USING ALGOL W AS A BASIS FOR A DISCRETE EVENT SIMULATION LANGUAGE

by

W. DOUGLAS STIRLING, B.Sc.

Thesis submitted for the degree of Master of Science of the University of St. Andrews

(August 1976)

ACKNOWLEDGMENTS

I would like to thank my supervisor, Michael Weatherill, for the advice and encouragement he has offered during the work. I am also indebted to the Computing Laboratory at St. Andrews for the use of their facilities and to Maureen Sanders and Seonaid McCallum for patiently typing the manuscript.

DECLARATION

I declare that the following thesis is a record of research work carried out by me, that the thesis is my own composition, and that it has not been presented in application for a higher degree previously.

W. DOUGLAS STIRLING

CONTENTS

CHAPTER I

INTRODUCTION

1.0 Modelling of Real-Life Systems

When a real-life system is being studied, it is often helpful to build a "model" of the system. This "model" is a 1-1 mapping of certain aspects of the real-life system which the modeller regards as relevant to the problem he is studying. A perfect model would be a 1-1 mapping of every detail of the system, but this would usually be extremely difficult and expensive to construct. Often the precise details of its operation are not even fully understood (e.g. in modelling the economy of a country). Moreover, it is often felt that only certain aspects of the real-life system are relevant to the problem being studied and therefore that only these need to be represented in the model.

The model that is made could be of many physical forms. For example, a railway network could be modelled by a children's model railway set with model trains representing real-life trains, etc. It could also be modelled in a computer with certain states of the computer being mappable onto states of the railway network (e.g. a certain bit might have a value 1 when a train is in a certain part of the track).

The "static description" of the real-life system consists of the physical entities which constitute the system and their properties at any point of time. The "static description" of the model representing the system similarly consists of the entities constituting the model with their properties. This will be mappable onto a subset of the "static description" of the real-life system.

In, for example, the modelling of a new design of bridge to measure stresses on components, this is all that is required to characterise the model. However, many real-life systems also have time-dependant behaviour which must be modelled. The "dynamic description" of the real-life system is a description of how its "static description" changes with time. The "dynamic description" of the model is similarly how the model's "static description" changes with the model's time scale and can be mapped onto a subset of the real-life system's "dynamic description". In particular, if S' is the part of the real-life system's "static description" which is modelled and D' is the part of its "dynamic description" which refers to changes in S', then the model's "dynamic description" can be mapped onto a subset of D'.

As a model usually only represents certain aspects of the system which the modeller regards as the most important characteristics, he will usually find that everything in the model is not completely deterministic and certain influences and characteristics must be represented by random factors.

Simulation is the modelling of a system with time-dependent characteristics. The model's time scale is generally not the same as real time and does not even have to be linearly related, although a 1-1 relationship must

exist. It is often possible, depending on the type of model, to simulate the passing of a large period of the system's time in a much smaller period of real time which might mean that 10 years of the system can be modelled in say 10 minutes. This shortening of the time scale gives one of the uses of simulation techniques which is the prediction of what will happen to a real system in the future.

Modelling (whether time-dependant or not) is also used where a real-life system cannot be experimented with. For example, when a new design for a bridge is proposed, it would not be desirable to build a bridge to this design to see if it collapses. A model would be built which would be tested. Another example would be if one was wanting to know what effect a rise in the bank rate would have on a country's economy. It would be impossible to perform a controlled experiment on the economy with 2 different values of the bank rate and other things remaining the same. However such an experiment could be performed on a model of the economy.

1.1 Simulation using a Digital Computer

If a digital computer is used to model a time-dependant system, the form of the model is restricted by the nature of the computer. A computer has a finite number of states and changes of state take place discretely. Changes of state of the model must therefore also be discrete. As most changes of state in real-life systems are continuous, they must be broken down into discrete changes before simulation on a digital computer can be possible. (These discrete changes can be made as small as desired, though, in general, fairly large changes will be found adequate). It will only be certain types of real-life system that can be satisfactorally broken down in this fashion.

A modeller will have to specify both the "statics" and the "dynamics" of the model. The "dynamics"of the model are usually specified as a set of discrete "changes of state" in the model, each of which is considered to occur at one instant of the model's time scale. These "changes of state" can be called "actions" or "events".

Several special purpose "discrete event simulation languages" have been developed to help a modeller specify the "statics" and "dynamics" of the model he is creating. All such simulation languages supply the modeller with a simulated time scale at discrete times on which, the state of the model can change. They also provide entities which the programmer must use to represent the "statics" of the real-life system. These entities might be simply integer, real or logical variables he can define and use, or they might have a more complex structure and properties which the user might be able to identify with properties of the real-life entities they represent (e.g. he might be able to call some of them "ships" or "cars" or "computer programs"). The modeller will also be able to specify the set of "changes of state" of the model in terms of the static program structures he defines. All simulation languages must provide a method for defining the order in which the specified "changes of state" are to occur, and at which values of the simulated time scale.

In a good simulation language, there should be available the power of a general purpose programming language for writing the "actions", entities to help represent the static model, a method of specifying the time and order in which "actions" are to be done and a behind-the-scenes method of doing the run-time scheduling and maintaining the simulated time scale.

1.2 Synopsis of Remainder of Thesis

A new simulation language called "SLIM" has been developed which is implemented by predeclared procedures which are called from a main program written by the user in ALGOL W. SLIM is "process-oriented" and allows the use of a statement of type, "WAITTILL (logical condition)" as well as "WAIT (time delay)" which is commonly available in process-oriented languages for scheduling. A method has also been introduced to allow one "transaction" to force another to do any specified set of actions which it did not itself "plan" to do. This is a generalisation of the concept of "PREEMPTING" as used in GPSS.

To implement a process-oriented language in ALGOL W, resort had to be made to a low-level language to allow transfers of control between transactions. (The method used involves modification of the ALGOL W run time stack).

Chapter 2 is a discussion of simulation languages in general to show the kind of facilities and methods available in some of the main languages. The methods of specifying the order of execution of actions and the behind-the-scenes organisation to implement this are discussed in some detail. A framework is developed for describing most of the scheduling commands available in event and process-oriented languages.

Chapter 3 gives a description of the basic statements that can be used in creating a simulation model in SLIM. These were chosen using the arguments of Chapter 2. It then gives a simple example of their use. The implementation of these statements and the system structure on which they operate are next explained in detail. Why resort has to be made to a low-level language is also explained here, though the actual details of the low-level procedures used are not described until Chapter 4.

Chapter 4 explains the use of low-level procedures and describes the structure of the run-time stack generated by the ALGOL W compiler upon which they act.

Chapter 5 explains some of the other features of the SLIM system, of most interest being those which generalise the "PREEMPT" block of GPSS. A few examples of their use are given.

In the last chapter are two programs written in SLIM, one of which simulates a traffic junction and the other, a time-sharing computer system. These programs illustrate the main features of the language.

CHAPTER 2

A COMPARISON OF SIMULATION LANGUAGES

2.0 Introduction

This chapter examines the main facilities available in current discrete event simulation languages and tries to develop a common framework for their description. It goes into detail on the distinctions and similarities between "event-oriented", "activity-oriented" and "process-oriented" languages and on how the behind-the-scenes runtime system organises itself.

Many different simulation languages have been developed over the last few years. However the discussion in this chapter uses versions of SIMULA, GPSS and SIMSCRIPT to illustrate the main facilities available as they are still the most widely known simulation languages.

GPSS III [1] was developed by IBM and is a "process-oriented" language which is obeyed interpretively. It has a wide range of statements (called "blocks"). Most blocks take a large number of parameters and this means that programs become very difficult to follow without a detailed knowledge of the language.

SIMSCRIPT II [2] was developed by the Rank Corporation and is an "event-oriented" language which is compiled before execution. It can be used as a general purpose programming language also and as such has a power somewhat greater than FORTRAN.

SIMULA 67 [3,4] was developed by Dahl, Myhrhang and Nygaard, in the Norwegian Computing Centre and is an extension of ALGOL 60. It makes a fairly powerful general purpose language as heaped structures are allowed in a general form. However the simulation facilities which have been provided (process-oriented) are very simple though the general structure of the language would make it quite possible to improve them into a fairly powerful form.

The other language used for the discussion is CSL [5] which, though not widely used, is an example of an "activity-oriented" language.

The discussion following does not attempt to describe all the facilities available in the above languages, but merely uses them to illustrate certain points about simulation in general. Likewise, it is not suggested that these languages are in any way superior to the many others which are not mentioned.

2.1 Methods of Specifying System Dynamics

When a simulation program is written in any simulation language, it can be thought of as being composed of a set of "ACTIONS". Each "action" represents a change in state in the model and can be thought of as being indivisible in the sense that each time the action is executed, it is executed to completion without the simulated time changing and without any other actions having been executed. Each action can be executed

several times during the running of the program, both at the same and at different values of the simulated time.

There are basically three different types of simulation language event-oriented, process-oriented and activity-oriented. The difference between the three lies mostly in the way that the sequence of actions to be executed (both at the same value of simulated time and at different values) is specified. There is part of the runtime system called the "SCHEDULER" which during execution obeys the logic of the specification given in the program to cause execution of the specified sequence of actions and update the simulated time as required.

The concepts used in the three types of language will next be explained.

Event-Oriented

In an event-oriented language like SIMSCRIPT II, an action is called an "EVENT", is separately declared like a procedure, and has a name associated with it. It might also have parameters connected with it.

An action can tell the schedules to execute one or more named actions at the same or a future value of simulated time, and can pass parameters to these other actions if they are required. The system is initialised before the start of the simulation to have certain actions pending and the only other way an action can be executed is after a call to specifically "SCHEDULE" it from another action.

In SIMSCRIPT II, the scheduling statement is:-

SCHEDULE actionname (parameters) IN time delay / NOW

Thus the exact time an action is to be executed must be known in SIMSCRIPT II before the scheduler is told about it.

There are many situations when one would like to use a statement like "SCHEDULE actionname (parameters) AS SOON AS condition". As soon as the condition specified became true, the scheduler would execute the specified action. This type of statement is not available in SIMSCRIPT II. An example of its use would be in a traffic-light simulation where one action was "car arriving at the lights". If they were at red, it would like to say "as soon as the lights become green execute action "enter junction". When the lights became green, the scheduler would then automatically recognise this and execute the action specified.

If this type of statement did not exist, the action to "change the lights" would also have to look and see if a car was waiting and if so, schedule the immediate execution of the action "enter junction". This is a very simple example and one can easily envisage a situation where the condition waited for is very complex. The approach without the above scheduling statement would be very difficult and long-winded to specify: (At each point of any action which might make the condition true, the condition would have to be checked to see whether the action required could be scheduled immediately.)

The special characteristics of event-oriented languages are that each action is declared separately and that the method of scheduling actions is by specific calls to the scheduler giving the condition or time at which the named action is to take place. The passing of parameters to scheduled actions is possible with this method and is a very helpful facility (e.g. the action "enter junction" above could be passed the car which was

to enter the junction instead of having to search itself to find which car was being referred to).

Activity-Oriented

In an activity-oriented language like CSL, an action is called an "ACTIVITY" and is declared like a procedure. An action here does not need to have a name (though it normally does have, e.g. CSL) and does not have any parameters. Associated with every action is a condition. As opposed to event-oriented languages, where the scheduler is told which actions are to be executed, here no information is directly passed from one action to the scheduler to schedule another. Instead, the scheduler actually looks for actions to execute. At each value of simulated time, the scheduler looks at each action in turn (in the order written) and tests the condition attached to it. If it yields a value true, then the scheduler executes the action. After all the actions have been looked at in this way, simulated time is updated, and the scheduler looks at all the actions again. One action could however tell the scheduler that it should re-check some previous actions before updating simulated time as their conditions might now be true.

As opposed to an event-oriented language where the scheduler can update simulated time to the time of the earliest scheduled action, the scheduler here does not know when the earliest time an action can be executed will be. Normally, it would therefore be updated one unit at a time. This however is not necessary if a method as adopted in CSL is used. Here there is a class of integer variables called "T-cells". Such variables can be declared by the programmer and can be "linked" to other entities.

These cells hold values which represent "time in the future". They can be assigned to and used like any other integer variable, but are also used by the scheduler. When it comes to the point for the scheduler to update simulated time, it looks at all T-cells to find the smallest, positive, non-zero value. It then subtracts this value from every T-cell which brings each "time in the future" that amount closer to the present time. (It should be noted that this means that it is impossible for the scheduler to update simulated time by 0).

In this way a function similar to "SCHEDULE action1 IN timedelay" can be made using a T-cell, X. Instead of the SCHEDULE statement would be the statement, X; = time delay and the condition attached to action1 would be X=0. When there is a scan of the actions in time delay units of simulated time, the schedules will have subtracted time delay from the T-cell X, and it will have the value 0.

The activity-oriented approach makes it fairly easy to deal with actions whose occurrence time depends on the state of the system. In fact all actions must be specified like this. It is easy for the user to specify such actions and they can be implemented efficiently compared with "SCHEDULE actionname AS SOON AS condition" in an event-oriented language.

A further advantage is in the simplicity of the scheduler. It can almost be as simple as the following:-

```
While  simulation_not_finished do
  begin for A : = every action in order written do
              if condition (A) then action (A);
        update simulated time
  end
```

However, there are 3 disadvantages with the activity-oriented approach:-

(i) parameters cannot be passed to an action as there is no scheduling call. Each action must therefore search for any such parameters required.

(ii) every occasion simulated time is updated, all the conditions are checked. This means probably that there will be a lot of time spent checking conditions that the programmer knows cannot be true at this point. This is specially true when the action would be scheduled as "SCHEDULE actionname IN timedelay" in an event-oriented language. The extreme example of this is the activity used to end the simulation, whose condition, say T.FINNISH=0, is checked every time simulated time is changed.

(iii) The order of writing the actions can be important. There can even be two actions, each of which can alter to true the condition attached to the other. In this case, neither order of writing would ensure that after the execution of one action, the system would recheck the other. There would therefore have to be a method provided of telling the scheduler to do this.

Process-Oriented

The fundamental notion of a process-oriented language is that an action is executed by something. These entities which conceptually execute actions are called "TRANSACTIONS" (this is the terminology used in GPSS). A transaction has a part which is invisible to the programmer and is used by the system

for scheduling purposes, called an "EVENTNOTICE" and possibly also a part containing several programmer-defined variables. There are several such transactions in the system at any time and the user-defined parts do not all have to be of the same form.

A transactions life is not limited to one action, but extends through many actions. When a transaction is no longer required, it must execute a statement that explicitly destroys itself. At every point, each transaction is linked to an action which it can either be executing at the time or which will be the next action which it will execute when it is given control. At any point, there may be more than one transaction linked to the same action. The transactions can independantly execute the action is this case. A "PROCESS" is the union of one transaction and the set of actions it can execute.

The set of actions in a process is usually written as a program interspersed with "SELF-SCHEDULING" statements. When a transaction executes a self-scheduling statement other than the one which destroys the transaction (e.g. TERMINATE block in GPSS), the system schedules the action starting at the statement following the self-scheduling statement, linking the current transaction with it. Therefore, the transaction executes statements in the normal programming order, except that when it executes a self-scheduling statement it experiences some kind of delay with respect to the simulated time. An example would be GPSS's"ADVANCE"block. That such a process can be split up into actions is obvious since each separate statement satisfies the criteria to be called an action (with an implied null self-scheduling statement between each). A better break-down into actions can be got by making the first statement following each self-scheduling statement the start of an action. All

the statements in each path from this statement are part of the same action apart from these which would be part of 2 or more actions by this rule. The first statement in each of these sections is similarly the start of an action.

There are two main types of self-scheduling statement which can be available:-

(i) "WAIT time delay" which schedules the next action with the current transaction in "time delay" units of simulated time.

(ii) "WAITTILL condition" which schedules the next action with the current transaction as soon as "condition" gives a value true.

During the execution of an action in a process-oriented language, there can also be statements similar to the SCHEDULE statements in an event-oriented language:-

(i) "SCHEDULE action name (parameters) IN time delay"

(ii) "SCHEDULE action name (parameters) AS SOON AS condition"

In this case a new transaction would be created with an undefined "user-defined" part and this would be linked to the scheduled action. A transaction with no user-defined part is really just an eventnotice, and these statements would be implemented in almost the same way here as in an event-oriented language because there, a similar "eventnotice" (usually a bit smaller) would have to be created by the scheduler to help it to execute the specified action at the right time. The bigger eventnotice is usually required to hold certain system information about the state of the transaction.

A further pair of scheduling statements might also be available, "DETACH" is like a self-scheduling statement in that it links the current transaction with the action starting at the next statement. However it does not tell the

scheduler when this next action is to be scheduled. The statement "RESUME transaction" would have to be used by another transaction to cause execution of the action linked to the "detached" transaction. The RESUME statement could be generalised into:-

(i) "RESUME transaction IN timedelay"

(ii) "RESUME transaction AS SOON AS condition"

It can therefore be seen that a good process-oriented language would have the scheduling statements of an event-oriented language and also have self-scheduling statements. WAIT and WAITTILL are actually the same as the two SCHEDULE statements, but have an implicit "next action" and pass on an implicit "transaction" as parameter. They could therefore be implemented in terms of SCHEDULE statements by making these explicit. DETACH and RESUME could not easily be expressed in terms of SCHEDULE statements, but where "SCHEDULE AS SOON AS" is available, there are not many circumstances where DETACH and RESUME are needed. A process-oriented language could otherwise by mapped onto a good event-oriented one by defining each action (found as explained above) as an event requiring a transaction as a parameter and having a name. WAIT and WAITTILL would be changed into the appropriate SCHEDULE statements with explicit "next action" and transaction passed. The actions ending in TERMINATE would simply have that removed. Conversely an event-oriented language could be mapped onto a process-oriented one by making each event a separate process and placing a TERMINATE at the end of each.

The main advantage a process-oriented language has over an event-oriented one is in ease of programming when the concept of a "transaction" can be identified with some type of object conceptually passing through the

the model (e.g. programs in a computer simulation, cars in a traffic simulation, orders in a job-shop simulation, etc). The process can be written as the "life-history" of such a transaction with WAIT and WAIT-TILL added in a fairly straight-forward fashion. In an event-oriented language, all the separate actions would have to be identified and defined, and suitable parameters passed between each. There is also a slight efficiency advantage in that self-scheduling statements do not require eventnotices to be created by the system as they already exist as part of the associated transaction whereas all scheduling statements in an event-oriented language need an eventnotice to be created. This would however probably be balanced out by the fact that eventnotices in a process-oriented language would normally be bigger and take more time to maintain.

The above discussion refers to the methods of specifying system dynamics in ideal process-oriented and event-oriented languages. It is not intended here to compare the actual facilities available in specific languages of the two types. However the scheduling statements available in GPSS III and SIMULA 67 will be described. It should be noted that as neither they nor SIMSCRIPT II have all the statements possible for their types, they cannot be mapped onto each other as the "ideal" languages could.

In GPSS, there are no statements or "blocks" comparable with either of the SCHEDULE statements in our model language, apart from the "SPLIT" block which schedules the immediate execution of a named action with a new transaction which is a copy of the current transaction. No parameters can be passed. DETACH and RESUME take the form of the blocks "LINK" which links the current transaction to a specified "USER CHAIN" as well as performing a DETACH and "UNLINK" which RESUMES a specified member of a given "USER CHAIN".

Most of the self-scheduling statements in GPSS can be expressed in terms of the two procedures, W and WT, defined as follows:-

W (time delay) ≡ if (time delay ≠ 0) then WAIT time delay

WT (condition) ≡ if (¬ condition) then WAITTILL condition

These two procedures are very similar to WAIT and WAITTILL, but the fact that there is no scheduling if timedelay = 0 or condition immediately holds can be important (e.g. if a "high priority" transaction is waiting for condition C and, while a "low priority" transaction is in control, C becomes true. If the "low priority" transaction executes WT (C), it would carry on, but if it executed WAITTILL C, the "high priority" transaction would be moved first and if C again became false, the "low priority" transaction would be blocked.) The "ADVANCE" block is directly equivalent to the W statement. The other self-scheduling statement, WT, appears in many disguises. The "TEST" and "GATE" blocks can be used as pure WTs, but many other blocks have scheduling aspects which can be expressed in terms of WT (e.g. the block to "SEIZE" a facility can be expressed as "WT (facility is empty), then take the facility"). There is however another type of self-scheduling statement that is not expressable in terms of W and WT. If a "BUFFER" option is used in a "PRIORITY" block, the effect of the block is to alter the transaction's priority, and then WAIT 0 (which allows transactions with higher priority to move before it).

In SIMULA 67, the "detach" and "resume" statements are the same as our DETACH and RESUME, and "HOLD" is the same as W defined above. The construct, "identifier:=new actionname (parameters)" is equivalent to the statement, "SCHEDULE actionname (parameters) IN 0". However here the identifier is also made to point to the created transaction. There is no other SCHEDULE statement and no WAITTILL.

Finally, it must be mentioned that there is one block in GPSS which affects the scheduling and does not fit in with the scheduling statements introduced so far. This is the "PREEMPT" block, After one transaction has "SEIZED" a "FACILITY" (say representing a CPU in a computer simulation) and is waiting there for a period of time, another transaction can "PREEMPT" the facility in question. This means that it forces the first transaction to leave the facility and then freezes it indefinitely, while it itself enters the facility. After it is finally finished with the facility in question and leaves it, the preempted transaction is put back into the facility and acts as if the time it had been out of the facility had not existed. The above would model a program seizing a computer CPU and holding it while it did a computation, but at some point being interrupted by an I/O channel for a period of time, and then continuing as if the interrupt had never occurred. The relevant aspect of this is that one transaction forces another to do something that it did not itself plan to do (ie. give up the CPU for a period). This cannot be expressed in terms of the scheduling statements introduced so far. The concept is only really relevant to process-oriented languages. It can be very useful in certain types of model and a generalised form of it is introduced in chapter IV.

Mixtures

As the event-oriented approach is more-or less a subset of the process-oriented approach, a mixture of the two would be meaningless. However the

activity-oriented approach could be mixed with either of the others. This would entail adding to the event or process-oriented language the ability to define actions with attached conditions. At any value of simulated time, as well as executing all actions scheduled at that time, the scheduler would test all the conditions attached to the other defined actions and execute them if they were true. In SIMSCRIPT II it is possible to add such condition/action pairs.

This simplifies the updating of simulated time compared with a pure activity-oriented language, as it can be updated according to the scheduled actions (although that means that an action associated with a condition cannot be executed at any time if there is not another action "scheduled" at that time). However, the problems of the order of writing actions in an activity-oriented language are made more complex by the problem of whether scheduled actions should be executed before conditions are tested. It is possible to conceive of complex situations where certain types of scheduled actions should be executed before certain conditions are tested and certain after, and, even worse, situations where no fixed order would do.

It would appear that there are different situations in which either activity or event-process or a mixture might be both easier to program in and more efficient. It would really depend on what was being simulated and on how good a language of each type was available.

2.2 Organisation of scheduled actions and operation of the "Scheduler"

In activity-oriented languages, actions are not specifically scheduled and the operation of the"scheduler", which is relatively simple, has been already explained.

As was mentioned in section 2.1, the scheduling statements available in an event-oriented language are a sub-set of those possible in a process-oriented language. Those available in our generalised process-oriented language are:-

(1) DETACH

(2) RESUME transaction name

(3) SCHEDULE action name (parameters) IN time delay

(4) WAIT time delay

(5) SCHEDULE action name (parameters) AS SOON AS condition

(6) WAITTILL condition

The first statement above does not concern the scheduler as it is only on a call of RESUME that the action is actually "scheduled". Action scheduled by the 2nd, 3rd or 4th statements can be called "TIME-SCHEDULED" and by the 5th or 6th, "CONDITION-SCHEDULED". All time-scheduled actions can be treated similarly, as the only difference is in the way in which the parameters "next action" and "transaction" are found. The same holds for all condition-scheduled actions. In (2) the transaction is passed to the scheduler and has stored with it in some way a representation of the "next action" put there by the DETACH. In (3) and (5), the scheduler creates a new transaction and the "next action" is passed by the call (It will normally be the address of the action's first statement, but could be a symbolic name). In

(4) and (6), the transaction scheduled is implied to be the current transaction which the scheduler will be able to find, and the call will be compiled into a form such that the address of the "next action" is passed to the scheduler (this could simply be the "return address" of the call). For the remainder of this section, therefore, a distinction will not be made between the different types of time-scheduling and of condition-scheduling.

In SIMULA 67 and SIMSCRIPT II, all actions scheduled are time-scheduled and the scheduler is therefore fairly simple to implement. In SIMULA 67, the scheduler keeps the eventnotices for all scheduled actions linked in a linear list. This list is ordered by the values of simulated time at which the actions are due to be executed. When an action is scheduled, the scheduler calculates the value of simulated time at which it should be executed and places its eventnotice in this list after all other eventnotices corresponding to actions scheduled at either this or an earlier value of simulated time. After one action has been completed, the scheduler takes the first eventnotice from this list and executes the action it is scheduled to do. If it was due to be executed at a time later than the current value of simulated time, simulated time is updated.

In SIMSCRIPT II, a similar method is used, except that there is a separate list for each type of action. When an action is scheduled, its eventnotice is placed on the list for this type of action in the same order as the list in SIMULA 67. The first action to be executed at any time is found by comparing the first eventnotices in every list. If there are two or more such eventnotices scheduled at the same time, the order of execution is resolved by the programmer who initially says which order the action types should be executed in if there is such a conflict.

When there are condition-scheduled actions, a more involved structure is required. The mechanism used in GPSS III will be explained as one solution to the problems involved.

Scheduling Structure in GPSS

The problem about the implementation of condition-scheduled actions is in how the scheduler should know when the scheduling condition has been satisfied. The eventnotice for each such condition-scheduled action must hold a method of recalculating the scheduling condition. It might be critical that the action scheduled should be executed immediately the condition is satisfied. It might also be critical that when 2 actions each have their condition satisfied, the first scheduled is moved first. One method of ensuring that the actions are executed at the right times would be to keep all condition-scheduled eventnotices in a linear queue, each new condition-scheduled eventnotice being placed at the end of the queue. Each time the scheduler was looking for a new action, it would look at each eventnotice in this queue, from the start in turn, testing its condition. As soon as it found a satisfied condition, it would remove the the eventnotice from the queue and execute the corresponding scheduled action. Otherwise, it would take the first time-scheduled action and execute it. The trouble with this method is that it is extremely inefficient as the scheduler would be spending a large amount of time checking conditions. For example, if the first ten condition-scheduled actions in the queue were held up for a long time, the scheduler would have to recheck all ten conditions between executing any other actions. If there were a lot of condition-scheduled actions in the system at a time, the problem would be very serious.

To help overcome this problem, the scheduler in GPSS recognises two different types of scheduling condition - certain standard conditions like "facility empty", "facility full" or "logical switch set", and other conditions which can be of any complexity. With the standard conditions, the scheduler knows what kinds of statement might free such a condition-scheduled action (e.g. "RELEASE facility", "SEIZE facility" and "SET logical switch" for the above 3 conditions) and knows that there is no point in retesting these actions' conditions until such a statement has been executed. They are therefore marked "passive" in the queue for condition-scheduled eventnotices. For each standard condition, there is a "delay chain". Actions delayed by a standard condition are also linked to its delaychain, so that when, for example, a transaction executes "RELEASE facility 1", this will mark all eventnotices on the delay chain for the delaying condition "facility 1 empty" as "active", and the scheduler will know to check their conditions. If, when the condition is checked by the scheduler, it is found still to be false, the eventnotice is again marked "passive". Otherwise it is removed from the queue of condition-scheduled eventnotices and from the delay chain, and the corresponding action is executed. With other kinds of more complex condition, the eventnotices are only kept on the queue and not linked to a delay chain. They are kept "active" until they are moved.

To be sure of moving transactions in the correct order, the scheduler would still have to scan the queue from the beginning before every action, the only simplification being that it would not have to check the conditions attached to eventnotices marked "passive". This could still be very time-consuming. The scheduler in GPSS therefore adopts a method which is "right most of the time". It does not start at the beginning of the queue each

time it looks for a new action, but often starts looking from where it left off in the queue the previous time. It starts from the beginning only when either simulated time is updated or certain types of statement have been executed in the last action. This normally is enough to ensure that actions are executed in the right order, but in certain circumstances can mean that they are not, or even that one is executed at the wrong value of simulated time. The most important types of statement which make the scheduler start its scan at the beginning of the queue are those altering the state of "facilities", "storages" and "logic switches" and the one which alters the "priority" of a transaction.

The above is the basic principle behind the GPSS scheduler, but the actual operation is slightly more complex because of the concept of "PRIORITY" Every transaction has a priority associated with it, which is an integer from 0 to 127, and can be changed by the user. The idea is that if there are several transactions which can move at a certain value of simulated time, those with the highest priority will always move first. There is a "FUTURE EVENTS CHAIN" which is a linear list of time-scheduled eventnotices ordered by their scheduled time. All time-scheduled actions are on this list apart from those scheduled for the current value of simulated time. There is also a "CURRENT EVENTS CHAIN" which is actually composed of 128 separate "PRIORITY QUEUES", one for each possible priority class. When an action is condition-scheduled, its eventnotice is put at the end of the appropriate priority queue. When simulated time is updated, all transactions on the future events chain which are scheduled at that time are

removed from it and placed at the end of the appropriate priority queues of the current events chain. They can then be considered as a special case of condition-scheduled actions with a condition that always yields the value "true". The current events chain therefore holds all actions which could possibly be scheduled at that value of simulated time. The scheduler then considers the chain as a linear queue with the first event-notice in one priority chain following the last in the next higher priority chain and scans the chain as explained before. When it reaches the end of the current events chain, simulated time is again updated.

This approach means that although eventnotices might be removed from anywhere in the current events chain, they are always added at the end of a priority queue. This makes it fairly efficient as the actual queue can be found by indexing with the priority. Also there is not much overhead in dealing with condition-scheduled actions (apart from the actual testing of conditions) as all that happens to them until they are moved is the setting of a flag to mark them "active" or "passive". The moving of time-scheduled actions from the future to the current events chain when simulated time is updated means that the future events chain does not have to be ordered by priority.

The method used in the GPSS scheduler is not the only method of treating condition-scheduled actions. However it shows the problems involved and gives an idea of the extra complexity of the scheduler required to solve them.

2.3 Structures to help specify the static model

This and the following sections of this chapter apply to all the three methods of specifying system dynamics discussed in the last two sections.

I give the name "STRUCTURE" here to an entity which has properties more complex than simply one integer, real or logical value. An array is an example of a structure. Another example is a group of values each of which is accessed by using a unique "field name" as well as the structure name; the values here do not have to be all of the same type (this is like a record in ALGOL W or a structure in ALGOL 68). The properties of a structure can however be of a not so easily defined nature. An example of this would be a "queue" where a statement would exist to put something into the queue and another to remove something from the queue. From the users point of view, this structure would not be a set of values, but would have the properties of holding the entities in the queue and maintaining a first-in, first-out order amoung them.

The use of these structures is that in a model they are usually representations of actual things in the real-life system being modelled. For example, a structure called a "CAR" with an integer field "no. of passengers" and two character string fields, "origin" and "destination" might be a static representation in the model of a real-life car.

There are three possibilities for the creation and destruction of structures:-

(i) There are a fixed number of them with fixed properties which are in the environment of any program of the language whither they are required or not. Specific ones might be given names by the programmer. This is how all structures in GPSS are treated apart from "transactions".

(ii) Different types of structure might be defined by the programmer with properties to suit his model. He must, however, state how many of each are required before the start of the simulation, and this number is fixed throughout the whole program. All structures in CSL and "permenant entities" in SIMSCRIPT II are like this.

(iii) It might be possible to define ones own types of structure as in (ii). However the number of these structures which exist at any time is not fixed, but varies dynamically with the running of the program. Statements exist to order the creation or destruction of such structures. All structures in SIMULA 67 and "temporary entities" in SIMSCRIPT II are like this. "Transactions" in GPSS are also dynamically created and destroyed, though there is a fixed upper limit to the number there can be at any one time, due to the implementation.

Apart from GPSS, most simulation languages allow the user to specify the form of the structures he will use. When he defines a structure type, he will list its properties, giving integer, real, or logical fields a field name. SIMULA 67 allows fields to be defined also which point to other structures and can be manipulated by the programmer. SIMSCRIPT II allows structures to be defined to also have the property that they "own" a list of other structures and there are statements to insert and remove specified structures from the list.

Structure types in GPSS are system-defined. A "FACILITY" has among other things the property of "being in use" or "not being in use". Its state can be changed by a transaction "SEIZING" or "RELEASING" it. It also collects statistics on its own usuage. A "STORAGE" is similar except that it is initialised to have a certain "size". It can"contain" transactions up to this number and also collects statistics. A "LOGIC SWITCH"

is to the user, little more than a logical variable, but like the previous two structure types, possesses "DELAY CHAINS" as the "standard conditions" are phrased in terms of these structures. These delay chains are used to help the scheduling as was explained in section 2.2. The main other type of structure in GPSS is a "GROUP". This is a list of transactions and blocks exist to insert and remove transactions from the list.

