need continual regeneration. Then the program would execute the next event, return to its inactive state, etc. This could be done quite easily by modifying the Agenda scan mechanism and using the real-time scheduling features available in Multics.

This would be similar to the present use of the SLEEP call to the supervisor in CTSS, which allows a program to specify a variable period of time, during which the program is inactive, and at the end of which the program automatically awakes and resumes computation. The only difference in this situation is the short duration of both the active and inactive periods. An active period might range from only a few micro seconds to several milli seconds and an inactive period from several hundred milliseconds to a few seconds. Whether a scheduling policy would allow an inactive period of only 200-500 milliseconds is

problematic at present. Also, the time to transmit and regenerate the display might be several seconds. Furthermore, the simulation system would have to be allowed a high priority so that it would be placed on or near the top of the ready list when the inactive period expired, so that its next execution phase would not be significantly delayed.

If the minimum sleep period were restricted to 1 second, as in the present CTSS system, this might constitute too low an upper limit on the number of events to be executed per unit of time. However, if such a system as proposed could be implemented, it would offer substantial economies in the amount of computer time required to control the display rate compared to the technique used in a stand-alone computer system.

If the user wished to 'freeze' a particular presentation so that he could study it, this is easily accomplished by using the interrupt feature described in Chapter 6. Although the simulation would be stopped, the display would remain since it would need no regeneration. When the user wished to continue the simulation the dynamic nature of the display would be resumed.

Summary

This Chapter has described the direct view storage tube display console that is being designed at Project MAC. It has discussed both the advantages and disadvantages of soft copy output. It has suggested

that the availability of a centralized facility for producing hard copy reproductions of graphical output is most important. Many possible uses of a graphical display in an on-line, interactive simulation system have been described. The effective use of graphical displays will allow the researcher conducting a simulation experiment on a time-shared computer to have a more complete degree of involvement in a simulation than is possible with just typewriter-like devices. This is one of the main goals of an incremental simulation system.

Chapter 11

SUMMARY

OPS-4 in Retrospect

This thesis has described the potentials for a new simulation system in a time-shared environment. Specifically the Multics environment is considered. A review of some of the current simulation languages shows them all to be unsuitable for use in this environment. A new simulation system called OPS-4 is described. It is based on a subset of the PL/1 language, but extends it considerably, by adding many new statements and the three data types of sets, queues, and tables.

OPS-4 emphasizes flexibility and ease of change. The goal of OPS-4 is to allow a model builder to continuously interact with his model as it is being structured. He is encourages to start using OPS-4 at a very early point and let it help him to evaluate alternate possibilities. OPS-4 allows him to build and test his model incrementally. Special facilities are provided to allow completely unstructured portions of the model to be portrayed directly by the user. More formally structured portions of the models may be written as normal programs and executed interpretively. These programs may be repeatedly modified and executed with no intermediate recompilation. Well defined portions of the model may be compiled procedures and run at full efficiency.

A simulation model may consist of any mixture of these three types of procedures. The construction of a simplified model of segment and page fault handling in Multics illustrates the use of these features.

The techniques used to implement OPS-4 in Multics include: the extension of the standard Shell procedure to allow more general parameter specifications, the addition of the concept of a global data base and its associated symbol table, a system for uniquely naming replications of an individual procedure's local data bases so as to insure that they all remain distinct from each other but yet are addressable, the isolation of the local data bases in separate segments which provides a multi-stack system so that the activities may be interrupted and be returned to at a later time without the control information in the stack being destroyed, and the use of a special ordered list called the Agenda which acts as an intermediary for simulation activities which desire to transfer control among themselves.

OPS-4 provides an extremely broad choice of both explicit and implicit sequencing statements allowing activities to be executed both conditionally and unconditionally and also placed on the Agenda relative to entries already on the Agenda. Statements for rescheduling, cancelling, interrupting and resuming activities are also provided in OPS-4. The Agenda which contains a list of all scheduled, or interrupted activities may be viewed by the user at any time, and the Agenda scan operation may also be monitored by the user. The various states of an

activity are defined and effect of the sequencing statements in changing states is described.

A large variety of tracing options are provided which allow the user to receive only summary information and gross flow of control information or obtain a complete trace of every statement as it is executed and view the value of every variable that is referenced in an OPS-4 program. Control facilities which allow individual procedures, portions of uncompiled procedures or the entire simulation model to be executed from a selected starting point and with specified stopping conditions are available in OPS-4.

