SIMULA - A Way of Thinking

Berth Eklundh

Lund Institute of Technology, Lund, Sweden

ABSTRACT

SIMULA as a tool for modelling and programming is reviewed. Some basic questions that arise in a simulation context are dealt with, and the evolution steps from a percepted reality to final simulation program are examined and exemplified. The modelling step is considered the most important and more effort is put in the analysis of this than in a semantic and syntactical definition. A specific example is given, and SIMULA is used to simulate that example. The importance of conceptually wellsuited building-blocks for simulationtasks is discussed, and the concepts of SIMULA are put in relation to program evaluation.

INTRODUCTION

The complexity of the world around us is growing at such a rate that even researchers and engineers, who deal with rechnical and economical problems in their everyday life, get stunned and find it harder and harder to grasp the essence of their work. Due to this, we try to find new ways to analyze and describe various fenomena in our environment. In some disciplines, simulation has come to be used in a growingly number of cases, but unfortunately the programming is done with the aid of programming languages, like FORTRAN and related languages, that are not designed for simulation tasks. What is needed for the analysis of our complex systems, is not just another programming language, of which some allow more elaborate constructions than others, but something that will help our bewildered brain to sort things out, i.e. something that is **conceptually** new and helpful. The title is an indication of the belief that SIMULA to a large extent is this "something", and that it can give us a new way of thinking. Through some advices and an example, which we penetrate quite thoroughly, we will try to convey some of this way of thinking, and we do this rather than give a tiring exposé of syntax and sematic in SIMULA. The way to tackle a simulation task that will be suggested, leads to a solution that might be called "structured simulation" in analogy with structured programming. We will also try to give some rational arguments for the SIMULA approach, of which the naturalness hopefully will be the major. But before we deal with SIMULA more in detail, let us look somewhat at what simulation is and why it is used.

As you can read in any book on system simulation, and presumably in these proceedings, simulation is "the process of designing a model of a real system and conducting experiments with this model for the purpose either of understanding the behavior of the system or of evaluating various strategies for the operation of the system" /3/. There are many types of simulation techniques, but the most common, which we will concern ourselves with here, is the discrete event simulation, in which the events may or may not be governed by random decisions. What makes the definition above differ from a definition of, for example, an analytic model, is the phrase "conducting experiments". If the simulation has random decisions, this phrase will indicate that simulation is an experimental investigation method and that the results thus are samples from the model which have to be analyzed by statistical methods. And if the results are only statistically correct we know that we, by no means can rely on them unconditionally but only with a certain likelihood can say that they tell the truth about the **model**, and the model only. Further, we have the constraint that our simulation program in a proper way has to map our model into some code understandable to a computer. In addition, the statistical data have to be properly collected and processed. If all these demands are met, we can state that our results, with a certain probability, give a fair view of the behaviour of our model. But then, if our model is not a good picture of the real system we want to investigate, all our results are useless and we cannot say anything about the behaviour of the real system considered.

We thus observe that there is a long chain of undertakings that have to be performed before we can get any statistically significant results for the real system. And we have to know that each link in the chain is correct before we can make any statements on the system. To be sure that each link is correct is, of course, impossible, and it thus becomes most important to choose a method in each step that minimizes the probability for that step to be erroneous. This is where SIMULA and the "way of thinking" comes in.

The problems mentioned above are of less importance, or does at least not trouble us to any larger extent, if the system we want to take a closer look at is small, or has a very simple behaviour. But in such a case we are seldom forced to use simulation to get information about the system, and we can thus suppose that the system in view is large, or has a complex way of working.

There are, in addition to the simulation approach, quite a few ways to analyze a system, e.g. analytical methods. Generally, these should be investigated first, and only if it is, with the

CH1437-3/70/0011-0020$00.75

1979 Winter Simulation Conference

knowledge and equipment at hand considered, impossible to apply some of these methods, simulation should be used. This may seem rigid a standpoint, but it is the auther's experience that simulation, due to its ostensible simpleness, is used much too often, and that many times too much faith is put in the results.

In spite of this, very few large and complex systems lend themselves to be analyzed by for example analytical methods, and thus simulation has to be used. It is then most important to have a working scheme to follow when performing the steps in the chain discussed above. This paper should be a tutorial mainly on SIMULA, but it is the auther's opinion that it is meaningless to produce a language tutorial that deals mostly with the details of the language at hand, and not with how to use it. By this reason we will embarke on the problem that from some unstructured reality build a model, find the essence of that model, write a program and point out how to measure the quantities desired. This means that any exact semantic or syntactical definition of SIMULA will not be given, but can of course be found in any book on SIMULA /1/. Instead we will try to, by a specific example and some general hints, show that the approach to system simulation that SIMULA provides, gives a final program that is structured, easy to read, correct, enlarge and that, maybe most important, the approach is quite natural and adapted to our usual way of thinking. It is, with the limited space and time available in proceedings and during conference, impossible to give a full account for SIMULA and its features. It is therefore assumed that the reader is familiar with ALGOL or at least its underlying principles, as the block structure.

