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

ADVANCED USE OF SIMULA

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

ABSTRACT

This paper is a tutorial on program development in Simula. It assumes a reading knowledge of Simula, and sketches the design of a local area network simulator (Cambridge Ring architecture) in five logical levels: machine interface, queueing, simulation primitives, data collection primitives and finally the network components. Besides program development technique, we also emphasize the value of class body actions, inner, the virtual mechanism and data protection.

INTRODUCTION

Simula was designed by Dahl, Myhrhaug and Nygaard at the Norwegian Computing Center (NCC), Oslo. It has been implemented on CDC, DEC, IBM, ICL, and Univac hardwares, amongst others to a mutually agreed standard as laid out in the Common Base definition (Dahl [1]). The NCC did not merely produce a language but also set up language custodian and user groups to guarantee continued uniformity. Language consistency is controlled by the Simula Standards Group and language development by the Simula Development Group. These groups are meant for implementors and language designers; for ordinary users there is the Assocation of Simula Users which has its own newsletter and annual conferences.

Simula (Dahl [1]) is the successor to Simula 1 [2] which was a pure simulation extension to Algol 60. (Nygaard [3] is an account of the development of the two Simulas.) It is important to realise that the new Simula was not intended solely (or even particularly) for discrete event simulation work. It was designed as an extendible kernel in which to write application packages (which we will call "contexts") for specialised areas. A context is a package hosted in SIMULA which extends Simula towards a particular problem area. It will define a library of concepts and methods associated with that one area, but leaves it to the programmer to apply them in his own way.

The implied methodology makes Simula an attractive proposition for an applications group, with different specialisations in the same general area. Since it takes several months to become an expert in another programming language, it is economically unjustifiable to expect every member of the applications group to be a expert programmer. Working with Simula, an applications group needs no more than one Simula expert, for the rest a nodding acquaintance will do (this knowledge will grow with exposure anyway). The expert will collaborate with his colleagues over the content of a context providing tools to cover their application area and angled towards their style of use. Then he will implement the context; note that it is at this level that expertise in Simula is most required. Once commissioned, a context becomes a platform for a myriad of applications. For example, literally thousands of discrete event Simulation programs have been based on the standard Simula context SIMULATION, and just about every interactive Simula program written for the DEC 10 system uses SAFEIO. All programs written on top of the context must conform to Simula's syntax and semantics, but have access to the extra high level the building blocks now provided. They usually (read 'always') turn out to be extremely user oriented and very restricted in format. Thus only a little formal Simula need be learnt before they can be used, and that can be picked up in a matter of hours rather than months. Applications programmers can either use the already written context

tailor-made for their group, or, if this proves insufficient, discuss their needs with the Simula expert and let him extend it for them. But as their knowledge of Simula grows by repeated use, the applications programmers will be able to add in their own fresh layers to the ones provided, or perhaps peel back some of the provided layers and produce more suitable ones of their own.

Simula is not an all things to all men language, but a kernel in which to write user oriented packages. Simula programs carry the overhead of this Simula kernel plus the appropriate context. However the Simula kernel is still a substantial chunk of software carrying a run time overhead of about 15 k words. This is due partly to the fact that it was deemed politically wise at the time of its design not to come up with a completely new language but to build on Algol 60. There already exist dozens of Simula contexts for such varied areas as discrete event simulation, continuous simulation, combined simulation, CODASYL type data bases, graphics, safe interactive i/o, text formatting, teleconferencing, etc. etc. A short bibliography, culled mainly from the Simula Newsletter, is included at the end of this paper.

USING SIMULA

There are two main reasons why so much excellent software has been developed in Simula - namely the class and context concepts. Classes are used to describe entire simulated components - both their attributes and their behaviour patterns. Classes have been used to define records (e.g. in data base applications), records with operators (e.g. queued loads in simulations. Primarily records, they also need operators to place them into and remove them from queues), and processes (e.g. the transmit and receive software processes in X25 nodes). In fact, class declarations are quite as general as FORTRAN, PL1, GPSS, or even SIMULA programs. Furthermore, many copies of the same declaration may exist at the same time. The class declaration is probably the most powerful programming language concept yet developed. We will use class declarations to describe the software the transmit hardware of network nodes, and to define processes representing incoming requests and periodic interrupts.

Contexts are libraries of pre-defined class, procedure and data definitions to which initialising actions may be added. Context definitions may be built up layer by layer, each corresponding to an entire logical software. For example, the ISO Open Systems Interconnection reference model has a seven layer peer-to-peer protocol interrelation. In Simula, this would be modelled by developing a context for each layer:

```
CLASS PHYSICAL;
BEGIN
  defines the physical media for inter-
          connection;
END;

PHYSICAL CLASS DATA_LINE;
BEGIN
  defines protocols ensuring error free
          transmission of packets;
END;

DATA_LINK CLASS NETWORK;
BEGIN
  defines software to accept input packets
          at a source node, forward
          packets destined for other
          nodes and accept packets at
          a destination node;
END;

NETWORK CLASS TRANSPORT;
BEGIN
  controls data movements from source to
          destination nodes;
END;

etc. etc.
```

Each layer isolates and resolves one problem. Concepts developed in one layer are passed onto the next one(s) by prefixing the next context declaration. Thus when dealing with incoming packets in level 3 (CLASS NETWORK), we may assume that they have been received intact and in order because appropriate procedures have already been provided in the prefixing previous layer (DATA_LINK). This methodology is very much in accordance with recent proposals for structured programming and system specification. The gap between the visualisation of a system and its implementation in Simula seems to me to be smaller than in any other generally available language.