These flexible and comprehensive trace and control features are efficiently implemented by a special inter-statement interpreter which executes compiled statements or interpretively executed statements. A special time attribute is used to act as an alarm which alerts the system to any changes in variable definitions, changes in model structure, or changes in the setting of trace options. Immediately after an alarm extensive checking is done to make sure that the changes are noted and acted upon. However, after this flurry of activity subsides the need for continual checking is not necessary and the system runs in a more efficient manner.

Although OPS-4 does not provide any automatic statistics processing, it does provide the user with certain important system-related data. Furthermore, it provides a number of special statements and functions

which make the collection and processing of statistical measures of a model's performance particularly simple. Both dynamic reports of the model's behavior over time and summary statistics are available in OPS-4.

Since OPS-4 operates in a paged memory environment the problems of memory management of list structures is discussed. A scheme for automatically reordering the Agenda to minimize out of page references and a garbage collection scheme for removing inactive entries in sets and queues are described.

OPS-4 Versus OPS-3

OPS-4 has not yet been implemented since the Multics system on which it depends is not completed. However, OPS-3 has been running on CTSS since 1965. It has been used successfully by several dozen students to construct a wide variety of simulation models. However, as pointed out in Chapter 1, it has several serious limitations which regulate it to the role of an experimental rather than a production system.

Many of the features of the OPS-4 are direct adaptations of those found in OPS-3. These are:

1. The generalized parameter accessing mechanism.
2. The inter-line tracing facilities.
3. The Agenda concept.
4. The conditional scheduling facilities.

5. The method for inferring the attributes of data structures resulting from various well defined operations.
6. The immediate error diagnostics.
7. The on-line information system.
8. A limited collection of statistical routines.

Because of the experience with the features in OPS-3 it is expected that there should be little difficulty in implementing them in OPS-4. OPS-4 does, however, include a substantial number of new features that were not tested in OPS-3. These are:

1. The concept of user defined programs.
2. Complete compatibility between all three types of programs in OPS-4.
3. The use of multiple processes to provide asynchronous execution.
4. More extensive scheduling statements which allow the user to schedule on the basis of position.
5. A more efficient interpretive and incremental compiler system.
6. Smooth and simple transfer of control between all three types of programs in OPS-4.
7. Extended data structures, including sets, queues and tables.
8. Set manipulation facilities.
9. More extensive control facilities.
10. More extensive statistics collection and processing statements.

11. An extension of the scheduling statements to include an interrupt facility.
12. A facility for linking activities to one another.
13. Attention to memory management techniques.
14. Integration of graphical monitoring and display facilities.
15. The use of segmentation concepts.
16. A more efficient and expanded Agenda mechanism.

Many of these new features are possible because of the facilities provided in Multics. Contingent on the exact final specifications of Multics, it may be necessary to modify some of these features of OPS-4 presented in this thesis. In particular, the method of allocating data among the various standard Multics segments and special OPS-4 segments may be modified. The method proposed for allocating the instances of the local data bases of activities may be changed. Since the proposed combination interpreter and incremental compiler proposed in Chapter 9 is a new feature in OPS-4 its exact implementation may change with experience. Also, the memory management techniques outlined in Chapter 8 are a new feature and may change as experience with a paged memory system is accumulated.

Conclusion

Taken as a whole, the OPS-4 system offers the model builder a powerful and flexible system for structuring models in a time-shared

environment. Its use of the PL/1 language provides a basic and relatively broad foundation for describing models. However, the choice of PL/1 does limit OPS-4 to a definite style. In particular, the major area for significant improvement of simulation languages such as OPS-4 lies in extending the data addressing and structuring facilities so that more natural ways of handling complicated data structures and relationships may be easily expressed. PL/1 has made significant advances in this direction compared to ALGOL and FORTRAN, but extensions to it are already being proposed.[67]

APPENDIX

STATEMENTS IN OPS-4

(All keywords are in upper case)

1. General Statements (standard syntax of PL/1 with the exception of the first two statements) - discussed in Chapter 1.

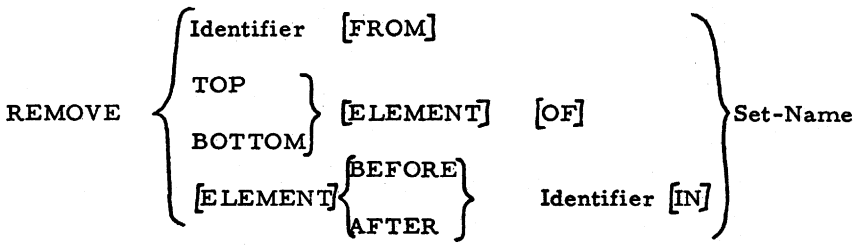
   These statements provide the general algebraic, control and data definition facilities as well as some of the debugging and storage allocation features of OPS-4.

