1981 Winter Simulation Conference Proceedings
T.I. Ören, C.M. Delfosse, C.M. Shub (Eds.)

INTRODUCTION TO DEMOS

Graham Birtwistle
Computer Science Department
University of Calgary, Alberta, Canada T2N 1N4

ABSTRACT

Demos [1,2] is yet another discrete event simulation language hosted in Simula. It was released in 1979 and is now running on IBM, DEC, UNIVAC, and CDC hardwares amongst others. The paper contains a short introduction to Simula's object and context features; an explanation of the process approach to simulation; a brief comparison of Simula and GPSS; and finally, the main features of Demos are presented via an example.

MAIN FEATURES OF SIMULA

This section is a short introduction to the highlights of Simula. Fuller accounts of Simula are found in Birtwistle [3], Dahl [4], and Franta [5]. Simula is an extension to ALGOL 60 and includes nearly all that language as a subset. The central new ideas in Simula are those of the OBJECT and of the CONTEXT. An object is used in Simula to mirror the characteristics and behaviour of a major component in the system under description, e.g. a boat in a harbour simulation or a furnace in a steel mill simulation. A context is a pre-compiled collection of object, procedure, and data definitions, and statements common to one particular topic. For example, HARBOUR context may contain definitions for boats, cranes, tugs, the tide, etc., and a TRAFFIC context may contain definitions for cars, trucks, etc.

Objects

Objects are used in Simula programs to mirror major components in the actual system under investigation, one for one. As an example, consider a steel mill simulation involving furnaces, ingots, etc. Each actual furnace may be represented in the Simula program by a corresponding furnace object. If so, the furnace object will have to reflect all those features of an actual furnace deemed relevant in the model; not just its physical characteristics such as capacity, current load, power requirements, etc., but also the actions it carries out as it runs through its schedule of operations.

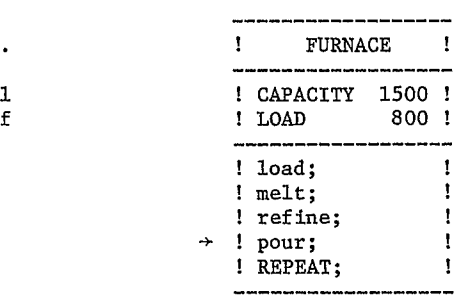
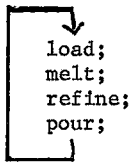


Figure 1. A furnace object.

Figure 1 introduces our standard way of depicting objects - as rectangular boxes divided into three levels. The top level gives the class of the object (here FURNACE), the middle level gives the attributes (data characteristics and local procedures) of the object (here CAPACITY and LOAD with current values of 1500 and 800 respectively, perhaps in tons), and the bottom level gives the action sequence of the furnace object. Here they are informally shown as the cycle

81CH1709-5/81/0000-0559$00.75 © 1981 IEEE



Where it sheds light on the situation, the current action of an object will be marked with an arrow, thus '→'. This marker (and there is one for each object) is called its LOCAL SEQUENCE CONTROL, or LSC for short. The furnace object in figure 2 represents an actual furnace which is pouring. Figure 2 shows how the real world situation involving two furnaces (one loading and one pouring) would be mapped into a Simula program. Notice the positions of the LSCs of the two furnace objects.

```
------------------      ------------------
!    FURNACE     !      !    FURNACE     !
------------------      ------------------
! CAPACITY  1500 !      ! CAPACITY  1200 !
! LOAD       800 !      ! LOAD        0 !
------------------      ------------------
    ! load;          !  → ! load;          !
    ! melt;          !    ! melt;          !
    ! refine;        !    ! refine;        !
 →  ! pour;          !    ! pour;          !
    ! repeat;        !    ! repeat;        !
    ------------------      ------------------
```

Figure 2. Objects representing the two furnaces.

Now although their individual data values are different, and they are currently performing different actions, the two furnace objects have exactly the same layout of attributes and the same action sequence. The objects are said to be 'of the same class' and are defined by a single CLASS DECLARATION. Here it is in Simula (partly informally):

```
CLASS FURNACE(CAPACITY); INTEGER CAPACITY;
BEGIN
  INTEGER LOAD;
  load;
  melt;
  refine;
  pour;
  REPEAT;
END***FURNACE***;
```

N.B. Program segments in this paper use a blending of Simula and English as it suits. Upper case letters and punctuation are formal language elements which are part of Simula itself. They have precisely defined meanings and are used strictly in accordance with the rules of Simula. When we wish to be informal, we will use lower case letters. Above, we have merely sketched the action sequence of CLASS FURNACE as its precise formulation in Simula is of no immediate relevance.

We need a class declaration for each type of object appearing in a Simula program. Each declaration can be thought of as a template from which objects of the same general layout can be created as and when required. Several objects of the same class may be in existence and operating at the same time. To create a new furnace object in a Simula program, we execute the command NEW FURNACE. For example, the code below creates two furnace objects, named F1 and F2, and initialises their capacities to 1500 units and 1200 units respectively.

```
F1 :- NEW FURNACE (1500);
F2 :- NEW FURNACE (1200);
```

(':-' is read 'denotes'). F1 and F2 are Simula variables of a type not found in Algol 60. They are REFERENCE VARIABLES of type REF(FURNACE) (read as 'REF to FURNACE') and are declared so

```
REF(FURNACE)F1, F2;
```

F1 and F2 are variables capable of referencing furnace objects or NONE (no object at all). NONE is the initial value of all reference variables.

When the program action is inside a particular object, the object will refer to its own attributes directly, as CAPACITY or LOAD. For example, suppose that a batch of ingots requires 200 units of refined metal and that part batches are not to be poured. More formal Simula code for "pour" would be:

```
WHILE LOAD >= 200 DO
BEGIN
  pour another 200 ton set of ingots;
  LOAD := LOAD - 200;
END;
```

Each furnace object loops while its own LOAD >= 200.

When we access the current data values of the attributes of objects from without (for example, from the main program), we have to specify the particular object as well as the attribute we are after. Simula uses the DOT NOTATION, typically

```
<object>.<attribute>
```

e.g. F1.CAPACITY or F2.LOAD. For example, to load F2 with 800 tons of scrap we could write

```
F1.LOAD := 800;
```

A run time error results if the value of the object reference in a remote access is NONE (asking for an attribute of a non-existant object is illegal). This will halt program execution with a suitable message.

The remote access problem has its analogs in every day life. For example, my phone number on campus is 6055. Off campus but in Calgary, it is 284 6055. Out of Calgary, but within North America it is 403 284 6055. When outside of North America, omitting the 403 prefix gives quite a different result - indeed, the probability is that

the line does not even exist. (Compare asking for LOAD within the main program.)