In process-oriented languages, there are also "transactions" which themselves are structures. The eventnotice part of transactions is invisible to the user. In GPSS the rest is mostly an array of integer values, though there are also a few properties like the transaction's creation time which can be accessed. In SIMULA 67 a transaction can be defined to have fields as can any other structure in the language. However the number of fields in a SIMULA 67 transaction can dynamically change. A process in SIMULA 67 is written like a procedure, and when a transaction enters a begin - end block with declarations in the process, the declared variables are added as extra fields of the transaction. The ability to dynamically add and remove fields from a transaction can be very useful.

2.4 Initialisation of system

In any simulation language, the programmer will want to define certain global variables and structures and to initialise them before the start of the simulation. In GPSS, this is not required as there are fixed numbers of variables, etc. defined by the system and initialised to default values of zero, empty, etc.. Initialisation to any other values must be done during the actual simulation by organising the program so that the first action executed does any other initialisation. In both CSL and SIMSCRIPT II, there are separate program segments which define the variables and other entities required by the program. In CSL, this segment also initialises the variables, whereas in SIMSCRIPT II, the initialisation is performed in a small control program which starts and stops the simulation. In SIMULA 67 global variables are defined and initialised in a block which is outer to the rest of the program.

In an activity-oriented language like CSL, this is the only initialisation of the system required. The scheduler can immediately look through all the activities in turn as explained in section 2.1. To stop the simulation, there can be a programmer-defined activity which is scheduled to happen at the required stopping time in the same fashion as any other activity.

In the other types of language, there have to be actions scheduled in the system before control can be passed to the scheduler to start the simulation. In GPSS, this is performed by the "GENERATE" block. The system recognises these blocks and before the simulation starts, schedules actions as specified in these blocks. These actions start at the blocks following the GENERATE blocks.

The normal method, as in SIMSCRIPT II and SIMULA 67, is to write a small control program which schedules the initially required actions, starts the simulation and after the end of the simulation prints out statistics. This control program actually satisfies the criteria to be a process (even in SIMSCRIPT) as there are at least 2 actions, done at different values of simulated time, the setup at time 0 conceptually and the output of statistics at the time the simulation is stopped. In SIMULA 67, the control program is in the same outer block that the global variables are defined in, whereas in SIMSCRIPT II, it is in a separate segment.

2.5 Random Factors

In most simulations there will be random variables required and most languages provide several pseudo-random number streams which generate random numbers in the range 0 to 1. Pseudo-random numbers are used so that experiments can be repeated with the "random" factors the same, but with some parameters of the model altered. This makes it much easier to evaluate the effect of the change.

Functions are usually also provided to generate samples from some probability distributions using a given random number stream. Most common are Uniform (min, max), Normal (mean, standard devn.), Negative exponential (mean) and Poisson (mean). Facilities are generally also provided to define one's own functions.

Another very useful tool is to be able to generate samples from empirical distributions, or, more general, to have empirical functions (which can be of a random variable). Both GPSS and SIMSCRIPT II are very good in this respect and one can specify as a function, several pairs of variables. These can be thought of as points on a graph. The system will either produce a step function from them or will smooth between the points with straight lines to give a more continuous function. A programmer should, however, be able to write procedures himself to give such an effect without too much difficulty. In SIMULA 67, one would have to write ones own procedures as the system gives no help.

2.6 Tracing execution and Statistic-gathering

It should be possible to find out exactly what happens, and when in a program for debugging purposes. GPSS does not allow write statements to be executed by a transaction, so the transaction cannot output when it is doing what in a way that is concise and in language that is appropriate to the model. Instead the system, if so requested, outputs a list of block numbers (the units of a GPSS program) along with the transaction numbers executing them, the simulated time and a few other standard variables. This is far from readable. Most other languages allow the addition of write statements which, although meaning more work for the programmer, if they are sensibly written at least provide an easily understood trace of execution. Some method of tracing execution is, however, essential to find if a model has been specified correctly.

Statistic gathering, like tracing, is not essential as far as the structure of the model goes and can normally be added after the model has been designed and is working. However, normally the purpose of the model is to collect empirical statistics about the model's operation and therefore the types of statistics that can be gathered and the ease with which the appropriate statements can be added to the model are of great importance.

In GPSS, the system automatically collects certain statistics about the model. These statistics collected are about certain structures that the programmer uses as part of his model, namely "facilities" and "storages". These statistics are fairly comprehensive about the structures' use. These are also special statistic-gathering structures called "QUEUES" and "TABLES" (of which, the tables have to be partly defined by the programmer).

Transactions can "JOIN" or "LEAVE" queues and can ask for certain values to be "TABULATED", and the system will calculate statistics about the queue use and will output a table of the numbers of transactions whose tabulated value was in certain programmer-defined ranges. It is however difficult to calculate any other "purpose-built" statistics as a write statement is not available to output them at the end of the simulation.

SIMULA 67, SIMSCRIPT II and CSL all allow the use of write statements, so a user can calculate whatever statistics he requires and output them in any format required. In SIMSCRIPT II, it is also possible to tell the system to keep certain averages, standard deviations, histograms and some other statistics, when defining the global variables and structures to be used in the model. This means that the programmer does not have to write statements in his actions to maintain such statistics. Instead, the system will automatically maintain them. In CSL, histograms can be defined and calls can be made in an action to perform a function like the GPSS "TABULATE" block. In SIMULA 67, however, the system gives little hlep in the collection of statistics.

CHAPTER 3

DESCRIPTION OF BASIC "SLIM" SYSTEM

3.0 Introduction

In the light of the arguments set out in Chapter 2, it was intended that the SLIM system should:-

(i) be process-oriented,

(ii) give the user the full power of a high-level general-purpose language to write his "actions",

(iii) have as rich as possible a set of scheduling statements to allow the user to specify system dynamics in the easiest way, and

(iv) be as efficient as possible.

Neither of the two process-oriented languages mentioned in the last chapter are totally adequate. GPSS does not allow many constructs available in general-purpose languages (e.g. procedure calls or real variables) and is inefficient. SIMULA 67 does not provide adequate scheduling statements.

As SLIM was intended to have the power of a general-purpose programming language, it was decided to implement it by embedding it in a high-level language as a set of pre-written procedures. It was hoped that these procedures could themselves be written in the language used, but no language the author was aware of would allow this totally. There were two special features

the language would require, the ability to store typed procedures as variables for later execution and the ability to transfer control back and forward between processes without either being executed to completion. ALGOL 68 [6] would allow the first, but not the second, whereas SIMULA 67 would allow the second but not the first. As neither was available to experiment with at any rate, it was decided to use ALGOL W [7,8] which meant that resort had to be made to a low-level language to implement both these requirements. It is regretted that SIMULA 67 was not used as SLIM would have then been better structured for the user and a bit less long-winded. However the scheduling statements and most of the system structure would have been the same, and most of what follows in the thesis would be as relevant.

Remainder of Chapter

Section 3.1 describes the basic scheduling statements that are available in SLIM, from a user point of view. (A few other features, including a more advanced scheduling concept are described in Chapter 5.) Section 3.2 gives a very simple example of the use of the scheduling statements to simulate an aeroplane shuttle-service.

Sections 3.3 and 3.4 explain in detail the behind-the-scenes system structure and how the scheduling statements use this structure. The points where resort has to be made to a low-level language and roughly what is done there are mentioned, though these low-level procedures are not fully described until Chapter 4.

3.1 Description of system from user point of view

Representation of a transaction

SLIM is a process-oriented language and every action is conceptually executed by a "transaction". A "transaction" always consists of a unique record of type EVENTNOTICE which is used by the system mainly and also possibly another user-defined record which holds variables private to the transaction. The 2 records are connected by the field TRANS of the EVENTNOTICE record which points to the user-defined record in the transaction.

The procedure, TRANSACTION (new record) connects a specified new record with the current transaction in this manner. From then on, when this transaction is "in control", the global reference variable CURRENT points to its user-defined record and CURRENTEVNOTICE, points to its EVENTNOTICE record.

Each transaction has a "priority" associated with it, with the same meaning as in GPSS. This is stored in the integer field PRIORITY of the EVENTNOTICE, can have a value from 0 to 100, and can be used and assigned to like any other integer variable.

Simulated Time

Simulated time is represented by an integer variable called SIMTIME This variable is zero at the start of the simulation and is maintained by the "scheduler" during the simulation.

SIMTIME must only be read. If it is assigned to, errors will probably occur.

Scheduling Statements

In the basic system, the two time-scheduling procedures are:-

(1) WAIT (integer TIME)

When a transaction executes this statement, it, in effect, tells the scheduler to schedule the action starting at the statement following the WAIT with the current transaction associated, after a delay, "TIME". The scheduler is then called to find the next transaction to give control to. If "TIME" is negative, the procedure does nothing.

(2) SCHEDULE EVENTS (integer DELAY, NO, PR; procedure DEST)

When a transaction executes this statement, it tells the scheduler to create "NO" of new transactions, none of which have any associated user-defined record and each with a priority "PR". These transactions are scheduled with a separately computed "DELAY" between each, and the first being "DELAY" after the current value of simulated time. The first part of the action each such transaction is scheduled to do is "DEST" which must end in a goto statement which links it to the rest of its "first action". (This method of specifying the "next action" is rather messy, but is necessary because of the implementation as seen later). The executing transaction continues after executing this statement.

The basic condition-scheduling procedure is:-

(3) WAITTILL (logical CON ; reference (DELAYCHAIN) DC)

When a transaction executes this statement it, in effect, tells the scheduler to schedule the action starting at the statement after the WAITTILL as soon as the condition, "CON" gives a value true. The executing transaction

is linked with this scheduled action, and the scheduler looks for the next scheduled action.

A "DELAYCHAIN" is a list of transactions which are held-up waiting for some condition to hold (not necessarily all for the same condition). This list is ordered by priority and then by order of being delayed (highest priority first, earliest delayed first). When a transaction is held up by a WAITTILL procedure as above, it is linked in order to the specified delay chain "DC". At the head of each delaychain is a record of type "DELAYCHAIN" whose integer field HELDUP holds the number of transactions which are on the chain at any time.

There is another procedure which is not really a scheduling procedure but can be grouped with them:-

(4) TERMINATE

This in effect destroys the transaction which executes it. The scheduler then looks for the next scheduled transaction to give control to.

A final procedure can be mentioned here:-

(5) STOP

When any transaction executes this procedure, the whole program is stopped in an orderly fashion. The effect is the same as if the last end in the program had been labelled "STOP" and "goto STOP" had been executed.

Prompts

The way condition-scheduled actions are treated in SLIM is a generalisation of the method used in GPSS for transactions delayed by "standard conditions" (which was explained in the last chapter). Whereas in GPSS, when a transaction is delayed by a standard condition, the "delaychain" in

which the transaction is to be kept is implied by the standard condition, in SLIM this delaychain must be specified by the WAITTILL statement. The condition can therefore be of any complexity. Also whereas in GPSS, when any transaction executes certain types of statement (e.g. "RELEASE facility 1") a "prompt" is automatically sent to the scheduler to retest whether transactions held up in a certain implied delaychain can now move (e.g. the one attached to the condition "facility 1 empty"), such prompts in SLIM must be explicitly programmed by the user. When a transaction is first held-up by a WAITTILL, a prompt is issued referring to it so that the scheduler will know to check its condition as it might immediately be true.

There are also two procedures to issue prompts to the scheduler to remind it to check the conditions of transactions held up in a specified delaychain:-

(1) TESTFIRST (reference (DELAYCHAIN) DC)

This prompts the scheduler to retest the condition of the first transaction on "DC". (This is its highest-priority transaction, and of these, the earliest delayed).

(2) TESTALL (reference (DELAYCHAIN) DC)

This prompts the scheduler to retest the conditions attached to all transactions on DC.

When a prompt has been given, the scheduler does not immediately test the transactions' delaying conditions, but "marks" them so that it knows to try to move them as soon as their priorities will allow. It takes them into consideration in future when looking for the next transaction to give control to, but immediately before actually giving any such transaction control, it retests the condition. If it is still not satisfied, the "mark" is removed and the transaction is ignored until another prompt has been received referring to it.

It should be noted that, as in GPSS, the scheduler completely ignores all delayed transactions until after it has received a prompt referring to them. In GPSS, it is arranged that any statement which can possibly alter a standard condition, automatically issues the appropriate prompt. Therefore it is assured that as soon as a transaction has its standard delaying condition changed, the scheduler will know that it might be able to move. However in SLIM, if a prompt is not programmed after statements are executed which alter the delaying condition of a transaction, the scheduler will not try to move this transaction. The responsibility for inserting prompts is therefore the programmer's and the program will not work properly if he misses some out. There is however no alteration in the correct running of a program if superfluous prompts are added, apart from a loss of efficiency. (The scheduler will itself test the delaying condition before giving any transaction control even though a prompt has been received). Therefore it is always better to err on the side of having too many.

Global variables and the "Control Process"

A user program in SLIM is written as a begin end block. All variables global to the simulation, and all delaychains required must be declared first. The process starting at the first statement of the block is called the "Control Process". The system initially creates one transaction with priority 0 and without local variables, which starts to execute the Control Process at simulated time zero. This transaction and process are of the same form, and therefore have the same capabilities as all other

transactions created later in the simulation. However, it must start by calling the procedure INITIALISE.

The normal form of the Control Process is as follows:-

(1) Global variables and record types are defined

(2) INITIALISE is called to initialise the system

(3) The global variables are initialised to appropriate values, including the delaychains, pointers to which must be assigned an initialised record of type DELAYCHAIN, NEWDELAYCHAIN,

(4) Transactions can be created and scheduled by repeated use of the procedure SCHEDULE_EVENTS

(5) The transaction executing the Control Process executes a self-scheduling statement which allows the rest of the created transactions to execute actions and interact for a length of simulated time (e.g. WAIT (1000)).

(6) When control is eventually returned to the Control Process (in the example after 1000 units of simulated time) it usually outputs certain statistics gathered up to that point

(7) Possibly, after some further re-initialisation, steps (5) and (6) might be repeated

(8) The procedure STOP would then be called to end the simulation

System Prelude

In a block, outer to the user program as defined in the last section, there must be added definitions of the system procedures and structures. These are in ALGOL W and have to be re-compiled for each user program as they all use common data structures. Most of these definitions can simply be copied or a deck of cards included.

However in this"system Prelude", there must be also included definitions of the types of record which are to be used as the user-defined parts of transactions. Also one system procedure and two variables refer to the names of these record types. These have to be copied out with the appropriate record type names included.

The form of a whole program is as follows:-

```
begin    "Copied system definitions";
         "User-defined record type definitions";
         "Copied system definitions referring to above record types"
         begin  "User Program"
         end
end.
```

The two lots of system definitions to be copied are given in Appendix A and any examples of programs from now on will omit them.

Restrictions

Because of the method of implementation of SLIM (necessitated by the structure of and code generated by ALGOL W) certain procedures must always be called from the "Outer Program Level". These procedures are:- INITIALISE, WAIT, WAITTILL, SCHEDULE_EVENTS and TERMINATE.

Statements in the "Outer Program Level" are those which are statically nested in the begin - end block surrounding the "User Program", but are not further nested within either:-

(i) a procedure declaration

(ii) a block expression

(iii) a further block with declarations

or (iv) certain kinds of block without declarations, namely one passed as a parameter to a procedure, one which is part of a for statement (e.g. "for I : = 1 until 5 do begin - - end) and one which is added as part of a list of statements, but whose begin - end pair is unnecessary (e.g. A:=0; begin - - end; B:=0).

This restriction is fairly serious and implies several others. As all processes, including the Control Process, must call the above procedures from the "Outer Program Level", they cannot be declared separately as procedures. Transfers of control within a process generally have to be by goto rather than procedure call if the transaction might be delayed while in the procedure. This explains why the procedure SCHEDULE_EVENTS passes as the action to be scheduled, a block of statements not including the restricted ones and ending in a goto statement.

There is a further restriction implied by this and the operation of the ALGOL W compiler on the IBM 360. The compiler creates one segment of code for the "Outer Program Level" and the amount of code allowed per segment is restricted to 4096 bytes. This is fairly large, but as large parts of every process will be in the "outer Program Level", difficulty might be found in a large program. The position can be helped by the sensible use of procedures which do not call the restricted procedures.

3.2 Simple Example

The following example simulates the operation of a shuttle service between London and Glasgow. It is very much simplified, but gives an idea of how a model can be created. There are 3 aeroplanes in the service, each of capacity 150. They each spend 30 minutes in Glasgow picking up passengers, fly to London in 60 minutes, wait in London 30 minutes picking up passengers, fly to Glasgow in 60 minutes and so on. There is therefore an hourly service as each plane takes 3 hours to complete the circuit. An aeroplane is considered as a transaction with a user-defined record, PLANE associated with it. When it is waiting in Glasgow, the global variable GLAS points to this record and similarly when it is in London, LON points to this record.

At the start of the simulation, the 3 planes are created, one hour between each, and after 200 units of time (i.e. minutes), one is in Glasgow one ten minutes out from Glasgow to London and one twenty minutes out from London to Glasgow. This can be thought of as a "random" starting position for the planes. Passengers are then generated for the real simulation run at Glasgow and separately at London.

It is assumed that passengers arrive in groups which cannot be split up (e.g. families or business parties). A group of passengers is represented by a transaction with a user-defined record of type PASSENGERS associated with it. Groups of passengers are generated in Glasgow at random intervals. They each have a random number of passengers and wait until there is a plane in Glasgow which has enough room for the group. They then enter the plane, keeping a note of which plane they are on and wait until this plane in in London. Finally, they leave the plane and the system. A similar sequence happens to passengers going in the opposite direction.

Delaychains and Prompts

It is instructive here to examine the problem of what delaychains to use and where prompts should be inserted.

The aeroplane transactions will only execute time-scheduling statements, as every delay between actions they encounter is of a fixed duration of either 30 or 60 minutes. They therefore do not use delaychains.

However each group of passengers can be delayed by a condition at two points. It will wait until "there is a plane in Glasgow (or London) which has enough room for the group" and after it has boarded this plane, it will wait until "the plane it is on has landed in London (or Glasgow)". When a group of passengers is waiting to board a plane in Glasgow (or London), it can only move immediately after a plane has arrived at that airport. A prompt must therefore be issued by any plane on arrival at an airport. Similarly when a group of passengers is waiting for its plane to land, it can only move immediately its plane has landed, and this again means that a plane must issue a prompt on arrival at an airport.

When a plane issues this prompt, there will therefore be two lots of passengers which the scheduler will now try to move, those waiting to leave the the plane and those waiting to board it. As it is essential that those waiting to leave do so before any others try to board, passengers waiting to leave planes should be given a higher priority.

One global delaychain could be used to hold all delayed groups of passengers. The prompt when a plane arrives at an airport would then refer to this delaychain. However, this would be very inefficient as the arrival of a plane in Glasgow would cause the scheduler to recheck the delaying condition of all groups of passengers waiting to board a plane in London, and also those waiting for another plane to land. It is known that such transactions still cannot move.

Two delaychains could therefore be used, one called "GLASQ" holding both passengers waiting to board a plane in Glasgow and those waiting for their plane to land in Glasgow. The other "LONQ", similarly holds passengers either waiting to board in London or waiting for their plane to land in London. The prompt issued on arrival of a plane in Glasgow will refer to "GLASQ" and similarly to "LONQ" on arrival in London.

It should be noted that the above method is not the most efficient either. On arrival of a plane in Glasgow, a prompt needs to be issued to the passengers waiting to board in Glasgow, but of those waiting to land in Glasgow, it is only those on the actual plane landing that need to be prompted. Passengers on the plane one hour behind the current one would also receive the prompt above. The method used in the program following does not issue prompts to passengers unnecessarily, and uses 5 delaychains. "GLASQ" and "LONQ" hold passengers waiting to board planes at Glasgow and London, and each aeroplane also has a private delaychain which holds passengers on that plane waiting to disembark. On arrival at Glasgow (or London) a plane will therefore issue a prompt referring to both its own delaychain and "GLASQ" (or"LONQ").

It can be helpful in deciding what delaychains to use, if one imagines delayed transactions waiting in physical queues in the real-life system, and creates a delaychain where each queue is. Thus passengers above could be imagined to be queued up at the airports waiting for planes to board, and also queued up on the plane they have boarded, waiting for it to land.

Program

```
1   begin   comment     "copy system definitions";
2           record      PLANE (integer ROOMLEFT; reference(DELAYCHAIN) PLANEQ);
3           record      PASSENGERS (integer NUMBER; reference(PLANE) OWNPLANE);
4           comment     "copy system definitions using PLANE and PASSENGERS";
5           begin       reference (DELAYCHAIN) GLASQ, LONQ;
6                       reference (PLANE) GLAS, LON;
7                       integer R; R:=5769432;
8                       GLASQ:=NEWDELAYCHAIN; LONQ:=NEWDELAYCHAIN;
9                       GLAS:=LON:=null;
10                      INITIALISE;
11                      SCHEDULE_EVENTS(60,3,10, goto PLANE_ROUTE);
12                      WAIT(200);
13                      SCHEDULE_EVENTS(RNEGEXP(1,R),5000,1,goto GLAS_PASS);
14                      SCHEDULE_EVENTS(RNEGEXP(1,R),5000,1,goto LON_PASS);
15                      WAIT(1000);
16                      STOP;
17          PLANE_ROUTE:TRANSACTION(PLANE(150,NEWDELAYCHAIN));
18                                  while   true   do
19                                  begin GLAS:= CURRENT;
20                                        TESTALL(PLANEQ(CURRENT));TESTALL
                                                                  (GLASQ);
21                                        WAIT(30);
22                                        GLAS:=null;
23                                        WAIT(60);
24                                        LON:=CURRENT;
25                                        TESTALL(PLANEQ(CURRENT));TESTALL
                                                                  (LONQ);
26                                        WAIT(30);
```

```
27                                  LON:=null;
28                                  WAIT(60)
29                              end;
30                  GLAS_PASS:TRANSACTION(PASSENGERS(1+RNEGEXP(.5,R),));
31                          WAITTILL(GLAS≠null and ROOMLEFT(GLAS)≥NUMBER
                                                    (CURRENT),GLASQ);
32                          ROOMLEFT(GLAS):=ROOMLEFT(GLAS)-NUMBER(CURRENT);
33                          OWNPLANE(CURRENT):=GLAS;
34                          PRIORITY(CURRENTEVNOTICE):=2;
35                          WAITTILL(LON=OWNPLANE(CURRENT),PLANEQ(GLAS));
36                          ROOMLEFT(LON):=ROOMLEFT(LON)+NUMBER(CURRENT);
37                          TERMINATE;
38                  LON_PASS:TRANSACTION(PASSENGERS(1+RNEGEXP(.5,R),));
39                          WAITTILL(LON≠null and ROOMLEFT(LON)≥NUMBER
                                                    (CURRENT),LONQ);
40                          ROOMLEFT(LON):=ROOMLEFT(LON)-NUMBER(CURRENT);
41                          OWNPLANE(CURRENT):=LON;
42                          PRIORITY (CURRENTEVNOTICE):=2;
43                          WAITTILL(GLAS=OWNPLANE(CURRENT),PLANEQ (LON));
44                          ROOMLEFT(GLAS):=ROOMLEFT(GLAS)+NUMBER (CURRENT);
45                          TERMINATE
46                  end
47  end
```

Prelude

Lines 1 to 4 constitute the prelude. The system procedures in Appendix A are copied there and the two types of record used to hold private variables of the two transaction types are defined. In the user program, a transaction can be considered as either a "group of passengers" or an "aeroplane" and has a record"

of type PASSENGERS or PLANE associated with it.

Control Process

The Control Process consists of lines 5 to 16. When execution of the program is started, this process is started first.

Lines 5,6,8 and 9 define and initialise the representations of Glasgow and London airports in the model and the delaychains for passengers waiting to board planes at the two airports.

Line 7 defines and initialises an integer variable R which is used with the integer procedure RNEGEXP to generate pseudo-random numbers. The facilities available for the generation of random numbers are not fully explained until Chapter IV.

Line 10 initialises the system.

Lines 11 and 12 create 3 aeroplanes; after 1 hr, 2 hr. and 3 hrs. On creation, each starts executing at "PLANE_ROUTE" and after 200 minutes, the three aeroplanes will all be in different positions round the route.

Lines 13 and 14 at time 200 schedule 5000 groups of passengers both from Glasgow and London. There is a random delay between creation of each group of passengers and they start to execute either at "GLAS_PASS" or "LON_PASS" with a priority of 1.

Lines 15 and 16 allow the planes and passengers to interact for 1000 minutes. At time 1200, the control process halts the whole program. In a model which is to be of any practical use, statistics would be collected by the different transactions and the control process would output them before executing "STOP".

Aeroplanes

The process for each of the 3 aeroplanes consists of lines 17 to 29.

Each aeroplane associates a PLANE record with 150 empty seats and an initialised private delaychain.

It then loops round lines 19 to 29 until the simulation is ended by the control process.

In lines 19 to 20 it lands at Glasgow and issues prompts to the scheduler telling it that the passengers on board the plane waiting to leave it and the passengers waiting at Glasgow airport for a plane to board might be able to move now.

In lines 21 to 23 it waits for 30 minutes in Glasgow, leaves Glasgow and waits a further 60 minutes which is the time taken to fly to London.

Lines 24 to 28 are the same as 19 to 23 only for the return flight.

Passengers

Passengers flying from Glasgow to London are generated at line 13 and execute the process from lines 30 to 37. Passengers travelling from London to Glasgow execute a similar process from lines 38 to 45.

Each group of passengers travelling from Glasgow associates with itself a new PASSENGERS record whose field NUMBER is the random number of passengers in the group.

In line 30 the group of passengers waits until there is a plane in Glasgow with as many empty seats as there are members of the group. If it cannot board the plane immediately, it waits in the delaychain, GLASQ with priority 1. All groups of passengers waiting in GLASQ are kept in order of arriving so the scheduler will allow them to board in this order when an aeroplane has arrived.

If there are more passengers waiting than seats, the earliest arrived passengers will board and the rest will automatically wait in GLASQ until another plane lands.

In lines 32 and 33, the group boards the plane, keeping a note of which plane has been boarded.

The group of passengers next waits in the plane's private delaychain until it is in London. The group has priority 2 here so that when the plane arrives in London, all passengers will leave the plane before any London passengers try to board.

In lines 36 and 37, the seats used by the group are returned and then they leave the system.

3.3 Details of System Structures and Variables

Eventnotice

The basic system-defined part of a transaction is a record of type EVENTNOTICE. This has the following fixed format:-

record EVENTNOTICE (reference (user-defined records) TRANS; reference (EVENTNOTICE)LPOINTER,RPOINTER,SUPERCEDING; reference (EVENTNOTICE, DELAYCHAIN)APOINTER,BPOINTER,;logical ACTIVE,FUTUREEV; integer TIME, PRIORITY,RET,THUNK,NUMBER,SUPBY)

The field TRANS can point to an optional user-defined record which will then be associated with the transaction. The use of the remainder of the above fields is explained as they are used in the following sections.

It should be noted here however, that in this chapter, we refer to the fields RET and THUNK of EVENTNOTICES as holding "representations" of the "next action" or "delaying condition" or "delay between the creation of transactions". The actual form of these "representations" is not explained until Chapter III. However, although defined as integer fields, it must be mentioned that they are not used as integers, but are used by low-level language procedures and hold machine addresses.

Future Events Chain

The "FUTURE EVENTS CHAIN" holds all transactions which are time-scheduled. It therefore holds transactions which have been:-

(1) created by SCHEDULE_EVENTS but not yet moved

(2) delayed by the WAIT procedure.

The Future Events Chain is a linked list of eventnotices, ordered by the value of simulated time at which they are scheduled to move. (This is kept in

the field TIME of each eventnotice). Within each value of simulated time, the eventnotices are ordered by priority (field PRIORITY of each eventnotice) and, within each priority class, by the order that they were delayed or scheduled in. The first eventnotice in the list is therefore scheduled for the earliest value of simulated time, of these has the highest priority and of these was the earliest delayed. The two pointer fields, LPOINTER and RPOINTER of each eventnotice in the list are used to doubly link the list, pointing to the previous and next eventnotices in the list, respectively. The field LPOINTER of the first eventnotice of the list and the field RPOINTER of the last are both null.

A reference (EVENTNOTICE) variable, FIRSTFE points to the first eventnotice in the list or, if the list is empty, it is null. This eventnotice must must always be moved before any other time-scheduled transaction and is therefore the only time-scheduled transaction that the scheduler needs to consider at any time when it is deciding which transaction to give control to.

It should be noted that, whereas in GPSS time-scheduled transactions scheduled to be moved at the current value of simulated time are removed from the Future Events Chain, they are kept on it in SLIM.

All transactions on the Future Events Chain have their logical field FUTUREEV true. Their field RET holds a "representation" of the transaction's "next action".

A call of SCHEDULE_EVENTS can ask for the creation of more than one transaction. If so, only one is created and placed on the Future Events Chain, and the number still to be created is assigned to the NUMBER field of its eventnotice. The field THUNK holds a "representation" of the parameter "DELAY" which was passed by the call of SCHEDULE_EVENTS, so that it can be recalculated later (if "DELAY" was a random variable, we would want a different random "DELAY"

to be produced between each transaction's creation). When this eventnotice is first moved from the Future Events Chain by the scheduler, a copy eventnotice can be created with its NUMBER field decreased by one. The "DELAY" can be recalculated and the copy scheduled after this time, before control is given to the original eventnotice. It can therefore be seen that transactions scheduled by WAIT on the Future Events Chain must have their NUMBER field zero so that the scheduler can tell when a copy has to be made.

Delayed Events Chain

The "DELAYED EVENTS CHAIN" holds all condition-scheduled transactions or, in other words, all transactions delayed by a WAITTILL procedure.

As with the "Current Events Chain" in GPSS, where there are also priorities, the Delayed Events Chain in SLIM actually consists of a set of "Priority Chains", one for each possible priority value, 0 to 100. The eventnotices of all transactions which have been delayed by a WAITTILL are kept in the appropriate priority chain in the order of their being delayed, and are doubly linked to form this list using their fields LPOINTER and RPOINTER. The field LPOINTER of the first and RPOINTER or the last eventnotice in each Priority Chain are null.

There are two reference (EVENTNOTICE) arrays, FIRSTDE(0::100) and LASTDE (0::100) which point to the first and last delayed eventnotices in each Priority Chain (priority is index of each array). If the Priority Chain for priority I is empty then FIRSTDE(I) = LASTDE(I) = null.

All transactions on the Delayed Events Chain have their logical field FUTUREEV false. Also all transactions have associated with them a condition which they are waiting to become true. The field THUNK of an eventnotice in the Delayed Events Chain holds a representation of its next action.

When a transaction executes a WAITTILL procedure, it is placed at the end of the appropriate Priority Chain of the Delayed Events Chain and is marked "active" (i.e. the field ACTIVE of the eventnotice is assigned the value true). During the same value of simulated time, the scheduler will test the transaction's condition and if it cannot then move, it is marked "passive" (i.e. field ACTIVE is made false). If, later still, the scheduler receives a "prompt" referring to the eventnotice, it will again be marked "active" and the condition will again be tested before simulated time is updated.

There are therefore two kinds of eventnotice on the Delayed Events Chain, "active" and "passive" ones. When the scheduler is looking for a new transaction to give control to, it completely ignores "passive" ones.

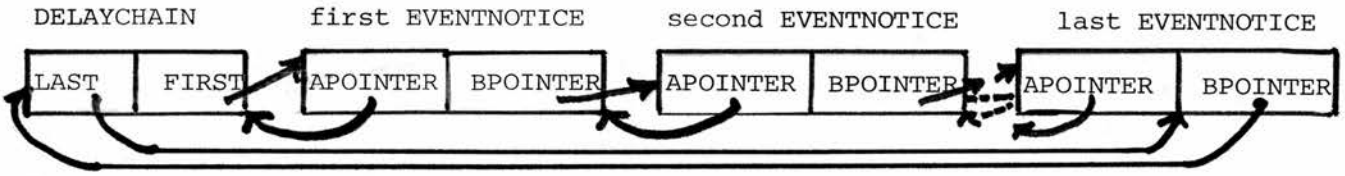
Conceptually, The Delayed Events Chain is best considered as one linear chain with the first eventnotice in the Priority Chain (I-1) following immediately after the last in Priority Chain I. (It is only structured differently, for efficiency of inserting new eventnotices). The reference (EVENTNOTICE) variable FIRSTACTIVEDE points to the first "active" eventnotice in the Delayed Events Chain when it is thought of like this. This is the first condition-scheduled transaction which the scheduler can try to move, being the highest priority, earliest delayed, "active" eventnotice. (At this stage the scheduler has not actually tested its condition and so does not know whether it will definately be able to move). If there are no "active" transactions on the Delayed Events Chain, FIRSTACTIVEDE is null.

A strict ordering relationship is maintained in each Priority Chain with the TIME field of each eventnotice on it, so that TIME (first eventnotice on the Priority Chain)<TIME(second eventnotice)..... <TIME (last eventnotice). This use of the TIME field has no connection with simulated time.

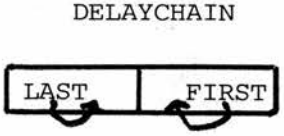
Delay Chains

Every condition-scheduled transaction on the Delayed Events Chain is also linked onto a user-defined "DELAYCHAIN" specified in the WAITTILL statement which delayed it.