   SET (assignment statement)
   SET-EVENT event-name(s)
   IF
   GO TO
   DO
   BEGIN
   END
   PROCEDURE
   DECLARE
   ENTRY
   RETURN
   ON
   SIGNAL
   REVERT
   ALLOCATE
   FREE

2. Set Manipulation Statements - discussed in Chapter 4.

   These statements provide a limited list processing facility for manipulating named variables and local data bases.

$$\text{ENTER Identifier}\left\{\begin{array}{l} \text{IN} \\ [\text{AT}]\left\{\begin{array}{l}\text{TOP}\\ \text{BOTTOM}\end{array}\right\} \quad [\text{OF}] \\ \left.\begin{array}{l}\text{BEFORE}\\ \text{AFTER}\end{array}\right\} \quad \text{Identifier } [\text{IN}] \end{array}\right\}\text{Set-Name}$$



CLEAR Set-Name

3. Global Symbol Table Manipulation Statements - discussed in Chapter 4.

These statements allow the user to manipulate directly the global symbol table of an OPS-4 model.

NAME	Identifier
USE	Identifier
CREATE	Identifier
CLEAR	Identifier
APPEND	Identifier
VIEW	

4. Global Data Base Manipulation Statements - discussed in Chapter 4.

These statements allow the user to manipulate directly the global data base of an OPS-4 model and to quickly clear, switch or modify the data base.

NAME-DB	Identifier
USE-DB	Identifier
CLEAR-DB	Identifier
APPEND-DB	Identifier

5. Local Data Base Manipulation Statements - discussed in Chapter 4.

These statements allow one activity to access the local data base of another activity.

CONNECT	Activity
DISCONNECT	Activity

6. Input-Output Statements (standard syntax of PL/1) - discussed in Chapter 1.

   These statements provide various degrees of control over input-output operations in OPS-4.

   OPEN
   CLOSE
   DELETE
   FORMAT
   GET
   PUT
   READ
   WRITE
   DISPLAY

7. Data Generation Statements - discussed in Chapters 1, 3 and 4.

   The DRAW statement is used to sample from a specified distribution.

   DRAW Identifier [FROM] { NORMAL [Mean ST-DV] | EXPONENTIAL [Mean] | UNIFORM [Lower-Bound Upper-Bound] | Array [C|D] }

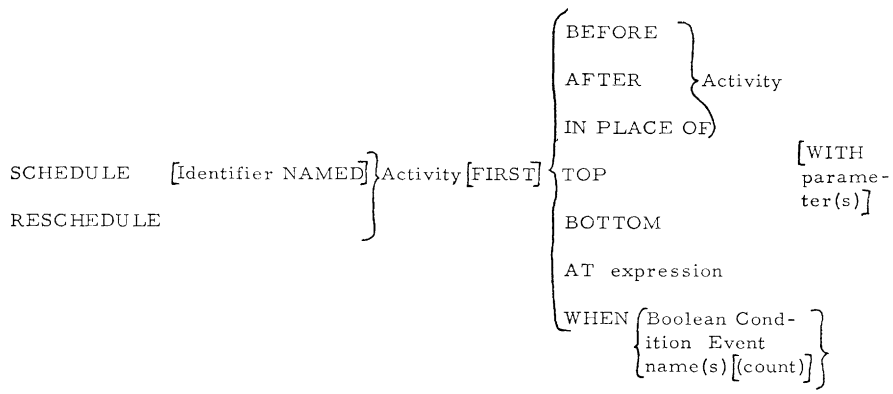
   The NEW statement dynamically creates a new data object and may assign it a name.

   NEW Identifier [NAMED Identifier]

8. Scheduling Statements - discussed in Chapter 5.

   A. External

      These statements allow one activity to affect the activation of another activity.



The last two options 'AT' and 'WHEN' may be combined by appending the WHEN phase after the AT phrase but before the WITH phrase; e.g.

AT expression [AND] WHEN { Boolean Condition / Event Name(s) [(count)] } [WITH parameter(s)]

CANCEL Activity
INTERRUPT Activity
RESUME Activity [FIRST]

B. Internal

These statements affect the state of the current activity.

