DISCRETE EVENT SIMULATION WITH DEMOS

Graham Birtwistle*

Department of Computer Science
University of Calgary

Paul Luker*

Department of Computer Science
University of Bradford

Abstract

Demos is a portable, strongly typed, extensible, process oriented, discrete event simulation language embedded in Simula. It extends Simula with a library of standard process synchronisation devices together with facilities for data collection, random number generation, tracing and automatic reporting. We review the synchronisation features of Demos programs and the structure of Demos programs and then illustrate modelling in Demos by a set of graded examples.

Keywords: Discrete event simulation. Process. Simula. Demos.

INTRODUCTION

Demos (1, 2, 3) is a discrete event simulation package hosted in Simula (4, 5). To Simula's general purpose algorithmic, data description, and file handling capabilities, Demos adds:

1) a simulation clock, an event list, and scheduling
2) the notion of an entity (= process) which can carry out tasks over a period of simulated time
3) data collection devices which enable data streams to be recorded unobtrusively
4) random number generators which automatically produce well-spread seeds
5) debugging facilities which include event list and queue snapshotting routines and event tracing
6) automatic reporting on the usage of all user created devices.

Demos has been implemented as a 2,000 line Simula context (prefixed class). Contexts are library units - they allow several interrelated ideas (e.g. clock time, event list, scheduling, snapshotting) to be externally compiled together and made accessible under a single name. In addition, external compilation ensures consistency and makes for rapid compilation. A 100 line Simula program backed up by a 2,000 line context costs scarcely more to compile than a 100 line Simula program without a context. Demos is made available to a user program by an external declaration, viz.

```
     EXTERNAL CLASS DEMOS,
```

and is called into use when it prefixes a user block

```
     DEMOS
        begin
            concepts of DEMOS are available here;
        end
```

DEMOS declarations are valid inside the block it prefixes. Inside that block, Demos statements may be freely intermingled with Simula statements, after all, Demos is written entirely in Simula.

Demos was released in 1979 and since then has been applied to a range of industrial problems including plant layout, transportation, shipping, and in the oil and gas industries. Because of its process base, it has also been used to help design software for local area networks, long haul networks and a telephone switching system.

In section 1 of this paper we introduce the entity, the structure of Demos programs, and introduce four basic synchronisations. In section 2 we present a number of Demos models.

*This work was supported in part by an NSERC grant.

DEMOS FACILITIES

Major components in Demos simulations are modelled as entities. Components of the same class are given a common declaration

```
ENTITY CLASS name;
BEGIN
   data;
   actions;
END;
```

The declaration reflects all the features of this class of object deemed to be relevant; both its physical characteristics and the actions it carries out as it wends its own way through the model.

Objects of the class are created and scheduled by

```
NEW name.SCHEDULE (delay)
```

When their action sequences are exhausted, entities are automatically deleted by the Simula system.

There are many ways in which entities can interact together: they may compete for resources, pass messages, cooperate over a period of time, or interrupt another entity's expected action sequence. There is a Demos device for each of these synchronisations.

RESources parallel GPSS's facility and storage and are used to implement mutual exclusion. For example, the code below creates PHONE as a resource of size 1.

```
PHONE :- NEW RES("TELEPHONE", 1);
```

An entity wishing to use the phone follows the protocol

```
PHONE.ACQUIRE(1);
HOLD(conversation time);
PHONE.RELEASE(1);
```

Acquire/Release parallel GPSS's enter/leave commands. Acquire blocks the requester until it is his turn to use the phone; release will awaken the first blocked entity, if any. RES objects may be initialised to values greater than 1, and acquired and released in chunks of any positive (integer) size.

Message passing is handled by BIN devices. A typical initialisation statement is

```
MESSAGES :- NEW BIN("MESSAGES SENT", Ø);
```

MESSAGES is used below to handle the synchronisation between a sender process and a receiver process - it makes sure that the receiver is blocked if no messages are there to be taken

```
SENDER:                         RECEIVER:
   produce new message;            MESSAGES.TAKE(1);
   MESSAGES.GIVE(1);               read this message;
   REPEAT;                         REPEAT;
```

Messages are usually handled by resources in other simulation languages. This is undesirable because tight checks can be made inside the release routine of RES objects to make sure that the releaser of m chunks has previously acquired at least m chunks of that resource. Such a check is meaningless for the producer/consumer synchronisation above. Furthermore, RES and BIN reports should be differently styled in order to reflect their different application.