OUTLINE OF THE SIMULATOR

We now describe the implementation of a simulator for a specific local area network architecture. The simulator described in level 5 is based upon a program developed by Bill May, Brian Ritchie and myself at the Edinburgh Regional Computing Centre in the summer of 1981. The contents of the first four levels herein described were actually provided by Demos [4, 5] in the real model. But for teaching purposes, I have chosen to work through some stripped down, purpose-built software coded in several thin layers.

Software layer	Level
user program	6
node hardware and software	5
reporting	4
simulation	3
queueing	2
safe input	1
Simula	0

The simulator is described in 5 levels, built upon Simula (at level 0). Working from the bottom up, level 1 defines routines for safe interactive input, and text handling. Level 2 is concerned solely with the notion of queueing. Level 3 uses the queueing concepts of level 2 to implement an event list and scheduling mechanisms. Level 3 also introduces processes. Level 4 introduces data collection devices and uses level 2 queues to implement automatic reporting. Only at level 5 do we get around to implementing the local area network stations and their software. At level 5, level 1 software is used to obtain the simulation run parameters, level 3 software allows us to describe software processes and hardware components, level 4 software enables us to gather statistical data in an unobtrusive fashion. Written in this way, we concentrate upon only one problem at a time. Each level is complete in itself and can be thoroughly proven before moving on to the next.

LEVEL 1 - SAFE INPUT

Interactive input should be guided by prompts and malformed or unexpected (out of order) input should never cause a program to blow up. To cater for these needs, we should supply routines which provide prompting for all types of input data (integer, real, boolean, character, text). These routines will examine the user response character by character making sure that the data supplied is acceptable. If not, a warning is given and request for data is made again. If the data is ok, it is accepted. In our mini-context, we outline how these ideas can be implemented for integer input (the paradigm is the same for other types), and for good measure throw in two text handling routines which have shown themselves to be generally useful.

```
CLASS SAFE_INPUT;
BEGIN
  PROCEDURE ASK(PROMPT, V, C);
    VALUE PROMPT; NAME V, C;
    TEXT PROMPT; INTEGER V; BOOLEAN C;
  BEGIN
    PROCEDURE valid input.....;

    BOOLEAN OK;

    WHILE NOT OK DO
    BEGIN
      OUTTEXT("? PLEASE INPUT (INTEGER) ");
      OUTTEXT(PROMPT);
      OUTIMAGE;
      INIMAGE;
      OK := valid input  AND C;
    END;
  END***ASK***;

  TEXT PROCEDURE CONC(A, B);
    VALUE A, B; TEXT A, B;
  BEGIN
    TEXT T; INTEGER LA, LB;
    LA := A.LENGTH;
    LB := B.LENGTH;
    T  :- BLANKS(LA + LB);
    T  := A;
    T.SUB(LA + 1, LB) := B;
    CONC :- T;
  END***CONC***;

  TEXT PROCEDURE EDIT(T, N);
    VALUE T; TEXT T; INTEGER N;
  BEGIN
    TEXT X; INTEGER LT;
    LT :- T.LENGTH;
    X  :- BLANKS(LX + 3);
    N  := N-N//1000*1000; !modulo 1000;
    X  := T;
    X.SUB(LX + 1,3).PUTINT(N);
    EDIT :- T;
  END***EDIT***;
END***SAFE INPUT***;
```

Suppose we wish to input a value for the number of epochs required in this run. The call

```
ASK("NUMBER OF EPOCHS",
     EPOCHS,
     EPOCHS >= 1 AND EPOCHS <= 24);
```

will write out

```
? PLEASE INPUT (INTEGER) NUMBER OF EPOCHS
```

and refuse any input that does not represent an integer. The final parameter (called by name and thus re-evaluated on every loop) further restricts the input to lie in (1, 24). The routine is incomplete - 'valid input' is the name of a local function which looks at the currently submitted input line and checks character by character to see that it is both an integer value and in range. If not valid input tells you why and returns false causing the prompt loop to be re-entered. If the input is valid, its value is assigned to V and valid input returns true. As shown above, C is used to place tighter bounds on the inputted value. If the value has no size restraints, use TRUE as the actual parameter.

SAFE_INPUT is very much a poor man's SAFEIO, an interactive package developed by Mats Ohlin in 1975 and distributed with the DEC 10 and DEC 20 Simula systems. SAFEIO is recommended reading. It is in daily use by many programs running on DEC 10 and DEC 20 systems.

Notice that SAFE_INPUT is a box of independent procedures which could be compiled separately and introduced into a Simula program by the external declarations:

```
EXTERNAL PROCEDURE ASK;
EXTERNAL TEXT PROCEDURE CONC, EDIT;
```

Wrapping them together in a single, separately compiled context means that they would ALL be available by the single declaration

```
EXTERNAL CLASS SAFE_INPUT;
```

LEVEL 2 - QUEUEING

It is possible to design classes which are intended to serve as prefixes, and will not be used as they stand. An obvious example of this

is provided when we come to develop a queueing mechanism. This tool will be used in later levels to queue blocked processes, link together arbitrary numbers of data recording devices and serve as basis for the event list. As common event list operations include inserting a process at an arbitrary position in the event list, and deleting scheduled processes (interrupts), a two-way linked queue will serve our purpose best. (Two way queues are in fact, already implemented in the standard built-in Simula context SIMSET, but we have a pedagogical need for a context involving two interrelated concepts).

Each member of a two-way queue is linked to its predecessor and its successor (the predecessor of the first member is NONE, the successor of the last member is NONE). It is important to realise that inserting members into a queue and removing them from a queue involves only manipulation of these pointers and need not depend in any way upon their other attributes. Accordingly we can implement a complete layer which encapsulates once and for all the ideas of two way lists. Each of our queues will be headed by a QUEUE object with pointers to its FIRST and LAST members. The queue head also maintains a count LENGTH of the number of items currently in the queue. Besides attributes SUC and PRED, each queue member maintains a reference Q to the head of the queue it currently lies in. M.Q == NONE implies that member object M is not currently in a queue.