A Delaychain is a linear list of eventnotices which is ordered by priority and within each priority class by order in which the transaction was delayed. At the head of each Delaychain is a record of type DELAYCHAIN (reference (EVENTNOTICE,DELAYCHAIN) FIRST,LAST; integer HELDUP). The field HELDUP of this record contains the number of eventnotices on the Delaychain. The fields FIRST and LAST of the DELAYCHAIN record and the fields APOINTER and BPOINTER of all eventnotices on the Delaychain are used to link the Delaychain together as follows:-



If there are no eventnotices on the Delaychain, the form is:-



This form means that from a pointer to any condition-scheduled eventnotice, it is possible to find which Delaychain it is on by following pointers.

Other Global System Variables

SIMTIME is an integer variable which holds the current value of simulated time. The scheduler maintains this variable.

CURRENTEVNOTICE points to the EVENTNOTICE record of the transaction in control at any time.

CURRENT points to the user-defined record that is associated with the transaction in control. Therefore CURRENT = TRANS (CURRENTEVNOTICE).

TEMP is a reference (EVENTNOTICE,DELAYCHAIN) variable and is used by the system procedures as a temporary workspace, It has no standard use.

DATASEG is defined as an integer variable, but is only used by the low-level procedures described in Chapter III and is used by them to hold a machine address.

3.4 Details of Basic System Procedures

The need for procedures in a low-level language

It was mentioned in section 3.0 that because the SLIM system was embedded in ALGOL W, resort has had to be made to a low-level language to implement both the storing of procedures for later execution and transfers of control between processes.

Procedures need to be stored in three circumstances:-

(1) On a call of WAITTILL, the transaction which is delayed must hold a representation of the delaying condition so that it can be re-checked later. This representation is kept in the pair of locations, DATASEG and the field THUNK of its eventnotice. The pair of locations can together be considered as a "reference (logical procedure)" variable. Such a type is not available in ALGOL W, so low-level procedures have to be used to store a logical procedure in this variable and later re-evaluate the logical procedure.

(2) On a call of SCHEDULE_EVENTS requiring the creation of more than one transaction, only one is created at the time and this eventnotice holds a representation of the delay between subsequent creations. The field THUNK of the eventnotice and DATASEG are again used to hold this representation and similarly to (1), can be thought of as a "reference (integer procedure)" variable. Again low-level procedures are required to store an integer procedure in the variable and later re-evaluate it.

(3) Also on a call of SCHEDULE_EVENTS, the created eventnotice must keep a representation of its "next action". This is given by the procedure

parameter passed by SCHEDULE_EVENTS , and DATASEG and the eventnotice field RET between them can be considered here as a "reference(procedure)" variable in which this procedure parameter can be stored by a low-level procedure. When this eventnotice's next action has to be executed later, this stored procedure is executed by another low-level procedure.

The representation of "next action" used in normal transfers of control between transactions is illustrated using the following piece of program:-

```
        ------
        ------
        WAIT(50);
LABEL:  ------
        ------
```

A transaction is held up by the WAIT procedure call. A representation of the "next action" must be stored in the eventnotice so that, when it is given back control after other transactions have been moved, it will start at LABEL. One way of doing this would be for the WAIT call to pass a procedure parameter, "goto LABEL" and for this to be stored as the representation in the same way as the "next action" of the newly created eventnotice is stored by SCHEDULE_EVENTS. This however means that every self-scheduling procedure would have to pass such a procedure parameter and every action would have to start with a label. This is unnecessary because the WAIT or WAITTILL procedure can itself find out the address of the statement after its call, LABEL, by dropping to a low-level language. This address, LABEL, is stored in the field RET of the eventnotice as a representation of the next action and there is another low-level procedure to cause execution to start from here when it is

required later to give this transaction control.

The actual low-level procedures are briefly described as they are encountered in this section. A full description of them is given in Chapter 4.

Procedures to add to and remove from Chains

(1) procedure FMERGE (reference (EVENTNOTICE)value EV)

This procedure inserts the eventnotice EV in the Future Events Chain in the correct position as specified by its TIME and PRIORITY fields, updating the pointer to the start of the Delayed Events Chain, FIRSTFE, if necessary.

(2) procedure UNHOOKFE (reference (EVENTNOTICE) value EV)

This procedure removes a general eventnotice EV from the Future Events Chain. It makes the links of the chain from either side of EV point to the eventnotice on the other side of EV. If it was the first on the chain, FIRSTFE is updated.

When it is known that it is the first eventnotice that is to be removed from the Future Events Chain, a simpler piece of code is used instead of calling UNHOOKFE:-

```
FIRSTFE:=RPOINTER(FIRSTFE);
If FIRSTFE ≠ null then LPOINTER(FIRSTFE):=null
```

(3) procedure DMERGE(reference(EVENTNOTICE)value EV; reference(DELAYCHAIN)value DC)

This procedure links EV in the appropriate place in both the Delayed Events Chain and the Delaychain DC.

It is first placed at the end of the Priority Chain in the Delayed Events Chain given by its PRIORITY field. The ordering relationship between the TIME fields in each Priority Chain is maintained by assigning to EV's TIME field zero if it is the only eventnotice on the Priority Chain or otherwise one greater than the TIME field of the previous last eventnotice.

EV is next added to the Delaychain DC.

DMERGE finally marks EV as "active" and updates FIRSTACTIVEDE if EV comes before FIRSTACTIVEDE in the Delayed Events Chain.

(4) procedure UNHOOKDE (reference(EVENTNOTICE)value EV)

This procedure removes EV from the Delayed Events Chain and also from the Delaychain it is on.

UNHOOKDE assumes that EV is marked "passive" and therefore that it does not need to update FIRSTACTIVEDE.

It should be noted here that UNHOOKDE has a side-effect of making TEMP point to the DELAYCHAIN record from which EV was removed. This is used in a procedure in Chapter 5.

System Procedure to find the next Transaction to be moved

The procedure "GETNEWFIRSTEV" finds which of the scheduled transactions should be moved first, makes CURRENTEVNOTICE point to its eventnotice and also makes CURRENT point to any associated user-defined record. If necessary, it also updates the simulated time. At any point it has only two scheduled eventnotices to consider, FIRSTFE which is the first time-scheduled transaction that can be moved, and FIRSTACTIVEDE which is the first condition-scheduled transaction which it can try to move. If GETNEWFIRSTEV decides to try to move the latter first, it might still not have its delaying condition satisfied. If this is the case, it is marked "passive" again, FIRSTACTIVEDE is updated to the next "active" eventnotice in the Delayed Events Chain, and the comparison between FIRSTFE and FIRSTACTIVEDE is made again. The procedure used to update FIRSTACTIVEDE above is called NEWACTIVE.

procedure NEWACTIVE

This procedure expects the eventnotice pointed to by FIRSTACTIVEDE to have been marked "passive". While FIRSTACTIVEDE has not reached the end of the Delayed Events Chain and does not point to an "active" eventnotice, it keeps on being moved to point to the next eventnotice in the Delayed Events Chain. This is either the following eventnotice in the same Priority Chain, or, if there are none, the first eventnotice in the next lower Priority Chain that is not empty. If it has reached the end of the Delayed Events Chain without finding an "active" eventnotice, FIRSTACTIVEDE is made null.

The actual procedure, GETNEWFIRSTEV can now be fully described:-

procedure GETNEWFIRSTEV

The first part of the procedure finds the eventnotice which is to be moved first. The logical variable FOUNDEVENT gives a value true if this is a condition-scheduled eventnotice and false if it is a time-scheduled one.

To do this, it loops round, marking FIRSTACTIVEDE as "passive" and updating it as long as:-

(1) There is at least one "active" condition-scheduled eventnotice left (i.e. FIRSTACTIVEDE≠null), and

(2) An attempt should be made to move FIRSTACTIVEDE before FIRSTFE, and

(3) FIRSTACTIVEDE's delayed condition is found to be not yet satisfied.

The condition (2) can be expanded into:-

(i) no time-scheduled eventnotices are left

or (ii) FIRSTFE is not scheduled for the current value of SIMTIME

or (iii) FIRSTFE is not marked for immediate execution (TIME SIMTIME, see Chapter 5) and its priority is not higher than FIRSTACTIVEDE's.

To evaluate condition (3), CURRENTNOTICE and CURRENT must first be set up to point to FIRSTACTIVEDE's transaction records as the delaying condition might be phrased in terms of them. The low-level language procedure, READY then evaluates the representation of the delaying condition held by the eventnotice and gives as a result the evaluated condition.

If the scheduler finds conditions (1) and (2) are satisfied and then tries to evaluate (3) it, as a side-effect of the evaluation, makes FOUNDEVENT true. If FIRSTACTIVEDE is then found not to be able to be moved, FOUNDEVENT is again made false. Therefore, when the loop above is finally terminated by the complex condition above being false, it can be false because one of the first two conditions above are false which means that the FIRSTFE should be moved and FOUNDEVENT will be false. Otherwise, if the third condition is false, this means that FIRSTACTIVE can actually be moved and FOUNDEVENT will be true.

A If a condition-scheduled eventnotice is to be moved, it is marked "passive", FIRSTACTIVEDE is updated and the eventnotice is unhooked from the Delayed Events Chain and the Delaychain it is on, CURRENTEVNOTICE is left pointing to the eventnotice and CURRENT pointing to its associated record.

B If a time-scheduled transaction is to be moved, it must be the one pointed to by FIRSTFE, so CURRENTEVNOTICE and CURRENT are set to point to its eventnotice and associated record. Simulated time is then updated if necessary and the eventnotice found is removed from the start of the Future Events Chain. If there is no time-scheduled eventnotice that can be moved either (FIRSTFE = null), an error message is printed and the simulation stopped.

If the field NUMBER of the eventnotice found is greater than zero, this means that it has been created by a call of SCHEDULE_EVENTS specifying that more than one transaction should be created. A new eventnotice is therefore created with its NUMBER field one less and other fields copied. The representation of "delay between creations" stored in the THUNK field and DATASEG is evaluated by the low-level procedure, EVALDELAY and the new eventnotice is linked to the Future Events Chain after this delay. The NUMBER field of the original eventnotice can then be zeroed.

Scheduling Procedures

Five low-level language procedures are called from the scheduling procedures. Their effects will first be briefly described:-

(i) SAVERETURNADD - called from WAIT and stores a representation of the next action in the executing eventnotices.

(ii) REPLACERETADD - called from WAIT, WAITTILL and TERMINATE, and transfers control to the next action of the next transaction to which control is to be given.

(iii) SAVETHUNKRE - called from WAITTILL and stores a representation of the next action and delaying condition in the executing transaction.

(iv) STOREDESTDELAY - called from SCHEDULE_EVENTS and stores a representation of the next action and delay between creation of eventnotices in the newly created eventnotice.

(v) LEVEL - used by all scheduling procedures and gives as a result an integer which represents the program level the scheduling procedure was called from. (DATASEG holds a representation of the "outer program level").

(1) procedure WAIT (integer value T)

A test is first made that the procedure has been called from the outer program level. WAIT then stores a representation of the transaction in control's next action, marks it as a "future event" (i.e. field FUTUREEV is made true) and puts it in the Future Event Chain after a delay T. The next transaction to be given control is then found, and control is transferred to its next action.

(2) procedure WAITTILL(logical value CON; reference (DELAYCHAIN)value DC)

A test is made first that WAITTILL has been called from the outer program level. Next, a representation of CON and the next action are stored in the transaction in control. The eventnotice is then marked a "delayed event" (i.e. field FUTUREEV is made false) and the eventnotice is linked to the Delayed Events Chain and the Delaychain, DC, and marked "active". The next transaction to be given control is then found and control is transfered to its next action.

(3) procedure SCHEDULE_EVENTS(integer value DELAY,NO,PR;procedure DEST)

A test is made that the procedure has been called from the correct level. Then a new eventnotice is created with priority PR and its field NUMBER set to the number of transactions still to be created, (NO-1). A representation of this transaction's next action and delay between creations, is stored with it, and it is linked to the Future Events Chain after a delay, DELAY. Control is returned to the transaction which executed the SCHEDULE_EVENTS.

(4) procedure TERMINATE

This procedure has a large part which is only applicable to the facilities introduced in Chapter IV. With the facilities introduced so far, however, the field SUPERCEDING of all eventnotices will always be null. The eventnotice in control is marked "terminated" (field RET=0), the next transaction to be given control is found and control is returned to its next action. The transaction which executed the TERMINATE is therefore not scheduled and is normally lost to the system. A test is also made here that the procedure has been called from the outer program level.

Prompts

All a prompt does is to mark either the first or all the eventnotices on a specified Delaychain as "active". However, if the first eventnotice on the Delaychain is "before" FIRSTACTIVEDE on the Delayed Events Chain, FIRSTACTIVEDE must also be updated. This notion of "before" is that either there are no active transactions at the time, or that the first eventnotice in the Delaychain has a higher priority than FIRSTACTIVEDE's or that their priorities are the same but it comes earlier than FIRSTACTIVEDE on the Priority Chain. (This comparison of places on a Priority Chain is made using the TIME fields which shows the use of the ordering relationship emposed on them).

It should be noted that even when all eventnotices on a Delaychain are to be marked "active", only the first needs to be compared with FIRSTACTIVEDE as the first on the Delaychain comes "before" all the rest of the Delaychain.

(1) procedure TESTFIRST (reference (DELAYCHAIN) value DC)

If there are any transactions held up on DC, and the first transaction on it is not already marked "active", it is marked "active" and , if it is "before" FIRSTACTIVEDE, FIRSTACTIVEDE is updated to point to it.

(2) procedure TESTALL (reference (DELAYCHAIN) value DC)

If there are any transactions held up on DC, FIRSTACTIVEDE is updated to point to the first if the first is "before" it, and all eventnotices on DC are marked "active".

Other Procedures

(1) procedure INITIALISE

This procedure initialises simulated time to zero and the Future Events Chain and Delayed Events Chain to empty. It then gives DATASEG a representation of the outer program level by assigning to it the low-level procedure, LEVEL, and creates an eventnotice to execute the Control Process.

(2) procedure TRANSACTION(reference (user-defined record type) value NEWTRANS)

This procedure links the given user-defined record, NEWTRANS, with the executing transaction by making both CURRENT and the executing event-notice's field TRANS point to it.

(3) reference (DELAYCHAIN) procedure NEWDELAYCHAIN

This procedure gives as a result a pointer to a newly created Delaychain which is initialised as empty by its field HELDUP being made zero and the fields FIRST and LAST pointing to the DELAYCHAIN record itself.

CHAPTER 4

LOW-LEVEL LANGUAGE PROCEDURES IN "SLIM"

4.1 Introduction

It was explained in the last chapter why resort has to be made in SLIM to a low-level language. What was required was a method of storing a representation of the delaying condition of a transaction held up by a WAITTILL, of the delay between creations of transactions just created by SCHEDULE_EVENTS, and of the next action of all scheduled transactions. The latter were seen to be split into two cases, transactions just scheduled by SCHEDULE_EVENTS where the next action is a procedure parameter passed to SCHEDULE_EVENTS, and transactions scheduled by self-scheduling statements (i.e. WAIT or WAITTILL) where the next action is of a different form.

The second representation of next action is the address of the statement following the call of the self-scheduling procedure which delayed the transaction. When a call is made of a self-scheduling procedure, a return address is passed to the procedure which is the address of the machine instruction after the point of call. This is to enable the procedure on normal exit, to return control to this point. The return address can be retrieved from where it is stored in the runtime stack by a low-level language procedure and stored as the

representation of next action. When, later on, a decision is made to return control to this transaction, this decision will also be made from within a call of either WAIT, WAITTILL or TERMINATE made by another transaction. The return address associated with this call will then be replaced by the next action of the transaction now given control, by another low-level language procedure. On "normal exit" from the self-scheduling procedure, control will then be passed to the appropriate next action. The two procedures mentioned here both use the structure of the run-time stack to retrieve and replace the return addresses.

The representation of delaying condition, delay between creations and the first type of next action are stored in what could be thought of as reference (logical procedure), reference (integer procedure) and reference (procedure) variables. As ALGOL W does not have these types, they must be simulated by low-level language procedures which store and execute the appropriate procedures in these variables. The entities which are to be saved in these variables are always first passed as parameters to a scheduling procedure and then stored by a low-level language procedure. The condensed form by which parameters are passed will be seen to be also suitable as a representation of the parameter for storing. The method normally used for the evaluation of parameters within an ALGOL W procedure can also be used by a low-level procedure to evaluate the stored representations. This is dependent on the methods used to pass and evaluate parameters in ALGOL W.

It can be seen therefore that all the low-level procedures used depend on the type of code generated by the ALGOL W compiler, so the structure of compiled ALGOL W programs will be described first. Then the actual low-level language procedures themselves will be described.

4.2 Structure of running ALGOL W programs

The ALGOL W compiler was written at Stamford University and this version is roughly described in [9]. However the version used here was modified from this, mainly at Newcastle University. The code generated is therefore very different from that described in [9].

The ALGOL W compiler is not intended to produce optimal code, but is more designed to facilitate debugging. If there is a large amount of optimisation perfored by a compiler, it becomes difficult to give detailed run-time debugging messages, because the structure of the program might be changed by the compiler during optimisation. Therefore the ALGOL W compiler produces standard code for the different syntactic structures.

As the compiler is for use on an IBM system 360, the code produced is in "Program Segments" which are each of size less than 4096 bytes, this being the largest displacement from a base and index register allowed in a machine instruction. These Program Segments are separately loaded and linked before execution. As well as the compiled Program Segments, certain "System Segments" are also loaded, which hold system routines such as garbage collection and certain constants. The running program has also a fairly large area of free space which it dynamically allocates from the bottom as a stack of "Data Segments". The rest of this area of free space is used as a heap for keeping dynamically allocated ALGOL W records. The operation of this heap does not concern us here, but checks have to be made when a new Data Segment is allocated to make sure that it does not overwrite part of the heap, in which case, a garbage collector would have to be called and the heap compacted.

The IBM System 360 has 16 general purpose registers, on which r1 to r15 can be used as base or index registers and r0 to R15 for arithmetic operations. The conventions for their use in ALGOL W programs are explained later. There are also 4 floating point registers, f0, f2, f4 and f6, which can be used for real arithmetic operations.

A description of the machine and the instructions that can be used in it is given in [10].

Program Segments

The compiler creates one new program segment for each of the following constructs:-

a) block expression with or without declarations

b) block with declarations

c) block without declarations which is either passed as a parameter to a procedure, or is included as part of a list of statements purely for visual grouping (e.g. a:=0; begin b:=0; c:=0 end; d:=0;)

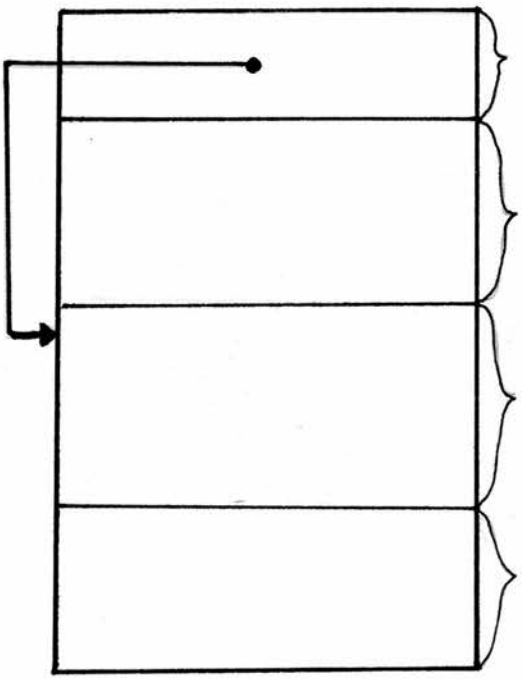
d) procedure or <type> procedure declaration, with or without formal parameters. If the procedure body is a block or block expression, this begin - end pair with declarations is considered to be part of the procedure declaration and is in the same program segment. (e.g. one segment for both:-

(i) procedure EX ; A:=0; and
(ii) real procedure EX (real value X);
begin integer A,B; A:=B:=1; (X+A+B) end;)

If, during execution of the program segment connected with one of the above constructs there is another such structure either nested within it, in the case of blocks, or called from it in the case of procedures, at this point there is placed a call of the program segment connected with the inner structure. In this way, all the program segments which

make up the program are connected together by calls.

A program segment has the form:-



Branch instruction to start of machine code

Certain fixed information about segment, including Constant Table, Branch Table and "REL REF" (see later)

Machine Code

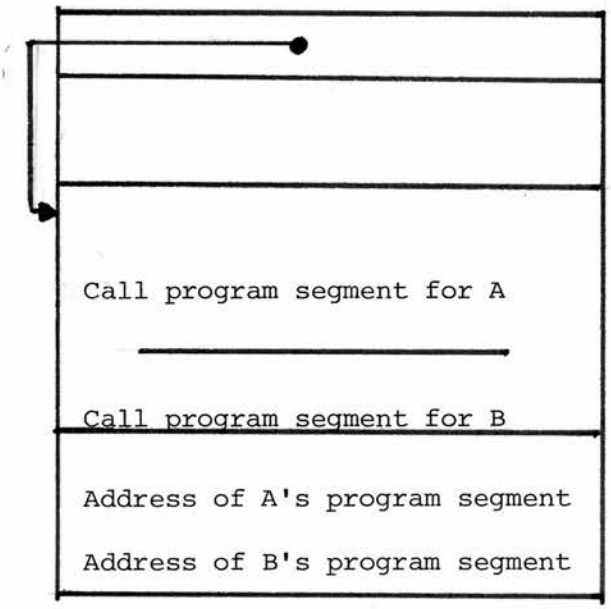
Address of all other program segments which it can call (filled in by the loader)

Thus the program

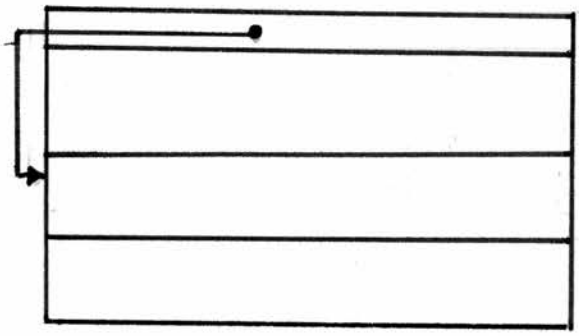
```
begin integer X, Y;
      procedure A;
          begin real R;
          end;

      A;
  B : begin integer Z;
          A;
      end
end
```

will be compiled into 3 program segments:-

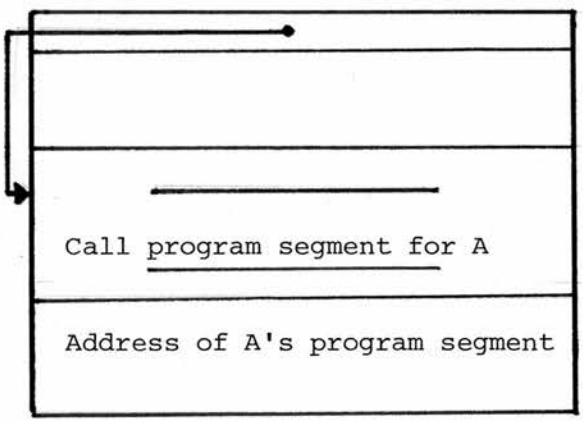
Program segment for outer block



Program segment for procedure A



Program segment for block labelled B



The 6th word of each program segment is called "REL REF" and consists of 2 half-words which are used by the ALGOL W garbage collector. If there are any reference arrays or reference variables declared in the block or procedure, these 2 half-words contain the addresses of the first reference array descriptor and the first reference variable in any data segment which "executes" this program segment relative to the start of the data segment (see next section). If there are no reference arrays or reference variables, the 2 half-words are zero.

Data Segments

Data segments are dynamically created and destroyed at run-time.

There are similarities between the concepts of a transaction and a Data Segment. A transaction conceptually executes a set of actions, but can be delayed for a while in the middle of them while other transactions execute some of their actions. When it finally regains control, it continues executing its actions at the point where it left off. Two transactions can be simultaneously in the process of executing the same set of actions, though, of course, at most one can be "in control" at any time. A transaction consists of a system defined part and a user defined part.

In the same way, a data segment can be thought of as conceptually executing a program segment. It can be delayed for a while if execution is passed by a call to either an inner block, or a procedure. In this case, a new data segment will be created which will execute the program segment of the inner block or procedure. When return is eventually made to the original data segment (by the inner block or procedure's execution being completed) it starts executing again at the point where it left off before (corresponding to the statement after the "call"). Two data segments can be simultaneously in the process of executing one program segment, though only one can be actually "in control" at any one time. This corresponds to a recursive procedure calling itself, in which case there will be a second data segment created which will also execute the recursive procedure. A data segment has a part which is only used by the system and is invisible to the user, and a part which is user-defined, consisting of the variables declared in the block or procedure it corresponds to.

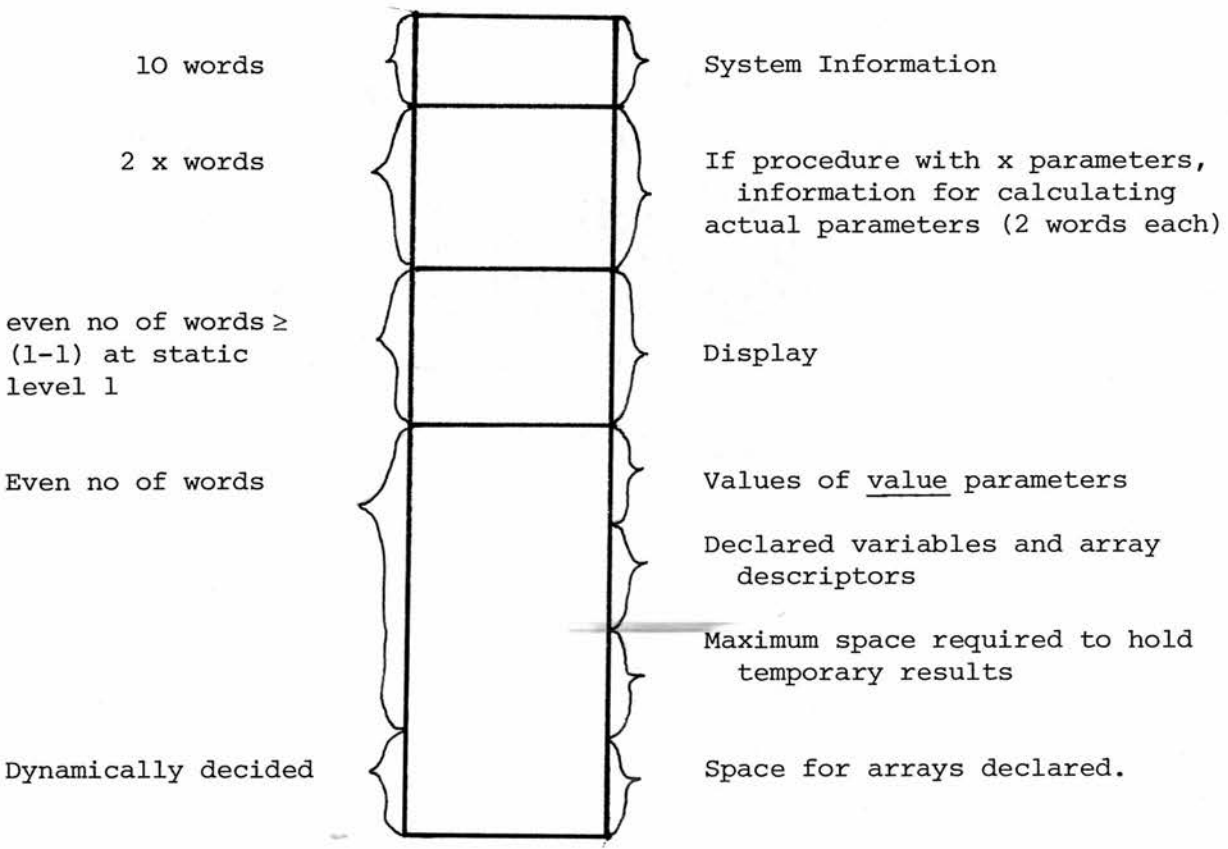
However, when a data segment executes part of its program segment which calls a new program segment, and a new data segment is created to execute this, it will not be given back control until the called data segment has completely finished executing its program segment and destroyed itself. With transactions, control could be passed back and forward between the two without either being executed to completion. As the called data segment must be destroyed before the calling can be given back control and destroyed, the data segments can be kept on a stack with each called data segment immediately above the one that called it. The data segment "in control" at any point will be on the top of the stack, and it is the only one that can be destroyed. Also, each time a program segment is called, the new data segment created to execute it can be placed on top of the stack.

When a data segment, DS1, executes code in its program segment, PS1, to call another program segment, PS2, it can pass parameters to the data segment DS2 which will execute PS2. A representation of each parameter is passed which, if the actual parameter is a constant or simple variable, consists of a pointer to the constant or simple variable. If the parameter is more complex, the representation consists mainly of a pointer to a "thunk". This "thunk" can be considered to be a separate program segment, which has only the "machine code" field, but in actual fact is a part of PS1 which cannot be executed by DS1. When DS2 wishes to evaluate this parameter, it creates a new data segment to evaluate it by executing the thunk. Thus, a data segment can either execute a program segment or a thunk.

In SIMULA 67, this comparison between transactions and data segments is extended by making a special kind of data segment, between instances of which control can pass without either being executed to completion. These data segments can execute "classes" and cannot be kept on a stack. They

can be considered as transactions in the language.

The format of a data segment in ALGOL W is as follows:-



The four sections will now be described.

Variables declared in block or procedure

In addition to the simple variables declared in the block or procedure being executed, there are also three more types of storage in the last section:-

1) If a procedure is being executed which has been passed value parameters, these parameters are considered as being declared as local storage but are initialised on entry to the procedure to the evaluated actual parameters.

2) Temporary storage might be required to hold temporary results during the evaluation of expressions. The maximum amount that can be required at any point in the data segment's life will be allocated.

3) If arrays have been declared, the amount of storage required for each might not be known until run time. If 2 such arrays have been declared, it will not be known where the storage for the second starts until runtime. Therefore the storage for arrays is placed at the end of the section, and fixed size "array descriptors" for each are placed in the second field. These array descriptors at run-time are assigned, among other things, a pointer to the actual storage area for the array.

This section is the only one whose information can be used by the programmer directly.

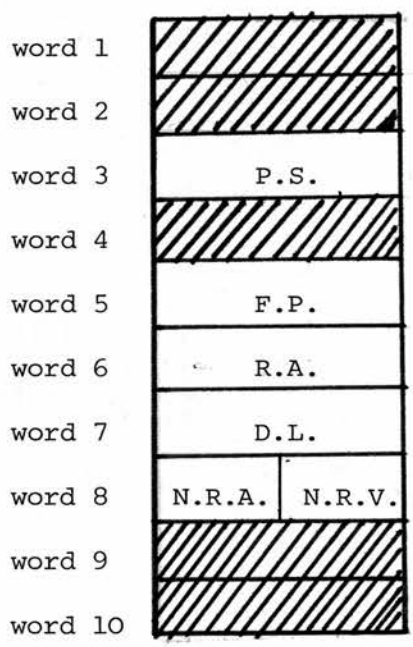
Parameters

When one data segment passes parameters to another which it calls, it puts a representation of each actual parameter in a pair of words in the second section of the called data segment. If the actual parameter is a constant or a simple variable, the first of the pair of words holds a pointer to the constant or variable. As an address is smaller than

the word length of the IBM 360/44, the first bit of the word is zero. The second word is not used. If the actual parameter is more complex, the address of the thunk to evaluate the actual parameter is put in the first word. This time, the first bit of the word is set to 1 so that 2 types of representation can be distinguished. (This can be done by an arithmetic comparison as a 1 in the first bit makes the word appear to be a negative integer.) The second word in this case points to the calling data segment because when the thunk is being evaluated the environment of the calling data segment must be restored. (Parameters must be evaluated in the context of their point of call.)

System Information

Of the system information in the first section of a data segment, the parts relevant to this thesis are:-



P.S. is a pointer to the program segment executed by this data segment.

All program segments start execution by being called. Corresponding to this call is a return address, stored in R.A., which is the point the calling data segment had reached in its program segment, and which is returned to when the called data segment finishes execution and is destroyed.

The dynamic link, D.L., is a pointer to the calling data segment and is therefore a pointer to the data segment immediately below this one on the stack.

The forward pointer, F.P., is similarly a pointer to the next data segment on the stack (at the time of the data segment's creation, this will be the top of the stack).

All data segments on the stack are therefore doubly linked by F.P. and D.L.

If there are any reference variables or reference arrays declared or used as value parameters in the data segment, NRV and NRA hold the number of each. All such reference variables are kept in adjacent locations of the data segment and similarly with all descriptors for reference arrays. The field "REL REF" of the program segment being executed will hold the relative addresses of the first of each in the data segment, so the garbage collector will then be able to find all reference variables and arrays in the data segment.

