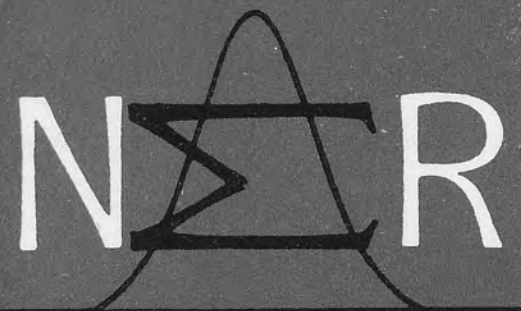


Publication No. S 55
NORWEGIAN COMPUTING CENTER
Forskningsveien 1B, Oslo 3,
Norway.



NORSK REGNESENTRAL
NORWEGIAN COMPUTING CENTER
OSLO 3 - NORWAY

AN INTRODUCTION TO SIMULATION

USING

S I M U L A

BY

P. R. HILLS

ROBIN HILLS (CONSULTANTS) LTD.

© Copyright NCC 1973

TM SIMULA is the registered trademark
of the Norwegian Computing Center,

The author would like to express thanks to Terje Bøen of the Norwegian Computing Center who checked the text and made the programs work.

INTRODUCTION

TYPE OF MODEL

Simulation in the sense we will use in this introduction means the building of working replicas of real life systems in order that experiments may be carried out to determine the real system's probable performance under various operating situations. We will not consider economic models, where performance, input-output and between system relationships are represented by mathematical equations, but concentrate on mechanistic models where actual movements and changes of state of components of the system are reflected in the model.

We will narrow the field still further by only considering discrete-event models and ignoring the continuous ones used by engineers. We are concerned therefore with the type of simulation carried out by Operational Research workers.

DISCRETE EVENT MODELS

In discrete event simulation we consider the model to consist of a number of multi-discrete-state objects. At any instant each one will be in a particular state and will remain so until at a later time, one or more will instantaneously change. A new composite state will then hold until a yet later time when other changes take place. The model will then jump from one state to the next. To illustrate this idea, we may consider a cine film. In this the continuous action is represented by single snap shots taken at small time intervals. If we took a sample of frames from a cine film, each one showing an event of importance, we could replace the film show with a "slide show". Indeed we could avoid the need for a cine film altogether by having the actors pose the important scenes, photographing each with a still camera. The resulting performance would lack much that the cine film would provide, but for the purpose of studying the way the plot developed it might well be sufficient. It would

certainly be far less expensive to produce. The parallel between such a cinema performance and an Operational Research study holds good, for in most cases the O.R. man is concerned with the way a system operates only in outline. He is concerned with the way events of importance to him will follow one another so that he can predict when more resources will be required or when overflow situations will occur or how often deliveries will be late and so on. He is not usually concerned with the minutiae of the real world. His useful "frames" will then be only a very small selection of the millions of second by second ones that could be produced to represent a real system.

It is the central problem of the O.R. man to decide what is sufficient in a model and what can be left out, and this is clearly outside the scope of any introduction to simulation, so we will concern ourselves here only with the production of chosen models.

MODEL CONSTRUCTION

To construct a simulation model the analyst will first have to decide which parts of the real world he will include. He will then have to describe his chosen 'outline system' in sufficient detail that a program can be written to replicate it. There are three fundamentally different approaches to this. These are:

- The activity approach, in which activities or tasks carried out in the real world are described in terms of the resources they require, how long they take and how they change the state of the resources they use.
- The event approach, in which events of importance are described in terms of the change to the overall model state which takes place and future events that can be predicted on their occurrence.
- The process approach, in which whole chains of actions of model components are described together with the ways in which each chain or process will interfere with others.

We will use the first approach, which is the easiest to follow for most of this text but illustrate the alternatives at the end.

EXAMPLES

Throughout the text we will refer to two models - a very simple one concerning an hotel bar and a more complicated and realistic one about a steel production plant.

BAR EXAMPLE

Our bar example is simple in the extreme, it consists of two men sitting at a bar drinking glasses of beer. They are served by a single barmaid who refills their glasses. To avoid unnecessary delays waiting for beer, they arrange to share a stock of glasses which the barmaid washes and fills as soon as they are empty. The men, as they finish a glass, put it down and take up a full one.

STEEL PRODUCTION

In our other example, we consider the production of slabs of steel. We have a furnace which first smelts a load of steel and then pours it into several sets of moulds as these become available, it is then replenished prior to the next smelt. The steel in the moulds is allowed to set, then the moulds are broken open, the ingots removed and the moulds are put together and cleaned for further use. The ingots, meanwhile, are loaded into soaking pits where they are held at a high temperature for a period of time until they are ready for rolling. Rolling turns the ingots into slabs, the end product of the system.

This example is taken from a real life study carried out at the Spencer Works of the British Steel Corporation.

We will in following sections consider how these example models can be:

- specified in terms of the activities that take place
- operated so that the activities follow one another in the right sequence
- programmed for computer operation.

ACTIVITY MODELS

MODEL COMPONENTS ENTITIES

In the activity approach we consider the model to be made up of entities representing real life objects such as cranes, cars, ships and so on, but maybe also less intrinsic things like messages or permissions. These entities may have many different states of existence, such as busy or idle or moving or full or empty, and at any given time will be fully described by having their states specified. In any model, the entities will interact with one another, so that, for example it will only be possible to have the entity 'crane' in the state of 'lifting' if we also have an entity - perhaps 'box' - in the state of 'being lifted'. In activity models we concentrate our attention on these interactions via the concept of any activity.

ACTIVITIES

An activity is a co-operation between entities to carry out some task. It is defined in terms of:

- A starting condition, composed of a list of entities together with their necessary states, which must be available before the activity can start.
- A start-up program, consisting of a list of actions (in terms of entity state changes), which takes place at the instant that the activity starts.
- A duration, in the form of a formula or value to specify how long the activity will take.
- An ending program, consisting of a list of actions (in terms of entity state changes) which take place at the end of the activity.

QUEUES

Entity states can be classified into two categories, those where the entities are engaged in activities, and those while they are not in activities. These latter states are when the entities are idle and often when they are in queues or waiting lines. Indeed it is convenient to always classify such idle entities as being in queues. If this is done then the starting conditions for an activity will always consist of testing appropriate queues to see if the necessary entities are there.

ACTIVITY CYCLE DIAGRAM

A diagrammatic representation of an activity model can be achieved using the activity cycle diagram. This traces the path of each type of entity in a model through its idle and active states. Since the activities are usually co-operative actions, the different entities' paths will coincide at them, reflecting the interaction that takes place. We will illustrate these ideas with the beer drinking and steel production examples described in section 1.