Here is a sketch of CLASS QUEUEING. Note that it is prefixed by SAFE_INPUT and thus any user of queueing also has access to the prompt and text handling routines of level 1.

```
SAFE_INPUT CLASS QUEUEING;
BEGIN

  CLASS QUEUE;
  BEGIN
    REF(MEMBER)FIRST, LAST;
    INTEGER LENGTH;
  END***QUEUE***;

  CLASS MEMBER;
  BEGIN
    REF(MEMBER)SUC, PRED;
    REF(QUEUE)Q;

    PROCEDURE INTO(Q); REF(HEAD)Q;.....;
    PROCEDURE OUT;.....;
    PROCEDURE FOLLOW(M); REF(MEMBER)M;....;
    PROCEDURE PRECEDE(M); REF(MEMBER)M;...;
  END***MEMBER***;
END***QUEUE***;
```

QUEUEING can now be separately compiled. Its concepts are made available to a user program by including the declaration

```
EXTERNAL CLASS QUEUEING;
```

Notice this time that QUEUE and MEMBER are interrelated (each contains references to objects of the other class) therefore they cannot be compiled separately.

Besides the six variables discussed above, we will also furnish four operations on queues which are written local to class MEMBER. A call M.OUT will remove M from its current queue, if any. (Members can be in at most one queue at a time since they are furnished with only one SUC/PRED pair. The first operation of the three queue insert routines is M.OUT. In addition M.INTO(Q) has no other effect if Q == NONE; otherwise M is added to the end of Q (becomes the new Q.LAST). M.PRECEDE(X) has no other effect if X == NONE or X.Q == NONE. Otherwise M is put into the same queue as X as its predecessor. The effect of M.FOLLOW(X) is now obvious.

A more complete version of CLASS MEMBER is:

```
CLASS MEMBER;
BEGIN
  REF(MEMBER)SUC, PRED;
  REF(QUEUE)Q;

  PROCEDURE OUT;
  ! removes from current queue, if any;
  IF Q =/= NONE THEN
  BEGIN
    IF THIS MEMBER == Q.FIRST
      THEN Q.FIRST :- SUC
      ELSE PRED.SUC :- SUC;
    IF THIS MEMBER == Q.LAST
      THEN Q.LAST :- PRED
      ELSE SUC.PRED :- PRED;
    SUC :- PRED :- NONE;
    Q.LENGTH := Q.LENGTH - 1;
    Q :- NONE;
  END***OUT***;

  PROCEDURE INTO(X); REF(QUEUE)X;
  BEGIN
    OUT;
    ! no effect if X == NONE;
    IF X == NONE THEN warning ELSE
    BEGIN
      IF X.LAST == NONE THEN
      BEGIN
        X.LAST :- THIS MEMBER;
        X.FIRST :- THIS MEMBER;
      END ELSE
      BEGIN
        X.LAST.SUC :- THIS MEMBER;
        PRED       :- X.LAST;
        X.LAST     :- THIS MEMBER;
      END;
      Q :- X;
      Q.LENGTH := Q.LENGTH + 1;
    END;
  END***INTO***;

  ! FOLLOW and PRECEDE are too obvious to
    need listing;
END***MEMBER***;
```

The class QUEUEING is an extended version of the standard Simula context SIMSET, see Dahl [1]. Dahl [6] gives an extremely elegant presentation of (the concepts of Simula and) SIMSET, which he calls TWLIST - Two Way LIST. Using assertions, Dahl is able to show that SIMSET lists are guaranteed to be well behaved provided that a user operates on lists with the provided and correct routines INTO, OUT, etc., and does not make ex-

plicit assignments to SUC, PRED, FIRST and LAST. Given this tight restraint, SIMSET - and likewise our QUEUEING - provides a reliable software level which cannot break down. However, certain common queue operations (e.g. scanning a list) need access to some or all of these pointers, and then the guarantee is no longer valid. Although the code may be correct, that you must prove. Simula has certain attribute protection features (HIDDEN and PROTECTION) which can be used to extend a Dahl type guarantee to cover even these cases. The rewrite of class QUEUEING is left over until we have introduced a scanning mechanism (used in levels 3 and 4) and found a simpler need for data protection.

LEVEL 3 - SIMULATION

It is not necessary that program components representing processes occurring in parallel be multiprogrammed in a computer; but it is necessary that the components should be able to suspend themselves temporarily, and be resumed later from where they left off. Processes in a simulation are represented by Simula objects operating in quasi-parallel under the control of a scheduling mechanism.

To implement simulated time, we give each process access to a global variable TIME which holds the current simulation clock time, and which is incremented on appropriate occasions by the time-control mechanism. The updating of this variable must be entirely independent of the passage of computing time during the simulation, since actions which take a long time on a computer may take only a short time in the real world, and vice versa. As far as simulated time is concerned, the active phases of the processes must be instantaneous.

To simulate the passing of time, a process simulating an active system component must relinquish control for a stated interval T of simulated time; and it must be reactivated again when the time variable has been incremented by T. This will be accomplished by the process calling HOLD(T). While a process is held, we need to record its reactivation time. We use a process attribute EVTIME for this purpose.

The method of holding for a specified interval is possible only if the process knows how long it has to wait before the next event in its life. Sometimes it may need to wait until the occurrence of some event in the life of some other process. For this we require the additional procedures WAIT(Q) and SCHEDULE(P).