Contexts

A Simula program must contain class declarations for all the objects it uses. For example, a steel mill simulation involving furnaces, ingots, etc. could have the format

```
BEGIN
     CLASS FURNACE.....;
     CLASS INGOT.......;
     REF(FURNACE)F1,F2;
     F1 :- NEW FURNACE;
     F2 :- NEW FURNACE;
     actions involving these declarations;
END;
```

When working on a particular problem, it is clear that some basic declarations will be useful through several experiments. It is tiresome and error prone to prepare much the same program each time. In Simula, the basic inter-related definitions and initialising actions can be collected together in a CONTEXT. In this case, we call the context STEEL and define it by:

```
CLASS STEEL;
BEGIN
     CLASS FURNACE.....;
     CLASS INGOT.......;
     REF(FURNACE)F1,F2;
     F1 :- NEW FURNACE;
     F2 :- NEW FURNACE;
END***STEEL***;
```

Normally STEEL would be compiled separately and the relocatable object code retained in a library. Users with an interest in the area and access to the library call up the compiled context code by an EXTERNAL DECLARATION. For example, the program below

```
BEGIN
  EXTERNAL CLASS STEEL;
  STEEL
    BEGIN
      user code;
    END;
END;
```

All the concepts inside STEEL (namely FURNACE, INGOT, F1, F2, etc.) are directly available in the user-defined block prefixed by STEEL. If you wish to protect or hide certain quantities from a user (e.g. SEED), there are mechanisms defined in Simula to do just that (Palme [6]). Importantly using contexts dramatically reduces compile times. The cost of compiling a 100 line program backed up by a 3000 line context is practically the same as that for compiling a 100 line program.

Prefixing

Whilst a steel mill simulation is running, the major components (furnaces, ingots) will compete with each other for resources, and we need a means of queueing components that are temporarily blocked. We could start from scratch and define our own queueing mechanism, but Simula contains a queueing context called SIMSET which is sufficient for the sequel. Each SIMSET list is represented by a HEAD object which has pointers to its FIRST and LAST members. The class declaration of objects that may be queued are defined, not as (say)

```
CLASS FURNACE.....;
```

with an empty prefix, but as

```
LINK CLASS FURNACE.....;
```

The LINK prefix augments the furnace part with queue membership attributes defined in CLASS LINK. These are pointers to its SUCcessor or PREDecessor in the list, and routines INTO, FOLLOW, PRECEDE for placing a member in a list, and OUT for removing a list member. Functional (but not strictly accurate) skeleton code for the predefined class SIMSET is:

```
CLASS SIMSET;
BEGIN
     CLASS HEAD;
     BEGIN
          REF(LINK)FIRST, LAST;
          ...................
     END***HEAD***;

     CLASS LINK;
     BEGIN
          REF(LINK)SUC, PRED;
          PROCEDURE INTO(H); REF(HEAD)H;...;
          PROCEDURE OUT;.....................;
          PROCEDURE FOLLOW(X);..............;
          PROCEDURE PRECEDE(X);.............;
     END***LINK***;
END***SIMSET***;
```

To make these list processing concepts available to steel mill programs, we use prefixing (yes, contexts may be prefixed by other contexts) and simply alter the heading of CLASS STEEL to read:

```
SIMSET CLASS STEEL;
```

Note that SIMSET is system defined and does not need an external declaration.

THE PROCESS APPROACH TO MODELLING

In this section, we use a cut down version of a steel mill model which is fully developed in the last section of this paper. Suppose the steel mill has two furnaces which work fairly independently. The furnaces are electric powered and have the operation cycle:

```
load; melt; refine; pour;
```

Each furnace is first filled with scrap metal (we assume an inexhaustible supply of scrap is available). The scrap is then heated until molten, which required 3 units of electric power. A maximum of four units of electric power is available to be shared between the furnaces; this effectively prevents two furnaces from 'melting' at the same time. Once the scrap is molten, two of these three units of power are freed, but one is kept for the remainder of a furnace's work cycle

to prevent its contents from setting solid before they have been poured out.

The molten metal is then refined, and finally the contents of the furnace are poured into a batch of moulds. Each batch of moulds is kept together as a unit, and is shunted around the mill floor on its own railway bogie. In this version, we assume that a batch is always available. However, pouring the molten metal into the moulds is a restricted process. Pouring requires use of a crane, and since there is only one crane available for this purpose, only one furnace may be pouring at a time.

We first go through the operation of the mill using a few 'easy' numbers. Let the furnaces be called 'F1' and 'F2'. We assume that loading and melting take 2 time units each, refining and pouring take 1 time unit each, and suppose that F1 starts its first cycle at time 0, and that F2 starts its first cycle at time 1.

Time	Furnace	Event taking place	Next event
0	F1	start loading	2
1	F2	start loading	3
2	F1	request 3 of power	
	F1	seize 3 of power	
	F1	start melting	4
3	F2	request 3 of power	
4	F1	release 2 to power	
	F1	start refining	5
	F2	seize 3 of power	
	F2	start melting	6
5	F1	request 1 of crane	
	F1	seize 1 of crane	
	F1	start pouring	
6	F2	release 2 to power	
	F2	start refining	7
	F1	release 1 to crane	
	F1	release 1 to power	
	F1	start loading	8
7	F2	request 1 of crane	
	F2	seize 1 of crane	
	F2	start pouring	8
8	F1	request 3 of power	
	F1	seize 3 of power	
	F1	start melting	10
	F2	release 1 to crane	
	F2	release 1 to power	
	F2	start loading	10

The state of the model changes only at certain critical times and only these times have been recorded. The trace records what is considered to be the essential behaviour of the system as a time ordered sequence of events, and we accept any program that can reproduce this trace as a 'correct' model. The first three columns of the trace give the 'when', the 'who', and the 'what' of each event. The 'who' of each event is at all times a furnace, and so the behaviour of the complete system can be rephrased in terms of the actions and interactions of F1 and F2. The process style of modelling follows this lead and splits the trace narrative into separate columns, one for each furnace, as below. Notice that each and every event appears once under the appropriate furnace name, and that no events have been omitted (we follow through only the first complete cycle of each furnace - later cycles follow the established pattern).

Event sequence	Time	
	F1	F2
start loading	0	1
request 3 of power	2	3
seize 3 of power	2	4
start melting	2	4
release 2 to power	4	6
start refining	4	6
request 1 of crane	5	7
seize 1 of crane	5	7
start pouring	5	7
release 1 to crane	6	8
release 1 to power	6	7

The table above is just a rehash of the trace in which we have followed through the actions of each furnace as an individual. Importantly, the action sequence for each furnace may be framed in exactly the same way. In Simula, this clearly invites the declaration of a CLASS FURNACE (informally):

```
CLASS FURNACE;
BEGIN
  load;
  request 3 of power;
  seize 3 of power;
  melt;
  release 2 to power;
  refine;
  request 1 of crane;
  seize 1 of crane;
  pour;
  release 1 to crane;
  release 1 to power;
  repeat;
END***FURNACE***;
```

from which two furnace objects will be created by executing NEW FURNACE at appropriate times in the main program.

The life cycle of a furnace is seen to be a sequence of phases (ACTIVITIES in discrete event parlance) each of which fits into the general pattern:

```
acquire extra resources;
carry out the task;
release unwanted resources;
```

Before we give Simula code for this example, we need to take a closer look at the tools provided for coding this pattern in the standard Simula context SIMSET CLASS SIMULATION.

CLASS SIMULATION

Simulation programs written in raw Simula are prefixed by the standard context SIMSET CLASS SIMULATION which extends Simula with a few primitives for discrete event simulation modelling. These are sufficient to write models in any of

the usual styles. It is interesting to write models in raw Simula using these primitives ONCE for it shows up in all its gory detail the mass of testing and reactivating that has to be done in order to get model components correctly synchronised. In practice, using the primitives supplied by SIMULATION turns out to be both tedious and error prone. More powerful tools need to be developed.