BEER DRINKING EXAMPLE

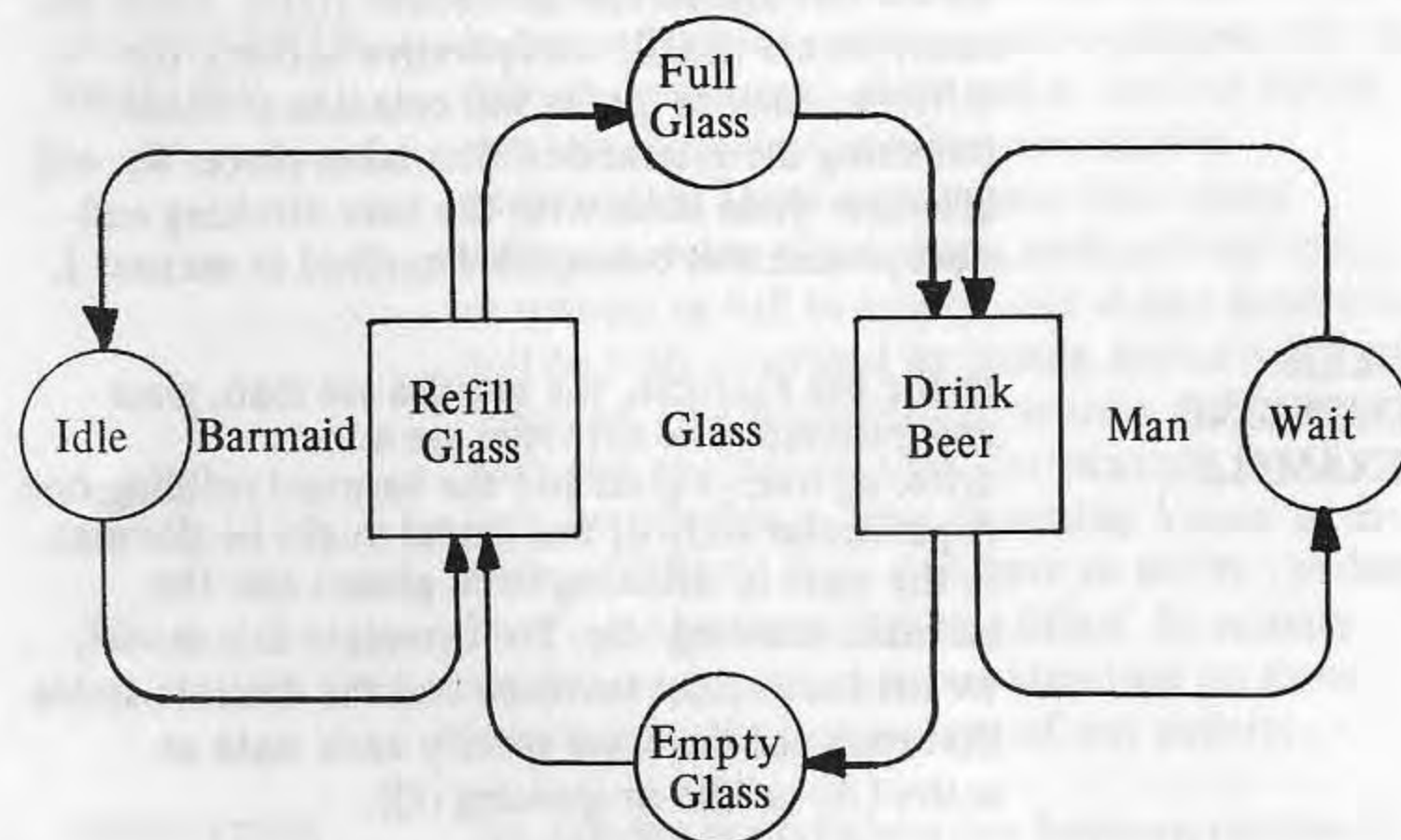
In the bar example, the entities are man, glass and barmaid. The activities are a man drinking from a glass and the barmaid refilling one. A particular state of the model might be the men in the state of drinking from glasses and the barmaid standing idle. To formulate this model, we list the entities involved and the discrete states that each can have; we specify each state as active (A) or idle or queuing (Q).

Entity	-	States
man	-	drinking (A)
	-	waiting (Q)
glass	-	being drunk from (A)
	-	waiting filling (Q)
	-	being refilled (A)
	-	waiting drinking (Q)
barmaid	-	refilling the glass (A)
	-	idle (Q)

In the activity diagram we represent an Activity by a square, and a Queue by a circle.

The Activity Life Cycles for the man, glass and barmaid are shown in the following diagram.

The directional arrows indicate the paths of the entities during their lives.

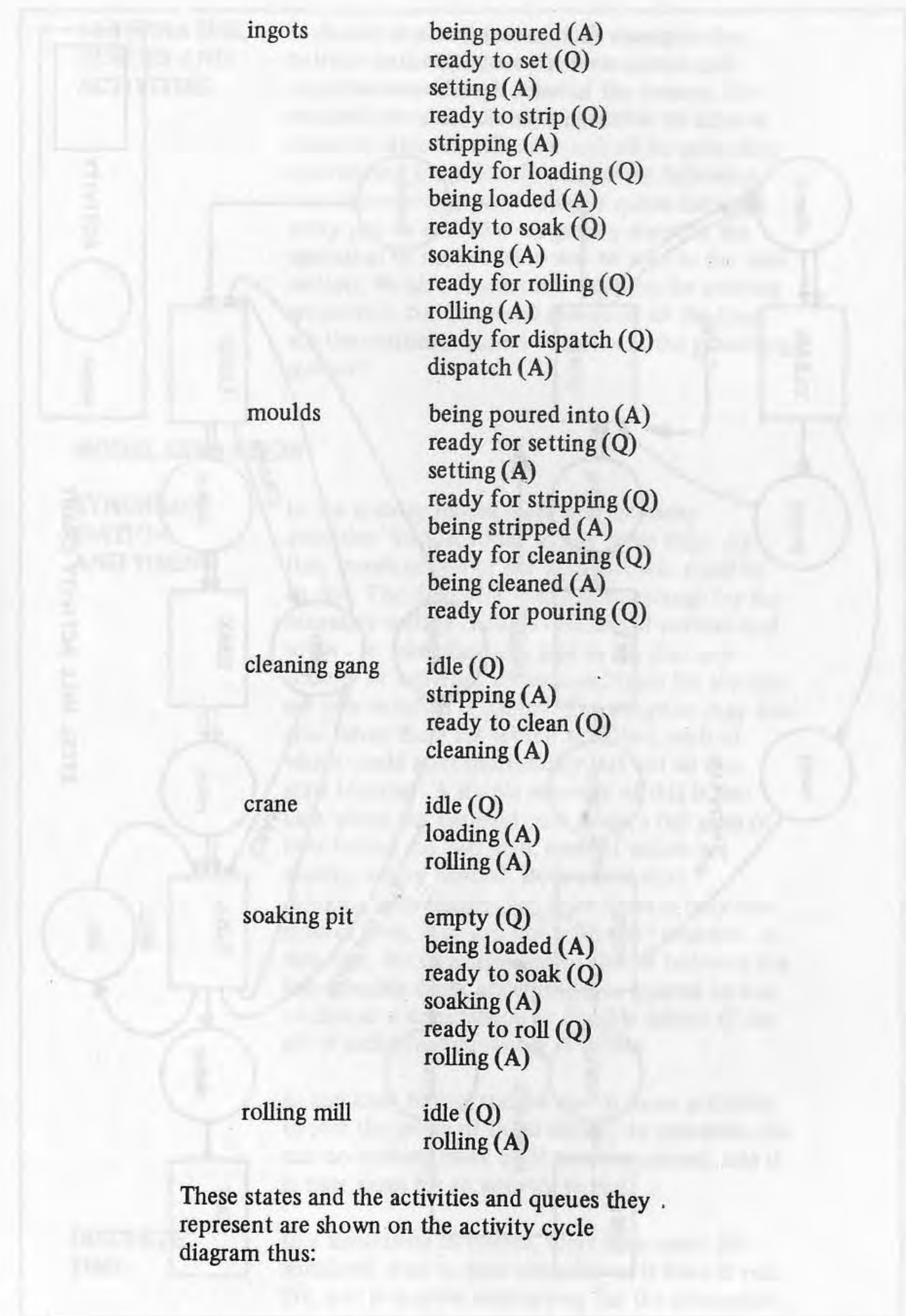


In this example, the entities, man and glass, are required for the activity 'drink beer' and barmaid and glass are needed for the activity 'refill glass'. This interaction is represented in the activity cycle diagram since the entity cycles join in these activities.