DELAY expression

WAIT { [UNTIL] Boolean Condition / [FOR] [EVENT] Event Name(s) [(count)] }

DELAY expression [AND] WAIT { [UNTIL] Boolean Condition / [FOR] Event Name(s) [(count)] } [EVENT(S)]

INTERRUPT
CONTINUE [NEXT]
RETURN [NEXT]

9. Trace Statements - discussed in Chapter 6.

The TRACE statements allow the user to monitor the action of the simulation.

```
                 CALLS [AND]        [PARAMS]
                 TIME
TRACE            ET
                 LABELS
                 ACTIVITIES
                 SCHEDULE
                 RESCHEDULE
                 CANCEL
                 INTERRUPT          ENTRIES
                 RESUME
                 DELAY
                 WAIT
                 CONTINUE
                 AGENDA
                 STATEMENTS
                 RESULTS
                 VARIABLE(S)        Identifier (s)
                 ERROR
                 FLAG
                 DEFINE
```

The syntax of the NO-TRACE statement is identical to the syntax of the TRACE statement.

10. Execution Control Statements - discussed in Chapter 6.

These statements allow the user to directly control the execution of the model

```
EXECUTE Procedure [FROM {Label      }]
                        {LINE number}

                [TO {Label      }] [NEXT number LINES] [number
                    {LINE number}                       TIMES]

EXIT

RESUME
```

START Procedure [FROM {Label / LINE number}

[STOP {AT time / AFTER count / WHEN {Boolean Condition / Event-Name(s) [(count)]}}]

STOP {AT time / AFTER count / WHEN {Boolean Condition / Event-Name(s) [(count)]}}

11. Statistics Collection Statements - discussed in Chapter 7.

These statements allow the user to collect and process statistical measures of a model's performance.

ACCUMULATE [NUMBER] [SUM] [AND] [SUMSQ] [OF]
expression [IN] Variable
[Variable] [AND] [Variable]

COMPUTE [MEAN Variable] [VAR Variable] [AND]
[ST-DV variable] {FROM variable [variable] [AND] [variable] / OF vector}

TABULATE Expression [IN] Table-name [Weight] [expression]

DISPLAY Table-name [FROM expression TO expression]
[CELL expression]

PLOT [CUM] [OF] Table name {[BAR] / [LINE]} [FROM expression to expression]
[CELL expression/expression]
[HEIGHT] [MIN expression]
[INTERVAL expression] [MAX expression]

CHANGE-PLOT {All / UNCONDITIONAL / CONDITIONAL / NORMAL} {UNITS expression / LABEL expression / DISTINCT}

12. Memory Management Statements - discussed in Chapter 8.

These statements allow the user to directly manage memory allocation of the AGENDA and sets.

REORDER AGENDA

PURGE { Activities / Lists } [continuously]

13. New Declaration Attributes

A. For procedures - discussed in Chapters 2, 3 and 9

1. SEQUENTIAL
2. SIMULTANEOUS
3. USER

B. For Variables - discussed in Chapters 4, 7 and 9

1. ACCESS Procedure 1 [Procedure 2] ...
2. SET
3. QUEUE
4. TABLE
5. PLOT [MIN expression] [MAX expression] [Plot-character]

14. Special OPS-4 Incremental Editor EDOPS - discussed in Chapter 9.

15. Special OPS-4 Functions

A. To obtain simulated times related to an activity - discussed in Chapters 5 and 7.

BTIME / ETIME / LTIME } [(Activity)]

B. Parameter Specification - discussed in Chapter 5

LATER-VALUE (Variable)
CURRENT-VALUE (Variable)

C. Set Manipulation (adopted from SIMULA) - discussed in Chapter 4.

HEAD (Set-Name)
SUCCESSOR (Identifier)
PREDECESSOR (Identifier)
SAME (Identifier Identifier)
SIMILAR (Identifier Identifier)
FIRST (Set-Name)
LAST (Set-Name)
MEMBER (Identifier Set-Name)
EXIST (Identifier)
EMPTY (Set-Name)

D. Activity References - discussed in Chapter 5.

CURRENT
STATE (Activity-Name)

BIBLIOGRAPHY

Abbreviations used in these references:

ACM	Association of Computing Machinery
AFIPS	American Federation of Information Processing Societies
FJCC	Fall Joint Computer Conference
SJCC	Spring Joint Computer Conference
ORSA	Operations Research Society of America

References, in order cited:

1. Greenberger, M., and Jones, M. M., "On-Line Simulation in the OPS System", Proceedings of the 21st National Conference, ACM, Thompson Book Company, Washington, D.C., 1966, pp. 131-138.