Finally, we may wish to interrupt a process and let it deal with a higher priority task before resuming its previous task. For this we provide a procedure INTERRUPT(P,T) whose parameters specify the particular process and the amount by which it must be delayed.

A sketch of the implementation is:

```
QUEUEING CLASS DES;
BEGIN
  REF(QUEUE)EL;
  MEMBER CLASS PROCESS;
  BEGIN
    REAL EVTIME;
  END***PROCESS***;

  PROCEDURE WAIT)Q); REF(QUEUE)Q;.....;
  PROCEDURE HOLD(T); REAL T;.....;
  PROCEDURE SCHEDULE(P); REF(PROCESS)P;...;
  PROCEDURE INTERRUPT(P, T);
    REF(PROCESS)P; REAL T;.....;

  EL :- NEW QUEUE;
END***DES***;
```

DES is intended to be used as a prefix to a simulation program or a further context. It is prefixed by QUEUEING and thus inherits the concept of a queue AND the prompting and text handling routines of SAFE_INPUT which prefixes QUEUEING. Simulation components must be prefixed by PROCESS. This will give them access to the time control routines, and the queueing mehcanisms since PROCESS is prefixed by MEMBER. Processes waiting for the elapse of their holding time are held in the event list EL. The processes therein are ranked according to their reactivation times. The process with the least event time is EL.FIRST and the scheduling routines see to it that EL.FIRST is always the operating process. Other processes in the event list are suspended.

Notice that DES contains not only declarations, but this time an action. Whenever a context is used as a prefix, its class body actions are executed before the next level is entered. Thus a user of DES is guaranteed that the event list is correctly set up before any of his code is executed.

```
QUEUEING CLASS DES;
BEGIN
  REF(QUEUE)EL;

  MEMBER CLASS PROCESS;
  BEGIN
    REAL EVTIME
    PROCEDURE RANK;
    BEGIN
      REF(PROCESS)X;
      X :- EL.LAST;
      IF X == NONE THEN INTO(EL)ELSE
      IF EVTIME >= X.EVTIME THEN INTO(EL)ELSE
      BEGIN
        X :- EL.FIRST
        WHILE EVTIME >= X.EVTIME DO
          X :- X.SUC;
        PRECEDE(X);
      END;
    END***RANK***;
    DETACH;
    INNER;
    OUT;
    IF CURRENT == NONE THEN error ELSE
      RESUME(CURRENT);
  END***PROCESS***;
```

```
PROCEDURE SCHEDULE(P); REF(PROCESS)P;
BEGIN
  IF P == NONE THEN error;
  IF P.Q == EL THEN error;
  P.OUT;
  P.EVTIME := TIME;
  P.PRECEDE(CURRENT);
  RESUME(P);
END***SCHEDULE***;

PROCEDURE WAIT(Q); REF(QUEUE)Q;
BEGIN
  REF(PROCESS)P;
  IF Q == NONE THEN error;
  P :- CURRENT;
  P.OUT;
  P.INTO(Q);
  IF EL.LENGTH = 0 THEN error;
  RESUME(CURRENT);
END***WAIT***;

PROCEDURE INTERRUPT(P, T);
  REF(PROCESS)P; REAL T;
BEGIN
  IF P == NONE THEN error;
  IF T < 0.0 THEN T := 0.0;
  IF P.Q == EL
    THEN T := P.EVTIME + T
    ELSE T := TIME + T;
  P.OUT;
  P.RANK;
END***INTERRUPT***;

PROCEDURE HOLD(T); REAL T;
BEGIN
  REF(PROCESS)P;
  IF T < 0.0 THEN T := 0.0;
  P :- CURRENT;
  IF CURRENT =/= EL.LAST THEN
  BEGIN
    P.OUT;
    P.RANK;
  END;
END***HOLD***;

PROCESS CLASS MAIN_PROGRAM;
BEGIN
L: DETACH;
   GOTO L;
END***MAIN PROGRAM***;

REF(PROCESS)PROCEDURE CURRENT;
  CURRENT :- EL.FIRST;

REAL PROCEDURE TIME;
  TIME := CURRENT.EVTIME;

EL :- NEW QUEUE;
MAIN :- NEW MAIN_PROGRAM;
MAIN.EVTIME := 0.0;
END***DES***;
```

First note that if a DES process, P, is scheduled, then P.Q == EL. CURRENT refers to EL.FIRST, the currently operating process. When a scheduling routine is called, it is always called by CURRENT. SCHEDULE(P) is an error if P is already in the event list; otherwise P is scheduled now and preempts the caller. WAIT(Q) places current in a specified queue and puts it to sleep. The new current (EL.FIRST) is resumed and the simulation clock time is stepped up to its EVTIME. INTERRUPT(P, T) extends the current active phase of P by T if P is active. If P is dormant, it is put into the event list at TIME + T. HOLD(T) reschedules current at TIME + T; (normally) a new current is resumed.

Detach and resume are Simula primitives which enable processes to be set up as separate program components and operate in quasi-parallel. RESUME(X) freezes the caller and resumes operation of X from where actions of X were left off the last time it was operating. The process MAIN plays the role of simulation program. The latter being a block cannot itself be inserted into the event list. But MAIN can. When it becomes current, the detach command causes the main simulation program to be reentered. This device is borrowed from SIMULATION (Dahl [1]) and is incorporated in Demos [4, 5]. Reference 4, Chapter 3, contains a sample trace of event list operations using this type of object.