The skeleton of CLASS SIMULATION is:

```
SIMSET CLASS SIMULATION;
BEGIN
  LINK CLASS PROCESS.....................;
  PROCEDURE HOLD(T); REAL T;.....;
  PROCEDURE ACTIVATE.....................;
  PROCEDURE WAIT(Q); REF(HEAD) Q;.....;
  PROCEDURE PASSIVATE;...................;
  REF(PROCESS)CURRENT;...................;
  REAL PROCEDURE TIME;...................;

ACTIONS:
  set up event list at time zero;
END***SIMULATION***;
```

SIMULATION is prefixed by SIMSET and thus inherits the concepts of two-way lists. The class body actions set up the future events list and set the simulation clock to zero.

The event list is usually implemented as a tree, but its behaviour is easier to explain if it is depicted as a time-ranked queue. Every object which undertakes time consuming tasks has to be scheduled in event list and its class declaration must be prefixed by PROCESS (hence it will be PROCESS CLASS FURNACE). At any time in a simulation run, several processes are carrying out tasks. They are posted in the event list for the simulated time at which their current phase is due to end, and are ranked according to this time. The process with the smallest event time lies at the front. It is called CURRENT. The scheduling mechanisms are so framed that it is always the actions of CURRENT that are being carried out, and the simulation time is taken to be the event time of CURRENT. The actions of all other processes in the event list are suspended. The LSC of each is poised on the statement it is due to execute when it next reaches the front of the event list and becomes the current process. When it becomes CURRENT, the simulation clock time is stepped up to its event time.

Resources are usually represented by integers in Simula programs, e.g.

```
INTEGER CRANE, POWER;
```

initialised by

```
CRANE := 1;  POWER := 4;
```

The most common scheduling routines are:

ACTIVATE - used to awaken a blocked process,
HOLD(t) - used to represent the duration of an activity,
PASSIVATE - used to remove CURRENT from the event list, and
WAIT(q) - used to remove CURRENT from the event list and block it in a specified queue. There it will remain until awakened and able to pick up the resources it requires.

The typical activity uses three of these routines. We illustrate the idea using 'pour'.

```
ACQUIRE:
     (1)  IF CRANE < 1 THEN
     (2)  BEGIN
     (3)    WAIT(POURQ);
     (4)    OUT;
     (5)  END;
     (6)  CRANE := CRANE - 1;
HOLD:
     (7)  HOLD(1.0);
RELEASE:
     (8)  CRANE := CRANE + 1;
     (9)  POWER := POWER + 1;
    (10)  IF CRANE >= 1 THEN ACTIVATE POURQ.FIRST;
    (11)  IF POWER >= 3 THEN ACTIVATE MELTQ.FIRST;
```

ACQUIRE (lines 1-6): In raw Simula, it is convenient to maintain a queue (here REF(HEAD) POURQ;) in front of each activity where processes can await the availability of the required resources. A process wishing to pour, first tests the availability of the resources it wants. If they are all free, it seizes them at once. If they are not all free, it removes itself from the event list and waits passively in a specified queue by executing WAIT(some queue), as in line 3. It is the responsibility of a process releasing the crane to awaken this dormant process. Once awakened, it takes itself out of its queue (by OUT;). Thus, whether delayed or not, once the request(s) have been granted, the process continues on its way by seizing the resources it needs, here CRANE := CRANE - 1.

HOLD (line 7): The furnace then advances its event time by the time required to carry out its CURRENT task, here by HOLD(1.0). The call on HOLD reschedules CURRENT FURTHER down the event list. When it becomes CURRENT again, one simulated time unit will have elapsed.

RELEASE (lines 8-11): The process now returns the resources it no longer needs back to the system pool

```
CRANE := CRANE + 1;
POWER := POWER + 1;
```

and awakens sleeping processes who can now go by

```
IF CRANE >= 1 THEN ACTIVATE POURQ.FIRST;
IF POWER >= 3 THEN ACTIVATE MELTQ.FIRST;
```

N.B. ACTIVATE has no effect if the queue is empty (FIRST == NONE). Notice that when the user decides the order in which queues are tested, he is in effect assigning priorities to these queues. If the blocked processes waiting in these queues are competing for common resources, one must be very careful and consistent with this ordering.

The coding for the remaining activites follows in

the same vein. We now give a complete program for the mill in raw Simula.

```
SIMULATION
BEGIN
  INTEGER CRANE, POWER;
  REF(HEAD)POURQ, MELTQ;
  REF(FURNACE)F1, F2;

  PROCESS CLASS FURNACE;
  BEGIN
  LOAD:
    HOLD(2.0);
  MELT:
    IF POWER < 3 THEN
    BEGIN
      WAIT(MELTQ);
      OUT;
    END;
    POWER := POWER - 3;
    HOLD(2.0);
    POWER := POWER + 2;
    IF POWER >= 3 THEN ACTIVATE MELTQ.FIRST;
  REFINE:
    HOLD(1.0);
  POUR:
    IF CRANE < 1 THEN
    BEGIN
      WAIT(POURQ);
      OUT;
    END;
    CRANE := CRANE - 1;
    HOLD(1.0);
    CRANE := CRANE + 1;
    POWER := POWER + 1;
    IF CRANE >= 1 THEN ACTIVATE POURQ.FIRST;
    IF POWER >= 3 THEN ACTIVATE MELTQ.FIRST;
    GOTO LOAD;
  END***FURNACE***;

  CRANE := 1;
  POWER := 4;
  MELTQ :- NEW HEAD;
  POURQ :- NEW HEAD;
  F1    :- NEW FURNACE;
  F2    :- NEW FURNACE;
  ACTIVATE F1 AT 0.0;
  ACTIVATE F2 AT 1.0;
  HOLD(200.0)
END;
```

This style of coding is very efficient, mainly because each process decides what to do next itself and knows where to look for blocked processes rather than relying upon a central intelligence which looks around the whole model polling everyone when deciding who should do what next. It is also very convenient in that one can calculate such data as process-through-times (or average cycle times) very easily.

However, even without data collection or reporting statements, the program is long winded and the style prone to error because the overall model logic gets swamped in a welter of detail. Notice too the danger in the separation of a request for a resource and its acquisition. It is all too easy to request a resource and forget to acquire it, or acquire a different size chunk. There is a similar danger in the separation of the release and awaken operations.

But take heart, you were never intended to write simulations in raw Simula! Simula invites you to explore an area, decide which features are generally useful, write them up as a context, and then use this context. Before deciding on what to include in Demos, I wrote 50-60 non-trivial programs in various styles and also examined the literature for complete models in ECSL, SIMSCRIPT, GPSS and SIMULA. GPSS gave most pointers on what should be included in Demos. In the next section, we have a look at a mini-GPSS context in Simula - its strengths and shortcomings.

GPSS AND THE TRANSACTION STYLE

GPSS (see Schriber [7]) is one of the oldest discrete event modelling languages, dating back to the early 1960's. GPSS uses EXACTLY the same approach to modelling as Simula - a Simula process is called a TRANSACTION in GPSS.