SIMULATION – THE MODELLING STEP

As stated in the introduction, building a simulation program consists of many different steps: define the considered reality, build a model of that reality, write a program that mimics the model and perform all the usual, but sometimes quite labours, steps that are involved in programming. Out of these steps the modelling step is the most crucial. In this step the structure of the final program is based, and very little can at a later occation be done to change this structure. In spite of the importance of this step it is hard to give any explicit guidance or rules how to go through it. Most of the working scheme has to be conveyed through examples, but we will try to give some general hints. The fact that most techniques used in modelling is gained through experience is perhaps one of the reasons why so many books and tutorial articles on the subject tend to be a bit "talky". We will try to avoid that trap and continue mostly by examples, but before that a few words about the nature of simulation and modelling.

The simulation technique can be divided into many different types: discrete, continous, processoriented, eventoriented; just to mention a few. Among these, SIMULA is most fit for the discrete. It can be used for other simulation-techniques as well, but since it is best adapted to the above mentioned, we will consider that approach here.

Most systems can be viewed as consisting of, physically or imaginary, welldefined parts that interact according to some rules. Further, in many systems we have some kind of flow of entities through the system, such as telephone calls through a local exchange or people shopping in a supermarket; where we can see the customers emerge from one set of doors and disappear through another. In a more abstract manner we can view money or thoughts as entities that circulate in the system. Whatever view we take of the system, time is a very relevant parameter, and things happen at district times: a new customer enters or leaves the shop; someone connected to the telephone exchange lifts the reciever and wants to make a call. This means that the history of the system can be described by a series of states where each state has a duration which ends with the occurance of a new event as exemplified above. When an event, such as the arrival of a new customer to a shop, occurs, this may effect the state of the system more than only through the fact that there is one more customer in the system. Perhaps the customer is a very prominent one and will be given immediate service. This is not likely if the service facility is an ordinary shop and the customer is a "real" customer. But if we view the concept customer in a more abstract manner and say that for example an alarm call to a telephone exchange is a customer, this could certainly be the case. Thus we find that we need some way to establish communication between different parts of the system; some way to do the timekeeping and some instrument to describe the parts that constitute the system. We will later find that SIMULA meet these requirements in a very elegant manner and that we can use the concepts provided by SIMULA in all kinds of simulation tasks, even if the final program is not to be written in SIMULA.

Now, if we want to break down a large system into parts which we can handle, and with these parts as building blocks build a model of the system, there are a few things we should bear in mind:

a) Try to get a clear view of the interactions and parts of the system. Try to avoid setting up any virtual borders or parts, which perhaps in some cases could simplify the programming step but which probably only bring confusion to the model, and which will make the interaction indistinct.

b) The environment of the system has to be modelled in some way. This could mean that we are forced to build an abstract model of the environment which has no direct equivalent in reality, which in our shop example would mean that we have to model the process which creates customers and let them in through the input gate.

c) If there are parts of a simulation which are in a way imaginary, as for example the calls that arrive to a telephone exchange, these should be modelled as something that do exist, but which is unable to act. The reason for this is that when we at some later occasion perhaps want to enlarge or make our model more precise, we are forced to get our inspiration from the real system we are modelling, and if we have put any decisions in the imaginary parts of the system it might be hard to incorporate this in our enlarged or more skillfully working model since these actions do not have any counterpart in reality and thus are hard to connect with the actual system.

d) Avoid, as far as possible, to think in or write programs at this stage. It is a very common mistake to pay to little attention to the modelling and embarke directly on the writing of programs. If this mistake is made it limits your freedom and can make you overlook solutions which later show to be much better than the first appearent.

This list could be made very long but we stop here and take some more details when we deal with a specific example in a subsequent paragraph.

Let us finally regard the problem of how to describe the model. We have to have some way to describe the intermediate step between our perception of the concrete system and the semantically and syntactically correct program. We will later use the term "a quasi formal description" for this intermediate step, since it is a hybrid between ordinary prose and a programming language. This hybrid is very useful since it does not demand any special programming language and therefore can be used in all

kinds of modelling. Its appearance is easiest showed through an example. Suppose we want to describe a counter in a super-market with respect to its way of working. A quasi formal description could be like this:

```
Counter;

Begin

   while time of day < closinghour do

   Begin
      if customers to serve
      then
      give customers service
      else
      wait for customers to arrive;
   end;
end;
```

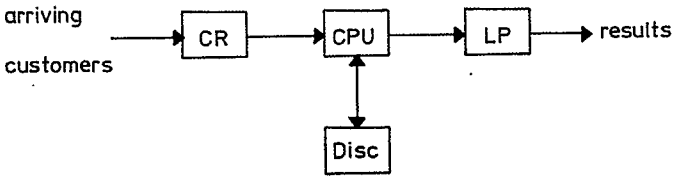
In this simple example the description may seem a bit over-worked, but when the system becomes more complex, it gives a very neat account for the system behaviour and will help to sort things out. We will use this technique when we now embarke on a small simulation task.

AN EXAMPLE

To choose a physical system, which could serve as an example in a tutorial on simulation and SIMULA, is, due to the heterogeneity of the readers, quite hard. It should be a system which is to some extent known to all, and it should at the same time be illustrative and small enough to be handleable in limited time and space. Since every simulation program has to be run on a computer, everyone that writes such a program will get in contact with a computer. Therefore we take an example which is a somewhat simplified computer system.