The body of PROCESS makes use of an extremely elegant Simula device - INNER. When a process has terminated its actions, it should be removed from the event list and the new current resumed. Thus the last two statements of every non-cyclic process declaration should be:

```
OUT;
RESUME(CURRENT);
```

and a programmer must remember to include them or else his simulation will terminate unexpectedly. The prefix/subclass actions are to be executed in the order

```
DETACH; sub-class actions; OUT; RESUME
          (CURRENT)
```

What INNER does is allow the initial and final actions to be written in the prefix (separated by INNER). When executing class body actions, meeting an INNER forces execution of the subclass actions first. Then control returns to the textually earlier level and the remaining actions are carried out. Thus the required actions are automatically always carried out no matter what the sub-class and users are relieved of a great responsibility.

The standard Simula context SIMULATION is usually implemented as a leftist priority tree (Dahl [8]); Franta [9] and Vaucher [7] have other proposals. based on lists. Vaucher's proposal is effective and particularly attractive due to its simplicity and brevity. Reference 5 contains Simula code for a complete set of leftist priority tree scheduling routines.

LEVEL 4 - DATA COLLECTION

Simula contexts for simulation usually contain several classes and procedures facilitating unobtrusive data collection and reporting - see Birtwistle [4, 5], Bredrup [10], Fishman [11], Franta [12], Landwehr [13] and Palme [14]. In this section, we tackle the problem another way by arranging for values of variables to be collected in external files (one file per variable, sectioned into epochs) for later analysis after

the program has been completed.

The aim of this level is to allow for an arbitrary number of variables and an arbitrary number of epochs. The user is expected to run his simulation model using such code as:

```
create the data collection structures
     and open their associated files;
FOR n epochs DO
BEGIN
  run next epoch and collect data;
MARK - give each open data collection
  file an end-of-epoch marker;
END;
CLOSE - close down all data files
  opened by the program.
```

MARK and OPEN are programmed as global routines in the body of DATA_COLLECTION which is listed below:

```
DES CLASS DATA_COLLECTION;
BEGIN
  REF(QUEUE)DATAQ;
  INTEGER PRE, FIELD;

  MEMBER CLASS DATA(TITLE, L);
    VALUE TITLE; TEXT TITLE; INTEGER L;
    VIRTUAL: PROCEDURE UPDATE;
  BEGIN
    REF(OUTFILE)O;
    PROCEDURE UPDATE(X); REAL X;
      O.OUTREAL(X, PREC, FIELD);
    O :- NEW OUTFILE(TITLE);
    O.OPEN(BLANKS(L));
    INTO(DATAQ);
  END***DATA***;

  PROCEDURE MARK;
  BEGIN
    REF(DATA)X;
    X :- DATAQ.FIRST;
    WHILE X =/= NONE DO
    BEGIN
      X.O.OUTIMAGE;
      X.O.OUTTEXT("++++");
      X.O.OUTIMAGE;
      X :- X.SUC;
    END;
  END***MARK***;

  PROCEDURE CLOSE;
  BEGIN
    REF(DATA)X;
    X :- DATAQ.FIRST;
    WHILE X =/= NONE DO
    BEGIN
      X.O.CLOSE;
      X :- X.SUC;
    END;
  END***CLOSE***;

  ASK("PRECISION", PREC, PREC > 4);
  ASK("FIELD WIDTH", FIELD, FIELD >
    (PREC + 5));
  DATAQ :- NEW QUEUE;
  INNER;
  CLOSE;
END***DATA COLLECTION***;
```

We cater for an arbitrary number of data collection devices by declaring

```
REF(QUEUE)DATAQ;
```

and take advantage of Simula's class body actions to enter each user generated data collection device into DATAQ automatically.

```
MEMBER CLASS DATA;
BEGIN
  ........
  INTO(DATAQ);
END***DATA***;
```

Each data collection object is tied to its own output file. On object creation, the files are opened and initialised (given a buffer). The file name and buffer length are class parameters.

The code for MARK and CLOSE is obvious. The initial class body actions prompt the user to supply a data precision indicator and a data field width and then generate DATAQ into which all user created data collection objects will insert themselves. INNER then causes the user program to be executed. On the latter's completion, control returns to this level and the final class actions guarantee that all files opened are correctly closed. (Control will then return back one level to finish off the final actions of DES).

No matter how many data objects are generated, each gets its own file, which is opened and supplied with a buffer. To use a data object, create it with a statement like

```
D :- NEW DATA ("THRU", 120);
```

which will open a file THRU, supply a buffer of length 120 characters, and insert D into DATAQ (where it can be located for the MARK and CLOSE operations). New values are appended to the file by calls typified by

```
D.UPDATE(TIME-START);
```

Each output value in character format to a fractional precision of PREC places an exponent in a field of total width FIELD places. If this default is inappropriate (for example, you may wish to write out real values in a different way, or write out quite different data types), then you need not write a completely new class for only its update routine would differ. Variants of DATA may now be declared which build upon DATA and supply the new update routine.

e.g.

```
DATA CLASS PAIR;
BEGIN
  INTEGER K;

  PROCEDURE UPDATE(X); VALUE X;
  BEGIN
    K := K + 1;
    O.OUTINT(K, 4);
    O.OUTFIX(X, 5, 12);
    O.OUTIMAGE;
  END***UPDATE***;
END***PAIR***;
```

which outputs sequentially numbered real values in fixed format one per line. PAIR objects being prefixed by DATA are automatically entered into DATAQ upon generations. The VIRTUAL specification at the DATA level has no effect for a DATA object. But for a sub-class like PAIR where a revised version of update is given, it causes the default routine at the DATA level to be erased and be replaced by the new version. The idea is very versatile and powerful. For it means that any context default specified as virtual may be overridden by a user. All the latter has to do is supply a new version of the routine with the same name. Such an overridden default becomes completely inaccessible, and its every occurrence is replaced by a call on the new version.