GPSS does not pretend to be a general purpose programming language. It is a dedicated simulation language supporting one specific approach to model building and is implemented as a closed package embracing this viewpoint. Since all GPSS models are written in the same style, the same needs keep cropping up. GPSS provides many building blocks, each one catering for a specific need. These are an enormous bonus because the GPSS simulator itself accomplishes many of the tasks which fall on the programmer if he is using raw Simula. For example, blocked transactions are automatically woken up when the resource they need is released to the system pool. Also, GPSS collects data describing model behaviour most unobtrusively, and automatically prints out summaries of this data at the end of each run. The model builder need not supply computational statements either for collecting or for summarising this data, nor provide statements indicating how it should be displayed. From a beginner's point of view, this is very desirable as it takes away the need to learn about output formatting and enables the new user to straightaway focus his attention, where it belongs, squarely on the model.

GPSS provides two built-in resource types which are adequate for many queueing network problems - the FACILITY and the STORAGE. A FACILITY is a resource of size 1 which can be seized and released. E.G. GPSS code to use the crane in our mill model for one time unit is:

```
SEIZE      CRANE
ADVANCE    1
RELEASE    CRANE
```

POWER in our mill model would be modelled by a STORAGE: a resource of arbitrary positive size which can be entered and left in chunks. GPSS code to declare POWER as a storage of limit 4, then use 3 chunks for 2 time units would be

```
STORAGE    POWER 4  {declaration of POWER
                       with limit 4}
ENTER      POWER, 3
ADVANCE    2
LEAVE      POWER, 2
```

Note that while facilities are implicitly defined in GPSS (the translator can work out that CRANE is a facility by its very usage), storages must be defined explicitly as their capacities are arbitrary.

The GPSS code for the activity 'pour' is simply:

```
SEIZE       CRANE
ADVANCE     1
RELEASE     CRANE
LEAVE       POWER, 1
```

A GPSS version of our simply furnace model is:

```
        STORAGE     POWER, 4

        GENERATE    1, 0, 2
L:      ADVANCE     2
        ENTER       POWER, 3
        ADVANCE     2
        LEAVE       POWER, 2
        ADVANCE     1
        SEIZE       CRANE
        ADVANCE     1
        RELEASE     CRANE
        LEAVE       POWER, 1
        TRANSFER    L
```

Notice that instead of waiting for all the resources it needs to be available before starting up the next activity, a GPSS transaction seizes them one at a time. A very strong plus for the GPSS code is that seize and enter are indivisible request and take operations; and release and leave are indivisible return and awaken operations. There is thus no chance of making a slip and testing for a resource and then taking either none of it or a different amount; nor can one forget to increment a resource and omit the awaken test.

Another bonus is that code for SEIZE, RELEASE, ENTER and LEAVE is extended to take care of maintaining statistics on device usage, average queue length, etc. and thus data collection can be done unobtrusively. One of the shocking things about Simula programs is that 50-75% of the code is concerned with data collection and reporting statements, and the program structure is buried under a welter of i/o and update statements.

Implementation in Simula

A skeleton Simula implementation of GPSS is given below. The idea is taken from a paper by Vaucher [8], which is well worth locating.

```
SIMULATION CLASS GPSS;
BEGIN
  REF(HEAD)FACILITY_Q, STORAGE_Q;

  PROCESS CLASS TRANSACTION;
  BEGIN
    INTEGER PRIORITY;
    PROCEDURE P_INTO(Q); REF(QUEUE)Q;
    BEGIN
      REF(TRANSACTION)E;
      IF Q == NONE THEN warning ELSE
      BEGIN
        E :- Q.LAST;
        IF E == NONE THEN INTO(Q) ELSE
        IF PRIORITY > E.PRIORITY THEN
            FOLLOW(E) ELSE
        BEGIN
          E :- Q.FIRST;
          WHILE PRIORITY >= E.PRIORITY DO
            E :- E.SUC;
          PRECEDE(E)
        END;
      END;
    END***PRIORITY INTO***;
  END***TRANSACTION***;

  LINK CLASS FACILITY;
  BEGIN
    REF(HEAD)Q;
    INTEGER FREE;

    PROCEDURE SEIZE;
    BEGIN
      REF(TRANSACTION)T;
      T :- CURRENT;
      T.P_INTO(Q);
      IF FREE = 0 THEN PASSIVATE;
      T.OUT;
      FREE := 0;
    END***SEIZE***;

    PROCEDURE RELEASE;
    BEGIN
      FREE := 1;
      ACTIVATE Q.FIRST;
    END***RELEASE***;

    INITIALISE:
    FREE := 1;
    INTO(FACILITY_Q);
    Q :- NEW HEAD;
  END***FACILITY***;

  LINK CLASS STORAGE(FREE); INTEGER FREE;
  BEGIN
    REF(HEAD)Q;

    PROCEDURE ENTER (M); INTEGER M;
    BEGIN
      REF(TRANSACTION)T;
      T :- CURRENT;
      T.P_INTO(Q);
      WHILE (M > FREE) DO
      BEGIN
        PASSIVATE;
        ACTIVATE SUC AFTER T;
      END;
      T.OUT;
      FREE := FREE - M;
    END***ENTER***;

    PROCEDURE LEAVE(M); INTEGER M;
    BEGIN
      FREE := FREE + M;
      ACTIVATE Q.FIRST;
    END***LEAVE***;

    INITIALISE:
    IF FREE < 1 THEN error;
    INTO(STORAGE_Q);
    Q :- NEW HEAD;
  END***STORAGE***;
```

```
    PROCEDURE REPORT;
    BEGIN
      cycle down FACILITY_Q and report on
           each FACILITY;
      cycle down STORAGE_Q and report on
           each STORAGE;
    END***REPORT***;

    PROCEDURE ADVANCE(T); REAL T;
      HOLD(T);

    PROCEDURE GENERATE(P, T); REF(PROCESS)P;
        REAL T;
      ACTIVATE P AT T;

    FACILITY_Q :- NEW HEAD;
    STORAGE_Q  :- NEW HEAD;
    INNER;
    REPORT;
  END***GPSS***;
```

IN GPSS all transactions are queued in priority order. This has been implemented in our mini-context. The procedure P_INTO is based upon the SIMSET procedures introduced earlier.

We start our GPSS context by expanding PROCESS into TRANS_ACTION adding in an INTEGER PRIORITY (initially zero, it can be altered dynamically by ordinary assignment statements such as PRIORITY := 2;), and a PROCEDURE P_INTO which when called, inserts its owner into a named queue in priority order (larger values at the tail end).

We also include simplified definitions of FACILITY and STORAGE, queues FACILITY_Q and STORAGE_Q into which each and every user created FACILITY object and STORAGE object respectively will place itself on generation and a (rough idea of the global) REPORT procedure which is included to show how it can be called automatically. The REPORT routine merely has to cycle through the two queues in turn and report on each object it finds there. Because FACILITY and STORAGE objects are automatically placed their respective queues, there is no chance of 'losing' a report due to oversight. The declarations of ADVANCE and GENERATE are merely renaming mechanisms and are included to make the GPSS user feel more at home.

The class body actions of GPSS create the FACILITY_Q and STORAGE_Q and the INNER passes control over to the user program. Once the latter's execution is complete, control returns back up to the GPSS context level and issues a call on REPORT.