Usually it is appropriate to let an entity needing several extra resources for its next phase to acquire them one by one. But this is not always so. For example, a crane may pick up from area A or from area B. When idle, it cannot predict where its next task will be dumped - it has to wait until there is a task and then go to the appropriate area and handle it. Demos provides a condition queue for this purpose. In the program segment below, we model the number of tasks awaiting handling by BINS AREA_A and AREA_B. The condition queue is initialised by

```
Q :- NEW CONDQ("CRANE AWAITS TASK");
```

and code for the crane entity has the form:

```
Q.WAITUNTIL(AREA_A.AVAIL > 0 OR
            AREA_B.AVAIL > 0);
WHILE AREA_A.AVAIL > 0 DO
BEGIN
   AREA_A.TAKE(1);
   pick up a load and dump it;
END;
WHILE AREA_B.AVAIL > 0 DO
BEGIN
   AREA_B.TAKE(1);
   pick up a load and dump it;
END;
REPEAT;
```

RES's and BIN's both have a local function AVAIL which returns how much is currently free. Once the crane has chosen a loading area, the while loops ensure that it exhausts one pick up area before checking the other one.

The final common synchronisation we present is the rendezvous (ADA's only built-in synchronisation). If entities wish to cooperate for a while, it is most convenient to let one be the master, the other(s) its slave(s).

A slave waits in a WAITQ by calling, for example

```
R_VOUS.WAIT;
```

The master entity coopts a slave from R_VOUS by executing

```
S :- R_VOUS.COOPT;
```

The call on COOPT blocks the master until a slave is available. When the period of cooperation is over, the master reschedules S by

```
     S.SCHEDULE(delay);
```

In addition to these synchronisations, Demos also supplies a portable random number generator with well-spread seeds (written in Simula), data collective devices, tracing and automatic reporting. Examples of their usage are given in (1). Their implementation can be found in (2).

In the next section, we present six Demos models using these synchronisations. Code for these models is presented in the format:

```
     BEGIN EXTERNAL CLASS DEMOS;
     DEMOS
        BEGIN
           declare static quantity names(resources,
              distributions, etc.);
           declare(dynamic) entities;
           create resource and distribution
              records;
           create entity streams;
           set simulation run time;
        END;
     END;
```

MODELLING IN DEMOS

Example 1. Gas Station.

Cars can be accommodated in one of two bays on either side of the single pump at a gas station. Cars arriving for gas drive to a free bay if there is one, otherwise, they form a single queue for pump bays. On acquiring a free bay, a car moves up to occupy it and then waits for the use of the pump.

The bays and the pumps are resources for which cars compete. These are modelled naturally in DEMOS using RES objects. The time taken for cars to move into a bay is constant, but the times taken to fill up the tank and pay the attendant are to be determined from random number distributions. Another random number distribution is used by each car to determine the arrival time of the next one. Data for the model is obvious from the program text.

The life history of a car is described as an entity in DEMOS. We give this life history first:

```
     ENTITY CLASS CAR;
     BEGIN
        NEW CAR("CAR").SCHEDULE(NEXT.SAMPLE);
        BAY.ACQUIRE(1);
        HOLD(5.0);
        PUMP.ACQUIRE(1);
        HOLD(FILL.SAMPLE);
        PUMP.RELEASE(1);
        HOLD(PAY.SAMPLE);
        BAY.RELEASE(1);
     END**CAR**;
```

The driving program for this simulation is:

```
     BEGIN EXTERNAL CLASS DEMOS;
     DEMOS BEGIN
        REF(RES) BAY, PUMP;
        REF(RDIST) NEXT, FILL, PAY;

        ENTITY CLASS CAR.....;

        BAY  :- NEW RES("BAY", 2);
        PUMP :- NEW RES("PUMP", 1);
        NEXT :- NEW NEGEXP("NEXT.CAR", 0.01667);
        FILL :- NEW UNIFORM("FILL TIME", 90.0,
                   15.0);
        PAY :- NEW UNIFORM("PAY TIME", 12.0, 2.0);
        NEW CAR("CAR").SCHEDULE(0.0);
        HOLD(60.0 * 60.0);  !simulation run length;
      END;
     END;
```

The simulation begins with one car just arriving and continues over one hour (3600 seconds). Initialisation of the RES objects shows how the quantity of a resource is defined.

Example 2. Conveyor belt.