The final Simula feature we wish to introduce is data protection. At this level for example, we want a user to have access to MARK, CLOSE and UPDATE, but if he masochistically or inadvertantly reassigned DATAQ, PREC, FIELD, or O the result could waste the entire run. This can be prevented by specifying these variables as HIDDEN and PROTECTED where they are declared. In our case, add in globally

```
HIDDEN PROTECTED DATAQ, PREC, FIELD;
```

and local to class DATA

```
HIDDEN PROTECTED O;
```

All access to these quantities outside the body of DATA_COLLECTION is now forbidden and will be spotted at compile time. A readable account of HIDDEN and PROTECTED is given by Palme [15]. It is an interesting exercise to rewrite our previous levels giving read only status to the queueing pointers (SUC, PRED, FIRST, LAST(Q)) and completely hiding the event list pointer EL etc.

LEVEL 5 - RING MODEL

We start off with a brief description of the Cambridge Ring architecture. A more detailed account is given in [16]. A ring consists of a number of transmitter/receiver stations, each station having a unique address in the range 1 through 254. Apart from a common address, the transmitter and receiver in a station are logically independent. A small fixed number of packets circulate continually round the ring. The number, typically 1, 2 or 4, depends on the ring delay. To transmit a packet, registers within the station are loaded by the host computer with an 8 bit destination address and 16 bits of data. The next empty ring packet arriving at the station is marked as being in use, and has the destination, source and data fields filled in. In the normal course of events, the packet circulates until it reaches the destination, where the data is extracted and the packet is modified to indicate successful delivery. The packet, still marked 'in use', continues to circulate until it is delivered back to the source, which marks it as free, and indicates the success of the data delivery to the host.

Several things may go wrong with this procedure. If the destination does not exist, or is not switched on, the packet will arrive back at the source without being modified by an away station. The source station can recognise this and notifies the host station that the packet was ignored. The destination host may be unable to keep up with the transmitter (the host may have other work to do and not be dedicated to the ring). If a packet arrives at a receiver before the destination host has dealt with the previous packet from the receive logic, then the station will mark the packet as rejected because the receive station was busy. This too will be notified to the sending host. Finally each receiver is equipped with a sourceselect register which allows the receiver to discriminate amongst stations from which it is willing to receive data. If the sourceselect register is set to 0, the receiver will reject packets from any source; if it is set to 255, it will accept packets from any source; any other possible value results in data packets being accepted from only the source whose ring address corresponds to the value in the source select register. An indication (unselected) is given to a transmitter if a packet is rejected in this case.

Judicious use of the source select register can greatly simplify the structure of the receiving software. A receiver expecting a large block of data, requiring many packets to transmit, can be set to receive only from one source, saving the software from having to concern itself with the problem of dealing with unexpected packets from other sources. The transmitter software, has of course, to be able to deal with the repercussions of this strategy.

Simulation aims

The purpose of this work was to build an accurate model of the ring software as it actually is, and then to experiment with different hardware/software strategies. E.g. fewer but longer packets (of length 4, 8, or 16 data bytes instead of 2); or ignoring sourceselect and allowing several senders to be received at a time. An accurate first model was built by examining the implementation source code, following the same branching logic in our model and incorporating exact timings for these branches. In practise, once the basic model was accepted, the modifications necessary for other strategies were only some ten or twenty lines.

Each node in a ring configuration is represented in the model by a STATION object. Each node works asynchronously and models the ring handling software itself. Three auxiliary processes are attached to each node. They too work concurrently. There is a transmitter T which delivers packets one by one to the destination station, and reports their acceptance status back to the sending station; a user U who arrives periodically with a block to transmit, and waits for it to be transmitted; and a clock interrupt CPUI which interrupts the station at regular intervals demanding service for higher level interrupts. These processes are documented one by one in the sequel.

The ring handler can best be described as a pair of independent processes, one transmitting packets and the other receiving packets. The transmit process accepts buffers handed down from higher level software and forwards them on to their destination two bytes at a time. The transmit process algorithm is:

```
TRANSMIT:
   await block to send;
   send header packet (includes no. of data
     packets, N);
   send receive port number;
   FOR k := 1 STEP 1 UNTIL N DO
     send data packet k;
   send check sum packet;
   return buffer to sender marked yea or
     nay;
   REPEAT;
```

The check sum is computed on the fly. It is accumulated as each data packet is sent. There are (N=3) packets per block, where N is the number of data packets. They are sent in strict sequence and the (k+1)st is not started until the kth has been accepted. Each packet is retried (up to a limit) until accepted. When the maximum retry level for an individual packet has been reached, the whole attempt to transmit is aborted. Also a timer is started when the header packet is first transmitted. A time-out occurring before the check sum packet has been both sent and accepted will also cause abortion of the block. When a block is aborted, the current buffer is returned to the sender marked with the reason for the failure, and the process returns to the head of its code and awaits the next buffer.

The receive block software process is:

```
RECEIVE:
   await arrival of a header.
     Set sourceselect and extract the
       packet length, N;
   get port packet.  Set P to port number;
   get buffer B;
   FOR k := 1 STEP 1 UNTIL N DO
     receive next data packet and copy to B;
   get check sum packet;
   hand on B to port P;
   request next buffer;
   REPEAT;
```

Accounting for the several possible failures must be imposed upon this pattern. The length of the incoming block, N, may be too long; there may be no buffer B available from the buffer manager when it is wanted (get buffer B); the checksum may fail; the port number may be invalid; the block may be timed out. Any of these will cause abortion of the attempt to receive. Then the buffer B (if any) is returned to the buffer manager and a request (a notification of intent, but not a seize) is made for another buffer, the sourceselect register is set back to 255, and the process awaits the arrival of the next header packet (non-headers are received but ignored).