2. Greenberger, M., "A New Methodology for Computer Simulation," Computer Methods in the Analysis of Large-Scale Social Systems, Joint Center for Urban Studies of The Massachusetts Institute of Technology and Harvard University, J. M. Beshers, Ed., Cambridge, Mass., 1965, pp. 147-162.

3. Licklider, J. C. R., Discussion on Simulation Models, Computer Methods in the Analysis of Large-Scale Social Systems, Joint Center for Urban Studies of The Massachusetts Institute of Technology and Harvard University, J. M. Beshers, Ed., Cambridge, Mass., 1965, pp. 163-165.

4. The Compatible Time-Sharing System: A Programmer's Guide, Second Edition, The M. I. T. Computation Center, P. A. Crisman, Ed., The M. I. T. Press, Cambridge, Mass., 1965.

5. Schwartz, J. I., "The SDC Time-Sharing System - Part I and Part II," Datamation, Vol. 10, Nos. 11, 12, (November and December 1964), pp. 28-31, and pp. 51-55.

6. Blunden, G.P., and Krasnow, H.S., "The Process Concept as a Basis for Simulation Modelling", presented at the 28th National Meeting, ORSA, Houston, Texas, November 4-5, 1965.

7. Laski, J.G., "On Time Sctructure in (Monte Carlo) Simulations", Operational Research Quarterly, Vol. 16, No. 3, (September 1965), pp. 329-339.

8. Markowitz, H., Hausner, B., and Karr, H., SIMSCRIPT, A Simulation Programming Language, Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1963.

9. Kiviat, P.J., "Introduction to the SIMSCRIPT II Programming Language", presented at the Symposium on Simulation Techniques and Languages, Brunel College, London, England, May 10-11, 1966.

10. Reference Manual, General Purpose Systems Simulator II, International Business Machines Corporation, White Plaines, New York, 1963.

11. Jones, M. M., "On-Line Version of GPSS II", Project MAC Memorandum MAC-M-140, March 10, 1964.

12. Knuth, D.E., and McNeley, J.L., "SOL - A Symbolic Language for General Purpose Systems Simulation", IEEE Transactions on Electronic Computers, Vol. EC-13, No. 4, (August 1964), pp. 401-408.

13. Knuth, D.E., and McNeley, J.L., "A Formal Definition of SOL", IEEE Transactions on Electronic Computers, Vol. EC-13, No. 4, (August 1964), pp. 409-414.

14. Dahl, O.J., and Nygaard, K., "SIMULA - A Language for Programming and Description of Discrete Event Systems, Introduction and User's Manual", Third Printing, Norwegian Computing Center, Forskningsveien 1 B, Oslo 3, Norway, May 1966.

15. Dahl, O.J., and Nygaard, K., "SIMULA - An ALGOL - Based Simulation Language", Communications of the ACM, Vol. 9, No. 9, (September 1966), pp. 671-678.

16. Dahl, O.J., Myhrhaug, B., and Nygaard, K., "SIMULA Tracing System", (Preliminary Version, August, 1966), Norwegian Computing Center, Forskningsveien 1 B, Blindern, Oslo 3, Norway.

17. Nygaard, K. "Report on the Use of SIMULA up to December 1965", Norwegian Computing Center, Forskningsveien 1 B, Blindern, Oslo 3, Norway.

18. Greenberger, M., Jones, M.M., Morris, J.H., Jr., and Ness, D.N., On-Line Computation and Simulation: The OPS-3 System, The M.I.T. Press, Cambridge, Mass.1965.

19. Corbató, F.J., and Vyssotsky, V.A, "Introduction and Overview of the Multics System", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 185-196.

20. Glaser, E.L., Couleur, J.F., and Oliver, G.A., "System Design of the GE 645 Computer for Time-Sharing Application", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965. pp. 197-202.

21. Vyssotsky, V.A., Corbató, F.J., and Graham, R.M. "Structure of the Multics Supervisor", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 203-212.

22. Daley, R.C. and Neumann, P.G., "A General Purpose File System for Secondary Storage", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 213-229.

23. Ossanna, J.F., Mikus, L.E., and Dunten, S.D., "Communications and Input/Output Switching in a Multiplex Computing System", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 231-241.

