*Proceedings of the 1987 Winter Simulation Conference*
A. Thesen, H. Grant, W. David Kelton (eds.)

# A PROCESS-ORIENTED SIMULATION PACKAGE BASED ON MODULA-2

Pierre L'Ecuyer and Nataly Giroux
Département d'informatique
Université Laval
Ste-Foy, Qué., Canada, G1K 7P4.

## ABSTRACT

*SIMOD* is a process-oriented, discrete-event simulation package, implemented as a set of precompiled modules written in Modula-2. It is not a new language; basically, a *SIMOD* program is simply a Modula-2 program. The package offers predefined data types and procedures, and run-time support facilities to manage the clock, event list, processes, resource acquisitions, etc. Using these tools, a programmer is able to express his model quickly and concisely, in a readable language. In this paper, we describe *SIMOD*, give some programming examples, and summarize the positive and negative aspects of our experience with Modula-2 as a host language for building a discrete-event simulation programming environment.

## 1. INTRODUCTION

Discrete event computer simulation is an important tool for decision makers. Many simulation programs have been (and still are) written in general purpose languages, like FORTRAN, (see Law and Kelton (1982) and Bratley, Fox and Schrage (1983)). To avoid reinventing the wheel again and again, people have developped subroutine libraries that provide the basic utilities for simulation programming (like GASP, for instance). On the other hand, specialized simulation languages were also created (like *e.g.* GPSS, SIMSCRIPT, SLAM, SIMAN, DEMOS, etc.).

Specialized languages usually offer higher level tools, leading to shorter and more readable programs. But they also have limitations, and in the most widely used simulation languages, people often have to call FORTRAN subroutines for operations that cannot be done in the specialized language. Compared to general-purpose languages, simulation language compilers are usually more expensive, not as well documented, produce less efficient code, and provide less debugging support. As Bratley, Fox and Schrage (1983, pp. 214–215) pointed out, none of the commonly used simulation languages is a modern language, and none has an easily understandable and predictable behavior. Another obstacle to specialized languages is that people have to learn them. This often imply a significant time investment, especially for one-time or occasional use. Programmers are more productive when they work in a familiar language. We feel that a good part of the time spent learning a new language could be better used working on more fundamental aspects of the problem.

For all these reasons, many simulation projects are programmed in general purpose languages, of which FORTRAN is still the most widely used. But FORTRAN is an outdated language, with many handicaps (see Kreutzer (1986)). More modern, more reliable, better structured and conceptually richer languages like *Pascal*, *Modula-2* and *ADA*$^{TM}$ are now becoming increasingly popular. Good and well supported compilers for these languages, and often very sophisticated programming environments, are available on most machines. *Pascal* is, in many respects, a marked improvement over FORTRAN, but still has its limitations, like not supporting modular design (with separate compilation) and coroutines. Implementing a process-oriented simulation package based on *Pascal* calls for a language extension, which means that one has to either rewrite a compiler or write a preprocessor (we intentionally forget the unelegant approaches which require the user to put labels all over his code). Rewriting or patching the compiler largely destroys portability. Some disadvantages of using a preprocessor are that two compilations are more time-consuming than one, error messages are more difficult to interpret, and the object code is sometimes slower. See Kaubisch, Perrot and Hoare (1976), Malloy and Soffa (1986), Vaucher (1984) and the references given in Kreutzer (1986) for ways of doing this.

*Modula-2* was developped recently by N. Wirth as a successor to *Pascal* (see Wirth (1985), Ford and Wiener (1985), Gleaves (1984)). It was designed to overcome *Pascal's* shortcomings, while keeping simplicity and a small size (in opposition to *Ada*, for instance). *Modula-2* provides access to low level machine functions on the one hand, and encourages good software engineering practices on the other hand. Large programs are built from separately compiled modules, which allow operations on the abstract objects (types, procedures, etc.) defined inside them, but hide the details of their implementations. A *coroutine* facility is supplied, on top of

which one can implement different process synchronization paradigms for quasi-concurrency. The module *Processes* suggested by Wirth (1985) is one example. *Modula-2* is already available on most computer systems, and has already been used successfully for various kinds of applications, like writing operating systems, compilers, expert systems, computer graphics, etc. Anybody already familiar with *Pascal* (which means almost every programmer) can learn it in a very short time. Hence, using it as a base language for developping a simulation software environment is quite appealing.