A production line involves 5 servers stationed along a conveyor belt. Items to be serviced arrive at a mean rate of 4 per minute (NEGEXP distributed). If unserviced they are carried along the conveyor passing a server every minute. If an item reaches an idle server, the item is picked off the conveyor, serviced (which takes UNIFORM 0.8->1.2 minutes) and stored away. If an item passes all the servers, then it is recirculated and reappears in front of server 1 after a delay of 5 minutes. The simulation runs for 480 minutes, and notes the number of recirculated items.

This example introduces an additional DEMOS data collection device, the count. It is used here to record the number of items processed, and the number which have to be recirculated. Counts are updated by calling their local procedure UPDATE.

The solution clearly demonstrates the value of embedding the simulation facilities within a standard simulation language. For example, there are five servers in the model, each of which is a resource. It is natural to combine the ARRAY of Simula with the RES of Demos to produce an array of RES objects. The code is also made easier to write and more concise through the use of standard SIMULA statements such as FOR.

We show the life history of an item in its ENTITY declaration below. A loop is used to step the item through the servers, with the controlled variable, K, being used to index the appropriate server resource. When an item has been serviced, any remaining passes through the loop and the recirculation control statements are skipped. If an item fails to receive service then it goes round again. REPEAT returns to the label LOOP.

```
      ENTITY CLASS ITEM;
      BEGIN INTEGER K;
        NEW ITEM("ITEM").SCHEDULE(ARR.SAMPLE);
      LOOP:
        FOR K := 1 STEP 1 UNTIL 5 DO
        BEGIN
          HOLD(1.0);
          IF SERVER(K).AVAIL = 1 THEN
          BEGIN
            SERVER(K).ACQUIRE(1);
            HOLD(SERVE.SAMPLE);
            SERVER(K).RELEASE(1);
            DONE.UPDATE(1);
            GOTO L;
          END;
        END;
        HOLD(4.0);
        AGAIN.UPDATE(1);
        REPEAT;
   L:END***ITEM***;
```

The driver program illustrates how the array of resources is declared and initialised, with use again being made of a FOR statement.

```
      BEGIN EXTERNAL CLASS DEMOS;
        DEMOS
          BEGIN REF(RES)ARRAY SERVER(1:5);
            REF(RDIST)ARR, SERVE;
            REF(COUNT)AGAIN, DONE;
            INTEGER K;

            ENTITY CLASS ITEM.....;

            ARR   :- NEW NEGEXP("ARRIVALS", 3.0);
            SERVE :- NEW UNIFORM("SERVICE", 0.8,
                                  1.2);
            AGAIN :- NEW COUNT("RE-CYCLES");
            DONE  :- NEW COUNT("ITEMS DONE");
            FOR K :- 1 STEP 1 UNTIL 5 DO
              SERVER(K) :- NEW RES(EDIT("SERVER",
                                       K), 1);
            NEW ITEM("ITEM").SCHEDULE(0.0);
            HOLD(480.0);
          END;
      END;
```

Example 3. Production line.

A small production line has three stages: the first assembles the inner and outer rings of bearings, the second greases the assemblage, the third packs them two to a box (the packers take two greased assemblages at a time). There are 3 assemblers, 1 greaser, and 2 packers. Timings (in minutes) are obvious from the program listing.

In the DEMOS program for this model, the assembler, greaser and packer are modelled as entities. No operator can work if the raw material required is not available. For example, the assemblers need an inner and an outer ring in order to assemble them. The assemblage is then passed on to the greaser, who cannot proceed without it. On completion of greasing, the greaser passes the greased units on to the packers, each of which requires two units in order to perform a packing operation. This producer/consumer synchronisation is modelled by the BIN of DEMOS. Two additional entities are included to generate the supply of rings, one for the outers and the other for the inners.