I The presuppositions

We suppose a computer system with the following structure:



CR: Card-Reader
LP: Line-Printer
CPU: Central-Processing-Unit
Disc: Secondary memory

and we assume the following way of working:

Customers arrive to the system with their Jobs and want to have them processed by the computer. A Job is specified on a number of cards which is read by the card-reader. The card-reader reads one Job at a time and then puts it in an input queue in the CPU. Each Job is processed by the CPU in a first-come-first-served manner, i.e. the CPU operates as a batch system. During its processing each Job has to read or write on a disc a numer of times. An access to the disc means that the processing in the CPU is interrupted and that the Job is handed over to the disc. When this occurs the CPU takes the next Job in the input queue, if any, and starts to process that Job. When a Job has finished its reading or writing on the disc it joins the tail of the CPU's input queue. After a while, when the Job has encounted the required amount of processing, and has made its visits to the disc, it is put out to a line-printer where the results are printed.

II The model

We are now in position to get our earlier statements exemplified. We see that we in some way have to model the environment which causes customers to arrive to the computer system to have their Jobs run. We can also observe that we have an abstract or imaginary part of the system: The concept Job. Further there are some concrete parts like Card-reader, Line-printer, CPU and Disc that have to be modelled.

If we follow a Job on its way through the system we notice that the time that the Job spends in the system will be divided into two parts: (1) time spent in wait for access to some of the requested service facilities and (2) time elapsed when actually in service at some of the service facilities. Since wait probably will occur, at least for some Jobs, at all parts of the system it seems reasonable to associate some waiting room with each of the physical parts of the system. We implement this waitingroom as a queue of infinite size, and assume in our model that when a Job encount some wait, i.e. the service facility is occupied by some other Job, it is put to wait at the end of the queue associated with that service facility.

This arrangement takes care of part (1). The other part, (2), is specific to the Job at service and it seems reasonable to give the Job some processingtime in the service facilities. Since the processing times in CR, LP, CPU and Disc not necessarily are related we should give separate values to the individual processing times. In the final implementation we are probably forced to use some stochastic distribution to assign these values, but at the present stage we do not have to concern with that. How long it takes to read a card, print a page and so on is only input data to the model and we could, as a first approximation, use some constant time for this activity. How well we model this activity is merely a question of accuracy and does not to any larger extent influence the way we separate the parts of the system. There will be some more clear examples of this later in the text. There are of course more details to be dealt with, but we stop here in order not to confuse the picture of the model. Let us instead put down our thoughts in the hybrid language we spoke of before.

III A quasi-formal description

We start with the imaginary parts: the Job and the environment, the latter modelled as a Job generator.

```
Job generator;

Begin
   while simulated time < intended duration of
   simulaton do

   Begin
      generate a new Job and put it in
      cardreaders queue;
      wait some interarrivaltime;
   end;
end;

Job;
Begin

Assign to the Job following quantities

a)  Number of cards to be read;
b)  Processing time when at CPU;
c)  Number of disctransfers;
d)  Processingtime when at Disc;
e)  Number of pages to be printed;

end;
```

and describe the concrete parts in the same manner:

```
Cardreader;
Begin

   while true do

   Begin
      if Job in queue
      then
      read cards and enter Job into CPU's queue
      else
      wait for Job to arrive;
   end;
end;

CPU;

Begin

   while true do

   Begin
      if Job in queue
      then

      Begin
         process Job until finished or
         discaccess required;
         if Job finished
         then
         hand over to lineprinter
         else
         hand over to disc;
      end
      else
      wait until Job arrives;
   end;
end;

Disc;

Begin

   while true do

   Begin
      if Job in queue
      then

      Begin
         look for required information;
         put Job back to CPU;
      end
      else
      wait until Job arrives;
   end;
end;

Lineprinter;

Begin
   while true do

   Begin
      if Job in queue
      then
      print result
      else
      wait until Job arrives;
   end;
end;
```

From this informal description it is possible to conclude that if we want to transfer it to some programming language we must demand certain properties of that language. Let us give a suggestive but not at all exhaustive list:

a) The language should contain routines for list handling, so that we can create queues and bring items out of and in to the list.

b) There should be some possibility to write a welldefined programsegment which could picture informal discriptions like LP, CR and so forth.

c) In some way we have to be able to establish communication between different parts of the program in an easy and safe manner.

d) The program system should be helpful in all scheduling situations, where we have to schedule some future event or put some part of the simulation away for an indefinite period of time.

Now, let us see what SIMULA can make for us on these matters.

SIMULA — THE TOOL

SIMULA is a general purpose language based on ALGOL, and contains elements that makes it well suited for simulation applications. This section gives a brief account for the most important features of the language, but a lot of details are omitted in order to give the essence rather than a full account. These omitted details are easily learned by the writing of a few exercise programs that can be found in some book on SIMULA, like /1/.

There are two major language elements in SIMULA that makes it differ from ALGOL:

The Class concept.
The Reference variables.

With these, and the ALGOL part of the language, two system-defined programming aids are built:

Class Simset

which contains routines for manipulation of doubly linked lists, sets in SIMULA terminology, and

Class Simulation

which contains a lot of routines, concepts and procedures which are at help when an event-simulation is to be programmed. Let us take a closer look at these four concepts in turn.