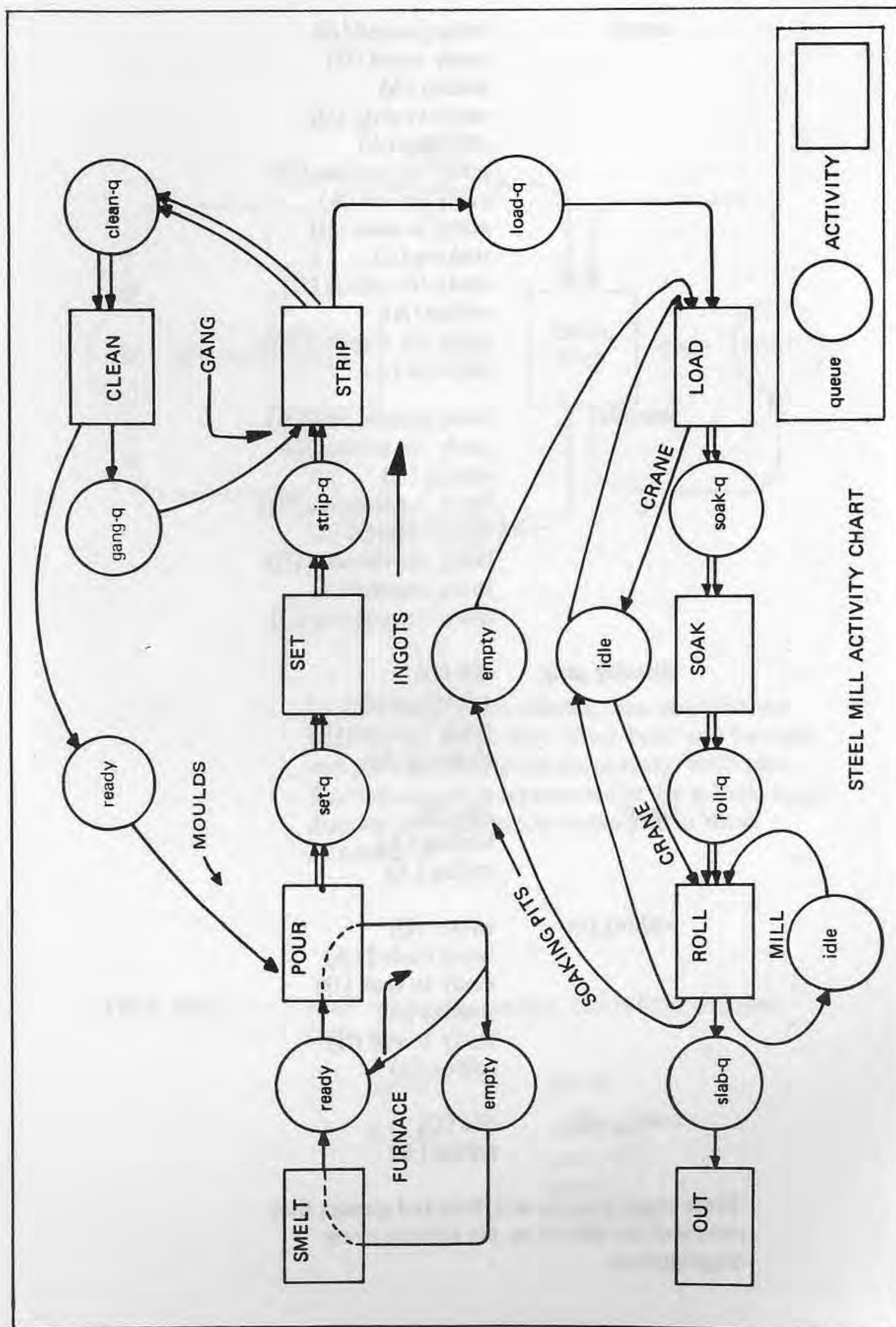
STEEL MILL

In the steel mill example, the entities and their states are:

Entity	States
furnace	smelt (A) ready (Q) pour (A) empty (Q)



These states and the activities and queues they represent are shown on the activity cycle diagram thus:



ALTERNATING QUEUES AND ACTIVITIES

It should be noted that in both examples the entities' paths alternate between queues and activities even though some of the queues, for example set-q, are clearly impossible to achieve, since the entities will come out of the preceding activity and immediately pass to the following one. However, by introducing a queue between every pair of activities we greatly simplify the operation of the model as will be seen in the next section. We also standardise the rules for starting an activity, for these will always be of the form - are the entities required available in the preceding queues?

MODEL OPERATION

SYNCHRONISATION AND TIMING

In the activity model there may be many activities "in operation" at any given time. As time passes so one of the activities will come to an end. The simulator then has to arrange for the necessary ending changes - release of entities and so on - to take place and also to see that any activity or activities whose conditions for starting are now satisfied is started. The situation may also arise when there are several activities, each of which could start individually but not all can start together. A simple example of this is the case where the barmaid puts down a full glass of beer before the two men, both of whom are waiting empty handed. Both could start drinking individually, but since there is only one glass of beer, they can not both start together. In this case, the simulator has to choose between the two possible drink activities, or in general he has to choose a simultaneously feasible subset of the set of individually feasible activities.

In any case, having started one or more activities to join the group of those already in operation, he can do nothing more until time has passed, and it is time again for an activity to end.

DISCRETE TIME

In a simulation of course, there is no need for simulated time to pass smoothly as it does in real life, and it is quite satisfactory for the simulation

"clock" to jump from one time to the next. In our parallel with the cinema film and slide show, the time which elapsed between important scenes in a plot might well be important, but in order to study the plot it could be sufficient if the time when each scene took place were known to the audience by say a clock on the wall of the stage.

The simulator can arrange then, that as soon as he has started all the activities able to start, he will move the simulation clock forward to the end time of the activity first due to end.

THREE PHASES

Simulation cycles then through three phases which are:

- Start any activities which can start
- Move simulation time to the end time of the activity in progress due to end first
- End that activity (and any others which happen to end at the same instant)

HAND OPERATION

This three stage operation can of course be programmed for a computer, since the actions that have to take place can be carried out in sequence even though the activities they control are operating in parallel. Equally well they can be carried out by hand, as it were playing the simulation like a board game! This mode of operation has several advantages including:

- testing the model - the simulator can see immediately that something is wrong or that his model is incomplete and make the necessary changes.
- demonstrating the model - the whole simulation exercise will only be of value if the manager or client implements the recommendations which stem from it. It may well be that he will need convincing that the model is complete and nothing of importance is neglected if he is to implement recommendations which are contrary to his own ideas or intuition about the likely behaviour of the system being

modelled. A hand simulation can be of great value in showing exactly how a model is built.

- Illustrating the recommendations — The recommended actions or new operating rules can often be explained to the staff who must implement them using a hand simulation.

The activity cycle diagram shown above provides a natural 'playing board' for hand simulation with the entities involved in the model being represented by tokens - coins, poker chips etc. The changes of state of the entities are represented by physically moving the tokens from queue to activity to queue. For example in the beer drinking example the start of the activity of drinking would be represented by moving the token representing a man from the man-idle queue circle to the activity drink square, and at the same time moving a token for a glass from the full glass circle to the same activity square.

COMPUTER OPERATION

When the computer is programmed to run the simulation, exactly the same concept of operating the model is followed, except of course the locations on the diagram and the entity tokens are represented by store values in the computer's memory. The necessary statements for such a program are available in any programming language, but of course a simulation language will take special care to provide the most convenient ones.

The following pages show how SIMULA can be used to deal with activity models.

THE SIMULATION FACILITIES OF SIMULA

INTRODUCTION While writing about programming we will assume a familiarity with ALGOL. Hopefully FORTRAN programmers will find sufficient similarity with their chosen language to be able to follow the text. There are several good textbooks on ALGOL and it is recommended that the intended user of SIMULA for simulation work should consult one if he does not have a working knowledge of this language. SIMULA is covered in full by 'SIMULA begin' by Birtwistle et al, published by Studentlitteratur, Stockholm.

CLASSES SIMULA introduces a construction of considerable power to an ALGOL base in its CLASS facility. The CLASS is defined in a similar way to the procedure. The vital difference is that whereas when the procedure is invoked, its variables are set up, its program is executed and then it disappears, this is not true for the CLASS. When a CLASS definition is invoked an object is created in store containing in effect the variable locations required, any local procedures and a copy of the CLASS program. This may be used immediately as in the case of the procedure or may be retained for later use. The program itself may be carried out in one step or in several stages at different times during the execution of the calling program.

REFERENCING A class object on creation may be given a reference so that it may be referred to later and SIMULA introduces a type of variable which can hold such references. This is known as a reference variable and has the sole use of storing references to class objects. Furthermore on declaration a reference variable is given a class to which it may refer and in use it is only allowed to refer to objects created to that class definition (or prefixed by it). When a reference variable is not storing a reference it takes the value NONE.

ATTRIBUTES As with procedures, classes may have parameters, that is, variables which have to be given values when the call of the class or procedure is made, and also

local variables and local procedures. We will call these together attributes. In the case of a procedure these attributes are only used while the procedure is being used. With classes, since the class object may exist for some time during the running of the main program, its attributes also exist for this time. Indeed even though the local program in the class object may be exhausted, while it is still referenced by a reference variable it remains in existence with all its attributes.

ACCESS

Attributes of a class object may be accessed either so that their values and other properties may be used outside the object or so that their values may be changed from outside the object. Access is available via any reference variable which is currently referring to the object. It should be noted however that only attributes declared at the outermost level in a class definition may be assessed in this way, that is, parameters or local variables or procedures declared at the head of the definition and not within any inner blocks.

FORM

A CLASS definition takes the form:

```
class NAME (PARAMETER1,PARAMETER2,...); *1  
    TYPE PARAMETER1,PARAMETER2,...;  
    begin TYPE LOCAL VARIABLES;  
        PROGRAM;  
        .....;  
        .....;  
    end;
```

The parameters or the local variables or the program may be omitted. A class object is created thus:

```
    new NAME (P1, P2,...);
```

where NAME corresponds with the declared name and P1, P2 etc. are expressions corresponding in type to the formally declared parameters

PARAMETER1, PARAMETER2, etc.