The complete program is shown below:

```
BEGIN EXTERNAL CLASS DEMOS;
  DEMOS
    BEGIN REF(COUNT)DONE; INTEGER K;
      REF(BIN)ASSEMBLER,GREASED,PACKED,
              INNERS,OUTERS;
      REF(RDIST)NEXTI,NEXTO,ASSEMBLE,
                GREASE,PACK;

      ENTITY CLASS IRING;
      BEGIN INNERS.GIVE(1);
        HOLD(NEXTI.SAMPLE);
        REPEAT;
      END***IRING***;

      ENTITY CLASSORING;
      BEGIN OUTERS.GIVE(1);
        HOLD(NEXTO.SAMPLE);
        REPEAT;
      END***OUTER RINGS***;

      ENTITY CLASS ASSEMBLER;
      BEGIN INNERS.TAKE(1); OUTERS.TAKE(1);
        HOLD(ASSEMBLE.SAMPLE);
        ASSEMBLED.GIVE(1);
        REPEAT;
      END***ASSEMBLER***;

      ENTITY CLASS GREASER;
      BEGIN ASSEMBLED.TAKE(1);
        HOLD(GREASE.SAMPLE);
        GREASED.GIVE(1);
        REPEAT;
      END***GREASER***;

      ENTITY CLASS PACKER;
      BEGIN GREASED.TAKE(2);
        HOLD(PACK.SAMPLE);
        DONE.UPDATE(1);
        REPEAT;
      END***PACKER***;

      ASSEMBLED :- NEW BIN("ASSEMBLED, 0);
      GREASED   :- NEW BIN("GREASED",   0);
      PACKED    :- NEW BIN("PACKED",    0);
      INNERS    :- NEW BIN("INNERS",   10);
      OUTERS    :- NEW BIN("OUTERS",   10);
      DONE      :- NEW COUNT("JOBS DONE");
      NEXTI     :- NEW NEGEXP("INNER", 6.0);
      NEXTO     :- NEW NEGEXP("OUTER", 6.0);
      ASSEMBLE  :- NEW NORMAL("ASSEMBLE",
                              0.5, 0.1);
      GREASE    :- NEW CONSTANT("GREASE",
                                0.16);
      PACK      :- NEW NORMAL("PACK", 0.6,
                              0.1);
      NEW IRING("I-RING").SCHEDULE(0.0);
      NEW ORING("O-RING").SCHEDULE(0.0);

      FOR K := 1 STEP 1 UNTIL 3 DO
       NEW ASSEMBLER("ASSEMBLER").SCHEDULE
                     (0.0);
      NEW GREASER("GREASER").SCHEDULE(0.0);
```

```
        FOR K := 1 STEP 1 UNTIL 2 DO
         NEW PACKER("PACKER").SCHEDULE(0.0);
        HOLD(480.0);
      END;
  END;
```

Example 4. Printing shop.

Each job for a printing shop consists of a number of pages, each of which consists of a number of components which comprise photographs, drawings and typed text.

The components of a page will each require the services of an appropriate operator: the photo process will require a photographer; the drawing process will require an artist and the type process will require a setter. These three types of operators are modelled as RES objects, with the three associated processes obviously being entities.

When all the components of a page have been prepared, the services of a platemaker (another RES) are required in order to assemble the page.

One essential feature of the simulation is to ensure that page assembly for any page is delayed until all its components are ready. Similarly, no job can be completed until all its constituent pages have been assembled. The required synchronisations are affected in this example by the BIN object of DEMOS. Each job has its own associated BIN object called JOB_OFFSPRING. When a job sets up its constituent pages, a pointer, c, of type REF(BIN), is passed to each page so that as each page is completed, it can signal this event to its own master job. By issuing a call of TAKE(PAGES), a job is blocked at that point until all PAGES constituent pages have been completed and issued a GIVE(1) to the job's BIN.

Exactly the same synchronisation is used between each page and its sub-tasks.