Each resource, be it FACILITY or STORAGE, is not just an integer giving how much is free, but also a queue for holding delayed transactions, and routines for acquiring portions of the resource and releasing portions back to it. The integer and the queue are protected from the user - he is only allowed to manipulate the resource via these two routines. The design is also very secure because checks can be made (not shown here) to ensure that a transaction does not return more than it acquired, ask for more than the resource limit, etc.

In these resource implementations, a transaction first joins the resource queue (in priority order) and then tests the availability of the resource. If sufficient is free, it decrements the integer, leaves the resource queue, and then proceeds without delay. If not, it remains in the resource queue and is allowed to proceed only when enough of the resource is available. Importantly, when a portion of a resource is returned, one knows precisely where all the other transactions awaiting upon that resource are - they are blocked in the resource queue. Thus the return routines (RELEASE and LEAVE), not only increment the integer, but also awaken all dormant transactions who can now go. This is very efficient, and also relieves the programmer of a major headache - explicitly reactivating blocked processes.

The Simula version of the mill using this GPSS context reads:

```
BEGIN EXTERNAL CLASS GPSS;
  GPSS
    BEGIN
      REF(FACILITY)CRANE;
      REF(STORAGE) POWER;

      TRANSACTION CLASS FURNACE;
      BEGIN
      LOAD:
        ADVANCE(2.0);
        POWER.ENTER(3);
        ADVANCE(2.0);
        POWER.LEAVE(2);
        ADVANCE(1.0);
        CRANE.SEIZE(1);
        ADVANCE(1.0);
        CRANE.RELEASE(1);
        POWER.LEAVE(1);
        GOTO LOAD;
      END***FURNACE***;

      CRANE :- NEW FACILITY;
      POWER :- NEW STORAGE(4);
      GENERATE(NEW FURNACE, 0.0);
      GENERATE(NEW FURNACE, 1.0);
      ADVANCE(200.0);
    END***GPSS version of the mill***;
END***PROGRAM***;
```

Aside: notice that this program is pretty much one for one in length with the original GPSS program.

Pros and cons of GPSS

The good things about GPSS are its resource types, tracing and reporting which make the job of the beginner very easy. Further, GPSS implementations have good error messages at compile time and run time; and TRACE and block counts are valuable when debugging. Although GPSS has a fixed-package framework, there are straightforward ways of extending its capabilities. For example, the random number generators provided by GPSS are poor. Fishman [9] shows how to use HELP routines to write your own in FORTRAN. Again, should more than one transaction type be desirable, several logically different types can be given in one combined description. Schriber

shows you how.

On the negative side, simulation is now being applied to systems with ever more components, with more complicated interactions amongst them, and requiring detailed modelling of their algorithmic aspects. All three of these points work against GPSS. One misses GPSS subroutines and extension mechanisms; the ability to read in data from files and output raw data to files for later analysis rather than an immediate report.

It seems to me that there is not much point in implementing both the FACILITY and the STORAGE. After all, a FACILITY is nothing but a STORAGE of size 1. Several other obvious synchronising mechanisms have been left out completely; for example, synchronisations for message passing, transaction-transaction co-operation, and transaction interruption.

Note that although GPSS implementations are usually very slow (see Virjo [10]), this is not inherent in GPSS. It is just that most implementations have taken the easy way out and interpret rather than produce machine code. Since GPSS is a dedicated language with a simple run time structure, GPSS implementations should be faster than Simula implementations. In practice they are usually 2 or 3 times slower, often much worse than that.

DEMOS

Obviously, much that I learned from GPSS has been carried over into Demos. Hosting Demos in Simula takes care of many of the objections to GPSS: it allows many transaction types, has good algorithmic capability (down to the character, but not to the bit level), good control structures, allows subroutines, etc. What Simula hasn't got at present are built-in resource types, tracing, report facilities, and much in the way of automation. It is a purpose of Demos to provide them.

If we begin our account of the design of Demos by asking the question "how can processes interact?", we wind up with a long list indeed. First there is mutual exclusion (storage in GPSS), but also there are producer/consumer, master/slave, waits until complicated conditions to arise, and the possibility of cancelling and interrupting other processes. In the rest of this chapter, we develop our mill model step by step, each step introduces a new complication and to solve it we need yet another synchronisation. Although this does not display the simplest possible use of these devices, they are introduced in the context of a fairly realistic model, and should imbue the reader with the confidence that they are indeed generally useful.

First of all, a CLASS ENTITY is defined with the structure:

```
CLASS ENTITY;
BEGIN
  INTEGER PRIORITY;
  PROCEDURE SCHEDULE(T); REAL T;
  PROCEDURE CANCEL;
  PROCEDURE INTO(Q); REF(QUEUE)Q;
  PROCEDURE REPEAT;
END***ENTITY***;

PROCEDURE HOLD(T);
```

Entities are the only objects which can be scheduled in the event list. Once there, they carry out activities by HOLDing themselves; on their life histories are completed, they CANCEL themselves from the event list. A process may SCHEDULE or CANCEL other entities - making HOLD global forces other entities to use SCHEDULE and CANCEL properly. A call on INTO puts an entity into the named queue in priority order. A call on REPEAT causes the class body actions to be repeated.

Mutual exclusion

The first resource type to be implemented is the RESource, which parallels a GPSS storage. Resources have an initial LIMIT, and thereafter can be ACQUIRED and RELEASED in integer chunks. Built into the resource is a queue in which blocked entities waiting upon the availability of that resource are kept.

```
CLASS RES(TITLE, LIMIT); VALUE TITLE; TEXT
     TITLE; INTEGER LIMIT;
BEGIN
  REF(QUEUE)q;
  PROCEDURE ACQUIRE(N); INTEGER N;
  PROCEDURE RELEASE(N); INTEGER N;
  INTEGER PROCEDURE AVAIL;
  IF LIMIT < 1 THEN error; q :- NEW QUEUE;
END***RES***;
```

Another visible attribute is AVAIL which returns how much of the resource is currently available. On generation of each RES object, the class body actions automatically check that its parametric LIMIT is at least 1. And a local queue q is created - this is where entities blocked on this resource will wait (in priority order). Using these tools, we could code our steel mill model in Demos by:

```
BEGIN EXTERNAL CLASS DEMOS;
DEMOS
  BEGIN
    REF(RES)POWER, CRANE;

    ENTITY CLASS FURNACE;
    BEGIN
      HOLD(2.0);
      POWER.ACQUIRE(3);
      HOLD(2.0);
      POWER.RELEASE(2);
      HOLD(1.0);
      CRANE.ACQUIRE(1);
      HOLD(1.0);
      CRANE.RELEASE(1);
      POWER.RELEASE(1);
      REPEAT;
    END***FURNACE***;

    POWER :- NEW RES("POWER", 4);
    CRANE :- NEW RES("CRANE", 1);
    NEW FURNACE("F").SCHEDULE(0.0);
    NEW FURNACE("F").SCHEDULE(1.0);
```

```
        HOLD(200.0);
      END;
    END;
```

which is one for one in length with the mini-GPSS context.