Display

This is concerned with the "static nesting" of the program. Any point in an ALGOL W program is statically nested within several blocks with declarations and procedure declarations. This is a fixed number at any point of the program (say n). At this point, there will be n sets

of variable names which can be used by the programmer, corresponding to the n blocks and procedure declarations which statically surround the point, the outermost being those declared in the outermost block of the program within which all the program is nested.

This point of the compiled program will be in a certain program segment, P.S. During the running of the program, when a data segment, D.S., is executing P.S., it will be able to directly address n sets of variables, each of which will be in a separate data segment. (These data segments will be in the process of executing the program segments corresponding to the blocks and procedure declarations that all points of P.S. are statically nested in.) The "display" is the set of n pointers, one to each of these data segments. It remains the same throughout D.S.'s execution.

It should be noted that the variables in the first data segment created (corresponding to the outermost block) are always accessible and therefore that the pointer of the display that refers to it never changes. It therefore does not need to be stored with the rest of the display. It should also be noted that the innermost display pointer points to D.S. itself as, of course, all its variables are directly accessible.

The display is stored from the innermost pointer first to the second outermost last, in the 3rd section of the data segment.

Conventions for register usage

The IBM System 360 has 16 general purpose registers which are used by convention as follows:-

Register 14 holds the base address of the program segment currently being executed.

Register 13 holds the address of a system segment which contains several system routines, constants and variables. The most important variable is T.D.S. (top data segment) which is the address of the data segment on top of the stack at any time. This register remains unaltered throughout the program.

Register 12 holds the address of the outermost data segment of the display, corresponding to the outermost block of the program. It remains unaltered throughout the program.

Register 11 downwards, as required, hold the rest of the display at the current point. Register 12 holds the outermost display pointer, and register 11 the second outermost, etc.

Register 15 and the remaining registers from 0 upwards are used as required.

4.2 Code Generated by ALGOL W Compiler

At the start of all program segments representing blocks or procedures, and of all thunks to evaluate actual parameters, there must be code to create a new data segment on top of the stack (this will be the one which will execute the program segment of thunk). At the end of all these, there must also be code to destroy the top data segment.

On entry to all program segments and thunks and on return to them after a call of another, steps must be taken to maintain the display registers. On entry to each, the address of the new data segment allocated must also be added to the appropriate display (according to its static level) and this new display stored in the new data segment. On return to a program segment, the whole display must be restored from its data segment.

Much of the code generated for calling and returning from program segments and thunks is to enable the above to be performed.

Details of the code generated for entry to and exit from program segments and thunks, for calling program segments and for evaluation of actual parameters are given in Appendix F. The calling conventions are also shown there.

4.3 Low-level Language Procedures

Representations

It can be seen that the restrictions mentioned in Chapter 3 for calling WAIT, WAITTILL, SCHEDULE_EVENTS, INITIALISE and TERMINATE mean that all these procedures are called from the same program segment and also that the same associated data segment is on top of the stack before all these calls. In a SLIM program, a global system variable (of type integer as far as the ALGOL W program is concerned) called DATASEG holds the address of this data segment.

When the procedures WAITTILL and SCHEDULE_EVENTS are called by the data segment pointed to by DATASEG, another data segment is created above this to execute the procedure called. Representations of the delaying condition for WAITTILL or delay between creations and next action for SCHEDULE_EVENTS are passed as parameters to this new data segment. These representations are 2 words whose form is either:

(i) the address of a constant or simple variable in word 1 and word 2 unused (not applicable as a representation of the procedure parameter for next action) or

(ii) the address of a thunk in word 1 with lst bit set to 1, and the address of the calling data segment in word 2.

It can be seen that the second word is either unused, or is the same as DATASEG. If the first word is stored in an appropriate field of the location in question, then, with DATASEG, a full representation of the parameter is stored. The field THUNK is used for delaying condition and delay between creations and the field RET for next action.

If the representation given by the event notice field and DATASEG is the address of a constant or simple variable, then it can be evaluated at any time when this variable or constant exists. It must be a variable or constant which was accessible to the data segment pointed to by DATASEG, and therefore can be evaluated as long as it exists. If the representation is the address of a thunk and DATASEG, then it can be evaluated as long as the display of the data segment pointed to by DATASEG exists, because the thunk will try to restore this display during evaluation. This again means that the data segment pointed to by DATASEG must exist. All attempts to evaluate these representations are made from low-level procedures called from scheduling procedures

which are themselves called from the data segment pointed to by DATASEG, so it is always possible to evaluate the representations.

The other representation of next action which is obtained when a self-scheduling procedure is called, was explained to be a return address of the self-scheduling procedure in section 4.1. The self-scheduling procedure is always called by the data segment pointed to by DATASEG and the data segment which is created to execute the self-scheduling procedure holds this return address in its field R.A. This representation of next action is stored in the RET field of the eventnotice being delayed. When, on a later call of a self-scheduling procedure, it is decided to execute this next action, the return address kept in the self-scheduling procedure's data segment is replaced by this stored next action. As return from the self-scheduling procedure is always made to the same program segment (executed by the data segment pointed to be DATASEG), the return will always be correctly made.

The two types of representation of next action can be distinguished because the representation produced by SCHEDULE_EVENTS has a 1 in the first bit of the RET field (as it is a pointer to a thunk).

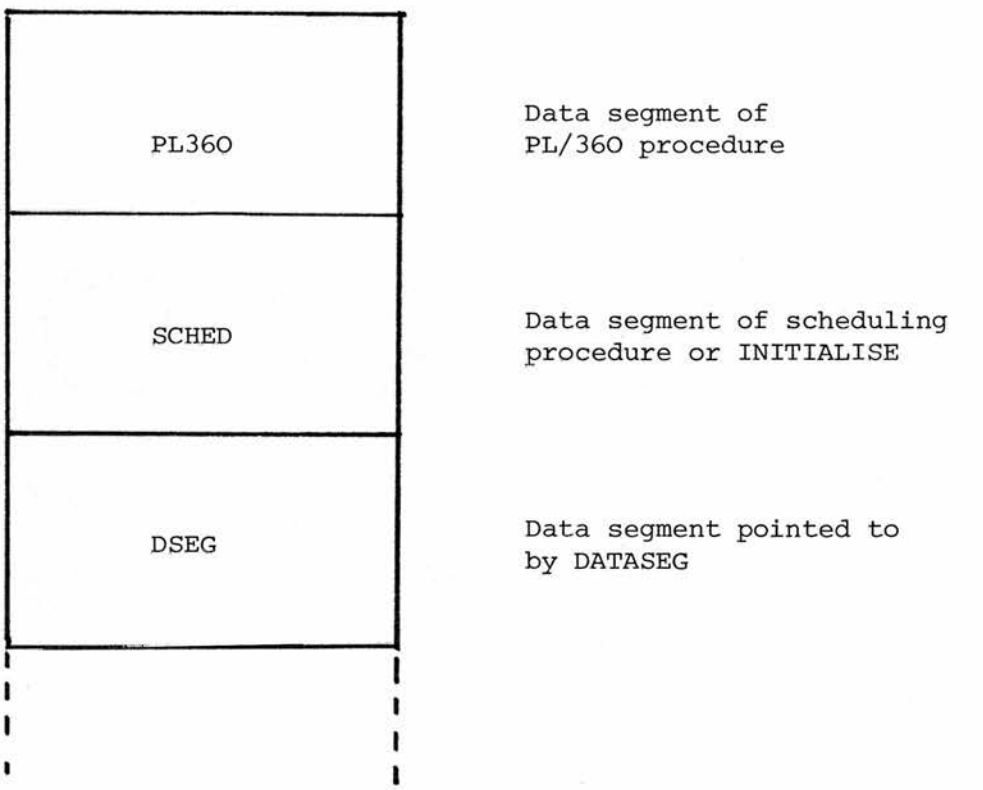
Actual procedures

The low-level language procedures used are written in PL/360 which can be called a high-level assembler. The language is a shorthand, structured way of writing assembly programs which allows access to all registers, allows execution of all machine instructions, and is very

easily mappable onto individual machine instructions. It is fully described in [11] and [12].

The PL/360 procedures used obey ALGOL W procedure calling conventions and are thought by the rest of the system to be ALGOL W procedures.

Calls of SAVERETURNADD, SAVETHUNKRETADD, STOREDESTDELAY, REPLACERETADD and LEVEL are all made from similar places. The data segment pointed to by DATASEG is on top of the stack, when a scheduling procedure, or INITIALISE is called, putting another data segment above it on the stack. Then the above procedures are called from this level and put their own data segment on the stack:-



The names PL360, SCHED and DSEG will be used in the following procedure descriptions to refer to these data segments. Using the dynamic links, PL360 can access any fields of the data segments below it to retrieve or replace any information stored in them.

The remaining PL/360 procedures used, EVALDELAY, EVALCON and STOP can be called from any level. The data segment which executes these procedures will also be called PL360 in the following procedure descriptions.

None of the data segments produced by the PL/360 procedures are fully filled in with the information a true ALGOL W procedure would insert. Their display does not need to be stored and the display registers do not need to be maintained because the procedures do not access any variables other than those passed as parameters. As none of the procedures declare any private variables, that section of the data segments can also be omitted.

Other than STOP and LEVEL , all procedures are passed parameters, and that field of the data segments (which is the top field used) is filled in by the data segment which calls the PL/360 procedure. It does so without checking to see if the "heap" storage has been overwritten (perhaps a small amount of free space is always kept between the stack and the heap). Therefore it must be assumed that it is not overwritten and that the rest of the data segment (i.e. the System Information section which is below the parameters passed) can similarly be filled in without checking. The PL/360 procedure does not itself pass parameters to other data segments, so there is no risk of overwriting the heap because of that.

The System Information section of the data segments is also not always filled out, but this is explained for each procedure separately.

The actual PL/360 procedures used are written in full in Appendix B.

(1) LEVEL

This procedure does not have any parameters and when called is given its return address in r1. R2 is assigned the address of SCHED from T.D.S. as this is also required on exit. This procedure gives as a result the address of DSEG and must therefore leave in r3 a pointer to this address. The address of DSEG is already held as the D.L. field of SCHED, so r3 is assigned the D.L. field's address before exit.

No System Information needs to be filled out in PL360 because no data segment is created on top of it on the stack.

(2) SAVERETURNADD

This procedure is always passed the field RET of an eventnotice record as parameter and assigns to it the return address of SCHED. The parameter is not a simple variable, so there is a thunk to evaluate it. The 11th word of PL360 gives the address of the thunk and the 12th gives the address of SCHED. To evaluate this thunk, a new data segment must be created, so some system information must be stored in PL360, namely the fields F.P, R.A. and D.L. The size of PL360 is 12 words and T.D.S. is made to point to it. The parameter is evaluated in the normal way except that there is no need to restore a display or make r14 point to SAVERETURNADD's program segment on return from the thunk as neither will be used. After evaluation of the thunk, r3 points to the RET field that was passed.

The address of SCHED is found from PL360 and its field R.A. is stored in the RET field found. Finally, T.D.S. is assigned the address of SCHED and a normal exit is made.

(3) SAVETHUNKRETADD

This procedure is always passed the RET field of an eventnotice as parameter and assigns to it the return address of SCHED. It also assigns to the next word (i.e. THUNK field of eventnotice) the representation of the delaying condition that has been passed to SCHED. (SCHED will be executing a WAITTILL procedure).

This procedure is the same as SAVERETURNADD, except that before exit, the 11th word of SCHED (which, with DATASEG, represents the delaying condition) is assigned to the word after the RET field that it was passed.

(4) STOREDESTDELAY

This procedure is called from SCHEDULE_EVENTS and is passed the RET field of an eventnotice as parameter. To this is assigned the representation of next action which was passed as parameter to SCHED and to the next word, the THUNK field, is assigned the representation of delay between creations which was similarly passed.

The procedure is the same as SAVERETURNADD except that, instead of assigning SCHED's return address to the RET field, the 17th and 11th words of SCHED (which represent the 2 parameters) are assigned to the RET and THUNK fields respectively.

(5) REPLACERETADD

This procedure is passed the RET field of an eventnotice and DATASEG, which is the representation of a next action. If this representation is a thunk, the thunk is executed, but if it is a return address, this return address is assigned to the R.A. field of SCHED.

It evaluates the thunk passed to it, to get a pointer to the RET field, in much the same way that SAVERETURNADD does, except that the size of PL360 is now 14 words, there being 2 parameters passed to it. The address of the program

segment for REPLACERETADD must also be restored to r14 from PL360, as it is required for the branch instructions implied by the if ..... then ..... else ..... construct.

The contents of the RET field are examined, and if there is a 0 in the first bit, this return address is placed in the R.A. field of SCHED and there is a normal exit. Otherwise the contents of DATASEG are also found and the thunk represented by them is evaluated. This thunk should contain a goto statement, so there should be no normal exit from REPLACERETADD in this case. If return is made to REPLACERETADD, the external ALGOL W procedure ERROR001 is called to write an error message and end the simulation.

(6) EVAL DELAY

This procedure is always passed the field THUNK of an eventnotice and DATASEG as parameters and evaluates the "delay between creations" they represent, giving this value as a result.

It finds the contents of the THUNK field passed to it in the same way as REPLACERETADD does with its RET field. If this value has a 1 on its first bit, it represents the address of a thunk and so the contents of DATASEG are found and the thunk they both represent is evaluated.

Whichever type of representation was used, now r3 points to the result, so the result itself is loaded into r3, which is the convention for passing an integer result from a procedure. A normal exit is finally made.

(7) EVAL CON

This procedure is always passed the field THUNK of an eventnotice and DATASEG as parameters and evaluates the delaying condition they represent, giving this value as a result.

It is the same as EVALDELAY, except that, as the result is a logical value, r3 is left pointing to the result on exit.

(8) STOP

This procedure causes exit to be made from the whole program. R12 always points to the outermost block's data segment, so the return address and dynamic link of this are fetched. T.D.S. is set to this dynamic link and a branch is made to this return address. This is the code that would normally be executed at the end of the program.

CHAPTER 5

OTHER FEATURES OF "SLIM"

5.0 Introduction

The basic scheduling features of SLIM were discussed in detail in Chapter 3. This chapter discusses the remaining features of SLIM. First are short sections on how random factors can be introduced and on statistic-gathering. The only built-in structure of SLIM, called a GROUP of event-notices is next explained.

Finally, another scheduling feature of SLIM is discussed which like the GPSS "PREEMPT" block does not fit into the general simulation language structure developed in Chapter 2. This is the idea of one transaction forcing another to do something that it did not itself plan to do. It is a more powerful and flexible generalisation of the GPSS PREEMPT and introduces a new idea for the representation of a transaction.

5.1 Random Factors

The basis of the random factors of SLIM is a pseudo-random number generator called RFRACTION which uses the algorithm [13] and is written in PL360 with ALGOL W linkage conventions. It is passed an integer variable as a parameter and generates a pseudo-random positive integer from it which is made the new value of the variable. This new integer is multiplied by a constant fraction to deliver as the procedure's result a random fraction, r, in the range, $0<r\leq 1$.

The programmer must define integer variables to use as random number streams and must assign to them, random initial positive values, If R is one such variable, RFRACTION(R) will give a fresh pseudo-random fraction between 0 and 1 each time it is called.

RFRACTION

The PL360 procedure calculates the address of the integer it is passed (by if necessary evaluating the thunk representing it). It then generates a pseudo-random positive integer from the contents of this location and stores this new value in the location. The new integer is then converted to a real number and multiplied by a constant (equivalent to the inverse of the maximum possible integer) to give a random fraction between 0 and 1. This is passed as a result of the procedure by being left in register f0 on exit.

There are also 3 procedures which can be used to generate random samples from common statistical distributions. All give results which are rounded to integer values as it is felt that this is the form usually required, simulated time being an integer quantity:-

(1) integer procedure RNEGEXP (real MEAN; integer R)

This generates a pseudo-random sample from the negative exponential distribution with mean, MEAN, using R as a random number stream. The random sample is rounded to integer before being passed as a result.

(2) integer procedure RDEVN (real DEVN; integer R)

This generates a sample from the uniform distribution, -DEVN to +DEVN, using random number stream, R. The random sample is again rounded to integer.

(3) integer procedure RNORMAL (real MEAN, SDEVN; integer R)

This generates a random sample from a distribution with mean MEAN and standard deviation, SDEVN, using random number stream,R. The distribution used is very similar to the normal distribution but is actually the sum of 12 independent random fractions (0 to 1), corrected to give the required mean and standard deviation. However, the distribution should be close enough to normal for most purposes.

5.2 Statistic Gathering

There are no statistics automatically gathered by the system. There are, however, a set of procedures and a type of record which are used only for statistic gathering. These gather queuing statistics and any other types of statistics required must be individually programed by the user.

The type of record in which queuing statistics are gathered is called a QSTAT:-

```
record QSTAT (integer NO, TOTAL, LASTTIME, STARTTIME, MAX, EMPTY,
                                                              INT)
```

This can be thought of as representing a queue of transactions in the model. The fields are used as follows:-

(i) NO represents the number of transactions that can be thought of as being in the queue at any time. (No note is kept of which transactions these are).

(ii) TOTAL gives tht total number of transactions which have entered the QSTAT since statistics started to be gathered for it.

(iii) LASTTIME gives the last value of simulated time at which either statistics started to be gathered or a transaction "entered" or "left" the QSTAT.

(iv) STARTTIME gives the value of simulated time at which the statistics started to be gathered.

(v) MAX gives the maximum number of transactions in the QSTAT at any one time since statistics started to be gathered.

(vi) EMPTY gives the total time the QSTAT has been empty since statistics started to be gathered.

(vii) INT gives the integral over simulated time of the contents of the QSTAT since statistics started being gathered. This is:-

$$\sum_{T=STARTTIME}^{SIMTIME\ -1}$$

(contents of the QSTAT between simulated time T and (T+1))

The reference (QSTAT) procedure NEWQSTAT gives as a result an initialised QSTAT record which has a content of zero and which starts collecting statistics from the current value of simulated time.

There are also 4 procedures which act on QSTAT:-

(1) procedure Q (reference (QSTAT) value X)

This procedure increases the contents of X by 1 and maintains the other fields of the QSTAT record.

(2) procedure UNQ (reference (QSTAT) X)

This procedure decreases the contents of X by 1 and maintains the other fields of the QSTAT record.

(3) procedure RESETQ (reference (QSTAT) X)

This procedure causes statistics to be gathered freshly from the current value of simulated time. The contents of the QSTAT however remain as they were before the RESETQ was called.

(4) procedure OUTPUTQ (reference (QSTAT) X)

This procedure requests that the statistics gathered for X be output. It prints out, in one line, the following statistics about the QSTAT since it was either created or reset last:-

(a) average length of X

(b) maximum length of X

(c) number of transactions which have entered X

(d) average time for transactions to pass through X

(e) percentage of time for which X was empty

(f) current length of X

(g) length of time for which statistics have been gathered.

As QSTATS do not keep a note of which transactions are in them at any time, it is possible for a transaction to execute "UNQ(X)" when it has never executed a "Q(X)". It is the programmers responsibility to ensure that this does not happen.

5.3 Groups

SLIM only has one built-in structure to help specify models, other than Delaychains (QSTATS are purely for statistic gathering). This is a GROUP.

From the user point of view, a Group is a linear list of Eventnotices. Each Group is headed by a record of type GROUP, which has an integer field COUNT which holds the number of eventnotices in the Group at any time. An initialised record of type GROUP is given by the reference (GROUP) procedure NEWGROUP (this group is empty).

An eventnotice can be a member of any number of Groups at a time. It can, in fact, even be a member of the same Group more than once at one point in time.

There is also, conceptually, a pointer, P, connected with each Group. This pointer can be thought of as referring to one eventnotice of the Group at any time.

There are basically, 2 procedures which act on Groups:-

(1) procedure INSERT (reference (GROUP) A; string (6) S; reference (EVENT-NOTICE)E)

This procedure inserts the eventnotice, E, in the Group, G. The place it is inserted in the list of eventnotices is determined by the string, S, which can have 4 values:-

(i) "START" - E is inserted as the new first eventnotice of G

(ii) "END" - E is inserted as the new last eventnotice of G

(iii) "BEFORE"- E is inserted immediately nearer the start of G than the eventnotice referred to by P.

(iv) "AFTER" - E is inserted immediately nearer the end of G than the eventnotice referred to by P.

If S = "START", the pointer, P, is also reset to refer to the first eventnotice in the Group.

(2) reference(EVENTNOTICE) procedure SEARCH (reference(GROUP) G; string (6) S; logical C)

This procedure looks at each eventnotice in the Group G in turn, to see if the condition C holds. (When it is testing the condition for an eventnotice, TEMP points to the eventnotice, so C will normally be phrased in terms of TEMP). It looks in this way at every eventnotice in the Group, from the one pointed to by P to the last eventnotice and then from the first eventnotice in the Group down to the one pointed to by P. The result given by the procedure is a pointer to the first eventnotice found in this way to have the condition, C, satisfied. If no eventnotice in the Group has the condition satisfied a result of null is given.

If no eventnotice has been found, P is left referring to the same eventnotice as before. Otherwise, what happens is dependent on the string S:-

(i) "UNLINK" - the eventnotice found is removed from the Group and P is left referring the the eventnotice after it (If the last eventnotice was found, P will refer to the first one).

(ii) "STAY" - the eventnotice found is left on the Group and P is made to refer to it.

(iii) "NEXT" - the eventnotice found is left on the Group and P is made to refer to the one after it (or the first eventnotice if the one found was last).

Three more procedures which act on Groups can be expressed in terms of the above 2:-

(3) procedure REMOVE (reference (GROUP) G; reference (EVENTNOTICE) E)

This procedure removes the eventnotice E from the Group if it was a member. It has a side-effect in that TEMP will point to the eventnotice removed (i.e. E) if it was found in the Group. If it was not found on the Group, TEMP will be made null, It can be expressed as "TEMP:=SEARCH(G, "UNLINK", TEMP=E)

(4) procedure JOIN (reference (GROUP) G; string (6) S)

(5) procedure LEAVE (reference (GROUP) G)

These two procedures have the same effect as INSERT and REMOVE except that the eventnotice referred to is implied to be CURRENTEVNOTICE. It should be mentioned here that, with the feature of "superceding", introduced in section 5.5, if CURRENTEVNOTICE is "superceding" another eventnotice, then this "superceded" eventnotice is implied (and if that one is "superceding" another it is, implied, etc).

There is one final procedure which acts on Groups:-

(6) procedure SET (reference (GROUP) G)

This sets the pointer P of G to the start of the Group.

Examples

(A) The Group GR has been defined and initialised. The eventnotices on GR have to be kept ordered by their priority and of those with the same priority, the earliest put in should come first.

The following procedure will insert a given eventnotice in the Group in the correct order (highest priority at the start of the Group):-

```
procedure  PRIORITY_IN     (reference (EVENTNOTICE)E);
begin      SET (GR);
           if SEARCH (GR, "STAY", PR(E)> PR (TEMP)) = null
             then INSERT (GR, "END", E)
             else INSERT (GR,"BEFORE", E)
end;
```

If, at some point, the eventnotice in GR with the highest value of the real field JN of its associated record is required, the following procedure will give it as a result (it is known that JN will never be negative):-

```
reference (EVENTNOTICE) procedure HIGHJN;
begin   real X;
        reference (EVENTNOTICE) E,F;
        X: = -1;  E: = null;
        while  begin F: = SEARCH(GR,"NEXT",JN(TRANS(TEMP))>X)
                       F  ≠ null
               end
        do   begin   X: = JN (TRANS(F));
                     E: = F
             end ;
        E
end;
```

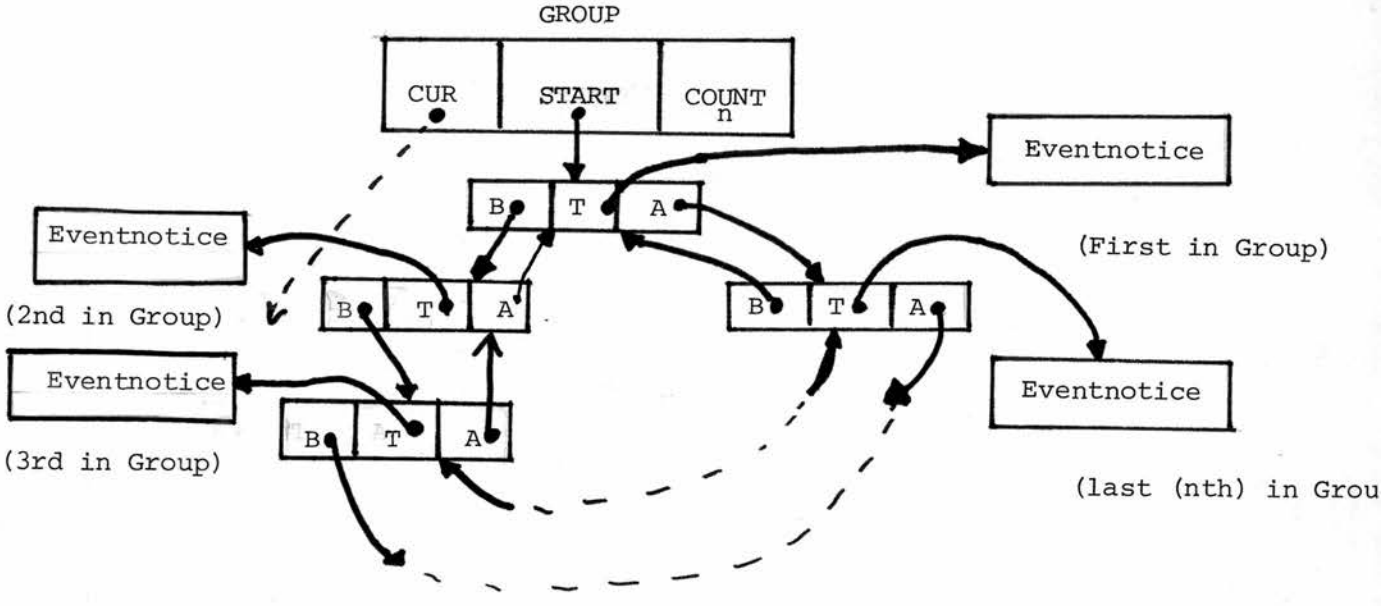
(B) A Group G will behave as a queue by using only INSERT(G,"END",E) and SEARCH(G,"UNLINK",true) to insert and remove eventnotices, or like a stack using only INSERT(G,"START",E) and SEARCH(G,"UNLINK",true).

5.4 Implementation of "Groups"

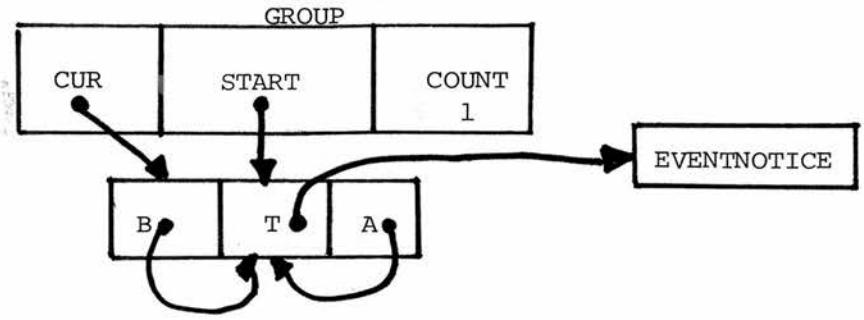
It was stated in the last section that a Group could be considered as a linear list of eventnotices with a pointer associated which referred to one eventnotice on the list. However, as an eventnotice can be in any number of Groups at once, it cannot be directly linked to form each list. Instead, the Group consists of a list of records of type LINK which are doubly linked to form the list with their reference(LINK) fields A and B. Each such LINK record also has a reference (EVENTNOTICE) field T which points to the eventnotice it is linking onto the Group. The list of LINK records is actually circular with the first LINK record coming immediately after the last one.

At the head of each Group is a record of type GROUP which has an integer field COUNT containing the number of eventnotices in the Group. The Group record also has 2 reference(LINK) fields START and CUR, of which START always points to the LINK record of the first eventnotice in the Group. The pointer (called P in the last section) referring to an eventnotice on the Group is implemented by CUR which points to the LINK record linking the required eventnotice to the Group.

A Group therefore has the following form:-



With only one eventnotice in the Group, the form is:-



When the Group is empty, there are no LINK records, START and CUR are both null and COUNT is zero. An empty Group of this form is given by the reference (GROUP) procedure NEWGROUP.

The procedures which act on Groups can now be explained:-

(1) procedure INSERT(reference(GROUP)value G; string(G)value S; reference (EVENTNOTICE)value EV)

This procedure creates a LINK record pointing to EV and puts this in the Group G at the place specified by S.

The COUNT field of G is first incremented by one to show that there is a new eventnotice in the Group. If there were no eventnotices in the Group before (i.e. START field = null), then a new LINK record is created which is pointed to by both START and CUR. Its field T points to EV and A and B point to the LINK record itself.

If there were other eventnotices in the Group, the course taken depends on S:-

(a) S="START" - the LINK record created points to the first LINK record of the Group and the one before it (i.e. the last LINK record in the Group) with its fields B and A. The fields START and CUR of the Group record

are made to point to this new LINK record.

(b) S="END" - the same happens as in (a) except that START and CUR are not updated.

(c) S="AFTER" - the new LINK record points to the one pointed to by CUR and the one after it with fields A and B

(d) S="BEFORE" or any other string - the new LINK record created points to the one pointed to by CUR and the one before it with fields B and A. If EV was to be inserted "before" the first LINK record in the Group, G's field START is made to point to it also.

In all 4 cases, the linking is completed by making the LINK records on either side of the new LINK record point to the new record.

(2) reference (EVENTNOTICE) procedure SEARCH (reference (GROUP) value G;
string(G) value S; logical C)

This procedure looks at all eventnotices in G to see if any satisfies the condition, C. The first eventnotice to satisfy C is given as the result of the procedure. The eventnotices are searched in turn from the one whose LINK record is pointed to by the CUR field of G.

If there are no eventnotices in the Group (i.e. field CUR=null) then a value of null is returned. Otherwise, L points to CUR and LST points to the LINK record before it. L is used to point to each LINK record in turn, from CUR round to LST until an eventnotice is found to satisfy C. When the condition is being tested, TEMP points to the eventnotice associated with L.

After this part of the procedure, either no eventnotices have been found to satisfy C, in which case, a value of null is returned and CUR is left

as it was, or an eventnotice will have been found, in which case, TEMP will point to it and L will point to its LINK record. TEMP will eventually be given as the result of the procedure, but first the Group must be modified according to the string S. There are three cases:-

(i) S="STAY" - G's field CUR is made to point to the LINK record of the eventnotice found (i.e. L).

(ii) S="UNLINK" - CUR is made to point to the LINK record after the one found and G's COUNT field is decreased by one. The LINK record found is then unlinked from the Group by making START and CUR null if the Group will now be empty, or making the LINK records on either side of it point to each other if there were others in the Group (If it was the first in the Group, the next LINK record must also be made the new first).

(iii) S="NEXT" (or otherwise) - G's field CUR is made to point to the LINK record after the one found.

(3) procedure REMOVE (reference (GROUP) value G; reference(EVENTNOTICE) value E)

(4) procedure JOIN (reference (GROUP) value G; string (G) value S)

(5) procedure LEAVE (reference (GROUP) value G)

These 3 procedures are expressed in terms of SEARCH and INSERT and were explained in sufficient detail in the previous section.

(6) procedure SET (reference (GROUP) value G)

This procedure makes G's field CUR point to the first LINK record in the Group (pointed to by its START field) so that on the next call of SEARCH, the searching will start from the start of the Group.

5.5 Superceding

In GPSS, when a transaction "Preempts" a facility, if there was another transaction in the facility, it is immediately halted in whatever it is doing and forced to leave the facility to allow the preempting transaction to enter it. It then remains completely inactive until the preempting transaction leaves the facility when it can return and resume the state it was in before being preempted (e.g. if at the time it was preempted it had 10 more units of simulated time to wait, it will still have 10 units to wait when it is allowed back into the facility).