Class

The SIMULA Class is similar to the ALGOL Procedure, i.e. it has a declaration and optionally a class body with some formal parameters and statements; in analogy with a procedure body. A typical class declaration could be

```
Class A (B,C);
Integer B,C;
Begin
   Locally declared variables and Procedures;
   Class body statements;

end;
```

Formal parameters as well as variables and procedures declared in the outmost block of the class body are called attributes of the class.

A class differs from a procedure in two major ways. A call to a procedure causes the statements in the procedure body to be executed to the end and the procedure gives back some information through its parameters or its name, and maybe through some global variables. A class is not called to in the same way, but through an operator, new, which is a keyword in SIMULA, which produces a copy of the class, and the copy is called an object of that class. A statement which creates a copy of our class A could be

new A (1,2);

with the actual parameters 1 and 2.

Now, when this statement is executed the statements of the class body, if any, are executed. So far the class behaves as a procedure, but the difference is that after the execution the copy, the object, remains in the memory and the attributes of that object are accessible through a special mechanism, called "dot notation", which will be explained below, when we have dealt with the reference variables. There will be more about classes later, but now we need

Reference variables

When we have made copies of our classes, i.e. we have made objects, we need some mechanism to reach and handle these objects. For this purpose there are the reference variables. A reference variable is of a type, like INTEGER, REAL et cetera, and has the formal declaration

Ref (Classname) Refname;

where "Classname" is the name of some in the program declared class, and "Refname" is the name of the reference variable. "Classname" is called the qualification of the variable and means that variable is allowed to make reference to objects of the type "Classname" but no other (a restriction that can be somewhat loosened). Reference variables are assigned values in the same way as other types of variables, but we use a different symbol for the assignment, :–. This means that if we assume that we already have made the previous declaration of class A, and that we declare a reference variable that is allowed to make reference to, or point at, as we will say from now on, objects of type A:

Ref (A) Pointer;

we will be allowed to write

Pointer:– new A (1,2);

which means that Pointer is pointing at a new object of type A.

The question is now what use we have of references to objects. This is answered by the construction we hinted at before: the dot notation. Through this mechanism, the attributes of an object are accessible in the following way:

if we for example like to access the attribute B of class A we can do this as

Pointer.B

and we can change the value of that attribute

Pointer.B:=0;

or transfer it to some other variable

D:=Pointer.B;

This is called dot notation and is one way to access attributes of an object, and the only considered here.

Prefixed Classes

In the same way as it is possible to build a hierarchical structure with blocks, where variables declared at, and outer to, the present level, are accessible, it is possible to build a hierarchical structure of classes. This is accomplished by prefixing. An exampel will make the point

Suppose we have declared a class

```
Class A (B,C)
Integer B,C;
Begin
.....
end;
```

then we can declare another class

```
A Class D;
Begin
.....
end;
```

We then say that we have the prefix A to class D. Now, if we make an object of class D we have to give values to the actual parameters of class A as well, i.e.

pointer:–new D (1,2)

and this object will contain attributes B and C, i.e. we can write

pointer.B

The advantages of these concepts have to be seen in examples. We will however postpone the use of examples until later when we have dealt with the systemdefined classes SIMSET and SIMULATION, which are built with the concepts just mentioned.

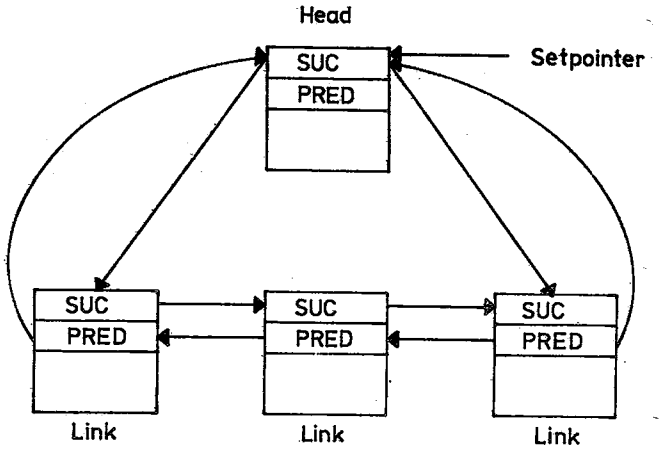
SIMSET

In our previous example we discovered that it would be convenient to have a queue associated with the service facilities. A queue could be implemented by a linked list and in SIMULA a class has been built which contains routines for the handling of doubly linked lists, called sets. This class is called SIMSET and we give a brief description of its way of working.