A reference to this object may be stored in a

*1 actual SIMULA program is given in lower case while informal statements are given in upper case.

reference variable created thus:

ref (NAME) var;

and the reference may be made thus:

var :- new NAME (P1, P2, ...);

Access may then be made to the objects attributes by a dot notation thus:

x := var.PARAMETER1;

EXAMPLE

An example of the above declarations and use is:

class exchange (a, b); real a, b;

begin real c;

c := a;

a := b;

b := c;

end;

ref (exchange) swap;

real d, e;

.....;

.....;

swap :- new exchange (4.5, 3.2);

d := swap.a;

e := swap.b;

The values attained by d and e are 3.2 and 4.5 respectively (not the other way round!).

PREFIXING

A class declaration may also be split into several parts so that properties shared by several different classes may have their common part declared once and their different parts declared separately. The common part is then used as a prefix to the separate declarations. For example in a simulation program we may wish to have use of several different types of motor vehicle all having common attributes of length width and speed but also having quite different additional attributes, a bus having a number of seats and a number of passengers, a truck having a capacity and a load. This could be declared thus:

class vehicle;

begin real length, width, speed; end;

vehicle class bus;

begin integer seats, passengers; end;

vehicle class truck;

begin real load, capacity; end;

A reference variable could then be declared as ref

(vehicle), ref (truck) or ref (bus). One declared as ref (vehicle) would ordinarily only be able to give access to vehicle attributes i.e. length, width and speed but one declared as ref (bus) or ref (truck) could access seats and passengers or load and capacity respectively as well as length, width and speed.

SYSTEM DEFINED CLASSES

In addition to the class structures of general use defined above there are special class structures of use in simulation programs. These are made available to the user by prefixing the whole of a program with the class SIMULATION. The special classes can then be thought of as being part of the language. These special classes since they are defined as part of the formal definition of SIMULA are known as system defined classes.

LIST PROCESSING

Two of the system defined classes are HEAD and LINK classes. Any class prefixed with HEAD has the additional property that it may be used as a list store in which objects of a class prefixed with LINK may be stored. An object of class HEAD alone can be used as a list store but of course it will have no user defined properties. There are a number of procedures available as attributes of HEAD and LINK objects. These are:

Link attributes

suc;	gives a reference to the following member in the list or <u>NONE</u> .
pred;	gives a reference to the preceeding member or <u>NONE</u> .
out;	removes the object from any list it may be in.
into (H);	places the object as last member of a list H, removing it from any previous list it may be in.
precede (L);	places the object in front of another link object L in whatever list L is in again removing it from any previous list. is in.
follow (L);	as with precede but behind L.

Head attributes

clear;	removes all link objects from the head object's list.
first;	gives a reference to the first member of the object's list.
last;	gives reference to its last member.
empty;	gives a boolean value <u>true</u> if the list has no members.
cardinal;	gives the number of members.

There are of course default actions if the procedures cannot take place for logical reasons such as if the program tries to add a member to an object which is not a head object.

EXAMPLE

To illustrate use of the list processing facilities we will refer back to the vehicle class objects mentioned on page . If we wanted to create several vehicles and add them to a list we could do the following:

```
link class vehicle;  
begin real length, width, speed; end;  
vehicle class bus;  
begin integer seats, passengers; end;  
vehicle class truck;  
begin real load, capacity; end;  
ref (head) list;  
list :- new head;  
new bus.into(list);  
new truck.into(list);  
new bus.into(list);
```

A subsequent reference to a member of the list might be:

```
ref (vehicle) v;  
v :- list.first;
```

PROCESS CLASSES

Another type of system defined class is the process class. Any class prefixed with PROCESS has the additional properties of:

- having as a real attribute a simulation time value called EVTIME.
- being able to have its program interrupted for continuation later in a simulation by special control instructions.

- having a local sequence control register so that if it is interrupted it can later be continued from the place it left off.
- being able to belong to a special list known as a time axis consisting of process class objects in order of their EVTIME.
- being able to refer to its neighbour in the time axis as its NEXTEV.

MAIN PROGRAM and TIME AXIS

Any program prefixed by SIMULATION also has an element called MAIN which is the outermost block within the SIMULATION prefix. This acts like a process and can be scheduled by being included in the time axis. At any time during execution there may be several process objects and the main program in existence. At least one of these will be in the time list and others may be considered dormant. Overall program control operates on the object or main program at the head of the time axis. When this is completed or goes dormant or reschedules itself for later execution control passes to the next object in the time axis.

SIMULATION TIME

A SIMULATION program also has a global real variable TIME which is used to represent the real world time. It is automatically set equal to the EVTIME of the process object at the head of the time axis.

ACTIVATION

A process class object on first creation is left dormant with its local sequence control set at its first instruction. Its program is only started when it is made subject to an ACTIVATE statement. The following sequence would create and then activate a process class object:

```
process class journey;  
begin .....;  
.....;  
.....;  
end;  
ref (journey) trip;  
trip :- new journey;  
activate trip;
```

alternatively we could shorten this to:

```
process class journey;  
begin .....;  
.....;  
.....;  
end;  
activate new journey;
```

The difference being that in the first case we are left with a separate reference to the process object which may be useful.

The actual mechanism employed is that when the activate statement which must be external to the process object is encountered the process object is put at the head of the time list and its evtime is set equal to the evtime of the previous head. In the second example above let us assume that the statement activate new journey; is in the main program. While the main program is being executed it will be at the head of the time axis. On encountering the activate statement a journey class object will be created and then it will be placed before main in the time axis. Control will pass to it taking up where its local sequence control stands and as it is a new object this will be at the first statement. When its statements are exhausted (or it goes dormant in some other way as explained below) control will pass to the next object in the time axis which will in this case mean returning to the main program and execution will continue from the place at which the sequence control stands which will be the statement following activate new journey.

EXTERNAL CONTROL STATEMENTS

The external control statements, that is those statements which control the operation of a process class object from some other part of the program are as follows:—

- activate PROCESS CLASS OBJECT;
As above this activates the object immediately.
- activate P.C.O. delay VALUE;
This object is put into the time axis for future activation with an EVTIME of TIME + VALUE.

- activate P.C.O. at VALUE;
The object is put in the time axis with its EVTIME = VALUE.
- activate P.C.O. before PROCESS OBJECT;
The object is put in the time axis before a second object.
- activate P.C.O. after PROCESS OBJECT;
As above but after the second object.
- reactivate P.C.O.;
All of the preceding statements may have activate replaced by reactivate. In this case the object is rescheduled even if it is already in the time axis. With activate if the object is already in the time axis no action is taken.
- activate P.C.O. at VALUE prior;
The activate and reactivate statements at or after VALUE described above will place an object behind another in the time axis which happens to have the same EVTIME. Use of prior will cause the object to be placed before one with the same EVTIME.
- cancel (P.C.O.);
If the object is in the time axis it will be deleted from it.

INTERNAL SCHEDULING

There is a second group of process control statements which are placed within the process object and control the object itself. These are:

- hold (VALUE);
Replaces the object in the time axis at an EVTIME = TIME + VALUE.
- passivate;
Takes the object out of the time axis.
- wait (HEAD OBJECT);
Is equivalent to: into.(HEAD OBJECT); passivate;

SYSTEM PROCEDURES

The following system procedures and variables are available.

- current;
Returns a reference to the currently active process object.
- time; Returns the EVTIME of the currently active process object.

PROCESS CLASS ATTRIBUTES

The following attributes are available to a process class object:

- idle; returns a value true if the object is not in the time axis.
- terminated;
returns a value true if the object is not in the time axis and its program is exhausted.
- nextev;
returns a reference to the object following in the time axis.
- evtime;
returns the EVTIME of the object.

EXAMPLE

As an example of the use of these process control statements we now give a program to simulate a cuckoo clock. We will imagine that we start the clock at 7.10 and leave it running. Every hour on the hour the cuckoo sounds the number of hours. The clock will run for 12 hours. At random times we go and read the time and if the clock has stopped rewind it.

PROGRAM 1

```
1 simulation
2 begin
3   process class noise;
4   begin
5     again:
6       outtext("cuckoo");
7       passivate;
8       goto again;
9   end *** noise ***;
10
11  process class clockwork;
12  begin
13    integer hours, minutes, run.I;
14    boolean rundown;
15
16    procedure windup;
17    begin
18      rundown := false;
19      run := 0;
20      activate clock after current;
21    end *** windup ***;
22
23  start:
24    while  $\neg$  rundown do
25      begin
26        if minutes < 60 then minutes := minutes + 1
27        else
28          begin
29            minutes := 1;
30            hours := if hours = 12 then 1 else hours + 1;
31            for I := 1 step 1 until hours do activate cuckoo;
32            outimage;
33          end;
34          hold(1);
35          run := run + 1;
36          if run  $\geq$  720 then rundown := true;
37        end;
38        passivate;
39        goto start;
40    end *** clockwork ***;
```

```

41
42  ref(clockwork) clock;
43  ref(noise) cuckoo;
44  integer I;
45  I:= 34567;
46  clock:- new clockwork;
47  cuckoo:- new noise;
48  clock.hours:= 7;
49  clock.minutes:= 10;
50  activate clock;
51  while time < 10000 do
52  begin
53    hold(normal(100,25,I));
54    outint(clock.hours,5);
55    outchar('.');
56    outint(clock.minutes,2);
57    if clock.rundown then
58    begin
59      outtext(" and stopped");
60      clock.windup;
61    end;
62    outimage;
63  end;
64 end;

```

A sequence of images of the computer store is now given to show how the SIMULA system would run this program. Each process object created will be represented by a square block giving the objects attributes. (The block images are not given at every instruction but only occasionally).