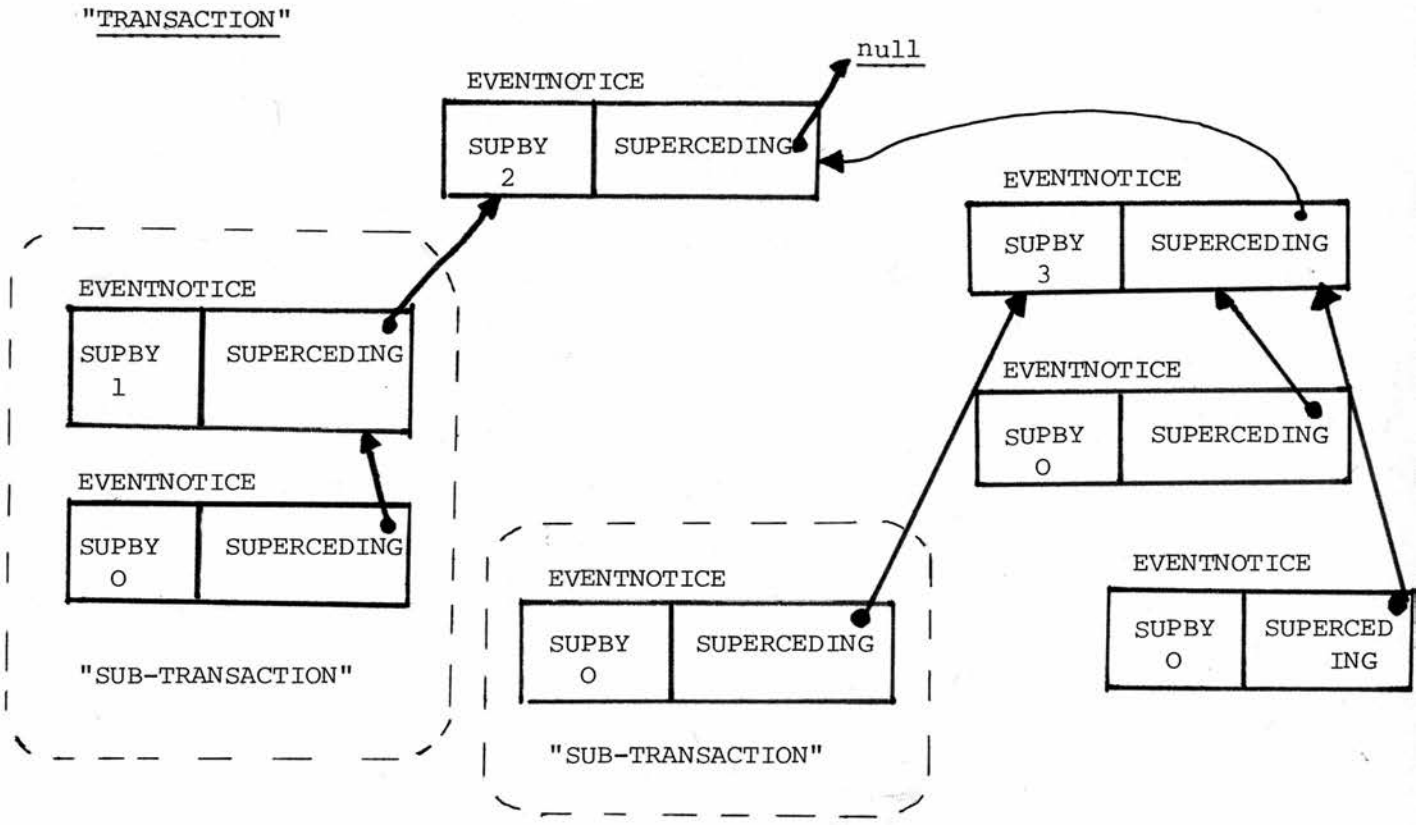
The important concept involved here is that the first transaction only planned to enter the facility, wait say 20 units of simulated time, and then leave the facility. This was all its executed code specified for it to do. However, the second transaction, after say 10 units of time, forced the first (by executing a PREEMPT block) to leave the facility, wait until the second had finished with the facility, enter it again, and then resume its original wait for the remaining 10 units of time.

In SLIM, a method has been developed to allow one transaction to force another to do any specified sequence of actions which it did not itself plan to do. The transaction which is forced to do the actions is stopped in the middle of what it is doing and, after the specified actions have been completed, returns to the exact state it was in before being interrupted.

To accomplish this, the concept of a transaction is generalised. Whereas previously a transaction was considered to be one eventnotice record with, optionally, a user-defined record associated, we shall now consider a transaction to be a tree of eventnotices, each with a user-defined record optionally associated with it. (Often all the eventnotices will have the same user-defined record associated with them, but this is not essential). All

eventnotices apart from the root eventnotice are considered to "supercede" the eventnotice immediately higher in the tree and are linked to this higher eventnotice by their reference (EVENTNOTICE) field SUPERCEDING. All eventnotices in the tree apart from the terminal node eventnotices are considered to be "superceded" by the eventnotices immediately below them in the tree. Their integer field SUPBY contains the number of eventnotices they are superceded by. The field SUPERCEDING of the root eventnotice is null as it does not supercede any other, and the field SUPBY of the terminal node eventnotices is zero as they are not superceded.

It should be noted firstly that the limiting case of a tree with only one eventnotice is equivalent to the notion of a transaction previously used. Also each of the sub-trees of a transaction could be thought of as transactions, the only difference from above being that their root eventnotice would be superceding another eventnotice outside this subtransaction.



No superceded eventnotices (i.e. field SUPBY>0) can be given control. The terminal node eventnotices (which are not superceded) however each independantly execute sets of actions and can be separately scheduled in exactly the same way as discussed in Chapter II. When they execute the procedure, TERMINATE, if they are superceding an eventnotice. Then they decrease its field SUPBY by one, and if it is now not superceded, then it can be scheduled again.

When an eventnotice becomes superceded by another, its field SUPBY is increased by one. If it was previously not superceded then it must remember the state it was in before being superceded which will be either

(i) "time-scheduled" with a certain time still to wait

or(ii) "in control".

or(iii)"condition-scheduled" waiting for a certain condition to hold, in a certain Delaychain.

OR(iv) "terminated"

The eventnotice is then totally ignored by the scheduler until it becomes not superceded when it will be returned to the state it was in before (except that if it was in control before, it will be time-scheduled after time zero).

The procedure which causes one transaction to be superceded by another is:-

```
procedure CALL_IMMEDIATE(reference (EVENTNOTICE) EV; procedure PROC)
```

When an eventnotice executes this statement, it causes a new eventnotice to be created with the same priority and user-defined record as EV. This new eventnotice is made to supercede EV as described above. The first action of the

created eventnotice is PROC, which, like the procedure parameter passed by SCHEDULE_EVENTS, must end in a goto.

The new eventnotice will immediately be given control and will start executing its first action. However, the eventnotice which executed the CALL_IMMEDIATE (if it has not caused itself to be superceded) will be scheduled to be given back control immediately the new eventnotice has executed a self-scheduling statement, irrespective of priorities.

The use of the procedure is made clearer by using the new definition of a transaction. The transaction EV is forced to stop what it was doing, execute the specified actions and, after these have been completed, return to the state it was in before the CALL_IMMEDIATE.

It should be noted that CALL_IMMEDIATE can only be called from the outer program level as defined in Chapter II.

Examples

(1) A car being made on a production line is represented by a transaction, and at one stage in its manufacture there are three people independantly working on it, fitting parts. It cannot move to the next stage of production until all these jobs are finished.

The car eventnotice would execute:-

```
CALL_IMMEDIATE (CURRENTEVNOTICE, goto START JOBS)
```

The car eventnotice would become superceded by a new eventnotice executing "STARTJOBS". This new eventnotice would execute:-

```
STARTJOBS : CALL_IMMEDIATE (SUPERCEDING(CURRENTEVNOTICE),goto FIRST_JOB);
            CALL_IMMEDIATE (SUPERCEDING(CURRENTEVNOTICE),goto SECOND_JOB);
THIRD_JOB :
```

This would create another two eventnotices which also would supercede the car eventnotice. The car eventnotice would then be superceded by three eventnotices independantly executing FIRST_JOB,SECOND_JOB and THIRD_JOB and would not be given control back until all three jobs were finished by their eventnotices executing TERMINATES. In other words, the car transaction would force itself to execute independantly all three jobs before continuing its execution at the statement after its CALL_IMMEDIATE.

It is instructive to examine the order the three jobs are started in (all at the same value of simulated time). The first CALL_IMMEDIATE executed at "STARTJOBS" causes the first job to be started immediately, and when it has been delayed, the second CALL_IMMEDIATE then starts the second job. Finally, when it too has been delayed, the third job will be started.

(2) In a workshop, there is a certain machine which is used in the repair of domestic and industrial appliances. These appliances are considered as transactions, domestic ones having an associated record of type IND. The repair of industrial appliances is considered to be urgent and they are allowed to use the machine before domestic appliances which are waiting for it. Also, if they wish to use the machine and a domestic appliance is using it at the time, the domestic appliance is removed from the machine until the industrial appliance is finished with it.

The machine itself is represented by a reference (EVENTNOTICE) variable, MACHINE, which points to the eventnotice using the machine at any time (or is null if the machine is unused). There is also a Delaychain, QUEUE, which

represents the queue of appliances waiting to use the machine.

A domestic appliance waits until the machine is unused in QUEUE with a priority 1 and then uses the machine with priority 4:-

```
PRIORITY (CURRENTEVNOTICE): = 1;
WAITTILL (MACHINE = null, QUEUE);
MACHINE: = CURRENTEVNOTICE;
PRIORITY (CURRENTEVNOTICE): = 4
WAIT (time using machine);
MACHINE: = null;
TESTFIRST (QUEUE);
```

An industrial appliance has priority 3 throughout. When it wants to use the machine, it can be either:-

(i) not used in which case it can immediately use the machine

(ii) used by an industrial appliance in which case it will wait until the machine is not used in QUEUE (it has higher priority than domestic appliances in QUEUE so will be moved before them).

(iii) used by a domestic appliance in which case the domestic appliance is forced to leave the machine temporarily by being superceded by an eventnotice which executes the actions at PREEMPT.

The industrial appliance will then use the machine itself:-

```
PRIORITY (CURRENTEVNOTICE): = 3;
if MACHINE ≠ null then
begin if TRANS(MACHINE) is IND then WAITTILL(MACHINE=null, QUEUE)
          else CALL_IMMEDIATE (MACHINE,goto PREEMPT)
end;
MACHINE: = CURRENTEVNOTICE;
WAIT (time using machine);
MACHINE: = null
TESTFIRST (QUEUE);
```

The eventnotice superceding a domestic appliance makes it give up the machine by executing the following:-

```
PREEMPT: MACHINE: = null
         PRIORITY (CURRENTEVNOTICE): = 2
         WAITTILL (MACHINE = null, QUEUE);
         MACHINE: = SUPERCEDING (CURRENTEVNOTICE);
         TERMINATE;
```

It marks the machine as unused and, with a priority of 2, allows the industrial appliance to use the machine and waits until the machine is again unused in QUEUE. It then puts the domestic appliance it is superceding back into the machine and destroys itself, which allows the domestic appliance to continue its wait in the machine.

Two points should be noted about this example:-

(A) When the industrial appliance executes CALL_IMMEDIATE, an eventnotice will be immediately created to execute PREEMPT and make the machine empty. When this eventnotice executes the WAITTILL, the industrial appliance will immediately be given back control and will itself use the machine before the scheduler tests the condition, "MACHINE = null" for the eventnotice. When it is tested, the machine will then be found to be used.

(B) The use of priorities is important. Of eventnotices waiting in QUEUE, industrial ones will be moved first, then preempted domestic ones and finally ordinary domestic ones. Also, domestic appliances have a priority of 4 when they are using the machine, so that, if they are scheduled to leave the machine at the same value of simulated time an industrial appliance wants it, they will leave it first, rather than being preempted.

5.6 Implementation of Superceding

The state of an eventnotice at any time can be determined as follows:

(i) field RET = 0 - eventnotice is "terminated"

(ii) field SUPBY >0 - eventnotice is "superceded"

(iii) field SUPERCEDING ≠ null - eventnotice is "superceding" another.

Also, if an eventnotice is neither terminated nor superceded.

(iv) field FUTUREEV = true - eventnotice is "time-scheduled"

(v) field FUTUREEV = false - eventnotice is "condition-scheduled"

(vi) CURRENTEVNOTICE points to it - eventnotice is "in control"

If an eventnotice is superceded, it keeps a note of the state it was in before being superceded:-

(a) field RET = 0 - was previously "terminated"

(b) field FUTUREEV = true - was previously either "in control" or "time-scheduled".

Field TIME holds the length of time it was still to be time-scheduled for when it was superceded (zero if it was in control before).

(c) field FUTUREEV = false - was previously "condition-scheduled". FIELD THUNK still holds the representation of the delaying condition and field APOINTER keeps a pointer to the Delaychain it was kept on.

The procedures CALL_IMMEDIATE and TERMINATE can now be fully explained:-

(1) procedure CALL_IMMEDIATE(reference (EVENTNOTICE)value EV; procedure PROC)

If EV is specified as null, nothing is done. Otherwise, a test is made of whether CALL_IMMEDIATE has been called from the correct level, EV is marked as "superceded" and the eventnotice in control is given a representation of its next action (which starts at the statement following the CALL_IMMEDIATE).

If the eventnotice executing the CALL_IMMEDIATE is causing itself to be superceded, then it is marked as "time-scheduled" with 0 time to go. Otherwise the eventnotice in control is scheduled to be given control at time, -1, and put at the start of the Future Events Chain. (It will therefore be given back control immediately the eventnotice created to supercede EV has been delayed, and GETNEWFIRSTEV called).

If EV was not in control or terminated and had not previously been superceded, it is either unlinked from the Future Events Chain using UNHOOKFE and marked with its time-to-go there, or unlinked from the Delayed Events Chain and the Delaychain it was on by a call of UNHOOKDE. UNHOOKDE has a side-effect of making TEMP point to the Delaychain EV was on, so a note is kept of this Delaychain in EV and a prompt is issued referring to it (in case EV had previously been marked "active" by a TESTFIRST but had not yet been moved, in which case the new first eventnotice in the Delaychain might be able to move).

If the eventnotice in control was superceded, CURRENT will already point to its associated record, and otherwise, it is made to point to the record associated with EV. In either case, a new eventnotice is now created and given control with the same priority and user-defined record as EV and superceding EV. PROC is then executed, being the start of the new eventnotice's first action. If this first action does not end in a goto statement and control is returned to CALL_IMMEDIATE, an error message is written and the simulation stopped.

(2) Procedure TERMINATE

The eventnotice in control is marked as "terminated" and a run-time check is made that TERMINATE has been called from the outer program level.

Before finding the next eventnotice to give control to and starting the execution of its next action, a test is made of whether the eventnotice being terminated was superceding any other. If so, the superceded eventnotice has its SUPBY field decreased by one. If this eventnotice is now not superceded and also not terminated, it is rescheduled according to the state it was in before being superceded. If it was time-scheduled then, it is again time-scheduled with the same time to go as it had when it was superceded. If it was condition-scheduled, it is again condition-scheduled in the same Delaychain and marked active as it might now be able to move.

5.7 Error Messages

If the "SLIM" language had been implemented by a specially written compiler, there are certain types of error that could have been picked up by the compiler. The main 2 would be the checking that all restricted procedures were called from the outer program level and that the procedure parameters passed as representations of first actions by SCHEDULE_EVENTS and CALL_IMMEDIATE ended in goto statements. As the ALGOL W compiler was used, compile-time error messages like the above could not be generated.

Luckily, a runtime check can be made during all calls of restricted procedures (using the low-level procedure LEVEL to compare with DATASEG) to make sure they are called from the correct program level. If it is found that a restricted procedure has not been called from the correct level, an error message is output and the whole program is stopped using STOP.

When SCHEDULE_EVENTS is called, a representation of the procedure parameter is stored with the newly created eventnotice. When this is later given control, the representation is evaluated by REPLACERETADD and, as it should end in a goto statement, control should never be given back to REPLACERETADD. If control is returned to REPLACERETADD, then there was no goto statement, and an error message is output and the simulation stopped. Similarly, in CALL_IMMEDIATE when control is given to the new superceding eventnotice and the procedure parameter is evaluated as its first action, return should not be made to CALL_IMMEDIATE if the procedure ends in a goto. If control is returned, the same error message as above is printed and the simulation stopped.

As well as these 2 types of runtime error message, an error message is also printed if the procedure GETNEWFIRSTEV cannot find a transaction to give control to. It should be noted that this can never happen if the control process is always time-scheduled and never superceded, as it can then always be given control even if no other transactions can be moved.

These are the only 3 error messages produced, though other types of error can exist. The most obvious are that SIMTIME should not be assigned to and that only the PRIORITY of transactions which are in control, superceded or terminated should be changed. It is, however, difficult to enforce such restrictions when the whole system has to be copied by the user and he can therefore directly get to any variable that is used by the system. Such types of error message have therefore been left out.

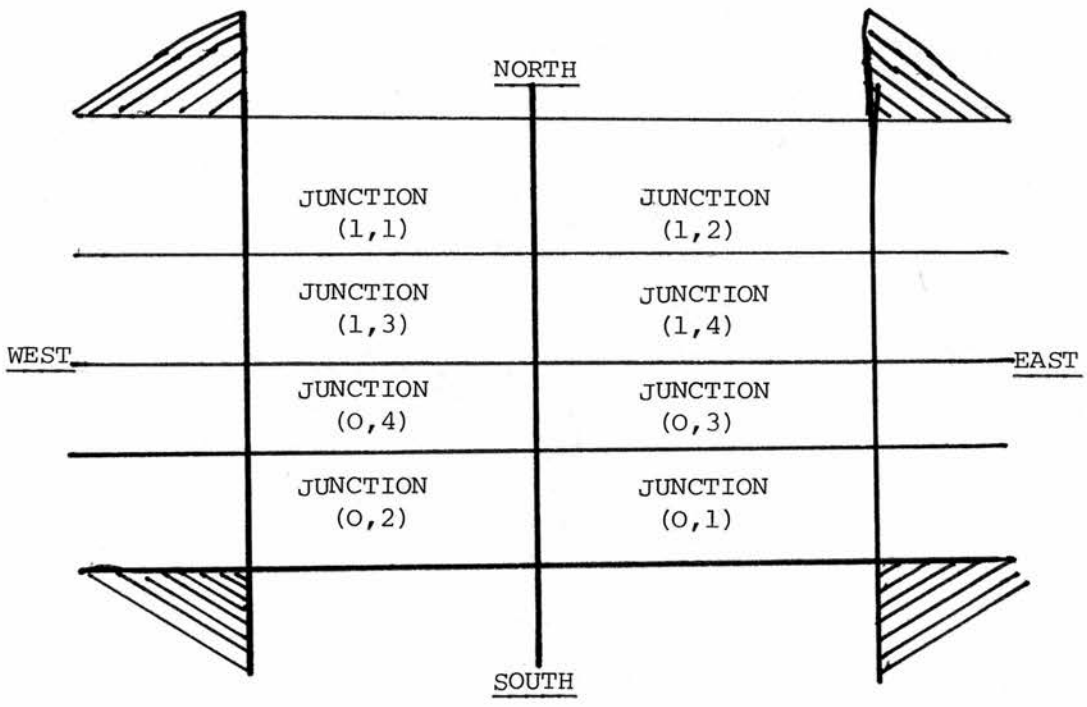
CHAPTER 6

TWO EXAMPLE PROGRAMS IN "SLIM"

6.0 Traffic Lights Simulation

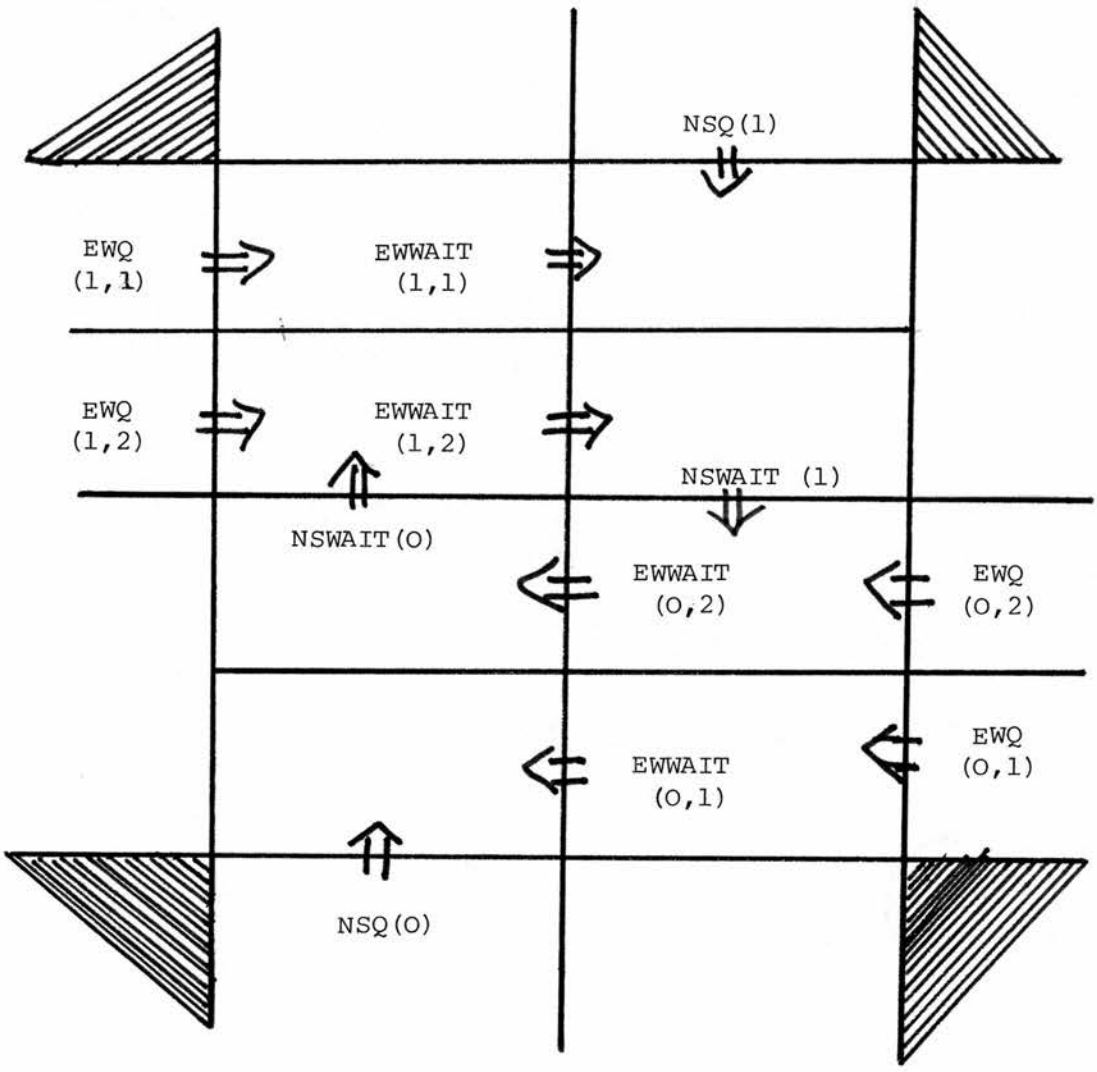
The real-life system being simulated here in a junction on a main East-West road with 2 lanes of traffic travelling each way. A North-South road with one lane each way meets it at a cross-roads with traffic lights.

In real life, the position of each car passing through the junction changes continuously. However, in the model, the junction has been split into 8 sections, each of which can either be occupied by a car or not. Every car in the model passes through the junction by moving discretely from one position in the junction to another. The junction is represented by an integer array JUNCTION (0::1, 1::4) which represents the 8 sections of the junction as follows:-



When a part of the junction has no car in it, the corresponding variable in the array has a value 0. Otherwise it has a positive value. When a car is facing in an East-West direction, it takes up one section of the junction at a time, but when it is facing in a North-South direction it covers 2 adjacent parts (i.e. (1,1) and (1,3) or (1,2) and (1,4) or (0,4) and (0,2) or (0,3) and (0,1)).

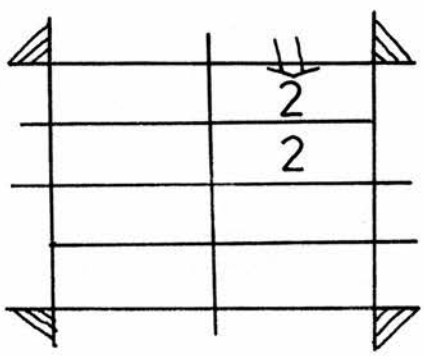
The delaychains in the model represent the places where cars (which are represented as transactions) can get held up waiting for conditions to hold. They can be thought of as being at the following points of the junction:-



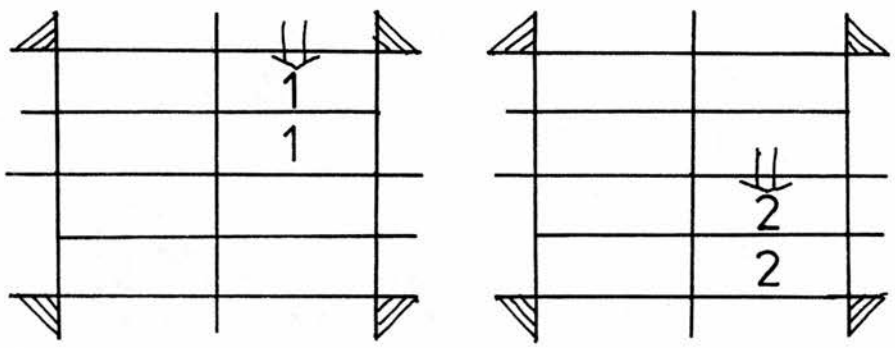
The arrays of Delaychains, EWQ and NSQ hold cars waiting to enter the junction in the lanes shown and the arrays EWWAIT and NSWAIT hold cars in the junction waiting to move to another part of the junction.

The course a car from the North takes through the junction depends on the direction it intends to leave. It enters the parts of the junction shown below in turn by assigning to the appropriate elements of the JUNCTION array the values shown. Before entering each part of the junction, the car might be delayed by a condition (e.g. another car being in that part of the junction) in the Delaychain shown. When it is in one part of the junction, it will also wait for a certain time before trying to pass on to the next part (or leaving the junction). This represents the time-taken for the real-life car to move:-

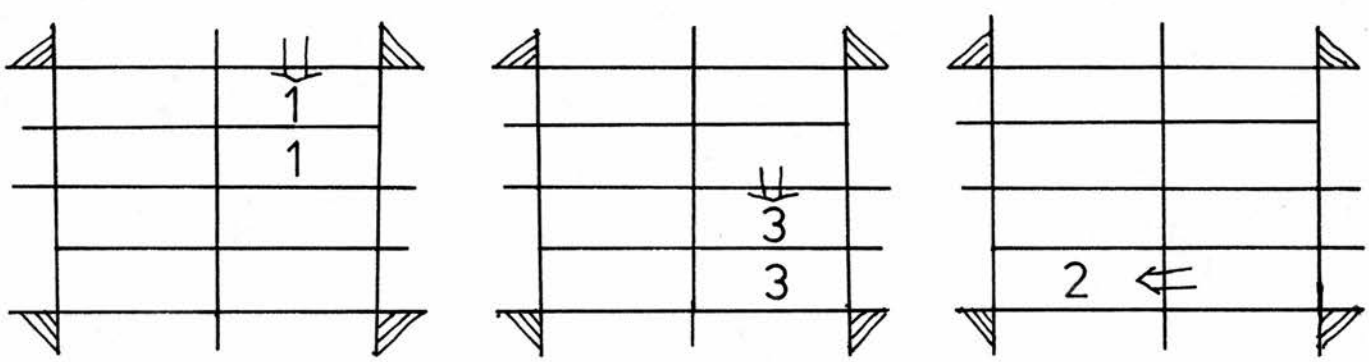
a) North car turning left:-



(b) North car going straight on:-



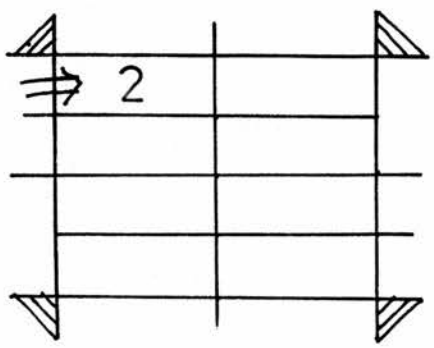
(c) North car turning right:-



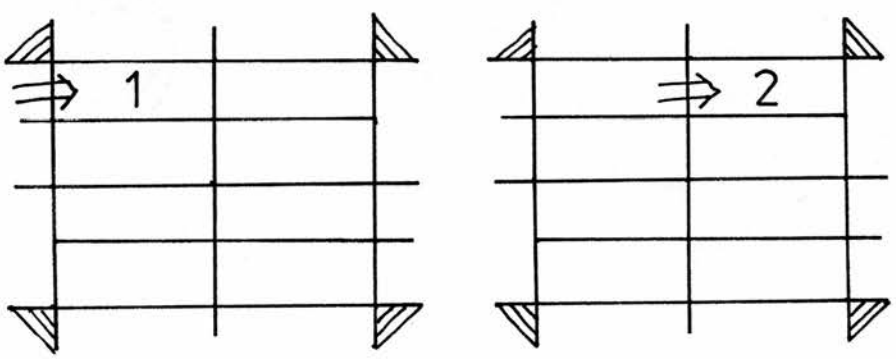
A car coming from the South goes through the same set of states but with NSQ(0) instead of NSQ(1), JUNCTION(0,2) and JUNCTION(0,4) instead of JUNCTION(1,2) and JUNCTION(1,4), EWWAIT(1,1) instead of EWWAIT(0,1), etc.

When a car comes from the West, it must decide which of the two lanes to take. If it is turning left, the left lane is taken and if it is turning right, the right lane is taken, but if it is going straight on, it can go in either lane. A car from the West goes through the junction in the following way:-

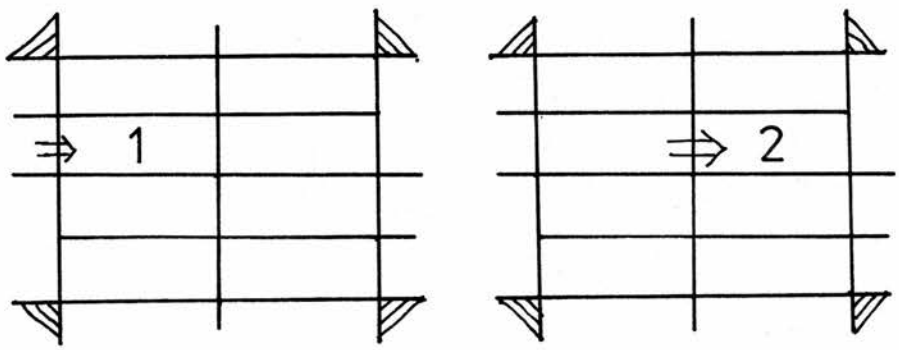
(a) West car turning left:-



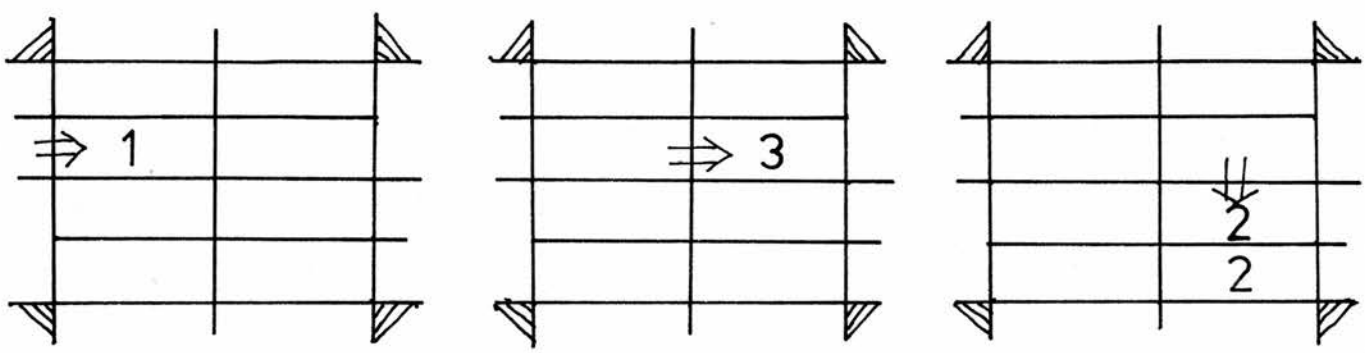
(b) West car going straight on (left lane):-



(c) West car going straight on (right lane):-



(d) West car turning right:-



A car coming from the East goes through similar states with JUNCTION(0,1) replacing JUNCTION(1,1), EWWAIT(0,2) replacing EWWAIT(1,2) etc.

It can therefore be seen that a section of the junction has a value 2 if there is a car in it whose next move is to leave the junction. A section has a value of 1 if a car has just entered the junction at this point and will move on to another part of the junction before leaving the junction completely. A section will have a value of 3 if the car in it is just about to make a right turn.

The state of the traffic lights is represented by an integer variable, LIGHTS, which can have 6 values:-

(i) LIGHTS = 0 - N. and S. red, E. and W. green

(ii) LIGHTS = 1 - N. and S. red, E. and W. amber

(iii) LIGHTS = 2 - N. and S. red/amber, E. and W. red

(iv) LIGHTS = 3 - N. and S. green, E. and W. red

(v) LIGHTS = 4 - N. and S. amber, E. and W. red

(vi) LIGHTS = 5 - N. and S. red, E. and W. red/amber

There is a transaction with no user-defined part which changes LIGHTS at regular intervals to the next higher value modulo 6.

There are four types of condition which can delay cars:-

(a) cars cannot move into a part of the junction if there already is another car there.

(b) Cars cannot enter the junction if the lights are not green for them, but once they are in the junction, they ignore the state of the lights.

(c) If a car from the West say, has entered JUNCTION(1,3) and the lights change before it has entered JUNCTION(1,4), a car from the North should not enter the junction until the West car has passed in front of it. This type of situation can occur with cars coming from all directions.