Each set contains at least one member and the first one is always an object from a systemdefined class named HEAD. HEAD is a subclass to SIMSET. All other members in the set are objects from another subclass to SIMSET, LINK. This means that each class that one wants to have in a set have to be prefixed by LINK, and that an empty set contains only the HEAD. Both HEAD and LINK have two reference variables, named SUC and PRED, which are pointing at the SUCcessor and PREDecessor of the member respectively. The set is referenced through a pointer with the qualification HEAD, i.e. it is declared with

Ref (HEAD) Setpointer;

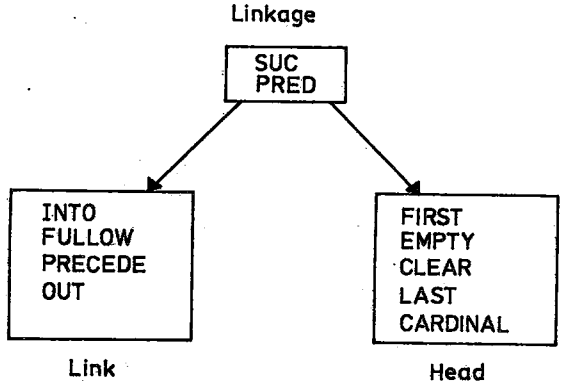
and the following figure is valid.



The reference variables SUC and PRED are collected in a, to LINK and HEAD, common prefixclass named LINKAGE. We get, for class SIMSET, the following skeleton

```
Class SIMSET;
Begin
   Class LINKAGE ..............;
   LINKAGE Class LINK  ..........;
   LINKAGE Class HEAD  ..........;
end*** SIMSET***;
```

Inside LINK and HEAD some procedures, which are useful for manipulation of the set, are declared. The relationship between these three classes is



These procedures are attributes of the class in which they are declared and can be reached by dot notation. Their use is quite straight-forward and we take a closer look at just some to show the underlying principle. Assume the deklarations

```
REF (HEAD) H;
REF (LINK) L1;
```

Then the statement L1.OUT removes L1 from its present set if any.

The statement L1.INTO (H) calls L1.OUT first and then inserts L1 as the last member of the set referenced by H.

The procedures FIRST and LAST in HEAD are REF (LINK) procedures and returns a reference to the first and last members of a set respectively, if any, i.e. a call

H.FIRST;

returns a reference to the first member in the set referenced by H.

EMPTY is a boolean procedure which returns true if the set is empty, i.e. if it has no members.

It is, in the same way as it is allowable to prefix classes, allowable to prefix blocks, and inside a class or block prefixed by SIMSET one is allowed to use LINKAGE, LINK and HEAD just as they are, or to prefix classes written by oneself.

A small example will straight things out. Suppose we want to create 10 objects of some kind and put them in a set. A way to do this is:

```
Simset
Begin
   Ref (HEAD) Set;
   Integer I;
   LINK Class Member;
   Begin
      Comment:  In the class body we could, in a more realistic
                example, have some attributes and statements
                that would model some part of a real system;
   end*** MEMBER***;

   Set:– new HEAD;

   for I:=1 step 1 until 10 do

   new Member.into (Set);

end*** SIMSET***;
```

SIMULATION

The system defined class SIMULATION is shaped very much like SIMSET, but its subclasses and procedures are designed to simplify the handling of simulation concepts, such as timekeeping, scheduling et cetera. But before we describe SIMULATION and its contents, we have to mention something about how to administrate an event simulation. As the name indicates such a simulation consists of a series of events that take place at different times. The decisions of the simulation are devoted to when the next active phase of some part of the system is to take place. The decision made results in a future eventtime and in some way we have to keep a record of these future eventtimes. The most common way to do this is to keep the events in a sorted list, usually called the event list. As the events of the simulation take place the simulated time progresses and we have to have some internal clock that is increased each time a new event occurs. We will now see how this is implemented in SIMULATION:

The class is a subclass to SIMSET and has the outline

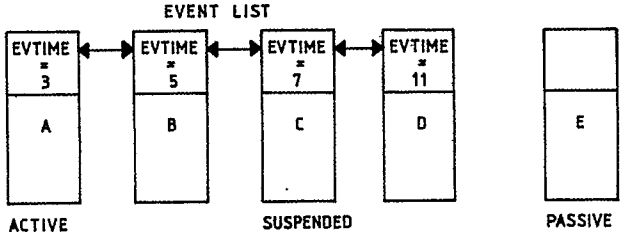
```
SIMSET Class SIMULATION;
Begin
   LINK Class PROCESS;
   Ref (PROCESS) Procedure CURRENT;
   Real Procedure TIME;
   Procedure HOLD;
   Procedure WAIT;
   Procedure CANCEL;
   Process Class main program;
   Ref (Main program) MAIN;
   < setup event list with system time zero >;
end*** simulation***;
```

Each class in the simulation that is to be scheduled should be prefixed by PROCESS, which has the outline

```
LINK Class PROCESS;
Begin
   Boolean Procedure IDLE;
   Real Procedure EVTIME;
   Ref (Process) Procedure NEXTEV;
end*** process***;
```

A PROCESS object of the simulation can be in one of three states: (1) active, (2) suspended and (3) passive. A fourth state, terminated, means that the object has been executed to its end and that it no more can take any active part in the simulation. We therefore concentrate on 1, 2 and 3.

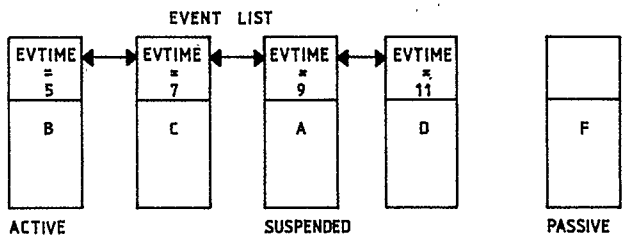
Look at the following snap-shot of the event list.