NAME	N	local sequence control
Var 1	X	present value
Var 2	X	present value
EVTIME	T	present value

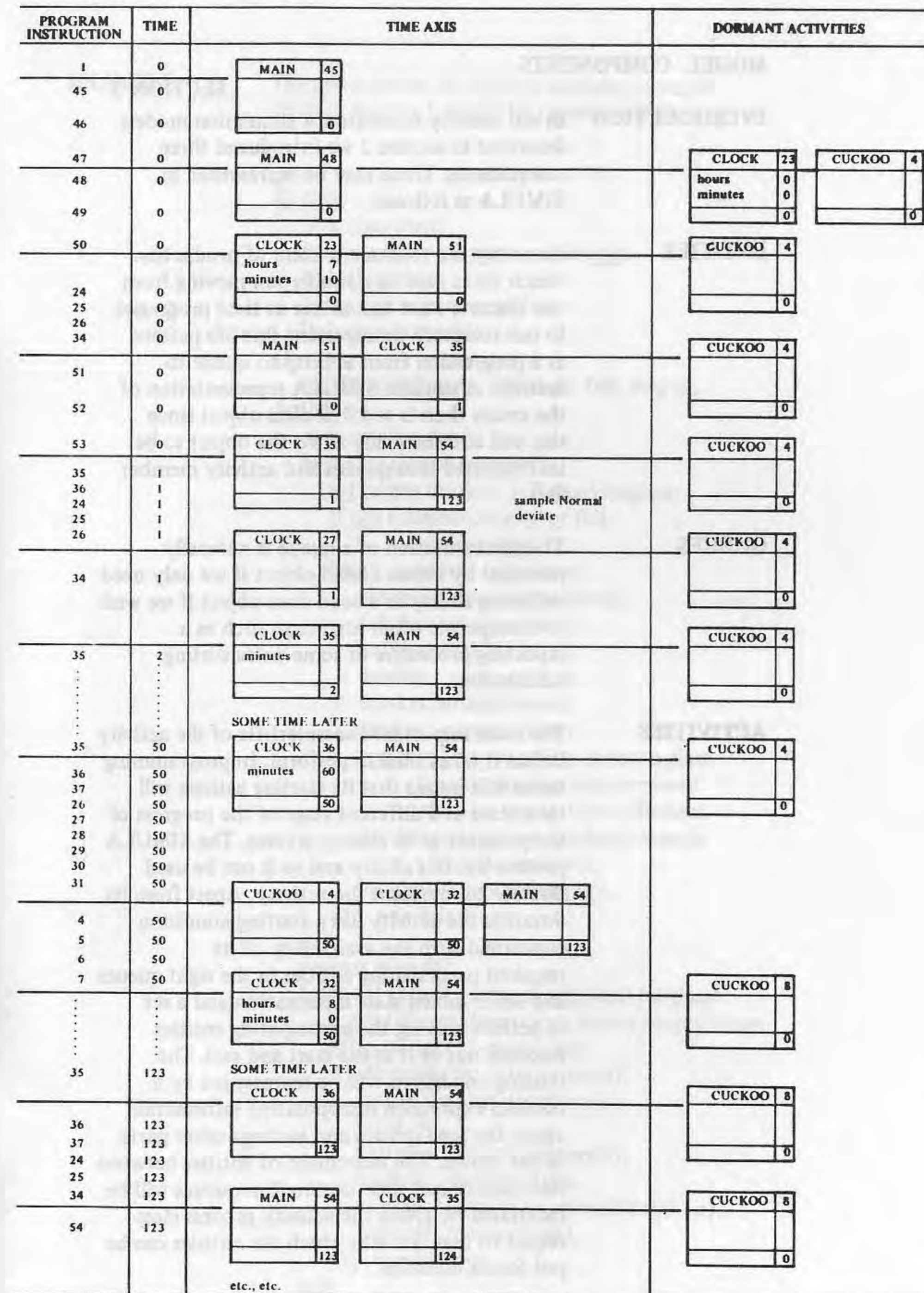


DIAGRAM 1

MODEL COMPONENTS

SECTION 5

INTRODUCTION In our activity structure for simulation models described in section 2 we introduced three components. These may be represented in SIMULA as follows:

ENTITIES An entity is a resource or item of production which takes part in a simulation moving from one discrete state to another as time progresses. In our structure we represent this life pattern as a progression from activity to queue to activity. A suitable SIMULA representation of the entity then is as a link class object since this will automatically allow the object to be incorporated into queues and activity member lists.

QUEUES The representation of a queue is naturally provided by either a head object if we only need its listing ability or a head class object if we wish to incorporate other attributes such as a reporting procedure or some extra sorting mechanism.

ACTIVITIES The most important characteristic of the activity is that it takes time to perform. In programming terms this means that its starting actions will take place at a different stage of the progress of the program to its closing actions. The SIMULA process has this ability and so it can be used directly to represent the activity. Apart from its duration the activity has a starting condition concerned with the availability of its required participating entities in the right queues and other model state information and a set of actions moving the participating entities into and out of it at the start and end. The starting conditions will be represented by a boolean expression incorporating information about the feed queues and perhaps other parts of the model. The movement of entities between their source and their destination queues will be facilitated by giving the activity process class object its own list into which the entities can be put for its duration.

EXAMPLE

The components of the beer drinking example given in section 1 would be represented thus:

Entities

```
link class man;  
begin real capacity, contents; end;  
link class glass;  
begin real contents; end;  
link class barmaid;;
```

Queues:

```
ref (head) idlemen, idlemaid, full, empty;
```

Activities:

```
process class drink;  
begin ref (head) partners;  
  ref (man) drinker; ref (glass) hisglass;  
  if not (idlemen.empty or full.  
    empty) then  
begin partners :- new head;  
  idlemen.first.into(partners);  
  full.first.into(partners);  
  hold(normal(10,2.5,r);  
  drinker:- partners.first;  
  drinker.into(idlemen);  
  hisglass:-partners.first;  
  comment partners.first is now a glass  
    the man having been removed  
    when he was put into idlemen;  
  drinker.contents := drinker.contents  
    + hisglass.contents;  
  partners.first.into(empty);  
end *** drink ***;  
end.  
process class fill;  
begin ref (head) partners; ref (glass) herglass;  
  if not (idlemaid.empty or empty.empty) then  
begin partners :- new head;  
  idlemaid.first.into(partners);  
  empty.first.into(partners);  
  hold(5);  
  partners.first.into(idlemaid);  
  herglass :-partners.first;  
  herglass.contents := normal(0.9,0.1,r);  
  partners.first.into(full);  
end *** fill ***;  
end;
```

CONTROL

Having created the elements of a model we then have to arrange a control mechanism so that they start and end their active phases in appropriate sequence. In overall terms we have to arrange three stages of operation. These are:

- An initiation when the components are created and set into a suitable state for our experiment.
- An operation stage when our simulation experiment is actually carried out.
- A reporting stage when the results of the run are presented.

The operation stage itself will, as we saw in section 3, be a cycling through three phases. These are:

- A start phase when each of the activities in turn is examined to see if its conditions for starting are satisfied and therefore it must be started.
- A time phase where simulated time changes until one of the activities in progress comes to an end.
- An end phase when one of the activities whose active state is due to end is actually ended.

Since the conditions for starting each activity are included within the process class declaration representing it, in order to carry out the start phase we merely have to activate a new instance of the process class. If its test is successful then this will continue to carry out its start up actions and then go into a hold state, being put into the time axis list. If on the other hand the test fails then the process object representing the activity will be completed and effectively disappear. At the end of the start phase we will be left with several process objects in the time axis. If we now allow the main program to become passive control will pass to the first of these. Since simulation time will take the EVTIME of this process object the second phase will have taken place automatically. The object at the head of the time axis will carry on with its program and this will look after the third phase. The one remaining thing to be done is to arrange to return to the start

phase so as to recycle through the phases. This can be done by incorporating an instruction to reactivate the main program in each process after its ending instructions.