(d) If the situation can arise where cars from the West are in JUNCTION(1,4)

waiting to turn right and in JUNCTION(1,3), and cars from the East are in JUNCTION(0,4) waiting to to turn right and in JUNCTION(0,3), none of these cars will ever be able to move. This must be avoided by making cars from East and West not enter the junction if there are already cars in JUNCTION (0,4) and JUNCTION(1,4) waiting to turn right. This can also happen to cars from the North and South.

It can be seen that the paths of cars from the North and West are mirror images of the paths of cars from the South and East. The method of indexing the arrays JUNCTION, NSQ, EWQ, NSWAIT and EWWAIT means that if a car transaction has an associated record of type CAR with 2 integer fields. THIS and THAT holding values 1 and 0 for cars from the North and West and 0 and 1 for cars from the South and East, then both can execute the same code provided they index the above arrays using THIS(CURRENT) and THAT(CURRENT). NSQ(THIS (CURRENT)) will therefore mean NSQ(1) to a North car and NSQ(0) to a South car. Similarly JUNCTION(THAT(CURRENT),3) will mean JUNCTION(0,3) for a West car and JUNCTION(1,3) for an East car.

All cars also have another integer field, DIRECTION in their associated CAR record. This has a value of 1 if the car is going to turn left, 2 if it it will go straight ahead or 3 if it will turn right.

The actual program is given in Appendix D. It has been designed so that the sequencing of the lights, the inter-arrival times of cars coming from all directions and the proportion of cars that turn left, right or go straight on, can all be easily changed for testing the characteristics of the model under different circumstances. Those and the times for cars to move from one part of the junction to another would probably be partly determined by empirical observations of real-life junctions.

The program will first be explained without reference to statistic gathering (i.e. ignoring the procedures Q, UNQ, SETUPSTAT, RESETSTAT and OUTPUTSTAT).

Control Process

The control process first initialises the SLIM system and the "statics" of the model. (This consists of the Delaychains used and the JUNCTION array both of which are initialised as empty, the LIGHTS which start as 0 and the "random number stream" R which is given an initial "random" positive value).

One transaction is scheduled immediately with priority 0 to change the traffic lights at regular intervals. "Car" transactions are also scheduled to arrive at random intervals from each of the four directions with a priority of 2. The first thing each "car" is scheduled to do is to associate with itself a CAR record with a random direction of 1, 2, or 3 (with probabilities ¼, ½ and ¼ respectively). The fields THIS and THAT of the CAR record are set according to the direction the car is coming from. The car will then go to either the common process for North and South cars or the common process for East and West cars.

The control process then waits for 1,000 $\frac{1}{10}$ ths of a second to allow some cars to pass through the system so that the next run of 40,000 $\frac{1}{10}$ ths of a second will start with cars already in the system at different points.

After this run, the control process stops the whole program.

Traffic Lights

The traffic lights have a fixed cycle of 0 for 15 seconds, 1 for 3 sec., 2 for 2 secs, 3 for 10 secs, 4 for 3 secs and 5 for 2 secs. The transaction which changes the lights loops, waiting for the appropriate time and then changing the lights to their next value.

If the lights are changed so that they are now green for East and West or North and South, prompts are issued referring to the Delaychains which hold cars waiting to enter the junction from these directions as the changing of the lights might now allow them to move.

Cars from the North and South

The process representing cars from the North is the same as that representing cars from the South. It will therefore be explained in terms of cars from the North and the fields THIS and THAT of their CAR records will be assumed to be 1 and 0 respectively.

A North car first waits in NSQ(1) until

(i) the lights are green for North and South

and (ii) JUNCTION(1,2) and JUNCTION(1,4) are empty.

and (iii) there are neither cars from the West in JUNCTION(1,1) going straight on nor in JUNCTION(1,3)

and (iv) there is not a "cross-over" situation where there is a car from the North in JUNCTION(0,3) waiting to turn right, a car from the South in JUNCTION(1,3) waiting to turn right, while the current car wants to either, go right or straight ahead.

The car then goes into JUNCTION(1,2) and JUNCTION(1,4) and, if it is turning left, waits there a random time, leaves the junction and destroys itself. Otherwise, it waits a random time in that part of the junction and waits until JUNCTION(0,3) and JUNCTION(0,1) are empty in NSWAIT(1). It then leaves the first part of the junction, enters JUNCTION(0,3) and JUNCTION(0,1) and waits there a random time. If it is going straight on, it then leaves the junction and destroys itself. If it is turning right, however, it waits in EWWAIT(0,1) with priority 1 until JUNCTION(0,2) is empty. (It has a priority of 1 here so that cars from the South will enter that part of the junction in preference to it, if they both can go at the same time). It then goes into JUNCTION(0,2), waits a random time, leaves it and destroys itself.

On leaving each of the three parts of the junction, suitable prompts must be issued:-

(a) on leaving JUNCTION(1,2) and JUNCTION(1,4), the first car waiting at NSQ(1) might be able to move and there might be cars in EWWAIT(1,1) and EWWAIT(1,2) waiting for it to leave.

(b) on leaving JUNCTION(0,3) and JUNCTION(0,1) cars from the East in EWQ(0,1) and EWQ(0,2) (if the lights have changed) or a car in NSWAIT(1) might be able to move. If there was a car in NSQ(1) waiting for a "cross-over" situation to resolve itself, it also might now be able to move.

(c) on leaving JUNCTION(0,2) a car in either EWWAIT(0,1) or NSQ(0) might now be able to move.

Cars from the West and East

Cars from both the West and East execute the same process, being distinguished by their fields THIS and THAT. The process will be explained in terms of a car from the West with the 2 fields being 1 and 0 respectively.

A test is first made of whether the car will go into the left lane or the right lane. It will go into the left lane if it is turning left, or if it is going straight on and the queue of cars waiting in the left lane is shorter than the queue of cars in the right lane - i.e. HELDUP(EWQ(1,1) ≤ HELDUP(EWQ(1,2) If it is going straight on and the queue in the right lane is shorter, it will still choose the left lane with probability, .5. Otherwise, it takes the right lane.

If the car takes the left lane, it waits in EWQ(1,1) until:-

(i) the lights are green for E - W traffic

and (ii) JUNCTION(1,1) is empty

and (iii) there is not a South Car in JUNCTION(0,2) and JUNCTION(0,4) which will want to go straight on or right.

It then enters JUNCTION(1,1) and, if it is turning left, waits there for a random time, leaves the junction and destroys itself. If it is going straight on, it waits a random time in JUNCTION(1,1), waits in EWWAIT(1,1), until JUNCTION (1,2) is empty, leaves JUNCTION(1,1) and enters JUNCTION(1,2). It then waits a random time, leaves the junction and destroys itself.

If the car initially takes the right lane, it waits in EWQ(1,2) until:-

(i) the lights are green for E - W traffic

and (ii) JUNCTION(1,3) is empty

and (iii) there is not a South car in JUNCTION(0,2) and JUNCTION(0,4) which will want to go straight on or right.

and (iv) there is not a "cross-over" situation with a West car in JUNCTION(1,4) waiting to turn right and an East car in JUNCTION(0,4) waiting to turn right.

It then enters JUNCTION(1,3), waits there a random time, waits in EWWAIT (1,2) until JUNCTION(1,4) is empty, leaves JUNCTION(1,3) goes into JUNCTION(1,4) and waits there a random time. If it is going straight on, it then leaves the junction and destroys itself, but if it is turning right, it waits with a priorit priority of 1 in NSWAIT(1) until JUNCTION(0,3) and JUNCTION(0,1), are empty. It then leaves JUNCTION(1,4) enters JUNCTION(o,3) and JUNCTION(0,1), waits there a random time, leaves the junction and destroys itself.

On leaving each part of the junction, prompts are issued. These are not explained here, but are similar in effect to the ones issued by North and South cars.

Statistics Gathered

There are 6 QSTATS which gather queuing statistics on the lanes of traffic waiting to enter the junction. NSQSTAT(1) and NSQSTAT(0) gather queuing statistics on the queues of traffic from the North and South and EWQSTAT(1,1), (1,2) (0,1) and (0,2) gather statistics on the left and right lanes of West traffic and the left and right lanes of East traffic. Cars join these QSTATS when they enter the system and leave them as soon as they enter the actual junction.

12 other QSTATS gather statistics on the total time taken for cars to pass through the system. These statistics are separately gathered for cars coming from every direction and for cars turning left or right or going straight on. Cars join the appropriate QSTAT on entering the system and leave it on leaving the system. NSTOT(1,*), NSTOT(0,*), EWTOT(1,*) and EWTOT(0,*) gather these statistics for North, South, West and East cars, the "*" being 1 for cars turning left, 2 for those going straight on and 3 for those turning right.

The control process initialises all the QSTATs as empty before the first simulation run of 1000 units of time. After this run, the statistics are output and the QSTATs are reset to start gathering statistics freshly (the QSTATs however will not be empty at this point). After the main simulation run of 40,000 units of time, a new page is taken (IOCONTROL(3)) and the statistics for this run are output. It will be seen from the results shown in Appendix D that the average lengths of the queues and times for cars to pass through the system are higher for the second time. This is because it starts from a more random state of the system, rather than with no cars in the system at all.

6.1 Time-Sharing Computer System Simulation

This example simulates the operation of a time-sharing computer system which is paged and whose pages are dynamically relocatable. For any program to run, all of its pages must be in core, but they may be transferred to disc storage when the program is not running. User programs are assumed to be submitted to the system at random intervals and are allowed to execute in slices of 10 seconds until they are completed. If several programs are in the store (which is of size 16 pages) then the user programs are given time slices in turn.

Two pages of the store are permenantly used by the system, but there are also other system programs (perhaps for file allocation, I/O, etc) which are normally resident on disc but when required are brought into store. Requests for these system programs are assumed to occur at random and, when they occur, they are immediately brought into store with the pages required if necessary being taken from any user programs in core at the time (these will be transferred to disc). Control is immediately given to the system program which executes to completion. If there is more than one system program required at one time, they are queued, but all get serviced before any user programs get executed.

It is assumed here that disc transfers take a negligible time. In practice, the model would probably have to be modified to allow for them.

Statics of Model

Both user and system programs are represented by transactions with an associated record of type PROGRAM. The logical fields USER of each PROGRAM record is true if the program is a user program or false if the program is a system one. Each PROGRAM record also has integer fields TIME_TO_GO which is the time it still has to execute for and PAGES which is the number of pages that constitute the program. Its integer field SLICE is used to temporarily hold the next time-slice the program will have.

An integer variable PAGES_LEFT represents the number of pages that are unused in the store at any time. SYSTEM_USED keeps the number of pages that are being taken by system programs (including 2 permenant system pages) and the Group, STORE holds all user programs in store at any time. The user programs in this Group are ordered by the order in which they will be given a time-slice. The last to be given a time-slice will be at the start of the Group and the first (or currently executing) will be at the end.

The integer variable PREEM also holds the number of user programs which have been forced to leave store to make room for system programs.

The CPU is represented by a reference (EVENTNOTICE) variable, CPU, which points to the transaction executing in the CPU at that time.

There are, finally, 2 Delaychains in the model, QUEUE which contains the programs waiting for pages to become available in the store, and CPUQ which holds the programs in the store waiting to use the CPU for a time-slice.

Purpose of Program

The program shown in Appendix E is what might be used to test that the model works correctly. The times between creation of system and user programs and their sizes and execution times are non-random and are not typical of values that would be expected in the real-life system. However these values mean that conflicts over storage and CPU will occur and it is hoped that this will shown up any errors in the model. Whenever a program executes any action, it outputs a description of what it is doing using the procedure OUT.

Because the program is used to trace the operation of the model, certain procedures and variables have been used which would be omitted when the model was later being run without tracing.

These are:-

(1) procedure OUT

(2) integer field PROGNO of PROGRAM records

(3) integers USERNO and SYSTEMNO

(4) the non-random form of integer procedures RSYSTEM_DELAY, RUSER_DELAY, RSYSTEM_TIME, RUSER_TIME, RSYSTEM_PAGES and RUSER_PAGES.

All statements which refer to OUT, PROGNO, USERNO and SYSTEMNO would also be omitted. The program will first be explained without reference to these entities.

Control Process

The Delaychains used and the CPU are initialised as empty, and the store is initialised as having 2 pages used by system programs (permanantly used), no user programs in STORE, 14 pages unused and no programs preempted from the store by system programs. The "SLIM" system is then set up and user programs with priority 0 and system programs with priority 51 are scheduled with a random delay between each. The control process then waits 100 seconds to allow the user and system programs to interact and, after this time, stops the simulation.

User Programs

Each user program first associates with itself a PROGRAM record which has a random execution time and a random number of pages. It is also marked as a user program by the field USER being true.

Its first waits in QUEUE with priority 0 until there are enough pages left unused in store for it to fit in. It then enters the store by joining the Group STORE at the start and decreasing PAGES_LEFT.

It next loops, taking time-slices, until it has used the CPU for its specified execution time. Each time it wants a time slice, it waits in CPUQ with priority 0 until the CPU is unused, takes the CPU by making the pointer CPU point to it, and evaluates the time slice it will get this time (maximum of 10 seconds). It then waits in the CPU for this time slice with priority 52, leaves the CPU, issues a prompt to CPUQ and reduces its TIME_TO_GO field by the time slice it has had. Finally it reduces its priority back to 0, and, if it still has to have another time slice, changes its position in STORE from the end to the start.

Once it has had sufficient time-slices to reduce its TIME_TO_GO field to zero, it leaves the store by leaving the STORE Group and increasing PAGES_LEFT by the number of pages it has used. It also issues a prompt to QUEUE before destroying itself.

System Programs

Each system program first associates with itself a PROGRAM record with random execution time and number of pages. (Its field USER is marked false to show that it is not a user program).

It then waits in QUEUE with priority 51 until it can enter the store. This will be when it can fit in with the system programs already there because it can force user programs to leave. When it finds it can enter it forces user programs to leave the store until there is enough unused space for it, by superceding them by eventnotices which execute the code at STORE_PREEMPT. The user programs are removed in the order they are kept in the STORE Group, the first removed being the user program at the start of the Group which is the last one due for a time-slice. The system program then enters the store itself by decreasing PAGES_LEFT and increasing SYSTEM_USED by its own size.

It then waits in CPUQ with priority 51 until the CPU is either unused or used by a user program. If it is used by a user program, this user program is forced to give up the CPU by being superceded. The system program takes the CPU itself, calculates its time-slice (which is its total execution time) and waits this time. It finally gives up the CPU, issuing a prompt to CPUQ, and leaves the store by increasing the number of pages unused, decreasing the number of pages used by system programs and issuing a prompt to QUEUE.

Forcing a User Program to leave Store

To do this, a system program causes the user program to be superceded by an eventnotice, E. The CALL_IMMEDIATE used causes the user program to be removed from the Group STORE and E's first statement is to increase PAGES_LEFT. There are 2 cases; either the user program was executing at the time or not:-

(1) Eventnotice being superceded was executing.

The user program being superceded here must also be forced to leave the CPU as it cannot be executing when it is not in store. With a priority of 50, E then waits in QUEUE until there is once again enough room in store for it. When it can fit into store, it decreases the number of empty pages, puts the eventnotice E at the end of the STORE group (it will be given a time-slice before other user programs) and waits in CPUQ until the CPU is unused.

At this stage, another system program might come which wants to force the user program out of store again. Because E has been placed in STORE instead of the user program itself, E will be superceded and only once it has been again allowed back into store will it try to regain the CPU.

When the CPU is unused, CPU is made to point to the user program, and the user program is put at the end of STORE instead of E. E then destroys itself, allowing the user program to resume its use of the CPU.

(2) Eventnotice being superceded was not executing

The global variable PREEM is increased by one as there is another eventnotice forced to leave the store only. If the eventnotice being superceded was itself forcing a user program to give up the CPU, E's priority is left as 50, but if a user program is being directly superceded, E's priority is made equal to PREEM. E then waits in QUEUE with the priority until it can fit back into the store. The eventnotice being superceded is then put back into the STORE Group, the space left is decreased and PREEM is reduced by 1.

Finally E destroys itself allowing the superceded eventnotice to resume its wait for the CPU.

Forcing a User Program to give up the CPU

The user program is superceded by an eventnotice, E, which makes it leave the CPU by making the pointer CPU null. The user program will be in the Group, STORE at this time. It is removed from the end of STORE, E is replaced there and E waits with priority 50 in CPUQ until the CPU is again unused.

At this time, as in case (1) above, another system program can arrive which wants to make this user program leave the store. Because E is in STORE now, rather than the user program eventnotice itself, E will be superceded and will stop trying to get the CPU until it has been allowed back into the store.

When the CPU is free, the user program that E is superceding is replaced for E at the end of STORE, and CPU is made to point to it. E then destroys itself to allow the user program to resume its use of the CPU.

Priorities

Of the Eventnotices waiting in QUEUE to enter the store, system programs have the highest priority, of 51. User programs which have been forced to leave both the store and the CPU have the next highest priority, of 50, and

ordinary user programs have priority 0. The priorities of user transactions in QUEUE which have been forced to leave the store but have not been forced to give up the CPU are more complex. The order of forcing such transactions to leave the store is from the least elligible for a time slice to the most elligible. They should however be put back in the opposite order, so each user program forced out of store has a hgiher priority than the one forced out before it, from a priority of 1 upwards (it can never be as high as 50 however).

In CPUQ, user programs again have priority 0, eventnotices forcing user programs to leave the CPU have priority 50 and system programs have priority 51.

Also, when a user program is actually using the CPU, it has priority 52 so that, if it is scheduled to leave the CPU at the same time a system program is scheduled to want it, it will leave first, thus avoiding being forced to leave.

Tracing

Each user program has a unique number in its field PROGNO to characterise it. Each system program similarly has a unique number. These are assigned when each program is created, using the global variables USERNO and SYSTEMNO. These are increased by one each time a program of the appropriate type is created. These variables are also used to assign execution times and sizes to programs. For example, the Ith user program will be given the Ith value in the case statement of RUSER_TIME as its execution time.

When the control process executes the SCHEDULE_EVENTS to create user programs, USERNO will be 0, so the first user program will be scheduled after the first delay in RUSER_DELAY. When the control process executes WAIT(100) and the first user program is given control, USERNO will be 1, so the second user program will be scheduled after the second delay in RUSER_ _DELAY. When the second user program is given control, and the third is scheduled, USERNO will be 2, and the 3rd delay in RUSER_DELAY will be chosen, etc. The same applies to the delays between creations for system programs.

The delays between creations, execution times and program sizes can therefore be adjusted to create "exceptional circumstances" which help to test the correctness of the model.

When any program executes an action, it calls the procedure OUT which prints the value of simulated time, a description of the program calling OUT and the action this program has just done.

CHAPTER 7

CONCLUSION

The object of this research was to provide facilities in ALGOL W to allow discrete event simulation. Whereas it would be fairly easy to add facilities for event or activity-oriented simulation, it has actually proved possible to give the appearance of a process-oriented language.

The language produced is far more structured than GPSS and has several features in a more general form (e.g. preempts, global structures, structure of a transaction). It should also be faster than GPSS as it is not interpreted, though a direct comparison has not been made. Compared with SIMULA, however, there is a lack of program structure because restrictions must be made on the points of call of certain procedures (e.g. WAIT, TERMINATE). However there are some advantages even compared with SIMULA as there are more advanced scheduling statements available.

Improvements in the structure of the SLIM language could obviously be made, but this would almost certainly require a separate compiler.

There are two ways in which the research could be of more general use. Firstly the concepts of the WAITTILL and CALL_IMMEDIATE commands could be added to process-oriented simulation languages like SIMULA to give them more power. Secondly, the implementation method used could also be applied to other high-level procedure-oriented languages. The features a language should have for this to be possible are user-defined records, a method of passing parameters by name and a mthod of linking external low-level procedures.

APPENDIX A

LISTING INTERNAL "SLIM" PROCEDURES TO BE COPIED

The following declarations must be copied in a block surrounding each user program in SLIM. It is given here with a record of type CAR being associated with transactions. This record declaration would be replaced by any declarations required for user-defined records to be associated with transactions and the definitions of CURRENT, EVENTNOTICE and TRANSACTION would be altered to refer to these record types rather than CAR:-

```
INTEGER SIMTIME,DATASEG;
RECORD DELAYCHAIN(REFERENCE(EVENTNOTICE,DELAYCHAIN)FIRST,LAST;
                                                INTEGER HELDUP);
REFERENCE(EVENTNOTICE)FIRSTFE,FIRSTACTIVEDE,CURRENTEVNOTICE;
REFERENCE(EVENTNOTICE) ARRAY FIRSTDE,LASTDE(0::100);
REFERENCE(EVENTNOTICE,DELAYCHAIN)TEMP;
PROCEDURE STORERETURNADD(INTEGER VALUE X); ALGOL "SAVERETURN";
PROCEDURE REPLACERETADD(INTEGER VALUE X,Y); ALGOL "REPLACERET";
PROCEDURE STORETHUNKRE(INTEGER VALUE X); ALGOL "SAVETHUNKR";
PROCEDURE STOREDESTDELAY(INTEGER VALUE X);ALGOL "STOREDESTD";
LOGICAL PROCEDURE READY(INTEGER VALUE X,Y); ALGOL "EVALCON";
INTEGER PROCEDURE EVALDELAY(INTEGER VALUE X,Y); ALGOL "EVALDELAY";
PROCEDURE STOP;ALGOL"STOP";
INTEGER PROCEDURE LEVEL; ALGOL "LEVEL";
PROCEDURE ERROR1; ALGOL "ERRCROO1";


PROCEDURE FMERGE(REFERENCE(EVENTNOTICE)VALUE EV);
BEGIN REFERENCE(EVENTNOTICE)E,F;
      E:=FIRSTFE;
      F:=NULL;
      WHILE E¬=NULL AND(TIME(E)<TIME(EV) OR(TIME(E)=TIME(EV) AND
                                          PRIORITY(E)>=PRIORITY(EV)))
            DO BEGIN F:=E;
                     E:=RPOINTER(E)
               END;
      IF F=NULL THEN FIRSTFE:=EV ELSE RPOINTER(F):=EV;
      IF E¬=NULL THEN LPOINTER(E):=EV;
      LPOINTER(EV):=F;
      RPOINTER(EV):=E
END;


PROCEDURE DMERGE(REFERENCE(EVENTNOTICE)VALUE EV;REFERENCE(DELAYCHAIN)
                                                VALUE DC);
BEGIN INTEGER P;  P:=PRIORITY(EV);
      IF LASTDE(P)=NULL
      THEN BEGIN FIRSTDE(P):=EV;
                 LPOINTER(EV):=NULL;
                 TIME(EV):=0
           END
```

```
      ELSE BEGIN RPOINTER(LASTDE(P)):=EV;
                 LPOINTER(EV):=LASTDE(P);
                 TIME(EV):=TIME(LPOINTER(EV))+1;
           END;
      RPOINTER(EV):=NULL;
      LASTDE(P):=EV;
      HELDUP(DC):=HELDUP(DC)+1;
      TEMP:=LAST(DC);
      WHILE(TEMP IS EVENTNOTICE AND P>PRIORITY(TEMP)) DO
                                              TEMP:=APOINTER(TEMP);
      APOINTER(EV):=TEMP;
      BPOINTER(EV):=IF TEMP IS DELAYCHAIN THEN FIRST(TEMP)
                                  ELSE BPOINTER(TEMP);
      IF TEMP IS DELAYCHAIN THEN FIRST(TEMP):=EV ELSE BPOINTER(TEMP)
                                                    :=EV;
      IF BPOINTER(EV) IS DELAYCHAIN THEN LAST(BPOINTER(EV)):=EV
      ELSE APOINTER(BPOINTER(EV)):=EV;
      ACTIVE(EV):=TRUE;
      IF FIRSTACTIVEDE=NULL OR PRIORITY(FIRSTACTIVEDE)<PRIORITY(EV) THEN
                                               FIRSTACTIVEDE:=EV;
END;


PROCEDURE UNHOOKFE(REFERENCE(EVENTNOTICE)VALUE EV);
BEGIN IF EV=FIRSTFE THEN FIRSTFE:=RPOINTER(FIRSTFE) ELSE RPOINTER
                                        (LPOINTER(EV)):=RPOINTER(EV);
      IF RPOINTER(EV)¬=NULL THEN LPOINTER(RPOINTER(EV)):=LPOINTER(EV)
END;


PROCEDURE UNHOOKDE(REFERENCE(EVENTNOTICE)VALUE EV);
BEGIN TEMP:=APOINTER(EV);
      WHILE TEMP IS EVENTNOTICE DO TEMP:=APOINTER(TEMP);
      HELDUP(TEMP):=HELDUP(TEMP)-1;
      IF APOINTER(EV)IS DELAYCHAIN THEN FIRST(APOINTER(EV)):=BPOINTER(EV
                    ) ELSE BPOINTER(APOINTER(EV)):=BPOINTER(EV);
      IF BPOINTER(EV) IS DELAYCHAIN THEN LAST(BPOINTER(EV)):=APOINTER(EV
                   ) ELSE APOINTER(BPOINTER(EV)):=APOINTER(EV);
      IF RPOINTER(EV)¬=NULL THEN LPOINTER(RPOINTER(EV)):=LPOINTER(EV)
                  ELSE LASTDE(PRIORITY(EV)):=LPOINTER(EV);
      IF LPOINTER(EV)¬=NULL THEN RPOINTER(LPOINTER(EV)):=RPOINTER(EV)
                  ELSE FIRSTDE(PRIORITY(EV)):=RPOINTER(EV)

END;


PROCEDURE NEWACTIVE;
    WHILE(FIRSTACTIVEDE¬=NULL AND ¬ACTIVE(FIRSTACTIVEDE)) DO
        IF RPOINTER(FIRSTACTIVEDE)¬=NULL THEN
                  FIRSTACTIVEDE:=RPOINTER(FIRSTACTIVEDE)
         ELSE BEGIN INTEGER I;
                    I:=PRIORITY(FIRSTACTIVEDE)-1;
                    WHILE I>=0 AND FIRSTDE(I)=NULL DO I:=I-1;
                    FIRSTACTIVEDE:=IF I=-1  THEN NULL ELSE
                                                  FIRSTDE(I)
              END;
```

```
PROCEDURE WAIT(INTEGER VALUE T);
      IF T>=0 THEN
      BEGIN IF LEVEL¬=DATASEG THEN ERROR;
            STORERETURNADD(RET(CURRENTEVNOTICE));
            TIME(CURRENTEVNOTICE):=T+SIMTIME;
            FUTUREEV(CURRENTEVNOTICE):=TRUE;
            EMERGE(CURRENTEVNOTICE);
            GETNEWFIRSTEV;
            REPLACERETADD(RET(CURRENTEVNOTICE),DATASEG)
      END;


PROCEDURE WAITTILL(LOGICAL VALUE CON;REFERENCE(DELAYCHAIN) VALUE DC);
BEGIN IF LEVEL¬=DATASEG THEN ERROR;
      STORETHUNKRE(RET(CURRENTEVNOTICE));
      FUTUREEV(CURRENTEVNOTICE):=FALSE;
      EMERGE(CURRENTEVNOTICE,DC);
      GETNEWFIRSTEV;
      REPLACERETADD(RET(CURRENTEVNOTICE),DATASEG)
END;


PROCEDURE GETNEWFIRSTEV;
BEGIN LOGICAL FOUNDEVENT;
      FOUNDEVENT:=FALSE;
      WHILE FIRSTACTIVEDE¬=NULL AND(FIRSTFE=NULL OR TIME(FIRSTFE)
          >SIMTIME OR TIME(FIRSTFE)=SIMTIME AND PRIORITY(FIRSTACTIVEDE)

          >=PRIORITY(FIRSTFE))
               AND(BEGIN CURRENTEVNOTICE:=FIRSTACTIVEDE;
                         CURRENT:=TRANS(FIRSTACTIVEDE);
                         FOUNDEVENT:=TRUE;
                         ¬READY(THUNK(CURRENTEVNOTICE),DATASEG)
                   END)
             DO BEGIN ACTIVE(FIRSTACTIVEDE):=FALSE;
                      FOUNDEVENT:=FALSE;
                      NEWACTIVE;
                END;
      IF FOUNDEVENT THEN
      BEGIN ACTIVE(FIRSTACTIVEDE):=FALSE;
            NEWACTIVE;
            UNHOOKDE(CURRENTEVNOTICE)
      END
      ELSE
      BEGIN IF FIRSTFE =NULL THEN BEGIN WRITE
("SIMULATION STOPPED --- NO TRANSACTIONS CAN MOVE");
                                        STOP
                                  END;
            CURRENTEVNOTICE:=FIRSTFE;
            CURRENT:=TRANS(FIRSTFE);
            IF TIME(FIRSTFE)>SIMTIME THEN SIMTIME:=TIME(FIRSTFE);
            FIRSTFE:=RPOINTER(FIRSTFE);
            IF FIRSTFE¬=NULL THEN LPOINTER(FIRSTFE):=NULL;
            IF NUMBER(CURRENTEVNOTICE)>0 THEN
            BEGIN EMERGE(EVENTNOTICE(NULL,,,NULL,,,FALSE,TRUE,
                      SIMTIME+EVALDELAY(THUNK(CURRENTEVNOTICE),DATASEG),
                      PRIORITY(CURRENTEVNOTICE),RET(CURRENTEVNOTICE),
                      THUNK(CURRENTEVNOTICE),NUMBER(CURRENTEVNOTICE)-1
                      ,0));
                  NUMBER(CURRENTEVNOTICE):=0
            END
      END;
END;
```

```
PROCEDURE SCHEDULE_EVENTS(INTEGER VALUE DELAY,NC,PR;PROCEDURE DEST);
      BEGIN IF LEVEL¬=DATASEG THEN ERROR;
      TEMP:=EVENTNOTICE(NULL,,,NULL,,,FALSE,TRUE,SIMTIME+DELAY,PR,
                                                    ,,NC-1,0);
      STOREDESTDELAY(RET(TEMP));

      FMERGE(TEMP)
END;


PROCEDURE TESTFIRST(REFERENCE(DELAYCHAIN)VALUE DC);
IF HELDUP(DC)>0 AND ¬ACTIVE(FIRST(DC)) THEN
      BEGIN ACTIVE(FIRST(DC)):=TRUE;
            IF(FIRSTACTIVEDE=NULL OR
               PRIORITY(FIRST(DC))>PRIORITY(FIRSTACTIVEDE) OR PRIORITY
                     (FIRST(DC))=PRIORITY(FIRSTACTIVEDE) AND TIME
                     (FIRST(DC))<TIME(FIRSTACTIVEDE)) THEN
               FIRSTACTIVEDE:=FIRST(DC);
      END;


PROCEDURE TESTALL(REFERENCE(DELAYCHAIN)VALUE DC);
IF HELDUP(DC)>0 THEN
      BEGIN TEMP:=FIRST(DC);
            IF (FIRSTACTIVEDE=NULL OR
                PRIORITY(TEMP)>PRIORITY(FIRSTACTIVEDE) OR PRIORITY(TEMP)
                   =PRIORITY(FIRSTACTIVEDE) AND TIME(TEMP)<TIME
                                    (FIRSTACTIVEDE)) THEN
                 FIRSTACTIVEDE:=TEMP;
            WHILE TEMP IS EVENTNOTICE DO
                  BEGIN ACTIVE(TEMP):=TRUE;
                        TEMP:=BPOINTER(TEMP)
                  END
      END;


PROCEDURE INITIALISE;
BEGIN SIMTIME:=0;
      FIRSTACTIVEDE:=FIRSTFE:=NULL;
      FOR I:=0 UNTIL 100 DO FIRSTDE(I):=LASTDE(I):=NULL;
      DATASEG:=LEVEL;
      CURRENTEVNOTICE:=EVENTNOTICE(NULL,,,NULL,,,,,,0,,,0,0);
      CURRENT:=NULL
END;


REFERENCE(DELAYCHAIN)PROCEDURE NEWDELAYCHAIN;
BEGIN TEMP:=DELAYCHAIN(,,0);

      FIRST(TEMP):=LAST(TEMP):=TEMP;
      TEMP
END;
```

```
PROCEDURE CALL_IMMEDIATE(REFERENCE(EVENTNOTICE)VALUE EV;PROCEDURE PROC);
IF EV¬=NULL THEN
BEGIN IF LEVEL¬=DATASEG THEN ERROR;
      SUPBY(EV):=SUPBY(EV)+1;
      STOREBETURNADD(RET(CURRENTEVNOTICE));
      IF EV=CURRENTEVNOTICE THEN BEGIN TIME(EV):=0;
                                       FUTUREEV(EV):=TRUE
                                 END
      ELSE BEGIN TIME(CURRENTEVNOTICE):=-1;
                 RPOINTER(CURRENTEVNOTICE):=FIRSTFE;
                 LPOINTER(CURRENTEVNOTICE):=NULL;
                 IF FIRSTFE¬=NULL THEN LPOINTER(FIRSTFE):=
                                                  CURRENTEVNOTICE;
                 FIRSTFE:=CURRENTEVNOTICE;
                 IF SUPBY(EV)=1 AND RET(EV)¬=0 THEN
                 BEGIN IF FUTUREEV(EV) THEN
                           BEGIN TIME(EV):=TIME(EV)-SIMTIME;
                                 UNHOOKFE(EV)
                           END
                       ELSE BEGIN IF ACTIVE(EV) THEN
                                  BEGIN ACTIVE(EV):=FALSE;
                                        IF FIRSTACTIVEDE=EV THEN NEWACTIVE
                                  END;
                                  UNHOOKDE(EV);
                                  APOINTER(EV):=TEMP;
                                  TESTFIRST(TEMP)
                            END
                 END;
                 CURRENT:=TRANS(EV);
           END;
      CURRENTEVNOTICE:=EVENTNOTICE(CURRENT,,,EV,,,FALSE,,,PRIORITY(EV),,
                                                          ,0,0);
      PROC;
      ERROR1;
END;


PROCEDURE TERMINATE;
BEGIN RET(CURRENTEVNOTICE):=0;
      IF LEVEL¬=DATASEG THEN ERROR;
      IF SUPERCEDING(CURRENTEVNOTICE)¬=NULL THEN
      BEGIN CURRENTEVNOTICE:=SUPERCEDING(CURRENTEVNOTICE);
            SUPBY(CURRENTEVNOTICE):=SUPBY(CURRENTEVNOTICE)-1;
            IF SUPBY(CURRENTEVNOTICE)=0 AND RET(CURRENTEVNOTICE)¬=0 THEN
            BEGIN IF FUTUREEV(CURRENTEVNOTICE) THEN
                  BEGIN TIME(CURRENTEVNOTICE):=TIME(CURRENTEVNOTICE)
                                                  +SIMTIME;
                        FMERGE(CURRENTEVNOTICE)
                  END
                  ELSE DMERGE(CURRENTEVNOTICE,APOINTER(CURRENTEVNOTICE))
            END
      END;
      GETNEWFIRSTEV;
      REPLACERETADD(RET(CURRENTEVNOTICE),DATASEG)
END;
```

```
RECORD LINK(REFERENCE(EVENTNOTICE)T;REFERENCE(LINK)A,B);
RECORD GROUP(REFERENCE(LINK)START,CUR;INTEGER COUNT);
REFERENCE(GROUP) PROCEDURE NEWGROUP;
      GROUP(NULL,NULL,0);


PROCEDURE INSERT(REFERENCE(GROUP)VALUE G;STRING(6)VALUE S;REFERENCE
                        (EVENTNOTICE) VALUE E);
BEGIN COUNT(G):=COUNT(G)+1;
      IF START(G)=NULL THEN
      BEGIN CUR(G):=START(G):=LINK(E,,);
            A(START(G)):=B(START(G)):=START(G)
      END
      ELSE BEGIN REFERENCE(LINK)L;
                 IF S="START" OR S="END" THEN
                          BEGIN L:=LINK(E,A(START(G)),START(G));
                                IF S="START" THEN START(G):=CUR(G):=L
                          END
                 ELSE IF S="BEFORE" THEN
                                BEGIN L:=LINK(E,A(CUR(G)),CUR(G));
                                   IF START(G)=CUR(G) THEN START(G):=L
                                END

                 ELSE L:=LINK(E,CUR(G),B(CUR(G)));
                 A(B(L)):=B(A(L)):=L
           END
END;


REFERENCE(EVENTNOTICE)PROCEDURE SEARCH(REFERENCE(GROUP)VALUE G;STRING
                                (6)VALUE S; LOGICAL C);
BEGIN REFERENCE(LINK)L,LST;
      LOGICAL NOTFOUND;
      L:=CUR(G);
      IF L=NULL THEN NULL
      ELSE BEGIN LST:=A(L);
                 NOTFOUND:=TRUE;
                 WHILE L¬=LST AND(BEGIN TEMP:=T(L);
                                        NOTFOUND:=¬C;
                                        NOTFOUND
                                  END)
                       DO L:=B(L);
                 IF L=LST AND NOTFOUND THEN
                 BEGIN TEMP:=T(L);
                       NOTFOUND:=¬C
                 END;
                 IF NOTFOUND THEN NULL
                 ELSE BEGIN CUR(G):=IF S="STAY" THEN L ELSE B(L);
                            IF S="UNLINK"THEN
                            BEGIN IF A(L)=L THEN START(G):=CUR(G):=NULL
                                  ELSE BEGIN IF START(G)=L THEN START(G)
                                                        :=B(L);
                                             A(B(L)):=A(L);
                                             B(A(L)):=B(L)
                                       END;
                                  COUNT(G):=COUNT(G)-1;
                            END;
                            TEMP
                      END
           END
END;
```

```
PROCEDURE REMOVE(REFERENCE(GROUP)VALUE G;REFERENCE(EVENTNOTICE)VALUE E);
      TEMP:=SEARCH(G,"UNLINK",TEMP=E);

PROCEDURE JOIN(REFERENCE(GROUP)VALUE G;STRING(6)VALUE S);
BEGIN REFERENCE(EVENTNOTICE) E;
      E:=CURRENTEVNOTICE;
      WHILE SUPERCEDING(E)¬=NULL DO E:=SUPERCEDING(E);
      INSERT(G,S,E)
END;

PROCEDURE LEAVE(REFERENCE(GROUP) VALUE G);
BEGIN REFERENCE(EVENTNOTICE) E;
      E:=CURRENTEVNOTICE;
      WHILE SUPERCEDING(E)¬=NULL DO E:=SUPERCEDING(E);
      REMOVE(G,E)
END;

PROCEDURE SET(REFERENCE(GROUP)VALUE G);CUR(G):=START(G);

RECORD QSTAT(INTEGER NC,TOTAL,LASTTIME,STARTTIME,MAX,EMPTY,INT);
REFERENCE(QSTAT)PROCEDURE NEWQSTAT;
      QSTAT(0,0,SIMTIME,SIMTIME,0,0,0);
PROCEDURE Q(REFERENCE(QSTAT)VALUE X);
BEGIN IF NC(X)=0 THEN EMPTY(X):=EMPTY(X)+SIMTIME-LASTTIME(X);
      INT(X):=INT(X)+NC(X)*(SIMTIME-LASTTIME(X));
      LASTTIME(X):=SIMTIME;
      TOTAL(X):=TOTAL(X)+1;
      NO(X):=NO(X)+1;
      IF NO(X)>MAX(X) THEN MAX(X):=NO(X)
END;
PROCEDURE UNQ(REFERENCE(QSTAT)VALUE  X);
BEGIN INT(X):=INT(X)+NO(X)*(SIMTIME-LASTTIME(X));
      LASTTIME(X):=SIMTIME;
      NC(X):=NC(X)-1
END;
PROCEDURE OUTPUTQ(REFERENCE(QSTAT)VALUE X);
BEGIN INT(X):=INT(X)+NC(X)*(SIMTIME-LASTTIME(X));
      IF NO(X)=0 THEN EMPTY(X):=EMPTY(X)+SIMTIME-LASTTIME(X);
      LASTTIME(X):=SIMTIME;
      I_W:=6;R_W:=10;

      WRITE("AVELEN=",INT(X)/(SIMTIME-STARTTIME(X)),"MAXLEN=",MAX(X),
          "ENTERED=",TOTAL(X),"TRANSTIME=");
      IF TOTAL(X)=0 THEN WRITEON("       ***  ") ELSE WRITEON(INT(X)/
                                          TOTAL(X));
      WRITEON("EMPTYFOR=",EMPTY(X)/(SIMTIME-STARTTIME(X)),"CURLEN=",
          NO(X),"TIMEPERIOD=",SIMTIME-STARTTIME(X));
      I_W:=R_W:=14
END;
PROCEDURE RESETQ(REFERENCE(QSTAT)VALUE X);
BEGIN MAX(X):=NO(X);
      TOTAL(X):=EMPTY(X):=INT(X):=0;
      LASTTIME(X):=STARTTIME(X):=SIMTIME
END;
```

```
REAL PROCEDURE RFRACTION(INTEGER VALUE RESULT R);  ALGOL "RFRACTION";
INTEGER PROCEDURE RDEVN(REAL VALUE DEVN;INTEGER VALUE RESULT R);
      ROUND(DEVN*(2*RFRACTION(R)-1));
INTEGER PROCEDURE RNEGEXP(REAL VALUE MEAN;INTEGER VALUE RESULT R);
      ROUND(-MEAN*LN(RFRACTION(R)));
INTEGER PROCEDURE RNORMAL(REAL VALUE MEAN,SDEVN;INTEGER VALUE RESULT R);
BEGIN REAL X; X:=-6;
      FOR I:=1 UNTIL 12 DO X:=X+RFRACTION(R);
      ROUND(X*SDEVN+MEAN)
END;


PROCEDURE ERROR;
BEGIN WRITE
("SIMULATION STOPPED --- RESTRICTED PROC CALLED AT WRONG LEVEL");
      STOP
END;


RECORD CAR(INTEGER DIRECTION,THIS,THAT);
REFERENCE(CAR) CURRENT;
RECORD EVENTNOTICE(REFERENCE(CAR)TRANS;REFERENCE(EVENTNOTICE)LPOINTER,
    RPOINTER,SUPERCEDING;REFERENCE(EVENTNOTICE,DELAYCHAIN)APOINTER,
    BPOINTER;LOGICAL ACTIVE,FUTUREEV;INTEGER TIME,PRIORITY,RET,THUNK,
    NUMBER,SUPBY);
PROCEDURE TRANSACTION(REFERENCE(CAR)VALUE NEWTRANS);
BEGIN CURRENT:=NEWTRANS;

      TRANS(CURRENTEVNOTICE):=NEWTRANS
END;
```