A new feature used in the program below is a further data collection device, the HISTOGRAM. This can provide a very concise and readily assimilated report on some aspect of a simulation behaviour.

```
  BEGIN EXTERNAL CLASS DEMOS;
    DEMOS
    BEGIN
      REF(RES)PLATEMAKER, PHOTOGRAPHERS,
              ARTISTS, SETTERS;
      REF(RDIST)NEXT JOB, PLATE_TIME,
                PHOTO_TIME, DRAWING_TIME,
                TYPE_SETTING_TIME;
      REF(IDIST)NR_PHOTOS, NR_DRAWINGS, NR_TYPES,
                SIZE;
      REF(HISTOGRAM)JOBTHRU;

      ENTITY CLASS JOB;
      BEGIN
        INTEGER PAGES, K;
        REAL ARRTIME;
        REF(BIN)JOB_OFFSPRING;

        NEW JOB("JOB").SCHEDULE(NEXT JOB.SAMPLE);
        ARRTIME    := TIME;
        PAGES      := SIZE.SAMPLE;
        OFFSPRING :- NEW BIN("PAGES", 0);
        FOR K := 1 STEP 1 UNTIL PAGES DO
          NEW PAGE("PAGE", JOB_OFFSPRING).
                   SCHEDULE(0.0);
        JOB_OFFSPRING.TAKE(PAGES);
        JOBTHRU.UPDATE(TIME-ARRTIME);
      END***JOB***;

      ENTITY CLASS PAGE(C); REF(BIN)C;
      BEGIN
        INTEGER PHOTOS, DRAWINGS, TYPING, ITEMS,
            K;
        REF(BIN)OFFSPRING;
        PHOTOS    := NR_PHOTOS.SAMPLE;
        DRAWINGS  := NR_DRAWINGS.SAMPLE;
        TYPING    := NR_TYPES.SAMPLE;
        ITEMS     := PHOTOS + DRAWINGS + TYPING;
        OFFSPRING :- NEW BIN("SUB-JOB", 0);
        FOR K := 1 STEP 1 UNTIL PHOTOS DO
          NEW PHOTO("PHOTO", OFFSPRING).SCHEDULE
                   (0.0);
        FOR K := 1 STEP 1 UNTIL DRAWINGS DO
          NEW DRAWING("DRAWING", OFFSPRING).
                   SCHEDULE(0.0);
        FOR K := 1 STEP 1 UNTIL TYPING DO
          NEW TYPE("TYPE",OFFSPRING).SCHEDULE
                   (0.0);
        OFFSPRING.TAKE(ITEMS);
        PLATEMAKER.ACQUIRE(1);
        HOLD(PLATE_TIME.SAMPLE);
        PLATEMAKER.RELEASE(1);;
        C.GIVE(1);
      END***PAGE***;

      ENTITY CLASS PHOTO(C); REF(BIN)C;
      BEGIN
        PHOTOGRAPHERS.ACQUIRE(1);
        HOLD(PHOTO_TIME.SAMPLE);
        PHOTOGRAPHERS.RELEASE(1);
        C.GIVE(1);
      END***PHOTO***;

      ENTITY CLASS DRAWING(C); REF(BIN)C;
      BEGIN
        ARTISTS.ACQUIRE(1);
        HOLD(DRAWING_TIME.SAMPLE);
        ARTISTS.RELEASE(1);
        C.GIVE(1);
      END***DRAWING***;

      ENTITY CLASS TYPE(C); REF(BIN)C;
      BEGIN
        SETTERS.ACQUIRE(1);
        HOLD(TYPE_SETTING_TIME.SAMPLE);
        SETTERS.RELEASE(1);
        C.GIVE(1);
      END***TYPE***;

      PLATEMAKER     :- NEW RES("PLATEMAKER", 1);
      PHOTOGRAPHERS  :- NEW RES("F'GRAPHERS", 3);
      ARTISTS        :- NEW RES("ARTISTS", 3);
      SETTERS        :- NEW RES("SETTERS", 3);
      JOBTHRU        :- NEW HISTOGRAM("JOB TIMES",
                        0.0, 20.0, 10);
      PLATE_TIME     :- NEW NORMAL("PLATE SETTING",
                        10.0, 1.0);
      PHOTO_TIME     :- NEW NORMAL("PHOTOGRAPH-
                        ING", 10.0, 1.0);
```

```
         DRAWING_TIME   :- NEW NORMAL("DRAW PIC-
                           TURE", 10.0, 1.0);
         TYPE_SETTING_TIME :- NEW NORMAL ("SET
                              TYPE", 10.0, 1.0);
         NEXT JOB       :- NEW NEGEXP("NEXT JOB",
                           0.010);
         SIZE           :- NEW RANDINT("SIZE", 2,
                           4);
         NR_PHOTOS      :- NEW RANDINT("PHOTOS PER
                           PAGE", 2, 4);
         NR_DRAWINGS    :- NEW RANDINT("DRAWINGS
                           PER PAGE", 2, 4);
         NR_TYPES       :- NEW RANDINT("TYPE FONTS",
                           1, 2);
         NEW JOB("JOB").SCHEDULE(NOW);
         HOLD(480.0);
       END;
     END;
```

Example 5. Dining philosophers.

Five philosophers are seated around a circular table which contains an inexhaustible supply of spaghetti within easy reach of all at its centre. Between each pair of adjacent philosphers is a fork. The philosophers have a simple life style

```
LOOP:  think;
       eat;
       REPEAT;
```

In order to eat, a philosopher requires both the fork on his left and the fork on his right. Each thus competes for resources with his immediate neighbours. The orgy is to last for 3 hours.