In our program we would introduce this extra statement as follows:

```
process class drink;
begin .....;
.....;
.....;
partners.first.into(empty);
activate main;
end;
end;
process class fill;
.....;
.....;
.....;
partners.first.into(full);
activate main;
end;
end;
```

The whole program with its control section is now:

```
PROGRAM 2
simulation
begin
link class man; .....; ref (man) jim;
link class glass; .....;
link class barmaid; .....;
ref (head) idlemen, idlemaid, full, empty;
process class drink; .....;
process class fill; .....;
comment initiation;
idleman :- new head; idlemaid :- new head;
full :- new head; empty :- new head;
new man.into(idlemen); new man.into(idlemen);
new barmaid.into(idlemaid); jim :- idlemen.first;
for i := 1 step 1 until 10 do new glass.into(empty);
comment operation;
while time lt 1000 do
begin ref (process) a;
for a :- new drink, new fill do activate a;
passivate;
end;
comment report;
outfix (jim.contents, 5,10);
end;
```

RECYCLING OF START PHASE

Of course such a simple example could be programmed in much more simple terms but the above structure is suitable for any model.

In the model described above it is unlikely that two instances of the same activity could start at the same instant in time. However there is the small chance that the two men could both start drinking at the same time. Further since the above structure should hold for any model, in general we must recognise the possibility of two or more instances of the same activity starting during the same start phase. We can look after this occurrence by adding the statement to activate a new instance of the activity process immediately after the condition test and the start up actions. In the example we would modify the drink activity thus:

```
process class drink;  
  begin ref (head) partners;  
    ref (man) drinker;  
    if not (idlemen.empty or full.empty) then  
      begin partners := new head;  
        idlemen.first.into(partners);  
        full.first.into(partners);  
        activate new drink;  
        hold(normal .....);  
        .....;  
        etc.
```

There is no need to do the same thing for the fill activity since there is only one barmaid and so only one fill can possibly start at a time.

In general we would incorporate a statement to check for the multiple starting of an activity in any for which this was remotely possible.

VARIATIONS OF CONTROL PATTERN

There are many possible variations of the control procedures and these will be examined in more detail in section 6. The structure shown has the great virtue of simplicity while not being too inefficient in terms of computer time. Indeed what it loses in possible extra run time it will probably make up for in gains of time which would otherwise be spent on de-bugging the program.

EXAMPLE

The second and more complicated problem of the steel mill is now presented. However some obvious simplification is now introduced. Firstly while it would have produced a satisfactory program if we had represented the state of the furnace by putting it in a 'ready' queue and an 'empty' queue these states are more conveniently represented by boolean attributes of the entity. Similarly limitations on the number of entities cooling, loading, unloading and being soaked simultaneously could have been represented by introducing the entities, moulds, cranes and soaking pits which would have to idle before these activities could start. However it is much simpler to use counters rather than entities and queues to check on resource availability in these cases. The structure of the activity is also simplified in that whereas above we have introduced a list called partners to each activity to hold the entities taking part this is clearly redundant when there is only one or even two entities taking part in the activity. In this case we can use a reference variable to store a reference to the entity taking part.

These simplifications do not alter the basic structure of the model. The program now follows.

PROGRAM 3

```
simulation  
begin  
  link class smelter;  
  begin  
    real load;  
    boolean empty, ready;  
  end *** smelter ***;  
  
  link class ingots(number,pourtime); integer number;  
    real pourtime;  
  begin  
    real procedure temp;  
      temp:= 500*exp(-(time-pourtime));  
  end *** ingots ***;
```

```

link class gang;
begin
  integer number;
end *** gang ***;

process class smelt;
begin
  if furnace.empty then
    begin
      furnace.empty:= false;
      hold(uniform(2,2,2,U1));
      furnace.ready:= true;
      furnace.load:= normal(1500,300,U2);
      activate main;
    end;
  end *** smelt ***;

  process class pour;
  begin
    integer number;
    if furnace.ready and setting < 3 then
      begin
        setting:= setting+1;
        furnace.ready:= false;
        number:= (if furnace.load >= 900 then 30 else
          furnace.load/30);
        hold(number*0.1+uniform(0.5,0.7,U3));
        new ingots (number,time). into(setq);
        furnace.load:= furnace.load-number*30;
        if furnace.load < 150 then
          begin
            furnace.load:= 0;
            furnace.empty:= true;
          end else furnace.ready:= true;
          activate main;
        end;
      end *** pour ***;

      process class set;
      begin
        ref(ingots) engaged;
        if !setq.empty then
          begin
            engaged:= setq.first;
            engaged.out;
            hold(2);
            engaged.into(stripq);
            setting:= setting-1;
            activate main;
          end;
        end *** set ***;
      end
    end
  end
end

```

```

process class strip;
begin
  ref(ingots) ing;
  ref(gang) gng;
  if ! (gangq.empty or stripq.empty) then
    begin
      ing:= stripq.first;
      ing.out;
      gng:= gangq.first;
      gng.out;
      gng.number:= ing.number;
      hold(ing.number+0.25);
      ing.into(loadq);
      gng.into(cleanq);
      activate main;
    end;
  end *** strip ***;

  process class clean;
  begin
    ref(gang) gng;
    if !cleanq.empty then
      begin
        gng:= cleanq.first;
        gng.out;
        hold(gng.number*0.02);
        gng.number:= 0;
        gng.into(gangq);
        activate main;
      end;
    end *** clean ***;

    process class load;
    begin
      ref(ingots) ing;
      if cranes > 0 and soaking < 20 and
        !loadq.empty then
        begin
          cranes:= cranes-1;
          soaking:= soaking+1;
          ing:= loadq.first;
          ing.out;
          hold(ing.number*0.01);
          cranes:= cranes+1;
          ing.into(soakq);
          activate main;
        end;
      end *** load ***;
    end
  end
end

```

```

process class soak;
begin
  ref(ingots) ing;

  real procedure soaktime(temp); real temp;
  begin
    real reheat;
    reheat:= (400-temp)/50;
    soaktime:= reheat+5;
  end;

  if  $\neg$ soakq.empty then
    begin
      ing:- soakq.first;
      ing.out;
      hold(soaktime(ing.temp));
      ing.into(rollq);
      activate main;
    end;
  end *** soak ***;

process class roll;
begin
  ref(ingots) ing;
  if  $\neg$ (rolling or rollq.empty or cranes = 0) then
    begin
      rolling:= true;
      ing:- rollq.first;
      ing.out;
      cranes:= cranes-1;
      hold(ing.number*0.1);
      ing.into(slabq);
      rolling:= false;
      cranes:= cranes+1;
      soaking:= soaking-1;
      activate main;
    end;
  end *** roll ***;

process class report;
begin
  if  $\neg$ slabq.empty then
    begin
      outint(slabq.first qua ingots.number,10);
      outtext("ingots produced at time");
      outfix(time,3,10);
      outtext("hours from start up");
      outimage;
      slabq.first.out;
      activate main;
    end;
  end *** report ***;

```

```

ref(smelter) furnace;
ref(process) act;
ref(head) setq,stripq,gangq,loadq,soakq,rollq,slabq;
integer I;
integer setting,soaking,cranes;
boolean rolling;
integer U,U1,U2,U3;
U := inint;
U1:= inint;
U2:= inint;
U3:= inint;
setq:- new head;
stripq:- new head;
gangq:- new head;
cleanq:- new head;
loadq:- new head;
soakq:- new head;
rollq:- new head;
slabq:- new head;
cranes:= 2;
furnace:- new smelter;
for I:= 1 step 1 until 3 do
  new gangq.into(gangq);
furnace.empty:= true;
while time < 250 do
  begin
    for act:- new smelt, new pour, new set,
      new strip, new clean, new load, new soak,
      new roll, new report do activate act;
    passivate;
  end;
  outtext("run ended at time");
  outfix(time,5,12); outimage;
end;

```

ALTERNATE MODEL STRUCTURES

IMPROVEMENTS TO ACTIVITY PROGRAM

In the preceding pages we have seen how a model may be set up and programmed using the activity approach. This approach is simple to specify and gives a clear and modular program. It is however inefficient in programming terms because whenever an activity ends every other activity is tested to see if it can start, even if it is not even remotely connected with the one just ended. Also in its SIMULA representation class objects are created in order to carry out the start phase many of which will fail their tests and no longer be required. This is expensive in terms of computer time both due to the unnecessary object creation and also to the subsequent garbage collection if the computer store becomes full of these terminated objects.