APPENDIX B

LISTING OF EXTERNAL "SLIM" PROCEDURES

The following procedures are compiled separately from any user program in SLIM and are linked to it at runtime. The first is in ALGOL W and the rest are in PL360.

```
CCCC --       PRCCEDURE ERRCR1;
C001 1-       BEGIN PROCEDURE STOP; ALGCL "STCP";
C004 --             WRITE
CC04 --       ("SIMULATICN STCPPED --- TRANSACTICN'S FIRST ACTICN HAS NC GOTC");
0005 --             STCP
0005 -1       END.

EXECUTICN CPTICNS: DEBUG,0 TIME=97269 PAGES=32767

CC0.12 SECONDS IN COMPILATICN, (00188, CCCCC) BYTES CF CODE GENERATED
```

```
   0001 -- GLOBAL PROCEDURE LEVEL(R1) BASE R14;
   CC02 1- BEGIN R2:=B13(72);
   CC03 --       LA(R3,B2(24));
   0004 -1 END.

   SEGMENT 014  NAME = LEVEL      LENGTH = CC1C  BASE REG = 14
```

```
   0001 -- GLCBAL PRCCEDURE SAVERETURNADD(R1) BASE R14;
   CC02 1- BEGIN R5=>B13(72);
   0003 --       R0:=R5+48;
   0004 --       STM(R0,R2,B5(16));
   CC05 --       LM(R3,R4,B5(40));
   CC06 --       BALR(R1,R3);
   0007 --       R5:=B2(24);
   CC08 --       R4:=B5(20)=>B3;
   0009 --       R5=>B13(72);
   0010 --       LM(R1,R2,B2(20));
   C011 -1 END.

   SEGMENT C14  NAME = SAVERETURN LENGTH = 0030  BASE REG = 14
```

```
0001 -- GLOBAL PROCEDURE SAVETHUNKRETADD(R1) BASE R14;
0002 1- BEGIN R5=>B13(72);
0003 --       R0:=R5+48;
0004 --       STM(R0,R2,B5(16));
0005 --       LM(R3,R4,B5(40));
0006 --       BALR(R1,R3);
0007 --       R5:=B2(24);
0008 --       R4:=B5(20)=>B3;
0009 --       R4:=B5(40)=>B3(4);
0010 --       R5=>B13(72);
0011 --       LM(R1,R2,B2(20));
0012 -1 END.

SEGMENT 014  NAME = SAVETHUNKR LENGTH = 0038  BASE REG = 14



0001 -- GLOBAL PROCEDURE REPLACERETADD(R1) BASE R14;
0002 1- BEGIN EXTERNAL PROCEDURE ERRCRC01(R1) BASE R14;NULL;
0003 --       R5=>B13(72);
0004 --       R0:=R5+56;
0005 --       R3=R3;
0006 --       STM(R14,R3,B5(8));
0007 --       LM(R3,R4,B5(40));
0008 --       BALR(R1,R3);
0009 --       R14:=B2(8);
0010 --       R3:=B3;
0011 2-       IF R3>0 THEN BEGIN R5:=B2(24);
0012 --                          R3=>B5(20);
0013 --                          R5=>B13(72);
0014 --                          LM(R1,R2,B2(20));
0015 -2                    END
0016 2-       ELSE BEGIN R1:=B2(48);
0017 --                  R4:=R1;
0018 --                  BALR(R1,R3);
0019 --                  R14:=B2(8);
0020 --                  ERRCR001;
0021 -2            END;
0022 -1 END.

SEGMENT 014  NAME = REPLACERET LENGTH = 006C  BASE REG = 14



0001 -- GLOBAL PROCEDURE STOREDESTDELAY(R1) BASE R14;
0002 1- BEGIN R5=>B13(72);
0003 --       R0:=R5+48;
0004 --       STM(R14,R2,B5(8));
0005 --       LM(R3,R4,B5(40));
0006 --       BALR(R1,R3);
0007 --       R5:=B2(24);
0008 --       R4:=B5(64)=>B3;
0009 --       R4:=B5(40)=>B3(4);
0010 --       R5=>B13(72);
0011 --       LM(R1,R2,B2(20));
0012 -1 END.

SEGMENT 014  NAME = STOREDESTD LENGTH = 0038  BASE REG = 14
```

```
0001 -- GLOBAL PROCEDURE EVALCON(R1) BASE R14;
0002 1- BEGIN R5=>B13(72);
0003 --       R0:=R5+56;
0004 --       R3-R3;
0005 --       STM(R14,R3,B5(8));
0006 --       LM(R3,R4,B5(40));
0007 --       BALR(R1,R3);
0008 --       R14:=B2(8);
0009 --       R3:=B3;
0010 2-       IF R3<0 THEN BEGIN R4:=B2(48);
0011 --                         R4:=B4;
0012 --                         BALR(R1,R3);
0013 -2                   END;
0014 --       R5:=B2(24)=>B13(72);
0015 --       LM(R1,R2,B2(20));
0016 -1 END.

SEGMENT 014  NAME = EVALCON     LENGTH = 0040  BASE REG = 14
```

```
0001 -- GLOBAL PROCEDURE EVALDELAY(R1) BASE R14;
0002 1- BEGIN R5=>B13(72);
0003 --       R0:=R5+56;
0004 --       R3-R3;
0005 --       STM(R14,R3,B5(8));
0006 --       LM(R3,R4,B5(40));
0007 --       BALR(R1,R3);
0008 --       R14:=B2(8);
0009 --       R3:=B3;
0010 2-       IF R3<0 THEN BEGIN R4:=B2(48);
0011 --                         R4:=B4;
0012 --                         BALR(R1,R3);
0013 -2                   END;
0014 --       R3:=B3;
0015 --       R5:=B2(24)=>B13(72);
0016 --       LM(R1,R2,B2(20));
0017 -1 END.

SEGMENT 014  NAME = EVALDELAY  LENGTH = 0048  BASE REG = 14
```

```
0001 -- GLOBAL PROCEDURE STOP(R1) BASE R14;
0002 1- BEGIN LM(R1,R2,B12(20));
0003 --       B13(72):=R2;
0004 -1 END.

SEGMENT 014  NAME = STOP        LENGTH = 0010  BASE REG = 14
```

```
0001 -- GLOBAL PROCEDURE RFRACTICN(R1) BASE R14;
0002 1- BEGIN LONG REAL NC1 SYN B13(280),NC2 SYN B13(272);
0003 --       R2:=B13(72);
0004 --       R5=>B13(72);
0005 --       R0:=R5+56;
0006 --       STM(R14,R2,B5(8));
0007 --       LM(R3,R4,B5(40));
0008 2-       IF R3<0 THEN BEGIN BALR(R1,R3);
0009 --                         R14:=B2(8);
0010 --                         R5:=R2;
0011 -2                   END;
0012 --       R1:=B3=>R0 SHRL 13 XOR R0=>R0 SHLL 18;
0013 --       R0 XOR R1 & %7FFFFFFF => B3;
0014 --       R0=>B13(276);
0015 --       XI(%80,B13(276));
0016 --       F01:=NC2+NC1*0.4656613'_9L;
0017 --       LM(R1,R2,B5(20));
0018 --       B13(72):=R2;
0019 -1 END.

SEGMENT 014  NAME = RFRACTION  LENGTH = 0070  BASE REG = 14
```