24. David, E.E., Jr., and Fano, R.M., "Some Thoughts About the Social Implications of Accessible Computing", AFIPS Conference Proceedings, Vol. 27 (1965 FJCC), Spartan Books,) Washington, D.C., 1965, pp. 243-247.

25. Saltzer, J.H., Traffic Control in a Multiplexed Computer System", Project MAC Technical Report, MAC-TR-30 (Thesis) July, 1966.

26. Control and Simulation Language, Reference Manual, Esso Petroleum Co., Ltd., London, England, March 1963.

27. Tocher, K. D., Handbook of the General Simulation Program, Department of O. R. and Cybernetics, United Steel Companies, Ltd., Sheffield, England.

28. MILITRAN Programming Manual, Prepared for the Office of Naval Research, Navy Department, Washington, D. C., by Systems Research Group, Inc., New York, June 1964.

29. Parente, R. J., "A Language for Dynamic System Description", IBM Advanced Systems Development Division, TR 17-180, 1965.

30. Conway, R. W., et. al, "CLP - The Cornell List Processor," Communications of the ACM, Vol. 8, No. 4 (April 1965), pp. 215-216.

31. Teichrow, D., and Lubin, J. F., "Computer Simulation - Discussion of the Technique and Comparison of Languages", Communications of the ACM, Vol. 9, No. 10, (October 1966), pp. 723-741.

32. IBM Operating System/360 PL/1: Language Specifications, Form C28-6571-3, International Business Machines Corporation, 1966.

33. Tocher, K. D., "Review of Simulation Languages", Operational Research Quarterly, Vol. 16, No. 2, (June 1965). pp. 189-217.

34. Krasnow, H. S., and Merikallio, R. A., "The Past, Present and Future of General Simulation Languages", Management Science, Vol. 11, No. 2, (November 1964), pp. 236-267.

35. McCarthy, J., et. al., LISP 1. 5 Programmer's Manual, The M. I. T. Press, Cambridge, Mass., 1963.

36. Shapiro, R. M., and Zand, L. J., A Description of the Input Language for the Compiler Generator System, CAD-63-1-S D,

37. An Introduction to COMIT Programming, The M. I. T. Press, Cambridge, Mass., 1962.

38. COMIT Programmers' Reference Manual, M. I. T. Press, Cambridge, Mass., 1962.

39. Whitelaw, S., "An Automated Stock Exchange", Unpublished Master's Thesis, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, June 1965.

40. Edwards, D.C., "Development of an On-Line Statistical Package within the OPS System," Unpublished Master's Thesis, Alfred P. Sloan School of Management, Massachusetts Institute of Technology, June 1967.

41. Kwok, B.C., "An Experimental On-Line Data Base System", Unpublished Master's Thesis, Alfred P. Sloan School of Management, Massachusetts Institute of Technology September 1966.

42. Newell, A., et. al., "Report on a General Problem-Solving Program," Proceedings, International Conference on Information Processing, Paris, UNESCO House, 1959.

43. Slagle, J.R., "Heuristic Program that Solves Symbolic Integration Problems in Freshman Calculus, "Journal of the ACM, Vol. 10, No. 4 (October 1963), pp. 507-520.

44. Strachey, C., "System Analysis and Programming," Information, W.H. Freeman and Co., San Francisco, 1966, pp. 56-75.

45. Saltzer, J.H. "CTSS Technical Notes," Project MAC Technical Report, MAC-TR-16, March 1965.

46. Morris, J.H., Jr., "Interpretive System in On-Line Programming," Unpublished Master's Thesis, Alfred P. Sloan School of Management, Massachusetts Institute of Technology. January 1966.

47. The Michigan Algorithm Decoder, University of Michigan, November 1963. (Rev. ed., March 1965.)

48. Scherr, A.L., "An Analysis of Time-Shared Computer Systems," Project MAC Technical Report, MAC-TR-18 (Thesis), June 1965.

49. Goodroe, J.R., and Leonard, G.F., "An Environment for an Operating System," Proceedings of the 17th National Conference, ACM, 1964, pp. E2.3-1 to E2.3-11.

50. _____, "More on Extensible Machines," Communications of the ACM, Vol. 9, No. 3, (March 1966), pp. 183-188.

51. Weizenbaum, J., "ELIZA - A Computer Program for the Study of Natural Language Communication Between Man and Machine," Communications of the ACM, Vol. 9, No. 1 (January 1966), pp. 36-45.