A simple improvement on this situation can be obtained by taking the tests out of the activity class specification and putting them in the main program. There is also an important extra gain by this device, for in this way all of the "Management control" of the model is brought to one place in the program. In situations where the reason for the simulation is to experiment with different priority systems or to examine the sensitivity of the model to adequate and timely control, then it is clearly advantageous to have those parts of the program which will be subject to change grouped together.

Program 4 shows the steel mill model programmed in this way. In this example these changes gave a factor of 3 improvement in running speed, compared with Program 3.

PROGRAM 4

```
simulation
begin
  link class smelter;
  begin
    real load;
    boolean empty, ready;
  end *** smelter ***;

  link class ingots(number, pourtime); integer number;
  real pourtime;
  begin
    real procedure temp;
    temp := 500 * exp(pourtime - time);
  end *** ingots ***;

  link class gang;
  begin
    integer number;
  end *** gang ***;

  process class smelt;
  begin
    furnace.empty := false;
    hold(uniform(2, 2.2, U1));
    furnace.ready := true;
    furnace.load := normal(1500, 300, U);
    activate main;
  end *** smelt ***;

  process class pour;
  begin
    integer number;
    setting := setting + 1;
    furnace.ready := false;
    number := (if furnace.load = 900 then 30
               else furnace.load / 30);
    hold(number * 0.1 + uniform(0.5, 0.7, U2));
    new ingots(number, time).into(setq);
    furnace.load := furnace.load - number * 30;
    if furnace.load < 150 then
      begin
        furnace.load := 0;
        furnace.empty := true;
      end else furnace.ready := true;
    activate main;
  end *** pour ***;
```

```

process class set;
begin
  ref(ingots) engaged;
  engaged:- setq.first;
  engaged.out;
  hold(2);
  engaged.into(stripq);
  setting:= setting-1;
  activate main;
end *** set ***;

process class strip;
begin
  ref(ingots) ing;
  ref(gang) gng;
  ing:- stripq.first;
  ing.out;
  gng:- gangq.first;
  gng.out;
  gng.number:= ing.number;
  hold(ing.number+0.25);
  ing.into(loadq);
  gng.into(cleanq);
  activate main;
end *** strip ***;

process class clean;
begin
  ref(gang) gng;
  gng:- cleanq.first;
  gng.out;
  hold(gng.number*0.02);
  gng.number:= 0;
  gng.into(gangq);
  activate main;
end *** clean ***;

process class load;
begin
  ref(ingots) ing;
  cranes:= cranes-1;
  soaking:= soaking+1;
  ing:- loadq.first;
  ing.out;
  hold(ing.number*0.01);
  cranes:= cranes+1;
  ing.into(soakq);
  activate main;
end *** load ***;

```

```

process class soak;
begin
  ref(ingots) ing;

  real procedure soaktime(temp); real temp;
  begin
    real reheat;
    reheat:= (400-temp)/50;
    soaktime:= reheat+5;
  end;

  ing:- soakq.first;
  ing.out;
  hold(soaktime(ing.temp));
  ing.into(rollq);
  activate main;
end *** soak ***;

process class roll;
begin
  ref(ingots) ing;
  rolling:= true;
  ing:- rollq.first;
  ing.out;
  cranes:= cranes-1;
  hold(ing.number*0.1);
  ing.into(slabq);
  rolling:= false;
  cranes:= cranes+1;
  soaking:= soaking-1;
  activate main;
end *** roll ***;

process class report;
begin
  outint(slabq.first qua ingots.number,10);
  outtest(" ingots produced at time");
  outfix(time,3,10);
  outtext(" hours from start up");
  outimage;
  slabq.first.out;
  activate main;
end *** report ***;

```

```

ref(smelter) furnace;
ref(head) setq,stripq,gangq,cleaq,loadq,soakq,
    rollq,slabq;
integer I;
integer setting,soaking,cranes;
boolean rolling;
integer U,U1,U2;

U := 12345;
U1:= 34567;
U2:= 56789;
setq:- new head;
stripq:- new head;
gangq:- new head;
cleaq:- new head;
loadq:- new head;
soakq:- new head;
rollq:- new head;
slabq:- new head;
cranes:= 2;
furnace:- new smelter;
for I:= 1 step 1 until 3 do
    new gang.into(gangq);
furnace.empty:= true;

while time < 250 do
begin
    if furnace.empty then activate new smelt;
    if furnace.ready and setting < 3 then activate
        new pour;
    if not setq.empty then activate new set;
    if not (gangq.empty or stripq.empty) then
        activate new strip;
    if not (cleaq.empty then activate new clean;
    if cranes > 0 and soaking < 20 and not loadq.empty
        then activate new load;
    if not soakq.empty then activate new soak;
    if not (rolling or rollq.empty or cranes = 0) then
        activate new roll;
    if not slabq.empty then activate new report;
passivate;
end;
outtext("run ended at time ");
outfix(time,5,12); outimage;
end;

```

RECYCLING

It should be noted that the situation described on page 28 where the start phase of an activity needs to be retested if it is successful does not need to be covered in this example. However if it were there the following test structure would suffice:

```

label: if (<activity condition>) then
begin
    activate new activity;
    goto label;
end;

```

EVENT MODELS

A completely different approach to simulation known as Event Models may also be taken. It has been used for example in the language SIMSCRIPT. This (as was noted on page 2) specifies the model control in terms of events when instantaneous changes take place. At each event a specific change in the model is specified and consequent later events are predicted.

Program 5 shows the steel mill example modelled in this mode. The resulting program is much the same in length and speed of operation as the activity model. It does however have the advantage that it allows the programmer to make use of his knowledge of the model's interactions. For example in the steel mill we know full well that if pouring has just finished it is only worth testing to see if we can either start a new pour or replenish the furnace. We also know that we must start a setting. In the activity model, we would also wastefully test to see if we could start all the other activities.

The consequent improvement in efficiency will not show up in a simple model like our example, but may prove important in larger more complex ones.

There are however two major disadvantages in the event approach. The first is that with each event we must specify all other certain and possible consequence events.

In a complex situation this may not be at all easy. In the activity approach we let the system do it (albeit inefficiently) for us. The second disadvantage is that the event equivalent to the starting of an activity may be conditionally consequent on several other events. In this case the condition test program will be repeated in each of these events programs with usage of core space and also more important a spreading of the control over the entire program. In situations where it is the control which is under examination it will be therefore that much harder to effect changes to the program.

PROGRAM 5

```
simulation
begin
  link class smelter;
  begin
    integer number;
    real load;
    boolean empty, ready ;
  end *** smelter ***;

  link class ingots(number, pourtime); integer
  number; real pourtime;
  begin
    real procedure soaktime;
    begin
      real temp;
      temp:= 500*exp(pourtime-time);
      soaktime:= (400-temp)/50+5;
    end *** soaktime ***;
  end *** ingots ***;

  link class gang;
  begin
    integer number;
    ref(ingots) ingot;
  end *** gang ***;
```

```
process class event; ;

event class loadsmelter;
begin
  furnace.empty:= false;
  furnace.load:= normal(1500,300,U);
  activate new smelterready delay uniform (2,2.2,U1);
end *** loadsmelter ***;

event class smelterready;
begin
  furnace.ready:= true;
  if setting < 3 then activate new startpour
  after current;
end *** smelterready ***;

event class startpour;
begin
  setting:= setting+1;
  furnace.ready:= false;
  furnace.number:= if furnace.load >= 900
    then 30 else furnace.load/30;
  activate new endpour delay furnace.number*0.1+
    uniform(0.5,0.7,U2);
end *** startpour ***;

event class endpour;
begin
  furnace.load:= furnace.load-furnace.number*30;
  new ingots(furnace.number,time).into(setq);
  activate new endsetting delay 2;
  if furnace.load < 150 then activate new loadsmelter
  after current else if setting < 3 then activate new
  startpour after current else furnace.ready:= true;
end *** endpour ***;

event class endsetting;
begin
  setting:= setting-1;
  if furnace.ready then activate new startpour;
  if 7 gangq.empty then activate new strip(setq.first)
  after current else setq.first.into(stripq);
end *** endsetting ***;
```

```

event class strip(ing); ref(ingots) ing;
begin
  ref(gang) strippers;
  strippers:- gang.first;
  strippers.ingot :- ing;
  ing.out;
  strippers.out;
  activate new endstrip(strippers) delay ing.
    number+0.25;
end *** strip ***;

event class endstrip(strippers); ref(gang) strippers;
begin
  activate new endclean(strippers)
    delay strippers.ingot.number*0.02;
  if cranes > 0 and soaking < 20
    then activate new load(strippers.ingot )
    else strippers.ingot .into(loadq);
  end *** endstrip ***;

event class endclean(strippers): ref(gang) strippers;
begin
  strippers.into(gangq);
  if ¬stripq.empty then activate new strip(stripq.first);
end *** endclean ***;

event class load(ing); ref(ingots) ing;
begin
  cranes:= cranes-1;
  ing.out;
  soaking:= soaking+1;
  activate new endload(ing) delay ing.number*0.01;
end *** load ***;

event class endload(ing); ref(ingots) ing;
begin
  cranes:= cranes+1;
  activate new endsoaking(ing) delay ing.soaktime;
  if ¬loadq.empty and soaking < 20
    then activate new load(loadq.first)
    else if ¬rollq.empty and ¬rolling
      then activate new roll(rollq.first);
  end *** endload ***;