Scheduling other entities

An entity may be scheduled from within the main block, or anywhere else in a Simula program. We now extend our basic model by adding in an INGOT component which traces the path of a batch of ingots from its creation (completion of a 'pouring'), to its exit from the mill as a rolled plate. Once poured, INGOTS are allowed time to set, that is form a hard crust so that they are self supporting when removed from their moulds. Ingots are then placed in soaking pits so that they come up to a uniform temperature throughout. Then they are lifted out of their soaking pit by a crane and rolled into sheets by a rolling mill. The ingot temperature has to be just right; if too cold, they crack the rollers; if too hot, they splash all around the shop floor. An informal description of an ingot is:

```
ENTITY CLASS INGOT;
BEGIN
  set;
  soak;
  roll;
END***INGOT***;
```

To formalise our description, we add into our model the declaration:

```
REF(RES)PITS, CRANE, MILL;
```

and their initialisations

```
PITS   :- NEW RES("SOAKING PITS", 20);
CRANE2 :- NEW RES("PIT AREA CRANE", 1);
MILL   :- NEW RES("ROLLING MILL", 1);
```

We can now give the full declaration of CLASS INGOT:

```
ENTITY CLASS INGOT;
BEGIN
SET FIRM:
  HOLD(SET.SAMPLE);
LOAD INTO PIT:
  PITS.ACQUIRE(1);
  CRANE2.ACQUIRE(1);
  HOLD(LOAD.SAMPLE);
  CRANE2.RELEASE(1);
SOAK:
  HOLD(SOAK.SAMPLE);
UNLOAD:
  MILL.ACQUIRE(1);
  CRANE2.ACQUIRE(1);
  HOLD(UNLOAD.SAMPLE);
  CRANE2.RELEASE(1);
ROLL AND QUIT:
  HOLD(ROLL.SAMPLE);
  MILL.RELEASE(1);
  PITS.RELEASE(1);
END***INGOT***;
```

which shows again how quite involved logic can be modelled resorting only RES type synchronisations. Hence the success of GPSS. The ingots are generated by including a call

```
NEW INGOT("INGOT").SCHEDULE(0.0);
```

within the body of CLASS FURNACE after the second HOLD(1.0) line. The parameters to HOLD are all calls on built in Demos distributions, see Birtwistle [11], these proceedings.

Producer/consumer

The producer/consumer synchronisation is well known to writers of operating systems and telecommunications software. It crops up quite often in discrete event problems, so it is rather surprising that no previous simulation language designer has felt impelled to include it in his language.

The simplest use involves two entities, one of which produces (widgets, say) which the other process consumes. The producer continues on making widget after widget; the consumer is blocked if there is no widget for it to consume.

```
ENTITY CLASS P;          ENTITY CLASS C;
BEGIN                    BEGIN
  make;                    W.TAKE(1);
  W.GIVE(1);               use;
  REPEAT;                  REPEAT;
END***P***;              END***C***;

    REF(BIN)W;
    W :- NEW BIN("WIDGETS", 0);
```

The appropriate Demos device is called a BIN and has the outline:

```
CLASS BIN(TITLE, INITIAL_SIZE); VALUE TITLE;
          TEXT TITLE; INTEGER INITIAL_SIZE;
BEGIN
  REF(QUEUE)q;
  PROCEDURE TAKE(N); INTEGER N;
  PROCEDURE GIVE(N); INTEGER N;
  INTEGER PROCEDURE AVAIL;

  IF INITIAL_SIZE < 0 THEN error;
  q :- NEW HEAD;
END***BIN***;
```

The producer gives (increments a local counter and awakens any one blocked who can now go). The consumer tries to take (and is held in a queue local to the BIN until his request can be granted). A BIN is thus very much like a RES: both have an integer count, a local queue, and a function AVAIL which returns the current count value. The pairs ACQUIRE/TAKE, RELEASE/GIVE are very similar too. But whereas a RES has an upper limit, a BIN has none. Importantly as the same entity will acquire and release a RES, checks can be built into ACQUIRE and RELEASE to ensure that a process does not try to acquire more of a RES than its limit, and release more than it acquired, and also when it terminates, that it has returned to the pool of resources all that it has acquired. None of these checks is appropriate in the producer/consumer synchronisation, and if the BIN and the RES are implemented as one device, the

possibility for these tight checks is lost. This synchronisation has proved very valuable in modelling computer and telecommunications systems.

In our model we introduce BOGIES as a BIN. A bogie carries a batch of moulds sufficient for one pouring. A furnace requires a bogie laden with moulds before it can pour. We can now complete our furnace description and include the request for a bogie and the generation of a new batch of ingots at the end of each pouring. We have also extended our previous description in two other ways. First a furnace load will fill four batches of ingots (neatly coded by a FOR-loop), and at the end of a pouring there is a small chance (modelled by CRACKED.SAMPLE) that the furnace needs relining. In which case it is allowed to cool off, and then a team of brickies repair the cracked lining before it starts loading again. Finally electricity in large blasts is expensive, so the mill restricts the furnaces so that only one can be 'melting' at a time. This is ensured by the extra RES MUTEX. (Although the parameters we have chosen: 1 furnace, 3 units of power for melting, and 4 overall would only allow one furnace in at a time anyway, this would not necessarily be the case if we increased the capacity of the mill.)

```
REF(RES)MUTEX, POWER, BRICKIES;
REF(BIN)BOGIES;

ENTITY CLASS FURNACE;
BEGIN
  INTEGER K;
  HOLD(LOAD_SCRAP.SAMPLE);
  MUTEX.ACQUIRE(1);
  POWER.ACQUIRE(3);
  HOLD(MELT.SAMPLE);
  MUTEX.RELEASE(1);
  POWER.RELEASE(2);
  HOLD(REFINE.SAMPLE);
  FOR K := 1 STEP 1 UNTIL 4 DO
  BEGIN
    BOGIES.TAKE(1);
    HOLD(POUR.SAMPLE);
    NEW INGOTS("INGOTS").SCHEDULE(0.0);
  END;
  IF CRACKED.SAMPLE THEN
  BEGIN
    BRICKIES.ACQUIRE(1);
    HOLD(RELINE.SAMPLE);
    BRICKIES.RELEASE(1);
  END;
  REPEAT
END***FURNACE***;
```

This description also shows the power of embedding a simulation package in a general purpose host which contains looping and branching constructs.

Master/slave

Once a batch of ingots has been poured, the bogie is shunted away into sidings and the ingots are allowed to set. After sufficient time has elapsed, the ingots are unloaded and dumped by the pitside. Here they are stripped of their moulds and loaded into a soaking pit. The stripping is carried out by a special crew, who not only clean the moulds, but also reassemble them, load them back onto the bogie and push the bogie back to the furnace area.

The actions of ingots and the crew coincide for a while (strip), but then diverge.

```
     crew                              ingots

                                       set
strip          <------------>          strip
reline                                 soak
reassemble                             roll
shunt
REPEAT;
```

Both ingots and crews HAVE to be modelled as entities. The question now is how do we neatly allow them to rendezvous (co-operate for a while) and then go their separate ways. This same mechanism was included in the author's earlier implementation of SIMON 75 and is a part of the ADA language. We choose one entity (according to taste) to be the master and let the other(s) be its slave(s). The slave waits for the rendezvous by executing

```
Q.WAIT;
```

where Q is a WAITQ object. When the master wishes to rendezvous, it executes the assignment

```
S :- Q.COOPT;
```

which removes the first waiting slave in the waitq and names it S. If the waitq is empty, then the master is blocked until the next slave executes a Q.WAIT. Only then will the assignment to S be completed. A sketch of WAITQ is:

```
CLASS WAITQ(TITLE); VALUE TITLE; TEXT TITLE;
BEGIN
  REF(QUEUE) masterq, slaveq;
  REF(ENTITY)PROCEDURE COOPT;
  PROCEDURE WAIT;
  INTEGER PROCEDURE LENGTH;
  MASTERQ :- NEW QUEUE;
  SLAVEQ  :- NEW QUEUE
END***WAITQ***;
```

The class body actions generate separate (hidden) queues for slaves and masters when a WAITQ object is created.