52. Weizenbaum, J., "Symmetric List Processor," Communications of the ACM, Vol. 6, No. 9 (September 1963), pp. 524-544.

53. Operating System/360 FORTRAN IV Language, International Business Machines, Form C28-6515-4, 1966.

54. Pugh, A. L., III, Dynamo User's Manual, The M.I.T. Press, Cambridge, Mass., 1961.

55. Naur, P., et. al., "Revised Report on the Algorithmic Language ALGOL 60," Communications of the ACM, Vol. 6, No. 1 (January 1963), pp. 1-17.

56. Evans, T.G., and Darley, D.L., "On-Line Debugging Techniques: A Survey," AFIPS Conference Proceedings, Vol. 29 (FJCC 1966,) pp. 37-50.

57. Feigenbaum, E.A., and Feldman, J., Computers and Thought, McGraw-Hill, 1963.

58. Collins, G.E., "PM, A System for Polynominal Manipulation," Communications of the ACM, Vol. 9, No. 8 (August 1966) pp. 578-579.

59. Morrissey, J.H.,"The QUICKTRAN System," Datamation, Vol. 11, No. 2, (February 1965), pp. 42-46.

60. Shaw, J.C., "JOSS: A Designer's view of an Experimental On-Line Computing System," AFIPS Conference Proceedings, Vol. 26. (FJCC 1965), pp. 455-464.

61. Lock, K., "Structuring Programs for Multiprogram Time-Sharing On-Line Applications," AFIPS Conference Proceedings, Vol. 27, (FJCC 1965), pp. 457-472.

62. Ryan, J.L. et. al., "A Conversation System for Incremental Compilation and Execution in a Time-Sharing Environment," AFIPS Conference Proceedings, Vol. 29, (FJCC 1966), pp. 1-21.

63. Stotz, R.H., Gronemann, U., Ward, J.E., "Specifications for a Dataphone-Driven Remote Display Console for Project MAC," Memorandum MAC-M-243, June 24, 1965.

64. Ninke, W.H., "Future Graphical Input/Output Equipment at Bell Telephone Laboratories," Project MAC Seminar, January 10, 1967.

65. Datamation, Vol. 12, No. 11, (November 1966) pp. 66-67.

66. DATATEXT: Terminal Reference Manual, For Y20-0037, International Business Machines, 1966.

67. Dodd, G.E., "APL-A Language for Associative Data Handling in PL/1", AFIPS Conference Proceedings, Vol. 29, (FJCC 1966), pp. 677-684.

BIOGRAPHICAL NOTE

Malcolm Murray Jones was born in Scarsdale, New York on October 24, 1935. He graduated, second in his class, from the Rivers Country Day School, Chestnut Hill, Mass. in June 1953. He entered the Massachusetts Institute of Technology in September 1953, where he studied Mechanical Engineering and Economics, receiving the degree of S. B. in June 1957, and also a commission as 2nd Lieutenant in the U. S. Air Force. He received the degree of S. M. in Economics from the Massachusetts Institute of Technology in June 1958. During the academic year 1957-58, and through November 1958, he worked as a research assistant in the newly formed M. I. T. Computation Center. This marked the beginning of his interest in digital computers.

From December 1958, through September 1962, Mr. Jones served as an electronics engineer in the U. S. Air Force, stationed at the National Security Agency, Fort Meade, Maryland. At NSA he worked on numerous computer oriented projects, including the design of an advanced data processing language for a large scale computer system. He received a letter of commendation from the Director of NSA for his work on the project. During his tour of duty in the Air Force, he took special courses in electrical engineering at the University of Maryland.

Mr. Jones returned to M. I. T. in September 1962, as a special student in the Department of Electrical Engineering. The following

summer he became one of the first members of the Project MAC staff, which was formed in July 1963, working as a research assistant. In September 1963, Mr. Jones enrolled in the Sloan School of Management as the first interdepartmental doctoral candidate in Computer Sciences. He also continued to work as a research assistant for Project MAC and the M.I.T. Computation Center until June 1965. At that time he was appointed an Instructor in the Sloan School of Management. He teaches courses in Information Systems technology. He has accepted a position as an Assistant Professor in the Sloan School of Management at M.I.T., beginning February 1967. Since April 1, 1965, he has also been a guest lecturer on Data Processing for the Katharine Gibbs School, Boston, Ma. His research interests are concentrated in the area of information systems design and digital computer simulation languages, the latter being the subject of his doctoral dissertation.