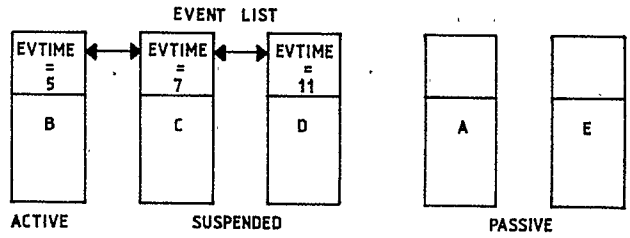
At the lower end of the list the object that is currently executed resides. This object is called active and a call to the procedure CURRENT inside a class or block prefixed by SIMULATION will result in a reference to this object. The time of the simulation is the event time of the active object and can be reached through a call to procedure TIME. Objects that are members of the eventlist but which are not active are called suspended. Objects, not members of the event list, but which have more active phases to come (i.e. they are not terminated), are called passiv.

In SIMULA there are three main ways, and related statements, to manipulate the event list: HOLD, ACTIVATE and PASSIVATE. Their effects are best shown by examples, and we use the figure above as initial configuration.

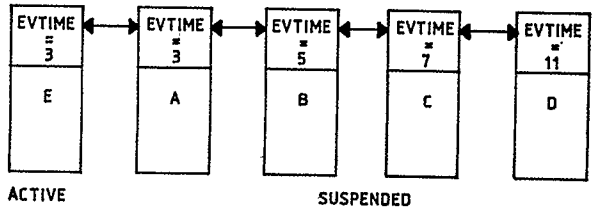
If we, in the object A above, execute a statement HOLD(6) it will have the effect that A is rescheduled at "time now + 6", and put in its proper place in the event list. Simulated time is stepped to the event time of the next object in the event list and the statements of that object will be executed. This results in differently shaped event list:



Execution of PASSIVATE will remove the active object from the event list and put it in a passive state. The next object in the event list will become active and we have:



Finally, ACTIVATE E will make E the new active object, i.e. the new CURRENT, and stop the execution of the previously active object at that statement. Simulated time will remain unchanged and the event list will look like:



We do not give a full account for all procedures declared in SIMULATION since this probably would just mix things up for the reader. Most of the procedures have names that are mnemonics, and the reader is encourage to consult the reference literature for a full account. Some of the possible statements, like, for example, ACTIVATE, have a few more elaborated forms which make them even more useful and we will use some of them in the fulfilment of our previous example but we postpone the explanation until then.

With the subset of the system defined classes SIMSET and SIMULATION we have showed so far, we will now proceed and use them to transform our earlier description of the computer system example into proper SIMULA. This will hopefully clearify how to use SIMULA to solve the problems that arose when we built our model. There are, of course, other ways to use SIMULA and to solve our example but the one presented here is one, and perhaps the most common.

THE EXAMPLE COMPLETED

Below we list one solution of our problem. Please observe the structure of the program: a lot of declarations and very few statements in the main program. This structure is very common in simulation programs written in SIMULA. All input and initial activation statements have been collected in separate procedures, a solution that gives the program a very "clean" appearance. There is a very specific reason why the input is collected in a separate procedure: since in SIMULA (and ALGOL and related

languages) it is not permissable to mix statements and declarations, it can, in large programs, be hard to get the declaration and assignment of variables in the same part of the program. In SIMULA all variables have an initial value, which for integers and reals are 0, and if an input to some variables are missed this initial value can effect the execution in just a minor way and cause harddiscovered errors. But if the declaration and the setting of values to variables are in the same part of the program it is easy to compare the list of declarations to the input list and control that all have been assigned values. This may be just a minor advantage but it could be accieved with a small effort. Observe also that at this stage no measuring routines have been introduced and that no variables for this purpose have been declared. How to measure and what variables to declare is very much dependent on what is to be measured and we will later give some examples.

Process Classes Cardreader, CPU, Disc and Lineprinter have all the same principal outline which means a cycle of removing jobs from the queue, make a HOLD proportional to the processing required by that Job and then take the next Job if any. When the queue is empty passivate the object and make sure that a component that puts a Job in an empty queue, i.e. to some idle component, activates the object to which the queue is associated.