In this model we represent the forks by REF(RES)ARRAY FORK(1:5) and initialise by

```
FOR K := 1 STEP 1 UNTIL 5 DO
  FORK(K) :- NEW RES(EDIT("FORK", K), 1);
```

We number the forks so that philosopher n finds FORK(n) on his left and FORK(n+1) on his right (FORK(1) in the case of philosopher 5). Then CLASS PHILOSOPHER can be written

```
ENTITY CLASS PHILOSOPHER(N); INTEGER N;
BEGIN REF(RES)L, R;
  L :- FORK(N);
  R :- FORK(IF N=5 THEN 1 ELSE N+1);
LOOP:
  HOLD(THINK.SAMPLE);
  Q.WAITUNTIL(L.AVAIL > 0 AND R.AVAIL > 0);
  L.ACQUIRE(1); R.ACQUIRE(1);
  HOLD(EAT.SAMPLE);
  L.RELEASE(1); R.RELEASE(1);
  Q.SIGNAL;
  REPEAT;
END***PHILOSOPHER***;
```

Usually a conditionqueue contains entities all waiting upon the same condition so that when one attempted awakening fails, there is no point in trying to awaken the rest of the queue. Occasionally, as here, we have several entities waiting upon different conditions in the same queue (it seems profligate to put each philosopher in his own queue). Besides waituntil and coopt, a condq has a boolean attribute ALL which when set to true ensures that a call on signal will test each and every condition queue member even though they have different conditions and some may fail. This gives us what we want: by using a single CONDQ with ALL set, philosophers are queued ranked according to their time of entry (their priorities are all zero) and every member of the condition queue will be tested. The complete program reads

```
BEGIN EXTERNAL CLASS DEMOS;
DEMOS
  BEGIN INTEGER K;
    REF(RES)ARRAY FORK(1:5);
    REF(IDIST)THINK, EAT;
    REF(CONDQ)Q;

    ENTITY CLASS PHILOSOPHER(N);........;

    Q     :- NEW CONDQ("AWAIT EAT");
    Q.ALL := TRUE;
    THINK :- NEW RANDINT("THINK", 20, 30);
    EAT   :- NEW RANDINT("EAT", 10, 20);
    FOR K := 1 STEP 1 UNTIL 5 DO
      FORK(K) :- NEW RES(EDIT("FORK", K), 1);
    TRACE;
    FOR K := 1 STEP 1 UNTIL 5 DO
      NEW PHILOSOPHER("P", K).SCHEDULE(0.0);
    HOLD(180.0);
  END;
END;
```

Example 6. Information system.

This problem has been used in several papers and books and so provides an interesting comparison (see for example (6)). The model represents an information retrieval system with a number of remote terminals each capable of interrogating a single processor (CPU). A customer with a query arrives at one or other of the terminals and queues, if necessary, to use it. The terminals are physically far apart and so no queue jumping is possible. When the terminal is free, the user keys in his request, and then signals his presence to the system. He then awaits his reply.

The queries are picked up by a scanner which looks at each terminal in turn. If there is no query outstanding, the scanner rotates on to the next terminal in turn. If there is a query, the scanner locks on to that terminal and does not rotate further until it has succeeded in copying the query to a buffer unit capable of holding three such queries at a time. The copying process is blocked if no buffer slot is available. When the copying has been completed, the scanner starts to rotate again and leaves the cpu to deal with the request.

The CPU processes the query and places the answer in the buffer slot overwriting the query. The answer is returned to the terminal by the buffer unit (without using the scanner) and then that buffer slot is freed. The customer reads the reply and then quits his terminal.

We describe the model in terms of two entity classes - QUERY and SCANNER. CLASS SCANNER describes the actions of the real scanner as it rotates from terminal to terminal. If the current terminal has no request pending, the scanner moves on, otherwise it awaits a buffer and transfers the query into it before rotating on.

CLASS QUERY describes the roles of the customer, his request and the reply. On arrival, the customer keys in his request and awaits his reply. The query (same object) is eventually passed from the terminal to a CPU buffer by the scanner. Since these actions are already described inside CLASS SCANNER, they are not repeated here - instead each QUERY object is coopted by the scanner for this part of its life. When it resumes its own life history, the query is already in a buffer. Then the query is processed by the CPU and the reply sent back to the appropriate terminal for reading.

Now that we have outlined the roles to be played by scanner and query objects and decided upon their interactions, we can tackle their declarations separately. We begin by detailing CLASS SCANNER.