This process description gives a neat explanation of the receive/transmit process roles, but node software is actually organised as a polling loop. The ring hardware can deliver packets at great speed, so that although block arrival/transmission events are rare, when they occur there is a great flurry of activity in both the sending and the receiving nodes. In between these flurries, a node will be busy doing other work and only turns its attention to the ring when a header is received or a block is handed down. In out line, we have:

```
PROCESS CLASS STATION;
BEGIN
  do other work;
RING POLLING CYCLE:
  switch contexts;
  WHILE ring active DO
  BEGIN
    IF packet to send THEN send it;
    IF packet arrived THEN deal with it;
  END;
  switch contexts;
  REPEAT;
END***STATION***;
```

Whilst in one cycle of the loop, one packet may be sent and another be received. It is important to bear in mind that switching contexts is a very expensive operation (approx. 500 machine instructions), and that once block transmission has started, packets can be accepted/sent very rapidly (approx. 50 machine instructions each). It therefore pays to keep the node software inside the ring polling cycle whilst there is a reasonable probability of some activity going on. In the actual implementation, a count of successive idle loops is maintained. Only when this reaches 10 is the ring polling cycle left. N.B. an idle loop takes approx. 30 machine instructions.

Whilst in the ring polling cycle, lower level interrupts are ignored. Thus the tready bit for packet arrival is inspected when the ring polling software is ready, not on actual receipt. But higher level interrupts are still attended to and take slices out of the ring polling cycle (thus adding or deleting items in the valid port list, disk io's and clock interrupts).

Transmitter processes

When a station wishes to transmit the next packet, it first ascertains that the transmitter is not currently sending (tready = true), loads up the ring registers, and activates the transmitter. The command SCHEDULE(T) sets the transmitter T to work at once and in parallel with the station. The transmitter has the work cycle:

```
tready := false;
X :- destination station;
hold an appropriate delay + send time;
IF X is down THEN return ignored ELSE
IF X.ring.source_select = 255 or me THEN
BEGIN
  IF NOT X.ready THEN
  BEGIN
    deliver packet;
    awaken X, if need be;
    return accepted;
  END ELSE return := busy;
END ELSE return unselected;
tready := true;
```

```
WAIT(TQ);
REPEAT;
```

The code is bracketted by assignments to tready which ensure that only one packet is being sent at a time. Hardware will delay sending if this is a retry packet (by 2 ring delays if oretry = 1, and by 16 of oretry > 1: a ring delay is taken as 0.005 milliseconds => 200,000 packets/sec). Then an empty ring packet is awaited.

The receive station, X, is found from a special register. If X.SOURCESELECT is 255 or the sender's address, the packet will be accepted if the software in X has taken care of the previous packet (X.rready = false); if not (X.rready = true), it is rejected and marked busy. If X.SOURCESELECT has been set to another station, the packet is marked unselected.

With ring status now being appropriately set, tready is set true and the transmit process goes to sleep in TQ. When next awakened by its own station, it will repeat the above cycle.

User processes

Class user plays the role of higher level software handing down blocks to be transmitted. A rough outline of a user is:

```
hand down next block, and awaken S if
  need be;
sleep until transmission over;
hold(think time);
REPEAT;
```

A user hands down a block to its station (S.npq := S.npq + 1), awakens S if S is idle (not in the ring cycle nor dealing with an interrupt) and then the user goes to sleep. Buffers for blocks are not modelled as such; we merely keep a count of how many requests are pending in S.npq. Once S has dealt with this block then S awakens the user who thinks for a while, and then hands down another block.

Since we have only one user per station, 0 <= S.PP <= 1. In future models, if we wish to attach several users to each station sending down different classes of message, a simple count will not be enough, for we will also wish to know who sent the message down. This can be solved by letting a user hand on a request block which includes a pointer to himself.

Clock interrupt processes

These processes mimic higher level interrupts arriving at regular intervals from the clock (50 per second). These interrupts take up a slice of cpu time, no matter what the node is currently engaged in. The class outline is:

```
hold(20.0);
delay S by the time to process the
  interrupts;
REPEAT;
```

The station S to which the cpu-interrupt is attached will be interrupted either when it is idle (doing other work) or when executing a hold (within the ring polling cycle, or perhaps switching contexts). Either way it doesn't matter because our interrupt routine defined at level 4 is geared to cater for both cases. We just code

```
PROCESS CLASS CPU_INTERRUPT(K); INTEGER K;
BEGIN
L: HOLD(20.0);
  INTERRUPT (S(K), t);
  GOTO L;
END***CPU INTERRUPT***;
```

Station processes

We elaborate 'IF packet to send THEN sent it;' to:

```
IF outputting THEN
BEGIN
  IF tready THEN
  BEGIN
    IF status accepted THEN try-packet-again
      ELSE
    IF op_count < op_length THEN send_next_
      packet ELSE
    IF check_sum THEN send-check_sum ELSE
    IF op_count >= op_length THEN finish_
      op_transfer;
  END;
END ELSE
IF PP > 0 AND tready THEN start_op_transfer;
```

If a block transmission has already started then outputting = true. If tready is false, the transmitter is dealing with a previous packet and there is nothing to do. If tready is true, then a packet is transmitted. If the last packet was not accepted (returned marked unselected, ignored or busy), then it is transmitted again. Otherwise, the last packet was accepted. The length of the current output block is op_length. Op_count gives the serial number of the packet we are currently trying to send. If op_count < op_length then we send another data packet. If op_count = op_length, then we send the check sum packet. If op_count > op_length, then the check sum has been accepted, i.e. the whole block successfully delivered, and we have to wrap up this transfer and prepare for the next.