APPENDIX C

SYSTEM ENTITIES AVAILABLE TO USERS

The following are the record types, variables and procedures which should be used by "users" :-

```
integer  SIMTIME

record  EVENTNOTICE(reference (user-defined records)TRANS; reference
                    (EVENTNOTICE)SUPERCEDING; integer PRIORITY, SUPBY)

reference (EVENTNOTICE) CURRENTEVNOTICE

reference (user-defined records) CURRENT

reference (EVENTNOTICE) TEMP


record DELAYCHAIN (integer HELDUP)

reference (DELAYCHAIN) procedure NEWDELAYCHAIN

procedure TESTFIRST (reference (DELAYCHAIN)DC)

procedure TESTALL (reference (DELAYCHAIN)DC)


procedure WAIT (integer T)

procedure WAITTILL (logical CON; reference (DELAYCHAIN)DC)

procedure TERMINATE

procedure STOP

procedure SCHEDULE_EVENTS (integer DELAY, NO, PR ; procedure DEST)

procedure INITIALISE


procedure CALL_IMMEDIATE (reference (EVENTNOTICE)EV; procedure PROC)
```

```
record GROUP (integer COUNT)
reference (GROUP) procedure NEWGROUP
procedure INSERT (reference (GROUP)G; string (6) S; reference (EVENTNOTICE)E)
procedure JOIN (reference (GROUP)G; string (6) S)
reference(EVENTNOTICE) procedure SEARCH(reference(GROUP)G; string (6) S;
                                                          logical C)
procedure REMOVE (reference (GROUP)G; reference (EVENTNOTICE)E)
procedure LEAVE (reference (GROUP)G)
procedure SET (reference (GROUP)G)

record QSTAT
reference (QSTAT) procedure NEWQSTAT
procedure Q (reference (QSTAT)X)
procedure UNQ (reference (QSTAT)X)
procedure OUTPUTQ (reference (QSTAT)X)
procedure RESETQ (reference (QSTAT)X)

real procedure RFRACTION (integer R)
integer procedure RDEVN (real DEVN ; integer R)
integer procedure RNEGEXP (real MEAN,; integer R)
integer procedure RNORMAL (real MEAN,SDEVN ; integer R)
```

```
0000 1-      BEGIN
0001 --
0001 --      COMMENT *****SYSTEM PROCEDURES COPIED HERE*****;
0001 --
0320 --      RECORD CAR(INTEGER DIRECTION,THIS,THAT);
0321 --
0321 --      COMMENT *****SYSTEM PROCEDURES REFERING TO "CAR" RECORD COPIED*****;
0321 --
0331 2-      BEGIN
0332 --      INTEGER ARRAY JUNCTION(0::1,1::4);
0333 --      REFERENCE(DELAYCHAIN)ARRAY EWQ,EWWAIT(0::1,1::2);
0334 --      REFERENCE(DELAYCHAIN)ARRAY NSQ,NSWAIT(0::1);
0335 --      INTEGER LIGHTS,R;
0336 --      REFERENCE(QSTAT)ARRAY EWQSTAT(0::1,1::2);
0337 --      REFERENCE(QSTAT)ARRAY NSQSTAT(0::1);
0338 --      REFERENCE(QSTAT)ARRAY NSTOT,EWTOT(0::1,1::3);
0339 --      PROCEDURE SETUPSTAT;
0340 --      FOR I:=0 UNTIL 1 DO
0340 3-      BEGIN FOR J:=1 UNTIL 2 DO EWQSTAT(I,J):=NEWQSTAT;
0342 --            NSQSTAT(I):=NEWQSTAT;
0343 4-            FOR J:=1 UNTIL 3 DO BEGIN NSTOT(I,J):=NEWQSTAT;
0345 --                                       EWTOT(I,J):=NEWQSTAT
0345 -4                                  END
0345 -3      END;
0346 --      PROCEDURE RESETSTAT;
0347 --      FOR I:=0 UNTIL 1 DO
0347 3-      BEGIN FOR J:=1 UNTIL 2 DO RESETQ(EWQSTAT(I,J));
0349 --            RESETQ(NSQSTAT(I));
0350 4-            FOR J:=1 UNTIL 3 DO BEGIN RESETQ(NSTOT(I,J));
0352 --                                       RESETQ(EWTOT(I,J))
0352 -4                                  END
0352 -3      END;
0353 --      PROCEDURE OUTPUTSTAT;
0354 --      FOR I:=0 UNTIL 1 DO
0354 3-      BEGIN WRITE(CASE(I+1) OF ("SOUTH","NORTH"));
0356 --            WRITE("QUEUE:--");
0357 --            OUTPUTQ(NSQSTAT(I));
0358 --            FOR J:=1 UNTIL 3 DO
0358 4-            BEGIN WRITE(CASE J OF ("LEFT ","AHEAD","RIGHT")," TRAFFIC:--");
0360 --                  OUTPUTQ(NSTOT(I,J))
0360 -4            END;
0361 --            WRITE(" ");
```

```
0362 --             WRITE(CASE(I+1) OF ("EAST","WEST"));
0363 --             FOR J:=1 UNTIL 2 DO
0363 4-             BEGIN WRITE(CASE J OF ("LEFT","RIGHT")," LANE QUEUE:--");
0365 --                   OUTPUTQ(EWQSTAT(I,J))
0365 -4             END;
0366 --             FOR J:=1 UNTIL 3 DO
0366 4-             BEGIN WRITE(CASE J OF ("LEFT ","AHEAD","RIGHT")," TRAFFIC:--");
0368 --                   OUTPUTQ(EWTOT(I,J))
0368 -4             END;
0369 --             WRITE(" ");
0370 -3       END;
0371 --       PROCEDURE SETUP;
0372 3-       BEGIN FOR I:=0 UNTIL 1 DO
0373 4-             BEGIN FOR J:=1 UNTIL 2 DO BEGIN EWQ(I,J):=NEWDELAYCHAIN;
0376 --                                             EWWAIT(I,J):=NEWDELAYCHAIN
0376 -5                                       END;
0377 --                   NSQ(I):=NEWDELAYCHAIN;
0378 --                   NSWAIT(I):=NEWDELAYCHAIN;
0379 --                   FOR J:=1 UNTIL 4 DO JUNCTION(I,J):=0
0379 -4             END;
0380 --             LIGHTS:=0;
0381 --             R:=46872;
0382 -3       END;
0383 --       INTEGER PROCEDURE RDIRECTION;
0384 3-       BEGIN REAL R1;
0386 --             R1:=RFRACTION(R);
0387 --             IF R1<.25 THEN 1 ELSE IF R1<.75 THEN 2 ELSE 3
0387 -3       END;
0388 --       LOGICAL PROCEDURE LLANE;
0389 --       DIRECTION(CURRENT)=1 OR DIRECTION(CURRENT)=2 AND(HELDUP(EWQ(THIS(CURRENT
0389 --                  ),1))<=HELDUP(EWQ(THIS(CURRENT),2)) OR RFRACTION(R)>=.5);
0390 --       PROCEDURE LIGHTTEST;
0391 --       IF LIGHTS=0 THEN
0391 33         BEGIN FOR I:=0 UNTIL 1 DO FOR J:=1 UNTIL 2 DO TESTFIRST(EWQ(I,J)) END
0392 --       ELSE IF LIGHTS=3 THEN
0392 --               FOR I:=0 UNTIL 1 DO TESTFIRST(NSQ(I));
0393 --       LOGICAL PROCEDURE NSREADY;
0394 3-       BEGIN INTEGER I,J;
0396 --             I:=THIS(CURRENT);
0397 --             J:=THAT(CURRENT);
0398 --             (LIGHTS=3 AND JUNCTION(I,2)=0 AND JUNCTION(I,4)=0 AND JUNCTION
0398 --                   (I,1)¬=1 AND JUNCTION(I,3)¬=1 AND¬(JUNCTION(J,3)=3 AND
```

```
0398 --                 JUNCTION(I,3)=3 AND DIRECTION (CURRENT)¬=1))
0398 -3       END;
0399 --       LOGICAL PROCEDURE EWREADY1;
0400 3-       BEGIN INTEGER I,J;
0402 --             I:=THIS(CURRENT);
0403 --             J:=THAT(CURRENT);
0404 --             (LIGHTS=0 AND JUNCTION(I,1)=0 AND JUNCTION(J,4)¬=1)
0404 -3       END;
0405 --       LOGICAL PROCEDURE EWREADY2;
0406 3-       BEGIN INTEGER I,J;
0408 --             I:=THIS(CURRENT);
0409 --             J:=THAT(CURRENT);
0410 --             (LIGHTS=0 AND JUNCTION(I,3)=0 AND JUNCTION(J,4)¬=1 AND
0410 --                   ¬(JUNCTION(I,4)=3 AND JUNCTION(J,4)=3))
0410 -3       END;
0411 --       PROCEDURE NSLEAVE1;
0412 3-       BEGIN INTEGER I;
0414 --             I:=THIS(CURRENT);
0415 --             JUNCTION(I,2):=JUNCTION(I,4):=0;
0416 --             TESTFIRST(NSQ(I));
0417 --             FOR K:=1 UNTIL 2 DO TESTFIRST(EWWAIT(I,K));
0418 --             IF DIRECTION(CURRENT)=1 THEN UNQ(NSTOT(THIS(CURRENT),1));
0419 -3       END;
0420 --       PROCEDURE NSLEAVE2;
0421 3-       BEGIN INTEGER I,J;
0423 --             I:=THIS(CURRENT);
0424 --             J:=THAT(CURRENT);
0425 --             JUNCTION(J,3):=JUNCTION(J,1):=0;
0426 --             TESTFIRST(NSWAIT(I));
0427 --             FOR K:=1 UNTIL 2 DO TESTFIRST(EWQ(J,K));
0428 --             IF DIRECTION(CURRENT)=3 THEN TESTFIRST(NSQ(I))
0428 --             ELSE UNQ(NSTOT(THIS(CURRENT),2));
0429 -3       END;
0430 --       PROCEDURE NSLEAVE3;
0431 3-       BEGIN INTEGER J;
0433 --             J:=THAT(CURRENT);
0434 --             JUNCTION(J,2):=0;
0435 --             TESTFIRST(EWWAIT(J,1));
0436 --             TESTFIRST(NSQ(J));
0437 --             UNQ(NSTOT(THIS(CURRENT),3));
0438 -3       END;
0439 --       PROCEDURE EWLEAVE11;
```

```
0440 3-       BEGIN INTEGER I,J;
0442 --             I:=THIS(CURRENT);
0443 --             J:=THAT(CURRENT);
0444 --             JUNCTION(I,1):=0;
0445 --             TESTFIRST(EWQ(I,1));
0446 --             TESTFIRST(NSWAIT(J));
0447 --             IF DIRECTION(CURRENT)=1 THEN UNQ(EWTOT(THIS(CURRENT),1));
0448 -3       END;
0449 --       PROCEDURE EWLEAVE12;
0450 3-       BEGIN INTEGER I;
0452 --             I:=THIS(CURRENT);
0453 --             JUNCTION(I,2):=0;
0454 --             TESTFIRST(EWWAIT(I,1));
0455 --             TESTFIRST(NSQ(I));
0456 --             UNQ(EWTOT(THIS(CURRENT),2));
0457 -3       END;
0458 --       PROCEDURE EWLEAVE21;
0459 3-       BEGIN INTEGER I,J;
0461 --             I:=THIS(CURRENT);
0462 --             J:=THAT(CURRENT);
0463 --             JUNCTION(I,3):=0;
0464 --             TESTFIRST(EWQ(I,2));
0465 --             TESTFIRST(NSWAIT(J));
0466 -3       END;
0467 --       PROCEDURE EWLEAVE22;
0468 3-       BEGIN INTEGER I;
0470 --             I:=THIS(CURRENT);
0471 --             JUNCTION(I,4):=0;
0472 --             TESTFIRST(EWWAIT(I,2));
0473 --             TESTFIRST(NSQ(I));
0474 --             IF DIRECTION(CURRENT)=3 THEN TESTFIRST(EWQ(I,2))
0474 --             ELSE UNQ(EWTOT(THIS(CURRENT),2));
0475 -3       END;
0476 --       PROCEDURE EWLEAVE3;
0477 3-       BEGIN INTEGER I,J;
0479 --             I:=THIS(CURRENT);
0480 --             J:=THAT(CURRENT);
0481 --             JUNCTION(J,3):=JUNCTION(J,1):=0;
0482 --             TESTFIRST(NSWAIT(I));
0483 --             FOR K:=1 UNTIL 2 DO TESTFIRST(EWQ(J,K));
0484 --             UNQ(EWTOT(THIS(CURRENT),3));
0485 -3       END;
```

```
0486 --
0486 --
0486 --       INITIALISE;
0487 --       SETUP;
0488 --       SCHEDULE_EVENTS(0,1,0,GOTO LIGHTCHANGE);
0489 --       SCHEDULE_EVENTS(RNEGEXP(50,R),1000,2,
0489 3-                                  BEGIN TRANSACTION(CAR(RDIRECTION,1,0));
0491 --                                      Q(EWTOT(1,DIRECTION(CURRENT)));
0492 --                                      GOTO EWCAR
0492 -3                                  END);
0493 --       SCHEDULE_EVENTS(RNEGEXP(50,R),1000,2,
0493 3-                                  BEGIN TRANSACTION(CAR(RDIRECTION,0,1));
0495 --                                      Q(EWTOT(0,DIRECTION(CURRENT)));
0496 --                                      GOTO EWCAR
0496 -3                                  END);
0497 --       SCHEDULE_EVENTS(RNEGEXP(150,R),1000,2,
0497 3-                                  BEGIN TRANSACTION(CAR(RDIRECTION,1,0));
0499 --                                      Q(NSTOT(1,DIRECTION(CURRENT)));
0500 --                                      GOTO NSCAR
0500 -3                                  END);
0501 --       SCHEDULE_EVENTS(RNEGEXP(150,R),1000,2,
0501 3-                                  BEGIN TRANSACTION(CAR(RDIRECTION,0,1));
0503 --                                      Q(NSTOT(0,DIRECTION(CURRENT)));
0504 --                                      GOTO NSCAR
0504 -3                                  END);
0505 --       SETUPSTAT;
0506 --       WAIT(1000);
0507 --       OUTPUTSTAT;
0508 --       RESETSTAT;
0509 --       WAIT(40000);
0510 --       ICCONTROL(3);
0511 --       OUTPUTSTAT;
0512 --       STOP;
0513 --       NSCAR:Q(NSQSTAT(THIS(CURRENT)));
0514 --             WAITTILL(NSREADY,NSQ(THIS(CURRENT)));
0515 --             UNQ(NSQSTAT(THIS(CURRENT)));
0516 --             JUNCTION(THIS(CURRENT),2):=JUNCTION(THIS(CURRENT),4):=
0516 --                   IF DIRECTION(CURRENT)=1 THEN 2 ELSE 1;
0517 3-             IF DIRECTION(CURRENT)=1 THEN BEGIN WAIT(30+RDEVN(10,R));
0519 --                                                NSLEAVE1;
0520 --                                                TERMINATE
0520 -3                                       END
```

```
0520 3-          ELSE BEGIN WAIT(20+RDEVN(7,R));
0522 --                     WAITTILL(JUNCTION(THAT(CURRENT),3)=0 AND JUNCTION(THAT
0522 --                              (CURRENT),1)=0,NSWAIT(THIS(CURRENT)));
0523 --                     NSLEAVE1;
0524 --                     JUNCTION(THAT(CURRENT),3):=JUNCTION(THAT(CURRENT),1):=
0524 --                                                DIRECTION(CURRENT);
0525 --                     WAIT(20+RDEVN(7,R));
0526 4-                     IF DIRECTION(CURRENT)=2 THEN BEGIN NSLEAVE2;
0528 --                                                       TERMINATE
0528 -4                                                 END
0528 4-                     ELSE BEGIN PRIORITY(CURRENTEVNOTICE):=1;
0530 --                                WAITTILL(JUNCTION(THAT(CURRENT),2)=0,EWWAIT
0530 --                                                  (THAT(CURRENT),1));
0531 --                                NSLEAVE2;
0532 --                                JUNCTION(THAT(CURRENT),2):=2;
0533 --                                WAIT(20+RDEVN(7,R));
0534 --                                NSLEAVE3;
0535 --                                TERMINATE
0535 -4                           END
0535 -3               END;
0536 --       EWCAR:IF LLANE THEN
0536 3-               BEGIN Q(EWQSTAT(THIS(CURRENT),1));
0538 --                     WAITTILL(EWREADY1,EWQ(THIS(CURRENT),1));
0539 --                     UNQ(EWQSTAT(THIS(CURRENT),1));
0540 --                     JUNCTION(THIS(CURRENT),1):=IF DIRECTION(CURRENT)=1 THEN 2
0540 --                                                                   ELSE 1;
0541 4-                     IF DIRECTION(CURRENT)=1 THEN BEGIN WAIT(30+RDEVN(10,R));
0543 --                                                        EWLEAVE11;
0544 --                                                        TERMINATE
0544 -4                                                  END
0544 4-                     ELSE BEGIN WAIT(20+RDEVN(7,R));
0546 --                                WAITTILL(JUNCTION(THIS(CURRENT),2)=0,EWWAIT
0546 --                                                  (THIS(CURRENT),1));
0547 --                                EWLEAVE11;
0548 --                                JUNCTION(THIS(CURRENT),2):=2;
0549 --                                WAIT(20+RDEVN(7,R));
0550 --                                EWLEAVE12;
0551 --                                TERMINATE
0551 -4                           END
0551 -3               END
0551 3-               ELSE BEGIN Q(EWQSTAT(THIS(CURRENT),2));
0553 --                           WAITTILL(EWREADY2,EWQ(THIS(CURRENT),2));
```

```
0554 --                          UNQ(EWQSTAT(THIS(CURRENT),2));
0555 --                          JUNCTION(THIS(CURRENT),3):=1;
0556 --                          WAIT(20+RDEVN(7,R));
0557 --                          WAITTILL(JUNCTION(THIS(CURRENT),4)=0,
0557 --                                                EWWAIT(THIS(CURRENT),2));
0558 --                          EWLEAVE21;
0559 --                          JUNCTION(THIS(CURRENT),4):=DIRECTION(CURRENT);
0560 --                          WAIT(20+RDEVN(7,R));
0561 4-                          IF DIRECTION(CURRENT)=2 THEN BEGIN EWLEAVE22;
0563 --                                                          TERMINATE
0563 -4                                                       END
0563 4-                          ELSE BEGIN PRIORITY(CURRENTEVNOTICE):=1;
0565 --                                     WAITTILL(JUNCTION(THAT(CURRENT),3)=0 AND
0565 --                                         JUNCTION(THAT(CURRENT),1)=0,NSWAIT
0565 --                                                        (THIS(CURRENT)));
0566 --                                     EWLEAVE22;
0567 --                                     JUNCTION(THAT(CURRENT),3):=JUNCTION
0567 --                                              (THAT(CURRENT),1):=2;
0568 --                                     WAIT(20+RDEVN(7,R));
0569 --                                     EWLEAVE3;
0570 --                                     TERMINATE
0570 -4                                  END
0570 -3                        END;
0571 --       LIGHTCHANGE:WHILE TRUE DO
0571 3-                        BEGIN WAIT(CASE(LIGHTS+1)OF (150,30,20,100,30,20));
0573 --                              LIGHTS:=(LIGHTS+1)REM 6;
0574 --                              LIGHTTEST
0574 -3                        END;
0575 -2       END
0575 -1       END.

EXECUTION OPTIONS: DEBUG,0 TIME=97369 PAGES=32767


 *NOTE 2031 NEAR COORDINATE 0230 - WARNING: NAME PARAMETER SPECIFIED
   (FOUND NEAR " C ) ")

016.90 SECONDS IN COMPILATION, (30696, 00000) BYTES OF CODE GENERATED
```

```
SOUTH
QUEUE:--
AVELEN=      0.283 MAXLEN=   2 ENTERED=   6 TRANSTIME=      47.2 EMPTYFOR=      0.719 CURLEN=   0 TIMEPERIOD= 1000
LEFT  TRAFFIC:--
AVELEN=      0.263 MAXLEN=   1 ENTERED=   2 TRANSTIME=      131. EMPTYFOR=      0.737 CURLEN=   0 TIMEPERIOD= 1000
AHEAD TRAFFIC:--
AVELEN=      0.206 MAXLEN=   1 ENTERED=   3 TRANSTIME=      68.7 EMPTYFOR=      0.794 CURLEN=   0 TIMEPERIOD= 1000
RIGHT TRAFFIC:--
AVELEN=     0.0550 MAXLEN=   1 ENTERED=   1 TRANSTIME=      55.0 EMPTYFOR=      0.945 CURLEN=   0 TIMEPERIOD= 1000

EAST
LEFT  LANE QUEUE:--
AVELEN=       1.92 MAXLEN=   4 ENTERED=  17 TRANSTIME=      113. EMPTYFOR=      0.162 CURLEN=   2 TIMEPERIOD= 1000
RIGHT LANE QUEUE:--
AVELEN=       2.53 MAXLEN=   5 ENTERED=   8 TRANSTIME=      316. EMPTYFOR=      0.196 CURLEN=   4 TIMEPERIOD= 1000
LEFT  TRAFFIC:--
AVELEN=      0.824 MAXLEN=   3 ENTERED=   7 TRANSTIME=      118. EMPTYFOR=      0.520 CURLEN=   0 TIMEPERIOD= 1000
AHEAD TRAFFIC:--
AVELEN=       1.82 MAXLEN=   4 ENTERED=  11 TRANSTIME=      166. EMPTYFOR=      0.153 CURLEN=   2 TIMEPERIOD= 1000
RIGHT TRAFFIC:--
AVELEN=       2.85 MAXLEN=   5 ENTERED=   7 TRANSTIME=      407. EMPTYFOR=      0.216 CURLEN=   4 TIMEPERIOD= 1000

NORTH
QUEUE:--
AVELEN=      0.348 MAXLEN=   2 ENTERED=   3 TRANSTIME=      116. EMPTYFOR=      0.687 CURLEN=   0 TIMEPERIOD= 1000
LEFT  TRAFFIC:--
AVELEN=          0 MAXLEN=   0 ENTERED=   0 TRANSTIME=       *** EMPTYFOR=       1.00 CURLEN=   0 TIMEPERIOD= 1000
AHEAD TRAFFIC:--
AVELEN=      0.191 MAXLEN=   2 ENTERED=   2 TRANSTIME=      95.5 EMPTYFOR=      0.875 CURLEN=   0 TIMEPERIOD= 1000
RIGHT TRAFFIC:--
AVELEN=      0.290 MAXLEN=   1 ENTERED=   1 TRANSTIME=      290. EMPTYFOR=      0.710 CURLEN=   0 TIMEPERIOD= 1000

WEST
LEFT  LANE QUEUE:--
AVELEN=      0.770 MAXLEN=   4 ENTERED=  13 TRANSTIME=      59.2 EMPTYFOR=      0.592 CURLEN=   2 TIMEPERIOD= 1000
RIGHT LANE QUEUE:--
AVELEN=       1.13 MAXLEN=   3 ENTERED=   7 TRANSTIME=      162. EMPTYFOR=      0.359 CURLEN=   3 TIMEPERIOD= 1000
LEFT  TRAFFIC:--
AVELEN=      0.210 MAXLEN=   1 ENTERED=   3 TRANSTIME=      70.0 EMPTYFOR=      0.790 CURLEN=   1 TIMEPERIOD= 1000
AHEAD TRAFFIC:--
AVELEN=       1.68 MAXLEN=   6 ENTERED=  12 TRANSTIME=      140. EMPTYFOR=      0.477 CURLEN=   3 TIMEPERIOD= 1000
RIGHT TRAFFIC:--
AVELEN=       1.07 MAXLEN=   2 ENTERED=   5 TRANSTIME=      213. EMPTYFOR=      0.229 CURLEN=   1 TIMEPERIOD= 1000
```

```
SOUTH
QUEUE:--
AVELEN=      1.05 MAXLEN=   7 ENTERED=284 TRANSTIME=      147. EMPTYFOR=      0.491 CURLEN=   1 TIMEPERIOD=40000
LEFT  TRAFFIC:--
AVELEN=     0.248 MAXLEN=   3 ENTERED= 64 TRANSTIME=      155. EMPTYFOR=      0.792 CURLEN=   0 TIMEPERIOD=40000
AHEAD TRAFFIC:--
AVELEN=     0.745 MAXLEN=   5 ENTERED=144 TRANSTIME=      207. EMPTYFOR=      0.525 CURLEN=   1 TIMEPERIOD=40000
RIGHT TRAFFIC:--
AVELEN=     0.388 MAXLEN=   4 ENTERED= 76 TRANSTIME=      204. EMPTYFOR=      0.702 CURLEN=   0 TIMEPERIOD=40000

EAST
LEFT  LANE QUEUE:--
AVELEN=      1.24 MAXLEN=   8 ENTERED=498 TRANSTIME=      99.7 EMPTYFOR=      0.390 CURLEN=   0 TIMEPERIOD=40000
RIGHT LANE QUEUE:--
AVELEN=      1.21 MAXLEN=   8 ENTERED=279 TRANSTIME=      174. EMPTYFOR=      0.428 CURLEN=   0 TIMEPERIOD=40000
LEFT  TRAFFIC:--
AVELEN=     0.602 MAXLEN=   5 ENTERED=183 TRANSTIME=      132. EMPTYFOR=      0.583 CURLEN=   0 TIMEPERIOD=40000
AHEAD TRAFFIC:--
AVELEN=      1.48 MAXLEN=   8 ENTERED=391 TRANSTIME=      152. EMPTYFOR=      0.304 CURLEN=   0 TIMEPERIOD=40000
RIGHT TRAFFIC:--
AVELEN=      1.43 MAXLEN=   7 ENTERED=203 TRANSTIME=      283. EMPTYFOR=      0.306 CURLEN=   1 TIMEPERIOD=40000

NORTH
QUEUE:--
AVELEN=     0.985 MAXLEN=   8 ENTERED=267 TRANSTIME=      148. EMPTYFOR=      0.470 CURLEN=   1 TIMEPERIOD=40000
LEFT  TRAFFIC:--
AVELEN=     0.295 MAXLEN=   3 ENTERED= 66 TRANSTIME=      179. EMPTYFOR=      0.756 CURLEN=   0 TIMEPERIOD=40000
AHEAD TRAFFIC:--
AVELEN=     0.638 MAXLEN=   5 ENTERED=132 TRANSTIME=      193. EMPTYFOR=      0.578 CURLEN=   1 TIMEPERIOD=40000
RIGHT TRAFFIC:--
AVELEN=     0.365 MAXLEN=   3 ENTERED= 69 TRANSTIME=      212. EMPTYFOR=      0.673 CURLEN=   0 TIMEPERIOD=40000

WEST
LEFT  LANE QUEUE:--
AVELEN=      1.20 MAXLEN=   6 ENTERED=492 TRANSTIME=      97.5 EMPTYFOR=      0.389 CURLEN=   2 TIMEPERIOD=40000
RIGHT LANE QUEUE:--
AVELEN=      1.13 MAXLEN=   9 ENTERED=254 TRANSTIME=      177. EMPTYFOR=      0.479 CURLEN=   1 TIMEPERIOD=40000
LEFT  TRAFFIC:--
AVELEN=     0.671 MAXLEN=   5 ENTERED=200 TRANSTIME=      134. EMPTYFOR=      0.543 CURLEN=   1 TIMEPERIOD=40000
AHEAD TRAFFIC:--
AVELEN=      1.28 MAXLEN=   6 ENTERED=360 TRANSTIME=      143. EMPTYFOR=      0.300 CURLEN=   3 TIMEPERIOD=40000
RIGHT TRAFFIC:--
AVELEN=      1.36 MAXLEN=  10 ENTERED=186 TRANSTIME=      293. EMPTYFOR=      0.373 CURLEN=   2 TIMEPERIOD=40000

049.46 SECONDS IN EXECUTION
```

```
0000 1-      BEGIN
0001 --
0001 --      COMMENT *****SYSTEM PROCEDURES COPIED HERE*****;
0001 --
0316 --      RECORD PROGRAM(INTEGER TIME_TO_GO,PAGES,SLICE,PROGNO;LOGICAL USER);
0318 --
0318 --      COMMENT *****SYSTEM PROCEDURES REFERING TO "PROGRAM"RECORD COPIED*****;
0318 --
0328 2-      BEGIN REFERENCE(DELAYCHAIN)QUEUE,CPUQ;
0330 --            REFERENCE(GROUP)STORE;
0331 --            REFERENCE(EVENTNOTICE)CPU;
0332 --            INTEGER PAGES_LEFT,SYSTEM_USED,PREEM,USERNO,SYSTEMNO;
0333 --            INTEGER PROCEDURE RUSER_DELAY; CASE USERNO+1 OF (0,9,1,7,49,100);
0335 --            INTEGER PROCEDURE RSYSTEM_DELAY;
0336 --                CASE SYSTEMNO+1 OF (40,2,28,1,4,100);
0337 --            INTEGER PROCEDURE RUSER_TIME;CASE USERNO OF (21,15,8,11,8);
0339 --            INTEGER PROCEDURE RSYSTEM_TIME; CASE SYSTEMNO OF (4,3,4,4,1);
0341 --            INTEGER PROCEDURE RUSER_PAGES; CASE USERNO OF (5,3,4,6,10);
0343 --            INTEGER PROCEDURE RSYSTEM_PAGES; CASE SYSTEMNO OF (6,6,5,2,3);
0345 --            PROCEDURE OUT(INTEGER VALUE I);
0346 --            WRITE(I_W:=3,SIMTIME,IF USER(CURRENT)THEN "USER PROG" ELSE
0346 --                  "SYSTEM PROG",S_W:=0,PROGNO(CURRENT),"(",TIME_TO_GO(CURRENT)
0346 --                  ,",",PAGES(CURRENT),")  ",
0346 --                  CASE I OF("CREATED",
0346 --                            "ENTERS STORE",
0346 --                            "TAKES CPU",
0346 --                            "GIVES UP CPU",
0346 --                            "LEAVES STORE",
0346 --                            "PREEMPTED(STORE,CPU)",
0346 --                            "GOES BACK IN STORE",
0346 --                            "TAKES CPU BACK(RETURN)",
0346 --                            "PREEMPTED(STORE)",
0346 --                            "GOES BACK IN STORE(RETURN)",
0346 --                            "PREEMPTED(CPU)"));
0347 --            PROCEDURE ENTER_STORE;
0348 3-            BEGIN IF USER(CURRENT) THEN JOIN(STORE,"START")
0349 --                      ELSE SYSTEM_USED:=SYSTEM_USED+PAGES(CURRENT);
0350 --                  PAGES_LEFT:=PAGES_LEFT-PAGES(CURRENT);
0351 --                  OUT(2)
0351 -3           END;
0352 --            PROCEDURE LEAVE_STORE;
0353 3-            BEGIN IF USER(CURRENT) THEN LEAVE(STORE)
```

```
0354 --                     ELSE SYSTEM_USED:=SYSTEM_USED-PAGES(CURRENT);
0355 --                PAGES_LEFT:=PAGES_LEFT+PAGES(CURRENT);
0356 --                TESTALL(QUEUE);
0357 --                OUT(5)
0357 -3           END;
0358 --           PROCEDURE TAKE_CPU;
0359 3-           BEGIN CPU:=CURRENTEVNOTICE;
0361 --                 SLICE(CURRENT):=IF¬USER(CURRENT) OR TIME_TO_GO(CURRENT)<10
0361 --                                    THEN TIME_TO_GO(CURRENT) ELSE 10;
0362 --                 PRIORITY(CURRENTEVNOTICE):=52;
0363 --                 OUT(3)
0363 -3           END;
0364 --           PROCEDURE GIVEUP_CPU;
0365 3-           BEGIN CPU:=NULL;
0367 --                 TESTFIRST(CPUQ);
0368 --                 TIME_TO_GO(CURRENT):=TIME_TO_GO(CURRENT)-SLICE(CURRENT);
0369 --                 PRIORITY(CURRENTEVNOTICE):=0;
0370 4-                 IF TIME_TO_GO(CURRENT)>0 THEN BEGIN LEAVE(STORE);
0372 --                                                    JOIN(STORE,"START")
0372 -4                                               END;
0373 --                 OUT(4)
0373 -3           END;
0374 --           PROCEDURE SWAP_IN_STORE(REFERENCE(EVENTNOTICE)VALUE E,F);
0375 3-           BEGIN REMOVE(STORE,E);
0377 --                 INSERT(STORE,"END",F)
0377 -3           END;
0378 --           USERNO:=SYSTEMNO:=0;
0379 --           QUEUE:=NEWDELAYCHAIN;CPUQ:=NEWDELAYCHAIN;
0381 --           STORE:=NEWGROUP;
0382 --           PAGES_LEFT:=14;
0383 --           PREEM:=0;
0384 --           SYSTEM_USED:=2;
0385 --           CPU:=NULL;
0386 --           INITIALISE;
0387 --           SCHEDULE_EVENTS(RUSER_DELAY,100,0,GOTO USERPROG);
0388 --           SCHEDULE_EVENTS(RSYSTEM_DELAY,100,51,GOTO SYSTEMPROG);
0389 --           USERNO:=SYSTEMNO:=1;
0390 --           WAIT(100);
0391 --           STOP;
0392 --     USERPROG:TRANSACTION(PROGRAM(RUSER_TIME,RUSER_PAGES,,USERNO,TRUE));
0393 --              USERNO:=USERNO+1;
0394 --              OUT(1);
```

```
0395 --                WAITTILL(PAGES_LEFT>=PAGES(CURRENT),QUEUE);
0396 --                ENTER_STORE;
0397 --                WHILE TIME_TO_GO(CURRENT)>0 DO
0397 3-                BEGIN WAITTILL(CPU=NULL,CPUQ);
0399 --                      TAKE_CPU;
0400 --                      WAIT(SLICE(CURRENT));
0401 --                      GIVEUP_CPU;
0402 -3                END;
0403 --                LEAVE_STORE;
0404 --                TERMINATE;
0405 --      SYSTEMPROG:TRANSACTION(PROGRAM(RSYSTEM_TIME,RSYSTEM_PAGES,,SYSTEMNO,
0405 --                                                              FALSE));
0406 --                  SYSTEMNO:=SYSTEMNO+1;
0407 --                  OUT(1);
0408 --                  WAITTILL(SYSTEM_USED+PAGES(CURRENT)<=16,QUEUE);
0409 --                  SET(STORE);
0410 --                  WHILE PAGES_LEFT<PAGES(CURRENT) DO
0410 --                   CALL_IMMEDIATE(SEARCH(STORE,"UNLINK",TRUE),GOTO
0410 --                                                       STORE_PREEMPT);
0411 --                  ENTER_STORE;
0412 --                  WAITTILL(CPU=NULL OR USER(TRANS(CPU)),CPUQ);
0413 --                  IF CPU¬=NULL THEN CALL_IMMEDIATE(CPU,GOTO CPU_PREEMPT);
0414 --                  TAKE_CPU;
0415 --                  WAIT(SLICE(CURRENT));
0416 --                  GIVEUP_CPU;
0417 --                  LEAVE_STORE;
0418 --                  TERMINATE;
0419 --      STORE_PREEMPT:PAGES_LEFT:=PAGES_LEFT+PAGES(CURRENT);
0420 --                    IF CPU=SUPERCEDING(CURRENTEVNOTICE) THEN
0420 3-                   BEGIN CPU:=NULL;
0422 --                         OUT(6);
0423 --                         PRIORITY(CURRENTEVNOTICE):=50;
0424 --                         WAITTILL(PAGES_LEFT>=PAGES(CURRENT),QUEUE);
0425 --                         OUT(7);
0426 --                         PAGES_LEFT:=PAGES_LEFT-PAGES(CURRENT);
0427 --                         INSERT(STORE,"END",CURRENTEVNOTICE);
0428 --                         WAITTILL(CPU=NULL,CPUQ);
0429 --                         OUT(8);
0430 --                         CPU:=SUPERCEDING(CURRENTEVNOTICE);
0431 --                         SWAP_IN_STORE(CURRENTEVNOTICE,SUPERCEDING
0431 --                                                  (CURRENTEVNOTICE));
0432 --                         TERMINATE
```

```
0432 -3                         END
0432 3-                         ELSE BEGIN OUT(9);
0434 --                                    PREEM:=PREEM+1;
0435 --                                    IF SUPERCEDING(SUPERCEDING(CURRENTEVNCTICE))
0435 --                                      =NULL THEN PRIORITY(CURRENTEVNOTICE):=PREEM;
0436 --                                    WAITTILL(PAGES_LEFT>=PAGES(CURRENT),QUEUE);
0437 --                                    INSERT(STORE,"START",SUPERCEDING
0437 --                                                         (CURRENTEVNCTICE));
0438 --                                    PAGES_LEFT:=PAGES_LEFT-PAGES(CURRENT);
0439 --                                    PREEM:=PREEM-1;
0440 --                                    OUT(10);
0441 --                                    TERMINATE;
0442 -3                               END;
0443 --       CPU_PREEMPT:CPU:=NULL;
0444 --                   OUT(11);
0445 --                   PRIORITY(CURRENTEVNOTICE):=50;
0446 --                   SWAP_IN_STORE(SUPERCEDING(CURRENTEVNOTICE),CURRENTEVNCTICE);
0447 --                   WAITTILL(CPU=NULL,CPUQ);
0448 --                   OUT(8);
0449 --                   CPU:=SUPERCEDING(CURRENTEVNCTICE);
0450 --                   SWAP_IN_STORE(CURRENTEVNOTICE,SUPERCEDING(CURRENTEVNCTICE));
0451 --                   TERMINATE
0451 -2       END
0451 -1       END.

EXECUTION OPTIONS: DEBUG,0 TIME=97369 PAGES=32767


 .NOTE 2031 NEAR COORDINATE 0222 - WARNING: NAME PARAMETER SPECIFIED
   (FOUND NEAR " C ) ")

012.36 SECONDS IN COMPILATION, (19360, 00C00) BYTES OF CODE GENERATED
```

```
 0  USER PROG     1( 21,  5)  CREATED
 0  USER PROG     1( 21,  5)  ENTERS STORE
 0  USER PROG     1( 21,  5)  TAKES CPU
 9  USER PROG     2( 15,  3)  CREATED
 9  USER PROG     2( 15,  3)  ENTERS STORE
10  USER PROG     1( 11,  5)  GIVES UP CPU
10  USER PROG     2( 15,  3)  TAKES CPU
10  USER PROG     3(  8,  4)  CREATED
10  USER PROG     3(  8,  4)  ENTERS STORE
17  USER PROG     4( 11,  6)  CREATED
20  USER PROG     2(  5,  3)  GIVES UP CPU
20  USER PROG     1( 11,  5)  TAKES CPU
30  USER PROG     1(  1,  5)  GIVES UP CPU
30  USER PROG     3(  8,  4)  TAKES CPU
38  USER PROG     3(  0,  4)  GIVES UP CPU
38  USER PROG     3(  0,  4)  LEAVES STORE
38  USER PROG     4( 11,  6)  ENTERS STORE
38  USER PROG     2(  5,  3)  TAKES CPU
40  SYSTEM PROG   1(  4,  6)  CREATED
40  USER PROG     4( 11,  6)  PREEMPTED(STORE)
40  SYSTEM PROG   1(  4,  6)  ENTERS STORE
40  USER PROG     2(  5,  3)  PREEMPTED(CPU)
40  SYSTEM PROG   1(  4,  6)  TAKES CPU
42  SYSTEM PROG   2(  3,  6)  CREATED
42  USER PROG     1(  1,  5)  PREEMPTED(STORE)
42  USER PROG     2(  5,  3)  PREEMPTED(STORE)
42  SYSTEM PROG   2(  3,  6)  ENTERS STORE
44  SYSTEM PROG   1(  0,  6)  GIVES UP CPU
44  SYSTEM PROG   1(  0,  6)  LEAVES STORE
44  SYSTEM PROG   2(  3,  6)  TAKES CPU
44  USER PROG     2(  5,  3)  GOES BACK IN STORE(RETURN)
44  USER PROG     1(  1,  5)  GOES BACK IN STORE(RETURN)
47  SYSTEM PROG   2(  0,  6)  GIVES UP CPU
47  SYSTEM PROG   2(  0,  6)  LEAVES STORE
47  USER PROG     2(  5,  3)  TAKES CPU BACK(RETURN)
47  USER PROG     4( 11,  6)  GOES BACK IN STORE(RETURN)
50  USER PROG     2(  0,  3)  GIVES UP CPU
50  USER PROG     2(  0,  3)  LEAVES STORE
50  USER PROG     1(  1,  5)  TAKES CPU
51  USER PROG     1(  0,  5)  GIVES UP CPU
51  USER PROG     1(  0,  5)  LEAVES STORE
51  USER PROG     4( 11,  6)  TAKES CPU
61  USER PROG     4(  1,  6)  GIVES UP CPU
61  USER PROG     4(  1,  6)  TAKES CPU
```

```
 62  USER PROG     4(  0,  6)  GIVES UP CPU
 62  USER PROG     4(  0,  6)  LEAVES STORE
 66  USER PROG     5(  8, 10)  CREATED
 66  USER PROG     5(  8, 10)  ENTERS STORE
 66  USER PROG     5(  8, 10)  TAKES CPU
 70  SYSTEM PROG   3(  4,  5)  CREATED
 70  USER PROG     5(  8, 10)  PREEMPTED(STORE,CPU)
 70  SYSTEM PROG   3(  4,  5)  ENTERS STORE
 70  SYSTEM PROG   3(  4,  5)  TAKES CPU
 71  SYSTEM PROG   4(  4,  2)  CREATED
 71  SYSTEM PROG   4(  4,  2)  ENTERS STORE
 74  SYSTEM PROG   3(  0,  5)  GIVES UP CPU
 74  SYSTEM PROG   3(  0,  5)  LEAVES STORE
 74  SYSTEM PROG   4(  4,  2)  TAKES CPU
 74  USER PROG     5(  8, 10)  GOES BACK IN STORE
 75  SYSTEM PROG   5(  1,  3)  CREATED
 75  USER PROG     5(  8, 10)  PREEMPTED(STORE)
 75  SYSTEM PROG   5(  1,  3)  ENTERS STORE
 78  SYSTEM PROG   4(  0,  2)  GIVES UP CPU
 78  SYSTEM PROG   4(  0,  2)  LEAVES STORE
 78  SYSTEM PROG   5(  1,  3)  TAKES CPU
 78  USER PROG     5(  8, 10)  GOES BACK IN STORE(RETURN)
 79  SYSTEM PROG   5(  0,  3)  GIVES UP CPU
 79  SYSTEM PROG   5(  0,  3)  LEAVES STORE
 79  USER PROG     5(  8, 10)  TAKES CPU BACK(RETURN)
 83  USER PROG     5(  0, 10)  GIVES UP CPU
 83  USER PROG     5(  0, 10)  LEAVES STORE

000.94 SECONDS IN EXECUTION
```

APPENDIX F

DETAILS OF CODE PRODUCED FOR CERTAIN CONSTRUCTS IN ALGOL W

Call of Block or Procedure without parameters

The following instructions are generated for the call of a program segment representing a nested block or procedure without parameters:-

(1) r14 := base of called program segment

(2) Call program segment (leaving return address in r1)

Expects on return : r2 = base of own data segment

(3) r14 := address of own program segment (restored from own data segment).

(4) If this is not the outermost block, restore display registers from own data segment (or if there is only one to restore, from r2)

Call of Procedure with parameters

The following instructions are generated for the call of a program segment representing a procedure with parameters:-

(1) Unconditional branch to (3)

(2) thunks for evaluation of actual parameters to be passed, other than constants and simple variables.

(3) r2 := base of own data segment

(4) r5 := F.P. (will be new data segment's base)

(5) For each actual parameter to be passed, store a pair of words, starting at the 11th and 12th from the top of the stack (r5), as follows:-

If variable or constant passed, 1st word := address of variable or constant
2nd word not used

otherwise, 1st word := address of thunk for parameter
2nd word := address of own data segment
Set 1st bit of 1st word to 1

(These representations of the parameters passed will be in the called data segment when it is created).

(6) -(9) Code as for"block call", (1) - (4)

Entry to Block or Procedure without Parameters

At entry to all program segments corresponding to blocks and procedures without parameters, including the outermost block (i.e. whole program is considered as a block inside a system program of sorts) the following code is generated:-

Expects : r14 = base of called program segment r1 = return address of segment

(1) r0 := no. of bytes required for data segment to be created (other than space required for arrays).

(2) Call system procedure to check that space is available for data segment

Expects on return : r0 = forward pointer r2 = dynamic link r5 = base of segment allocated

(3) r3 := 0 (initially no reference variables or reference arrays)

(4) Store r14, r15, r0, r1, r2, r3 as System Information in allocated data segment P.S.:= r14, F.P.:= r0, R.A. := r1, D.L. := r2, N.R.V./N.R.A. := r3)

(5) Gives all variables declared in data segment "undefined" value.

(6) T.D.S. := base address of allocated data segment (r5)

(7) Load r5 into appropriate display register (allocated statically

down from r12)

(8) Code to allocate any arrays and set up their array descriptions, updating F.P. to point past these arrays.

(9) Store information about any reference variables and arrays for garbage collector, in N.R.A. and N.R.V.

(10) If not outermost block, store display in data segment.

Entry to Procedure with Parameters

At entry to all program segments corresponding to procedures, the following code is generated:-

Expects : r14 = base of called program segment
r1 = return address of segment
r2 = base of calling data segment (i.e. dynamic link of data segment to be created)
r5 = base of data segment to be created

(1) r0 := no. of bytes required for data segment (apart from space required for arrays)

(2) Calls system procedure to check that space is available.

Expects on return : r0 = forward pointer

(3) - (10) As in (3) - (10) of block entry

(11) For each value parameter, "Evaluate Parameter"
Store value in data segment.

Block or Procedure Exit

On exit from all program segments, the following code is generated:-

(1) r1 := R.A. and r2 := D.L. (loaded from own data segment)

(2) T.D.S. := r2 (i.e. destroy top data segment)

(3) Unconditional branch to R.A. (i.e. r1)

Gives : r2 = base of data segment which called this segment

If a value is to be returned by a <type> procedure with parameters it is left before the exit code as follows:-

(i) If the value returned is integer, then it is passed in r3

(ii) If the value returned is logical, then its address is passed in r3.

The methods of returning other types of value are not relevant to this thesis.

If a value is to be returned by a block expression or <type> p rocedure without parameters, its address is always passed in r3, irrespective of type.

THUNK

A thunk is part of the program segment that passed it as a parameter. However its code is similar to that generated for entry to and exit from program segments. When the thunk is evaluated, it is evaluated in the context of the point of call of the procedure which the thunk was passed to; so the display from there must be restored before the thunk is executed properly.

Assuming the data segment D.S., executing program segment P.S. has passed the thunk as a parameter (the thunk will actually be part of P.S.) the following code is generated for the thunk:-

Expects : r1 = return address of thunk r4 = address of D.S.

(1) r14 := address of P.S., restored from D.S.

(2) restore display from D.S. (i.e. display at time thunk was passed as parameter)

(3) - (11) as in block entry, (1) - (9)

(12) Code to evaluate actual parameter (and place the result in thunk's data segment, if there is one).

(13) r3 := address of value found (if thunk is not of type procedure)

(14)- (16) As in block or procedure exit, (1) - (3)

Gives : r2 = base of data segment which called thunk to evaluate it r3 = address of parameter value (if parameter is not of type procedure

Note that a procedure parameter is passed and evaluated in the same way as an integer or logical parameter, except that the thunk does not return a value.

Evaluate parameter

When a procedure wants to evaluate an actual parameter passed to it, it does so using the two words passed as a representation of the parameter, to its data segment on procedure call. For a value parameter, the actual parameter is only evaluated once, on entry to the procedure, but otherwise the actual parameter is evaluated each time it is used. The code generated each time an actual parameter is evaluated is as follows:-

(1) load r3 and r4 with pair of words passed to represent actual parameter

(2) if r3<0 (i.e. thunk is to be evaluated)

then (2A) Call thunk (return address in r1)

Expects on return : r2 = base of own data segment r3 = address of value of parameter

(2B) r14 := address of own program segment (restored from own data segment)

(2C) restore display from own data segment

Gives : r3 = address of value

If the parameter is of type procedure, the test, "r3<0" is not made as the representation will always be by a thunk. Also, of course, there is no value returned.

REFERENCES

1. "General Purpose Simulation System /360, User's Manual", GH20 - 0326 - 4.

2. Markovitz, H., et al., "SIMSCRIPT", Prentice-Hall.

3. Dahl, O., Myhrhang, B. & Nygaard, K., "SIMULA 67 Common Base Language", Norwegian Computing Centre.

4. Birtwhistle, G.M., "Simula Begin", Student Litteratur, Sweden.

5. Buxton, J.N. & Laski, J.G., "Control and Simulation Language", Computer Journal 5.

6. Van Wijngaarden, A., et al., "Report on the Algorithmic Language Algol 68", Numerische Mathematik 14.

7. Wirth, N. & Hoare, C.A.R., "A Contribution to the Development of ALGOL", CACM 9.

8. "ALGOL W Reference Manual", CL/72/8, St. Andrews Computing Laboratory.

9. Bauer, H., Becker, S. & Graham, S., "ALGOL W Implementation", Technical Report No CS 98, Standford University.

10. "IBM System /360 Principles of Operation", GA22-6821-8.

11. Wirth, N., "PL360, A Programming Language for 360 Computers", JACM 15.

12. "PL360 Reference Manual", CL/72/6, St. Andrews Computing Laboratory.

13. Whittlesey, J.R.B., "A Comparison of the Correlational Behaviour of Random Number Generators for the IBM 360", CACM 11.