```

```

event class endsoaking(ing); ref(ingots) ing;
begin
  if cranes > 0 and ¬rolling then activate new roll(ing)
    else ing.into(rollq);
  end *** endsoaking ***;

event class roll(ing); ref(ingots) ing;
begin
  cranes:= cranes-1;
  ing.out;
  activate new product(ing) delay ing.number*0.1;
end *** roll ***;

event class product(ing); ref(ingots) ing;
begin
  cranes:= cranes +1;
  soaking:= soaking-1;
  rolling:= false;
  if ¬loadq.empty then activate new load(loadq.first)
    else if ¬rollq.empty then activate new roll
      (rollq.first);
  outtext("at time ");
  outfix(time,2,8);
  outtext(" we made");
  outint(ing.number,5);
  outtext(" ingots");
  outimage;
end *** product ***;

integer cranes,setting,number,soaking,U,U1,U2,I;
boolean rolling;
ref(smelter) furnace;
ref(head) stripq,gangq,loadq,rollq,setq;
stripq:- new head;
gangq:- new head;
loadq:- new head;
rollq:- new head;
setq:- new head;
furnace:- new smelter;
for I:= 1 step 1 until 3 do new gang.into(gangq);
U := 12345;
U1:= 34567;
U2:= 56789;
cranes:= 2;
activate new loadsmelter;
hold(250);
end;

```

PROCESS MODELS

A further way of specifying simulation models and indeed that which has been traditionally used with SIMULA, is the process approach. In this we consider a whole series of "activities" of an "entity" to form a process and we program the interactions between such processes explicitly. This is taking the intervention of the programmer a step further than in the event approach, for now, not only is he programming the control of the model but he is also explicitly programming the physical interactions, whereas in both the activity and the event approaches the concept of the co-operation of entities to engage in tasks was central to the specification and dealt with physical interactions. However we obtain a shorter program and one which was significantly faster. Program 6 gives the steel mill in process mode.

PROGRAM 6

```
simulation
begin
  process class smelter;
  begin
    real load;
  smelt:
    hold(uniform(2,2.2,U1));
    load:= normal(1500,300,U);
  pour:
    if setting >= 3 then passivate;
    setting:= setting+1;
    number:= if load > 900 then 30 else load/30;
    hold(number*0.1+uniform(0.5,0.7,U2));
    activate new ingots(number,time);
    load:= load-number*30;
    if load >= 150 then goto pour else goto smelt;
  end *** smelter ***;
```

```
process class ingots(number,pourtime); integer number;
  real pourtime;
begin
  ref(gang)handlers;
  real procedure soaktime;
  begin
    real temp;
    temp:= 500*exp(pourtime-time);
    soaktime:= (400-temp)/50+5;
  end *** soaktime ***;

set:
  hold(2);
  setting:= setting-1;
  activate furnace;
strip:
  if gangq.empty then wait(stripq);
  out;
  handlers:= gangq.first;
  handlers.out;
  hold(number+0.25);
  activate new clean(handlers,number);
load:
  while cranes = 0 or soaking = 20 do wait(loadq);
  cranes:= cranes-1;
  soaking:= soaking+1;
  hold(number*0.01);
  cranes:= cranes+1;
  activate loadq.first;
  activate rollq.first;
soak:
  hold(soaktime);
roll:
  while rolling or cranes = 0 do wait(rollq);
  out;
  rolling:= true;
  cranes:= cranes-1;
  hold(number*0.1);
  soaking:= soaking-1;
  rolling:= false;
  cranes:= cranes+1;
  activate loadq.first;
  activate rollq.first;
  report(time,number);
end *** ingots ***;
```

```

process class clean(g,n); ref(gang) g; integer n;
begin
  hold(n*0.02);
  g.into(gangq);
  activate stripq.first;
end *** clean ***;

link class gang; ;

procedure report(time,number); real time;
  integer number;
begin
  outtext("time is ");
  outfix(time,2,8);
  outint(number,5);
  outtext(" ingots just rolled");
  outimage;
end *** report ***;

ref(head) stripq,gangq,loadq,rollq;
integer setting,number,soaking,cranes,U,U1,U2,I;
boolean rolling;
ref(smelter) furnace;

U := 12345;
U1:= 34567;
U2:= 56789;
stripq:- new head;
gangq:- new head;
loadq:- new head;
rollq:- new head;
furnace:- new smelter;
for I:= 1 step 1 until 3 do new gang.into(gangq);
cranes:= 2;
activate furnace;
hold(250);
end;

```

COMPARISON OF METHODS

Table 1 shows how the four programs compare in computer terms.

Approach	Program	Number of Cards	CPU Time	Core Used
Activity (tests in objects)	3	199	0.866	7843
Activity (tests together)	4	179	0.292	1579
Event	5	165	0.238	1953
Process	6	99	0.096	856

TABLE 1.

In general the three methods may be characterised as follows.

<u>Activity</u>	Easy to specify. Easy to explain (hand simulate). Groups control statements together. Inefficient.
<u>Event</u>	Not so easy to specify in complex situations. Easy to explain once specified. Control statements scattered and so hard to modify. More efficient but larger in core size than activity model.
<u>Process</u>	Hard to specify. Hard to explain. Hard to modify. Efficient and compact.

CHOICE OF METHODS

The choice between these alternative methods is clearly, to some extent, dependent on the model being created. An attempt to give guidelines to a choice follows.

- | | |
|-----------------|--|
| <u>Activity</u> | <p>Good for models which are</p> <ul style="list-style-type: none"> - complex or ill-defined - created to study changes in control - where the real plant is naturally thought of in terms of jobs to be done. <p>● workshops, warehouses, and ports are classic examples</p> |
| <u>Event</u> | <p>Good for models which are</p> <ul style="list-style-type: none"> - well defined in real life - where the real situation is defined in terms of events or break points. <p>● air traffic control, military models, communication and computer systems are examples.</p> |
| <u>Process</u> | <p>Should be used when</p> <ul style="list-style-type: none"> - efficiency is required - a well-defined simulation or previously specified model (in activity or event terms) exists. - The real plant is a process - <p>● Mass production plants, refineries and continuous process plants are examples.</p> |

RANDOM SAMPLING AND STATISTICAL OUTPUT

The design of experiments both in physical and simulation studies is outside the scope of this introduction and we will merely list the facilities provided by SIMULA.

RANDOM SAMPLING

The random sampling procedures in SIMULA all make use of an external integer variable (called U below) to store the next value in a random number stream. When any of them is called it makes use of the current value of U and then changes it using a congruence method. In this way, by using a number of integers (U1, U2, U3...say), we have use of an unlimited number of random number streams. In addition, if a negative value is chosen first for a number stream, a series of antithetic values to the corresponding stream started with a positive value will be formed. The procedures are:

Boolean Procedure Draw (A,U);
gives the value false or true such that
 $\text{Prob}(\text{true}) = A$

Integer procedure Randint (A,B,U);
gives the integer value A or A+1 or A+2 or ... or B with equal probability.

Real procedure Uniform (A,B,U);
gives a value randomly sampled from a uniform distribution between A and B.

Real procedure Normal (A,B,U);
gives a value randomly sampled from a normal distribution mean A, standard deviation B.

Real procedure Negexp. (A,U);
gives a quantity randomly sampled from a negative exponential distribution mean A.

Integer procedure Poission (A,U);
gives a random drawing from the
Poission distribution with mean A.

Real procedure Erlang (A,B,U);
gives a drawing from the Erlang
distribution with mean $1/A$ and
standard deviation $1/(A*\text{Sqrt}(B))$;

STATISTICAL PROCEDURE

Since it is easy for the user to
introduce his own statistical collection
procedures in SIMULA only one
general one is provided. It is:

Procedure Accum (A,B,C,D)

- requires as parameters real quantities
A,B,C,D.
- accumulates a time integral
value A as follows:

A: = A+C*(time-B);

B: = time;

C: = C+D;