In our mill example, we declare

```
REF(WAITQ)STRIPQ;
```

and let the crew be the masters. The appropriate declaration for class crew is:

```
ENTITY CLASS CREW;
BEGIN
  REF(INGOTS)I;
  I :- STRIPQ.COOPT;
  HOLD(STRIP.SAMPLE);
  I.SCHEDULE(0.0);
  HOLD(CLEAN.SAMPLE + SHUNT.SAMPLE);
  BOGIES.GIVE(1);
  HOLD(TO_SETTING_AREA.SAMPLE);
  REPEAT;
```

```
END***CREW***;
```

Notice that a furnace acquired the moulds, and a crew returns them - they are modelled as BINs because different entities pick them up and drop them.

Waits until

We can now return to the ingots:

```
ENTITY CLASS INGOT;
BEGIN
  HOLD(SET.SAMPLE);
  STRIPQ.WAIT;

  PITS.ACQUIRE(1);
  CRANE2.ACQUIRE(1);
  ..................
END***INGOT***;
```

Note that there is now a 'hole' in the description of an ingot; having placed itself in STRIPQ, an INGOT is taken through the strip activity by its master crew. Only when this activity has been completed is the ingot scheduled again by its master. The ingot then tries for a place in the soaking pit. Of course, the ingots cool down as they wait for a place in the soaking pits. If the pits are fully utilised, a queue of waiting ingots will form. If this queue persists, it makes sense to dump the oldest (and coldest) ingots outside until a later slack period arises, and use a more recent arrival instead.

We introduce the strategy that if the pits are full (PITS.AVAIL = 0) and the length of the queue in front of the pits is 4 or more (PITQ.LENGTH >= 4), then a batch of ingots is dumped outside by a hoist crane (HOIST). Only when the pit utilisation is low (say, PITS.AVAIL >= 10) is such an ingot brought back and loaded into a soaking pit.

When decisions have to be made involving separate routes using disparate resources, one has to wait and see, wait until one is certain, before picking up resources and moving ahead. Here a CONDQ should be used:

```
CLASS CONDQ(TITLE); VALUE TITLE: TEXT TITLE;
BEGIN
  REF(QUEUE)q;
  PROCEDURE WAITUNTIL(C);
  PROCEDURE SIGNAL;
  INTEGER PROCEDURE LENGTH;

  q :- NEW QUEUE;
END***CONDQ***;
```

Each CONDQ contains a local queue for entities waiting upon its condition to arise LENGTH returns the number of currently waiting entities. An entity calling Q.WAITUNTIL(C) is blocked in Q if C is not true. Whenever C changes it is the responsibility of the entity that caused the change to signal any CONDQs that might be affected.

Complicated conditions are easy to code, e.g. corresponding to choosing route A or B

```
if A then .... else
if B then .... else
```

we code

```
Q.WAITUNTIL(A OR B OR ....)
```

and the caller is blocked until one or other or both of these conditions is true.

In activity mode languages, waituntil prefixes every activity. In process mode, given RES, BIN and MASTER/SLAVE synchronisations, they are hardly ever needed. Inserting an automatic CONDQ sniffer slows down a simulation run by a factor of 2 or 3, but this can be unboundedly worse. In Demos, the decision was made to force the user to explicitly wake up entities blocked in CONDQs. Every time a process effects an action which causes a waituntil condition to change, it must send a signal to that CONDQ.

In our mill model, we use a CONDQ called PITQ for the pit side area, and a second CONDQ called OUTQ for the outside area. The body of INGOT now reads:

```
REF(CONDQ)PITQ, OUTQ;

PITQ :- NEW CONDQ("PIT ENTRY");
OUTQ :- NEW CONDQ("DUMPED INGOTS");

ENTITY CLASS INGOT;
BEGIN
  BOOLEAN COLD;
  REAL T;
  HOLD(SET.SAMPLE);
  STRIPQ.WAIT;
  IF PITQ.LENGTH > 4 THEN PITQ.FIRST.
          SCHEDULE(NOW);
  PITQ.WAITUNTIL(PITS.AVAIL > 0 AND CRANE2.
          AVAIL > 0 OR PITQ.LENGTH >= 4);
  IF PITQ.LENGTH > 4 THEN
BEGIN
  COLD := TRUE;
  HOIST.ACQUIRE(1);
  HOLD(OUTTIME.SAMPLE);
  HOIST.RELEASE(1);
  OUTQ.WAITUNTIL(PITS.AVAIL >= 10 AND
          HOIST.AVAIL >= 1);
  HOIST.ACQUIRE(1);
  HOLD(INTIME.SAMPLE);
  HOIST.RELEASE(1);
  OUTQ.SIGNAL;
END;
PITS.ACQUIRE(1);
CRANE2.ACQUIRE(1);
HOLD(LOAD.SAMPLE);
CRANE.RELEASE(1);
PITQ.SIGNAL;
HOLD(IF COLD THEN LONGER.SAMPLE ELSE
        SHORTER.SAMPLE);
MILL.ACQUIRE(1);
CRANE2.RELEASE(1);
PITQ.SIGNAL;
HOLD(ROLL.SAMPLE);
MILL.RELEASE(1);
PITS.RELEASE(1);
PITQ.SIGNAL; OUTQ.SIGNAL;
END***INGOTS***;
```

Breakdowns and interrupts

We distinguish between two types of stoppage. (1) a breakdown by which we mean a process is using equipment which faults thus causing him a delay to fix the fault plus a reset time, but then the process carries on with its expected program. (2) interrupt where its behaviour pattern is altered (perhaps radically) by the unexpected.

Catering for a breakdown is intrinsically the easier. A process is carrying out a task and is in the event list at time denoted by its EVTIME. It thus has (P.EVTIME - TIME) left to go. If a breakdown now occurs, this time left is recorded, P removed from the event list, the breakdown cause repaired, and then P is scheduled again with delay.