The imaginary object TASK is a passive participant of the simulation and does only assign data to itself. We have used the name TASK for the Job concept since variables and classes are not allowed to have the same name. If we had used the class-name Job we would have been unable to use that name as a reference variable which we wanted to do since it mostly is the reference variables that occur in the program, as you can see. The data assignments to tasks is by no means statistically well-argumented, but is merely an example of how it could be done. RANDINT is a standard procedure in SIMULA that with equal probability gives an integral value between some upper and lower bounds. Processing times in CPU and DISC are chosen from an exponential distribution and NEGEXP gives a value from such a distribution. Variable names are self explanatory even if it sometimes can be hard to see what words they consist of. The listing is produced by a UNIVAC 1108 computer, and we ask the reader to overlook the somewhat vertical layout of the program. This is due to the effort of fitting program to the narrow columns of proceeding prints.

```
SIMULATION
BEGIN

   REF(CPU) PROCESSOR;
   REF(DISC) STORE;
   REF(LINEPRINTER) PRINTER;
   REF(CARDREADER) READER;
   REAL SIMPERIOD,ARRIVALINTENS;
   REAL TIMETOPRINTPAGE,TIMETOREADCARD;
   INTEGER STARTRANDGEN,MINNROFCARDS;
   INTEGER MAXNROFCARDS,MAXNROFDISCTRANSFERS;
   INTEGER MINNROFDISCTRANSFERS;
   INTEGER MINNROFPAGES,MAXNROFPAGES;
   REAL CPUPROCESSINGINTENS;
   REAL DISCPROCESSINGINTENS;

   PROCEDURE INPUT;

   BEGIN
      SIMPERIOD                    :=INREAL;
      ARRIVALINTENS                :=INREAL;
      TIMETOREADCARD               :=INREAL;
      TIMETOPRINTPAGE              :=INREAL;
      CPUPROCESSINGINTENS          :=INREAL;
      DISCPROCESSINGINTENS         :=INREAL;
      MINNROFCARDS                 :=ININT;
      MAXNROFCARDS                 :=ININT;
      MINNROFDISCTRANSFERS         :=ININT;
      MAXNROFDISCTRANSFERS         :=ININT;
      MINNROFPAGES                 :=ININT;
      MAXNROFPAGES                 :=ININT;
      STARTRANDGEN                 :=ININT;
   END*** INPUT ***;


   PROCESS CLASS JOBGENERATOR;


   BEGIN
      WHILE TIME < SIMPERIOD DO
      BEGIN
         HOLD(NEGEXP(ARRIVALINTENS,
                     STARTRANDGEN));
         NEW TASK.INTO(READER.INPUTQUEUE);
         IF READER.IDLE
         THEN
         ACTIVATE READER;
      END;
   END***JOBGENERATOR***;


   LINK CLASS TASK;

   BEGIN

      INTEGER NROFCARDS,NROFDISCTRANSFERS;
      INTEGER NROFPAGES;

      NROFDISCTRANSFERS :=
           RANDINT(MINNROFDISCTRANSFERS,
                   MAXNROFDISCTRANSFERS,
                    STARTRANDGEN);
      NROFPAGES :=
           RANDINT(MINNROFPAGES,
                   MAXNROFPAGES,
                   STARTRANDGEN);
      NROFCARDS :=
           RANDINT(MINNROFCARDS,
                   MAXNROFCARDS,
                   STARTRANDGEN);
   END*** TASK ***;
```

```
PROCESS CLASS CARDREADER;

BEGIN

   REF(HEAD) INPUTQUEUE;
   REF(TASK) JOB;

   INPUTQUEUE :- NEW HEAD;
   WHILE TRUE DO
   BEGIN
      WHILE NOT INPUTQUEUE.EMPTY DO
      BEGIN
         JOB :- INPUTQUEUE.FIRST;
         HOLD(JOB.NROFCARDS*
              TIMETOREADCARD);
         JOB.INTO(PROCESSOR.INPUTQUEUE);
         IF PROCESSOR.IDLE
         THEN
         ACTIVATE PROCESSOR
      END;
      PASSIVATE;
   END;
END*** CARDREADER ***;


PROCESS CLASS CPU;

BEGIN

   REF(HEAD) INPUTQUEUE;
   REF(TASK) JOB;



   INPUTQUEUE:- NEW HEAD;
   WHILE TRUE DO
   BEGIN
      WHILE NOT INPUTQUEUE.EMPTY DO
      BEGIN
         JOB :- INPUTQUEUE.FIRST;
         HOLD(NEGEXP(CPUPROCESSINGINTENS
                     ,STARTRANDGEN));
         IF JOB.NROFDISCTRANSFERS > 0
         THEN
         BEGIN
            JOB.NROFDISCTRANSFERS :=
            JOB.NROFDISCTRANSFERS-1;
            JOB.INTO(STORE.INPUTQUEUE);
            IF STORE.IDLE
            THEN
            ACTIVATE STORE;
         END
         ELSE
         BEGIN
            JOB.INTO(PRINTER.INPUTQUEUE);
            IF PRINTER.IDLE
            THEN
            ACTIVATE PRINTER;
         END;
      END;
      PASSIVATE;
   END;
END*** CPU ***;


PROCESS CLASS DISC;

BEGIN

   REF(HEAD) INPUTQUEUE;
   REF(TASK) JOB;

   INPUTQUEUE :- NEW HEAD;
   WHILE TRUE DO
   BEGIN
      WHILE NOT INPUTQUEUE.EMPTY DO
      BEGIN
         JOB :- INPUTQUEUE.FIRST;
         HOLD(NEGEXP(DISCPROCESSINGINTENS
                     ,STARTRANDGEN));
         JOB.INTO(PROCESSOR.INPUTQUEUE);
         IF PROCESSOR.IDLE
         THEN
         ACTIVATE PROCESSOR;
      END;
      PASSIVATE;
   END;
END*** DISC ***;


PROCESS CLASS LINEPRINTER;

BEGIN

   REF(HEAD) INPUTQUEUE;
   REF(TASK) JOB;

   INPUTQUEUE :- NEW HEAD;
   WHILE TRUE DO
   BEGIN
      WHILE NOT INPUTQUEUE.EMPTY DO
      BEGIN
         JOB :- INPUTQUEUE.FIRST;
         HOLD(JOB.NROFPAGES*
              TIMETOPRINTPAGE);
         JOB.OUT;
      END;
      PASSIVATE;
   END;
END*** LINEPRINTER ***;


PROCEDURE START;

BEGIN
   PROCESSOR :- NEW CPU;
   STORE     :- NEW DISC;
   PRINTER   :- NEW LINEPRINTER;
   READER    :- NEW CARDREADER;
   ACTIVATE PROCESSOR;
   ACTIVATE STORE;
   ACTIVATE PRINTER;
   ACTIVATE READER;
   ACTIVATE NEW JOBGENERATOR;
END*** START ***;

INPUT;
START;
HOLD(100000000);
END*** SIMULATION ***;
```