The input 'IF packet arrived THEN deal with it' elaborates to:

```
IF inputting THEN
BEGIN
  IF rready THEN
  BEGIN
    IF ip_count  = ip_length THEN copy_to_
      buffer ELSE
      finish_ip_transfer;
  END;
END ELSE IF rready THEN start_ip_transfer;
```

Action is only taken if rready is true. When dealing with the last packet received, extracting the data bytes from the hardware ring register, sets rready to false. Now a new packet can be received by hardware. When hardware has an accepted packet, it loads it into ring rdata/source and then sets rready. The next time the station

input polls, it can accept this new packet.

If any away station sends a header and then times out before getting a response, it may send a second header. A second header reinitialises block acceptance. Otherwise, the port is remembered and data bytes loaded into a buffer two by two. Ip_count and ip_length are analogous to op_count and op_length. When the checksum has been received (ip_count = ip_length), the checksum is checked, port validated and the user is awakened.

Results

A complete description of the model, the experiments upon it and result obtained is slated for completion at the end of 1981 [17]. (We are currently awaiting more extensive and reliable traffic data to complete our model validation.) Results so far confirm the good behaviour of the current design under both light and heavy ring loadings, particularly with regard to real time response guarantees. Experiments with longer packet lengths (6-8) offer no advantage under light loading (1%), but do offer increased throughput for no extra station utilisation when the ring traffic is heavy (15%). Dispensing with source select and accepting packets from several sources (up to a maximum) certainly increases throughput, but at the cost of much extra work by a receiving station. This strategy would seem to be useful only with stations dedicated to ring activity or very much faster than the PDP 11/40 stations currently in use.

ACKNOWLEDGEMENTS

The author spent the summer of 1981 at the Edinburgh Regional Computing Centre as the recipient of an SRC Visiting Fellowship. The Demos model sketched in level 5 of this presentation is based upon its PDP 11/40 implementation written by B. Gilmore and S. Binns. Bill May and Brian Ritchie were co-developers of the model.

REFERENCES

1) O. J. Dahl, B. Myhrhaug and K. Nygaard, Simula 67 Common Base Language, NCC publication, 3rd Edition, 1981.

2) O. J. Dahl and K. Nygaard, Simula A language for the programming and description of discrete event systems, NCC publication, 5th Edition, 1967.

3) K. Nygaard and O. J. Dahl, The development of the Simula languages, ACM Conference on the History of Programming Languages, LA, 1979.

4) G. M. Birtwistle, Discrete Event Modelling On Simula, Macmillan Press, 1979. Distributed in North America by Gage Publishing, Ontario.

5) G. M. Birtwistle, Demos Reference Manual, 2nd Edition, 1981. (Available from the author.)

6) O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured programming, Academic Press, Chapter 3, 1972.

7) J. Vaucher and D. Davey, Self-optimising partitioned sequencing sets for discrete event simulation, INFOR, vol. 18, no. 1, pp. 41-61, 1980.

8) O. J. Dahl and R. Jonassen, Analysis of an algorithm for priority queue administration, BIT, vol. 15, no. 4, pp. 409-422, 1976.

9) W. R. Franta and K. Maly, An efficient data structure for the simulation event set, CACM, vol. 20, no. 8, pp. 596-602, 1977.

10) B. Bredrup, A report generator for Simula, ASU Conference, Brighton, 1975.

11) G. S. Fishman, Concepts and methods in discrete event simulation, Wiley, 1973.

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

13) C. Landwehr, Abstract data types in Simula 67, TOPLAS, vol. 3, no. 1, 1981.

14) J. Palme, Putting statistics into a Simula program, FOA Report C 10030-M3(E5), 1975.

15) J. Palme, A new feature for module protection in Simula, Simula Newsletter, vol. 4, no. 1, 1976.

16) M. V. Wilkes and R. M. Needham, The Cambridge Digital Communication Ring, Local Area Network Symposium, Boston, 1979.

17) G. M. Birtwistle, W. D. May and B. Ritchie, A simulation model of the Cambridge Ring. In preparation.

BIBLIOGRAPHY

K. G. Muller, On the Simulation of Generalised Activity Networks in Simula, Simula Newsletter, vol. 2, no. 3, 1974.

D. O'Sullivan, Simula used for production planning, Simula Newsletter, vol. 2, no. 4, 1974.

P. A. Houle and W. R. Franta, On the structural concepts of Simula and simulation modelling, Simula Newsletter, vol. 3, no. 2, 1975.

J. Cunningham and R. Sim, Continuous simulation in Simula, Simula Newsletter, vol. 4, no. 2, 1976.

F. Schumacher, Modeling of discrete systems with extended Petri Nets, Simula Newsletter, vol. 4, no. 3, 1976.

O. Leringe, A simulation study of ice breakers, Simula Newsletter, vol. 4, no. 3, 1976.

B. Unger, Oasis a Simula extension for systems software and simulation, Simula Newsletter,

vol. 5, no. 2, 1977.

J. Palme, Moving pictures show simulation to the user, Simula Newsletter, vol. 5, no. 3, 1977.

L. Larsson, Specifying a new operating system in Simula, Simula Newsletter, vol. 5, no. 3, 1977.

D. Belsnes, K. Bringsrud, and E. Lovdal, The use of Simula for real time system implementation, Simula Newsletter, vol. 6, no. 3, 1978.

L. Larson, I. Lekteus, I. Sjors, Condis class basic flying, Simula Newsletter, vol. 9, no. 3, 1981.

K. Helsgaun, Disco a simula based language for continuous combined and discrete simulation, Simulation, vol. 35, no. 1, 1980.

E. McQuade, A. M. Salih, and H. J. Gray, Simulation of a telecommunications multiprocessor switching system, Simulation, vol. 36, no. 5, 1981.