Mr. Jones married the former Lesly Sheldon Weaver of Evanston, Illinois, in June 1966. He is a consultant to the National Security Agency and holds the rank of Captain in the Air Force Reserve. He is a member of the Association for Computing Machinery, Institute of Electrical and Electronic Engineers, American Association for the Advancement of Science, and Tau Beta Pi.

Publications

"On-Line Simulation in the OPS System," Proceedings of the 21st National Conference, ACM, The Thompson Book Co., Washington, D.C., 1966 (with M. Greenberger).

On-Line Computation and Simulation: The OPS-3 System, M.I.T. Press, Cambridge, Mass., 1965 (with M. Greenberger, J.H. Morris, Jr., and D.N. Ness).

"On-Line Version of GPSS II," Project Mac Memorandum, MAC-M-140, March 10, 1964.

CS-TR Scanning Project
Document Control Form

Date : 12/11/95

Report # LCS-TR-48

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:_____

Document Information

Number of pages: 250 (257-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☒ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:_____

Check each if included with document:
☒ DOD Form ☒ Funding Agent Form ☒ Cover Page
☐ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other: _____

Page Data:

Blank Pages (by page number):_____

Photographs/Tonal Material (by page number):_____

Other (note description/page number):

Description : Page Number:
IMAGEMAP: (1-250) UNNUMBERED TITLE, ABSTRACT, ACK PAGES, iv-viii, 1-242
(251-257) SCANCONTROL, COVER, FUNDING AGENT, DOD, TRGT'S (3)

Scanning Agent Signoff:
Date Received: 12/11/95 Date Scanned: 12/19/95 Date Returned: 12/28/95

Scanning Agent Signature: Michael W. Cobb

UNCLASSIFIED

Security Classification

DOCUMENT CONTROL DATA - R&D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY *(Corporate author)*
Massachusetts Institute of Technology
Project MAC

2a. REPORT SECURITY CLASSIFICATION
UNCLASSIFIED

2b. GROUP
None

3. REPORT TITLE
Incremental Simulation on a Time-Shared Computer

4. DESCRIPTIVE NOTES *(Type of report and inclusive dates)*
PhD. Thesis, Sloan School of Management, February 1967

5. AUTHOR(S) *(Last name, first name, initial)*
Jones, Malcolm M.

6. REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
January 1968	252	67

8a. CONTRACT OR GRANT NO.	9a. ORIGINATOR'S REPORT NUMBER(S)
Office of Naval Research, Nonr-4102(01) b. PROJECT NO. NR 048-189 c. RR 003-09-01 d.	MAC-TR-48 (THESIS) 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*

10. AVAILABILITY/LIMITATION NOTICES
Distribution of this document is unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
None	Advanced Research Projects Agency 3D-200 Pentagon Washington, D. C. 20301

13. ABSTRACT This thesis describes a system which allows simulation models to be built and tested incrementally. It is called OPS-4 and is specifically designed to operate in the environment of the Multics system. It represents a major expansion and improvement of the OPS-3 system implemented in CTSS and also includes many features adapted from other current simulation systems. The PL/1 language, augmented by many additional statements and new data objects, provides the basis for defining models in OPS-4. A list of desirable features for an incremental simulation system is presented and it is shown how OPS-4 incorporates these features, whereas other current simulation systems satisfy only some of them and are not suitable for use in time-shared environment.

A simplified model of page and segment fault handling in Multics illustrates some of the features OPS-4 provides to allow the user to continuously interact with a model during its construction, testing and running phases. It also illustrates how the user himself may portray portions of a model that are not yet defined.

14. KEY WORDS

Computers	Multiple-access computers	Simulation systems
Incremental simulation	On-line computers	Time-sharing
Machine-aided cognition	Real-time computers	Time-shared computers

DD FORM 1 NOV 65 1473 (M.I.T.)

UNCLASSIFIED

Security Classification

Scanning Agent Identification Target

Scanning of this document was supported in part by the **Corporation for National Research Initiatives,** using funds from the **Advanced Research Projects Agency** of the **United states Government** under Grant: **MDA972-92-J1029.**

The scanning agent for this project was the **Document Services** department of the **M.I.T Libraries.** Technical support for this project was also provided by the **M.I.T. Laboratory for Computer Sciences.**

Scanned

Date: 12/19/1995

M.I.T. Libraries
Document Services