So far we have not bothered about how to measure quantities in our model. This is a very cumbersome part of the simulation and in many ways harder to give any specific rules for. The reader is on this issue referenced to some of the abundant literature on the subject. We will, however, give an example of how to measure a couple of quantities in our model. Suppose we want to find out the average time a job spends in the system and the mean queue length of the inputqueue in the CPU. To accomplish this, we have to measure all, or some, of the jobs what their time in system is concerned and take samples of the CPU queuelength. For this purpose we write a special measuring class – which is a PROCESS – and let that class become active with regular intervals. This class will keep a record of the accumulated number of measurements and the accumulated queuelength, which makes it possible to compute the mean queuelength. The average time a Job spends in system can be calculated in a simular way: in TASK we declare a variable entrytime which we set when a new Job enters the system. When leaving, it has spent a time in the system which is the difference between time at that occasion and the value of entrytime. A procedure local to class Measure will keep a record of the necessary quantities. This means we have to add to the program the following sections and we do not give all details of input, initialization and declarations:

We add the Class MEASURE

```
PROCESS CLASS MEASURE;

BEGIN

   REAL TOTALTIME;
   INTEGER TOTALNUMBEROFJOBS,NUMBEROFJOBS;
   INTEGER NUMBEROFMEASUREMENTS;



   PROCEDURE COLLECTDATA(JOB);
   REF(TASK) JOB;

   BEGIN
      TOTALTIME := TOTALTIME+TIME-
                   JOB.ENTRYTIME;
      NUMBEROFJOBS := NUMBEROFJOBS + 1;
   END*** COLLECTDATA ***;

   WHILE TIME < SIMPERIOD DO
   BEGIN
      HOLD(MEASUREINTERVAL);
      NUMBEROFMEASUREMENTS :=
      NUMBEROFMEASUREMENTS+1;
      TOTALNUMBEROFJOBS:=TOTALNUMBEROFJOBS
         + PROCESSOR.INPUTQUEUE.CARDINAL;
   END;
END*** MEASURE ***;
```

and declare a specific procedure for output which, together with the end of the program, will have to outline:

```
   PROCEDURE OUTPUT;

   BEGIN
      .
      INSPECT MEASURECLASS DO
      BEGIN
         OUTTEXT(@ CPU MEAN QUEUELENGTH:@);
         OUTFIX(TOTALNUMBEROFJOBS/
                NUMBEROFMEASUREMENTS,4,8);
         OUTIMAGE;
         OUTIMAGE;
         OUTTEXT(@AVERAGE TIME IN SYSTEM:@);
         OUTFIX(TOTALTIME/NUMBEROFJOBS,4,8);
         OUTIMAGE;
      END;
   END*** OUTPUT ***;

   INPUT;
   START;
   HOLD(100000000);
   OUTPUT;
END*** SIMULATION ***;
```

The inspect statement in procedure OUTPUT is a way to avoid repeated dot notation. Inside the inspection block statements are executed as if they were written inside the inspected class, in this case MEASURE. Other quantities can be measured in a similar way and by this we have completed our example.

CONCLUDING REMARKS

We have, in the preceeding pages, tried to give the reader a fair view of what simulation is and how to use SIMULA in order to simplify the development and understanding of simulation programs. We have thereby concentrated on the basic ideas of SIMULA rather than on a detailed description of the language. You do not learn languages through tutorials but we hope we have succeeded to bring out the naturalness in modelling and program evaluation when SIMULA is in use. Some of the advises we have given may seem simple and fussy, and so they are, when the programs you write are small. But when they grow in size and complexity, which they do, it gets more and more important to follow some kind of guideline in the steps to a complete program. Today, when costs for hardware and computers on the hole is decreasing, a good programming methology can achieve large savings in the balance, since evaluation, debugging and documentation will account for the major part. A well structured program is also more realiable, a most important aspect in modern systems.

References

1 Birtwistle G.M., Dahl O.-J., Myhrhaug B., Nygaard K., **SIMULA Begin**, Auerbach Publishers Inc., Philadelphia, Pa., 1973.

2 Franta W.R., **The Process View of Simulation**, Elseview North-Holland, Inc., N.Y., New York, 1977.

3 Shannon R.E., **System Simulation: The art and science**, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1975.