The scanner rotates from terminal 1 to terminal 6, and then repeats. At each terminal N (1 <= N <= 6), it rotates (HOLD(0.0027)) and then tests to see if any query is pending (B := REQUESTQ (N).LENGTH > 0). This test also takes 0.0027 minutes to complete. If there is a request, then the scanner locks on to that terminal, acquires a buffer (by BUFFERS.TAKE(1). This may imply a delay.), and then transfers the request to the buffer (HOLD(0.0117)). Once this has been accomplished, the query is then released (Q.SCHEDULE(0.0)) and the scanner is free to rotate to the next terminal.

The full declaration is

```
ENTITY CLASS SCANNER;
BEGIN INTEGER N; BOOLEAN B: REF(QUERY)Q;
  FOR N := 1 STEP 1 UNTIL 6 DO
  BEGIN
    HOLD(0.0027);
    B := REQUESTQ(N).LENGTH > 0;
    HOLD(0.0027);
    IF B THEN
    BEGIN
      Q :- REQUESTQ(N).COOPT;
      BUFFERS.TAKE(1);
      HOLD(0.0117);
      Q.SCHEDULE(0.0);
    END;
  END;
  REPEAT;
END***SCANNER***;
```

A query object first generates the next query object, notes its arrival time (T := TIME), and then chooses its terminal (N). That terminal is then seized (by TERMINAL(N).ACQUIRE(1), perhaps after a wait), and the request is keyed in (HOLD(KEYIN. SAMPLE)). Now the query waits passively in REQUESTQ(N).

When it becomes active again, the scanner has acquired a buffer slot on its behalf and copied the request into that buffer. The request is now processed (HOLD(PROCESS.SAMPLE)) and the reply returned to the appropriate terminal (HOLD(0.0397)). After the transfer has been completed, the buffer slot is freed (BUFFERS.GIVE(1)), possibly unblocking the scanner. Then the reply is read and the terminal vacated (TERMINAL(N).RELEASE(1)), which allows in the next query, if any. Finally, a histogram of through times (THRU) is updated by the elapsed time of this query through the system.

```
ENTITY CLASS QUERY;
BEGIN INTEGER N; REAL T;
  NEW QUERY("QUERY").SCHEDULE(ARRIVALS.SAMPLE);
  T := TIME;
  N := TERMINALS.SAMPLE;
  TERMINAL(N).ACQUIRE(1);
  HOLD(KEYIN.SAMPLE);
  REQUESTQ(N).WAIT;

  HOLD(PROCESS.SAMPLE + 0.0397);
  BUFFERS.GIVE(1);
  HOLD(READ.SAMPLE);
  TERMINAL(N).RELEASE(1);
  THRU.UPDATE(TIME-T);
END***QUERY***;
```

The driving program contains these definitions plus the various resources, queues and distributions and one histogram. It generates the scanner and the first query and runs for 60 minutes. Unspecified timings are obvious from the program listing.

```
BEGIN EXTERNAL CLASS DEMOS;
DEMOS
  BEGIN INTEGER K;
    REF(HISTOGRAM)THRU;
    REF(WAITQ)ARRAY REQUESTQ(1:6);
    REF(RES)ARRAY TERMINAL(1:6);
    REF(BIN)BUFFERS;
    REF(RDIST)ARRIVALS, KEYIN, PROCESS, READ;
    REF(IDIST)TERMINALS;

    ENTITY CLASS SCANNER........;

    ENTITY CLASS QUERY..........;

    ARRIVALS  :- NEW NEGEXP("ARR", 2.0);
    TERMINALS :- NEW RANDINT("TERMINALS",
                             1.6);
    KEYIN     :- NEW UNIFORM("KEYIN", 0.3,
                             0.5);
    PROCESS   :- NEW UNIFORM("PROCESS", 0.05,
                             0.10);
    READ      :- NEW UNIFORM("READ", 0.6, 0.8);
    THRU      :- NEW HISTOGRAM("THRU", 1.0,
                               11.0, 10);

    FOR K := 1 STEP 1 UNTIL 6 DO
    BEGIN
      REQUESTQ(K) :- NEW WAITQ(EDIT("REQUEST",
                               K));
      TERMINAL(K) :- NEW RES(EDIT("TERMINAL",
                               K), 1);
    END;

    BUFFERS :- NEW BIN("BUFFER", 3);
    NEW SCANNER("SCANNER").SCHEDULE(0.0);
    NEW QUERY("Q").SCHEDULE(0.0);
    HOLD(60.0);
  END;
END;
```