*SIMOD* is a simulation package which is implemented as a structured set of precompiled modules written in *Modula-2*. These modules offer predefined data types and procedures, and encapsulate a run-time executive system which keeps the simulation clock, manages the event list, takes care of process synchronization, etc. *SIMOD* is process-oriented, which means that an active object in the system is represented by a process, which holds its private data and describes the sequence of actions it experiences throughout its life. The process view of simulation is recognized as a very natural way to program a complex model (see Birtwistle (1979), Unger et al. (1984), and Kreutzer (1986)). Real world systems often consist of independent objects (such as people, aircrafts, robots, vehicles, etc.) which interact with other objects to accomplish their tasks. There may be different types of processes in a model, and many instances of the same process type at any given time. In *SIMOD*, processes are implemented as coroutines. Event-scheduling facilities are also offered, so that one can mix the process-oriented and event-oriented views, if desired. *SIMOD* borrows some of its ideas from the simulation package *DEMOS* (Birtswistle (1979)), which is based on the language *Simula* (Birtwistle *et al.* (1979)), and from the *SIMSCRIPT II.5* language (Russel (1983)). As in *DEMOS*, different patterns of process-synchronization are available, like mutual exclusion through capacity-constrained resources, producer-consumer synchronization with a buffer (or a bin), waiting on conditions, and master-slave synchronization (see Kreutzer (1986)).

Other facilities provided by *SIMOD* include a predefined type `List` (a doubly linked list) with an extensive set of list management tools, multiple random variate generators, and tools to collect statistics. The user also has access to the vast *Modula-2* libraries which are available on most systems. Before giving an overview of *SIMOD*, which we do in section 3, we discuss a small example in the next section, to give the reader a basic feeling of the *SIMOD* programming style. A more elaborate example appears in section 4. In section 5, we discuss the positive and negative aspects of our experience with *Modula-2* as base language for discrete-event simulation. The current implementation of *SIMOD* is a prototype version, based on Logitech's *Modula-2* (Eckhardt et al. (1985)), under VAX/VMS.

Other simulation packages have been proposed, based on languages that support quasi-concurrency. There are packages based on *ADA* (Unger et al. (1984)), *C* (Schwetman (1986)), *SMALLTALK* (Knapp(1986)), etc. *SMALLTALK* is a nice language, but current implementations are rather slow. *C* is efficient, but not easily readable. *ADA* is a well designed language, but it is also big, complex and difficult to master, at least for a novice. While efficient implementations of *Modula-2* are readily available on most micro-computers, *ADA* is not. We do not claim that *Modula-2* is the perfect language for writing simulation software (see our negative comments in section 5), but it is certainly attractive and worth trying.

## 2. A SIMPLE EXAMPLE

Consider a $M/U/s$ queue. Customers arrive according to a Poisson process, wait for one of the $s$ servers to be available, and use this server for the duration of their service time, which is the value of a Uniform random variable. A *SIMOD* program to simulate this system appears in figure 1. The list of identifiers following the reserved word `IMPORT` are the names of *SIMOD*'s modules from which objects are imported. In the program, the identifiers of those imported objects are qualified (i.e. prefixed) by the name of the module from which they come.

Every customer in the system is viewed as a process, whose lifetime is described by the procedure `ProcCustomer`. A customer arrives, requests one server, waits if no server is available, occupy the server for a certain duration (its service time), releases the server, and terminates his activities. `Request` and `Release` are imported from the module `RES`, who provides the `Resource` type and resource management facilities. The procedure `Terminate` terminates the life of the process. It is imported from the module `PROCS`, who offers the basic tools for working with processes. Upon calling the procedure `Delay`, also imported from `PROCS`, the customer waits for the simulation clock to move forward for a "delay" equal to his service time. This service time is a random variable, uniformly distributed between 12 and 16 units of time, and whose value is produced through the generator number 2. The procedure `Uniform` is imported from module `RAND`. A second procedure, called `ProcEndSim`, describes an event which marks the end of the simulation. It prints a complete statistical report on resource `Server`, and stops the simulation.