For example, suppose we have a reserve generator. Should the main power supply go down, the reserve generator will supply 1 unit of power to each active furnace keeping them at the same temperature. A furnace engaged in a melt will be delayed by precisely the time involved in getting the main supply back up again. This we can code by

```
REF(FURNACE)M;

ENTITY CLASS FURNACE;
BEGIN
  ..........
  MUTEX.ACQUIRE(1);
  M :- THIS FURNACE;
  POWER.ACQUIRE(1);
  HOLD(2.0);
  POWER.RELEASE(2);
  M :- NONE;
  MUTEX.RELEASE(1);
  ..........
END***FURNACE***;

ENTITY CLASS GREMLIN;
BEGIN
  HOLD(time to next failure);
  zap the power supply;
  effect a repair;
  restart power to the furnace (if any)
    that was melting;
  REPEAT;
END***GREMLIN***;
```

In more detail, we fill out GREMLIN to

```
ENTITY CLASS GREMLIN;
BEGIN
  HOLD(MTBF.SAMPLE);
  IF MUTEX.AVAIL = 1 THEN
  BEGIN
    MUTEX.ACQUIRE(1);
    HOLD(REPAIR.SAMPLE);
    MUTEX.RELEASE(1);
  END ELSE
  BEGIN
    T := M.EVTIME - TIME;
    M.CANCEL;
    HOLD(REPAIR.SAMPLE);
    M.SCHEDULE(0.0);
  END;
  REPEAT;
END***GREMLIN***;
```

Interrupts

Interrupts give rise to distortions in process behaviour. For example, suppose the rolling mill breaks down, what then? The mill may not be in use, may have an ingot en route from the pit but not be rolling (response: return the ingot to its pit at once), or be rolling (in which case the shape of the ingot will be distorted and it will not fit back into its vacated pit slot. It is then 'plated', i.e. recycled for loading). An entity E may be interrupted by a call E.INTERRUPT (n), which sets a variable E.INTERRUPTED to N, and then reschedules E behind its interrupter (CURRENT). When E becomes current, it decides for itself what to do next.

```
REF(INGOT)MILL_USER;

ENTITY CLASS MILL_BREAKDOWN;
BEGIN
  PRIORITY := 2;
  HOLD(MTTNB.SAMPLE);
  IF MILL_USER =/= NONE THEN
       MILL_USER.INTERRUPT(1);
  MILL.ACQUIRE(1);
  HOLD(REPAIR.SAMPLE);
  MILL.RELEASE(1);
  REPEAT;
END***MILL_BREAKDOWN***;

ENTITY CLASS INGOT;
BEGIN
  ..........
L:
  MILL.ACQUIRE(1);
  CRANE2.ACQUIRE(1);
  HOLD(LOAD.SAMPLE);
  IF INTERRUPTED > 0 THEN
  BEGIN
    INTERRUPTED := 0;
    MILL.RELEASE(1);
    HOLD(LOAD.SAMPLE);
    CRANE2.RELEASE(1);
    GOTO L;
  END;
  CRANE2.RELEASE(1);
  HOLD(ROLL.SAMPLE);
  IF INTERRUPTED > 0
    THEN PLATED := PLATED + 1
    ELSE GOOD   := GOOD   + 1;
  MILL.RELEASE(1);
  PITS.RELEASE(1);
END***INGOTS***;
```

The program is completed by declaring and creating the distribution objects, and scheduling 4 furnaces.

CONCLUSIONS

Demos is implemented as a Simula context (prefixed by SIMSET, but not by SIMULATION) and extends over roughly 2000 cards. Along with suitable documentation and testing, DEMOS was approximately a 9 man-month project. The resulting system is portable and has proved easy to extend,

alter and maintain.

Implementing a Demos compiler from scratch was out of the question because Demos is Simula plus, and Simula is itself a 15 man-year project. However a Demos compiler would have definite advantages. It could, for example, give error messages written in Demos (rather than Simula) terminology, detect the possibility of deadlock at compile time, accept a Demos program written with waits until and itself insert the correct calls on SIGNAL, give resource usage cross-reference lists, do away with the need for explicitly titling every Demos object (as in NEW INGOT("INGOT"), etc.). Without a compiler, the same effect can be achieved by using a pre-processor, which is much easier to write but more expensive to run.

There are many advantages to coding Demos in Simula. Firstly, Simula is widely implemented and to a good standard. The implementors of Simula systems meet regularly in the SIMULA STANDARDS GROUP. The hope being that this would ensure that Simula systems are compatible now and will remain so in the future. Experience with porting Demos has been fairly trouble free, only the CDC implementation giving rise to non-trivial problems. Demos was implemented on DEC System 10 hardware and has been ported to DEC System 20, IBM 360/370, and ICL System 4, and ICL 2900 hardware with no modifications at all. The UNIVAC 1100 implementation does not yet support virtual labels, so that version of Demos has to make do without REPEAT. Both the NDRE and CDC Simula implementations quote key words. Once that hurdle has been accounted for, the NDRE version supports neither virtual labels nor functions returning references, so that REPEAT is out, and the function REF(ENTITY)PROCEDURE NEXTEV has had to be rewritten as a procedure. The CDC Simula compiler still does not treat virtual quantities correctly, and the Demos code for the reset and report routines had to be 'bent' to fit. Altogether, the experience has not been too bad (have you tried porting FORTRAN programs from one machine to another?). The situation should improve in the very near future when the Norwegian Computing Center's portable Simula implementations are released, making identical Simula compilers available on an even wider range of hardware.

Secondly, Simula's object and context features are considerably ahead of anything offered by other languages. It is easy to 'see' how to implement Demos facilities as Simula objects. The context feature enables an implementation to progress in an orderly fashion layer by layer, each new step adding in a few new interrelated ideas. Simula has very strict security and consistency checks so that many mistakes are picked up as soon as possible at compile time.

Finally, Demos is not the end of the road. The user should proceed in standard Simula fashion to develop his own more specialised contexts for his own areas of interest, e.g.

```
DEMOS CLASS STEEL(N_MILLS, N_PITS);
          INTEGER N_MILLS, N_PITS;
BEGIN
  ENTITY CLASS FURNACE;.....;
  ..........................
  REF(RES)MILLS, PITS;
  ...................
  MILLS :- NEW RES("MILLS", N_MILLS);
  PITS  :- NEW RES("PITS",  N_PITS;
END***STEEL***;
```

The distribution of Demos

Demos is an ordinary Simula program and will run on any computer that supports Simula. The Demos Reference Manual gives full documentation of the complete Demos system, and includes a Simula source listing of Demos as an appendix.

The Demos system (both source code and reference manual) are distributed in machine readable form as unlabelled files on IBM standard, 9 track tapes.

REFERENCES

1) G. M. Birtwistle, Discrete Event Modelling on Simula, Macmillan Press, 1979.

2) G. M. Birtwistle, DEMOS Reference Manual, 278 pp., 2nd Edition, July, 1981.

3) G. M. Birtwistle, O-J. Dahl, B. Myhrhaug, and K. Nygaard, Simula begin, Studentlitteratur, Lund, Sweden, 1973.

4) O-J. Dahl, B. Myrhaug, and K. Nygaard, Simula 67 Common Base Language, NCC Publication S-52, Norwegian Computing Center, Oslo, 1970.

5) W. R. Franta, The process view of simulation, North Holland, 1978.

6) J. Palme, Simula adopts data security, Simula Newsletter.

7) T. Schriber, Simulation using GPSS, Wiley, 1974.

8) J. Vaucher, Simulation data structures using Simula 67, Proc. Winter Simulation Conference, 1971, pp. 255-260.

9) G. S. Fishman, Concepts and Methods in Discrete Event Digital Simulation, Wiley, 1973.

10) A. Virjo, Comparison of Discrete Event Simulation Languages, NORDData 72 Conference, Helsinki, 1972.

11) G. M. Birtwistle, A portable random number generator with built-in well spread seeds, These proceedings.