OUTPUT:

```
                              CLOCK TIME =     60.00
**********************************************************************
*                                                                    *
*                                R E P O R T                         *
**********************************************************************

                            D I S T R I B U T I O N S
                            *************************

TITLE       /    (RE)SET/   OBS/TYPE       /         A/          B/       SEED
arr                 0.000   122 NEGEXP           2.000                33427485
terminals           0.000   122 RANDINT              1           6    22276755
keyin               0.000   122 UNIFORM          0.300       0.500    46847980
process             0.000   122 UNIFORM      5.000&-02       0.100    43859043
read                0.000   121 UNIFORM          0.600       0.800    64042082

                               H I S T O G R A M S
                               *******************

                                  S U M M A R Y

TITLE       /    (RE)SET/   OBS/  AVERAGE/EST.ST.DV/  MINIMUM/   MAXIMUM
thru                0.000   120     1.582      0.596     1.077     4.239

CELL/LOWER LIM/      N/    FREQ/   CUM :
                                          I-------------------------------
   0 -INFINITY       0     0.00     0.00  I
   1     1.000      95     0.79    79.17  I******************************
   2     2.000      18     0.15    94.17  I******
   3     3.000       6     0.05    99.17  I**
   4     4.000       1     0.01   100.00  I
                                                **REST OF TABLE EMPTY**

                                          I-------------------------------

                               R E S O U R C E S
                               *****************

TITLE       /    (RE)SET/    OBS/ LIM/ MIN/ NOW/  % USAGE/ AV. WAIT/QMAX
terminal 1          0.000     20    1    0    1    40.459      0.186    1
terminal 2          0.000     24    1    0    1    49.862      0.639    3
terminal 3          0.000     22    1    0    1    46.063      0.427    2
terminal 4          0.000     21    1    0    0    43.847      0.301    2
terminal 5          0.000     16    1    0    0    34.458  4.816&-02    1
terminal 6          0.000     17    1    0    1    35.523      0.310    2

                                    B I N S
                                    *******

TITLE       /    (RE)SET/    OBS/INIT/ MAX/ NOW/ AV. FREE/ AV. WAIT/QMAX
buffers             0.000    121    3    3    2     2.743  1.701&-04    1
```

```
                    W A I T   Q U E U E S
                    *********************

TITLE        /   (RE)SET/   OBS/ QMAX/ QNOW/ Q AVERAGE/ZEROS/  AV. WAIT
requestq 1         0.000     20     1     0      0.000    20      0.000
requestq 1   *     0.000     20     1     0  3.226&-03     0  9.679&-03

requestq 2         0.000     24     1     0      0.000    24      0.000
requestq 2   *     0.000     24     1     0  3.165&-03     0  7.912&-03

requestq 3         0.000     22     1     0      0.000    22      0.000
requestq 3   *     0.000     22     1     0  4.007&-03     0  1.093&-03

requestq 4         0.000     22     1     0      0.000    22      0.000
requestq 4   *     0.000     22     1     0  3.304&-03     0  9.011&-03

requestq 5         0.000     17     1     0      0.000    17      0.000
requestq 5   *     0.000     17     1     0  2.400&-03     0  8.472&-03

requestq 6         0.000     17     1     0      0.000    17      0.000
requestq 6   *     0.000     17     1     0  3.328&-03     0  1.175&-02
```

SUMMARY

Demos is, effectively, an extension of Simula which provides high-level features for the description and implementation of discrete event models. In this paper, the major synchronisation facilities of Demos have been described and their ease of use has been illustrated by several examples. The examples have been chosen from different application areas, which serve to show the wide applicability of Demos.

Demos runs on any of the twelve or so machines that currently support Simula (the list includes IBM, UNIVAC 1100, ICL, CD, PRIME, DATA GENERAL, DEC 10, DEC 20, VAX, Honeywell and Motorola hardware).

Future developments for Demos already slated include extensions for dealing with combined continuous and discrete models, and an interactive front end for inputting model specifications and automatically deriving 'correct' Demos programs (correct means consistent with specs).

BIBLIOGRAPHY

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

2. G. M. Birtwistle, "The Demos Implementation Guide and Reference Manual", University of Calgary Research Report 81/70/22.

3. G. M. Birtwistle, "An approach to discrete event modelling", University of Calgary Research Reports 81/72/24, 81/73/25, 81/74/26, 81/75/27, 81/76/28.

4. G. M. Birtwistle, O-J. Dahl, B. Myhrhaug and K. Nygaard, "Simula Begin", Studentlitteratur, Sweden, 1979.

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

6. A. A. B. Pritsker and P. J. Kiviat, "Simulation with GASP II", Prentice-Hall, 1969.