In the main program, the procedure `Create`, imported from `PROCS`, defines a process type associated with the pro-

```
MODULE Queue;

IMPORT SIM, EVENT, PROCS, RAND, RES;

VAR
   Server   : RES.Resource;
   Customer : PROCS.ProcessType;
   EndSim   : EVENT.EventType;

PROCEDURE ProcCustomer;
   BEGIN
   RES.Request (1, Server);
   PROCS.Delay (RAND.Uniform (12.0, 16.0, 2));
   RES.Release (1, Server);
   PROCS.Terminate;
   END ProcCustomer;

PROCEDURE ProcEndSim;
   BEGIN
   RES.Report (Server);
   SIM.Stop;
   END ProcEndSim;

BEGIN
PROCS.Create (Customer,ProcCustomer, 1000);
EVENT.Create (EndSim, ProcEndSim);
RES.Create
      (Server, RES.Fifo, 3, 'Service facility');
RES.CollectStat (Server);
SIM.Init;
EVENT.Schedule (EndSim, 1000.0, NIL);
PROCS.StartPoissonArrivals
      (Customer, NIL, 10.0, 1, 0);
SIM.Start;
END Queue.
```

**Figure 1.** Simulation of a $M/U/s$ queue, version 1.

cedure ProcCustomer. Its third parameter gives the number of memory words in which each instance of this process will execute. The variable Server was declared as a resource type. RES.Create creates that resource (the service facility), with a single Fifo (first in first out) waiting queue, and a service capacity of 3 units (3 identical servers). The character string "Service facility" will be used by *SIMOD* for the heading of its statistical report on this resource. The call to RES.CollectStat indicates that automatic statistical collection should be done for the resource. SIM.Init initializes the simulator. The event EndSim, marking the end of the simulation, is scheduled to occur in 1000 units of time, and the arrival process of customers is started. Customers arrive according to a Poisson process, with mean times between arrivals of 10 units, the interarrival times are generated with generator number 1. SIM.Start then starts the simulation, which will go on for 1000 units of simulated time.

In *Modula-2*, imported objects can also be used without qualification, provided that they are imported explicitly from their modules of origin. This is illustrated by the program in figure 2, which is equivalent to the one in figure 1, but written differently. In figure 2, some objects are imported explicitly

```
MODULE Queue;

IMPORT SIM, EVENT, PROCS, RES;
FROM RAND  IMPORT Uniform;
FROM EVENT IMPORT EventType, Schedule;
FROM PROCS IMPORT ProcessType, Delay, Terminate;
FROM RES   IMPORT Resource, Request, Release;

VAR
   Server   : Resource;
   Customer : ProcessType;
   EndSim   : EventType;

PROCEDURE ProcCustomer;
   BEGIN
   Request (1, Server);
   Delay (Uniform (12.0, 16.0, 2));
   Release (1, Server);
   Terminate;
   END ProcCustomer;

PROCEDURE ProcEndSim;
   BEGIN
   RES.Report (Server);
   SIM.Stop;
   END ProcEndSim;

BEGIN
PROCS.Create (Customer, ProcCustomer, 1000);
EVENT.Create (EndSim, ProcEndSim);
RES.Create
      (Server, RES.Fifo, 3, 'Service facility');
RES.CollectStat (Server);
SIM.Init;
Schedule (EndSim, 1000.0, NIL);
PROCS.StartPoissonArrivals
      (Customer, NIL, 10.0, 1, 0);
SIM.Start;
END Queue.
```

**Figure 2.** Simulation of a $M/U/s$ queue, version 2.

and used without qualification, while others are referenced through qualified identifiers. It is up to the programmer to decide which choice is best, for each imported object, in order to improve program's readability. One obvious constraint is that all identifiers which are imported explicitly must be unique with respect to each other and to locally declared identifiers. For instance, in *SIMOD*, the identifier Create may represent a procedure to create a List, a Resource, a Bin, etc. according to weither it is imported from module LIST, or RES, or BIN, etc. respectively. If Create is to be imported from two or more of these modules in the same program, then it should be qualified.

## 3. OVERVIEW OF SIMOD

This section briefly describes the principal modules comprising *SIMOD*, as viewed by the user, and the basic features they provide. A more elaborate user's guide, to be available in the near future, gives a precise functional definition of each object.

**UTIL** contains basic tools for objects idenfication by characters strings, reports printing, etc.

**RAND** is a random number generation package. Its kernel is an adaptation of the package proposed by L'Ecuyer and Côté (1987), and it is based on the 32-bit generator proposed by L'Ecuyer (1987). Multiple "virtual" generators can run in parallel (by default, the package has 16 generators, but that number can be increased to more than a thousand), and each generator has its sequence of numbers partitioned into $2^{20}$ disjoint substreams of length $2^{30}$ each. A simple initialization procedure permits to make any generator to jump ahead to the beginning of its next substream, back to the beginning of its current substream, or back to the beginning of its first substream. Initially, each generator is automatically set to the first value of its first substream (its initial seed). A simple switch (one per generator) permits to change from regular to antithetic variates or vice-versa. Other functions permit to generate values according to different probability laws, like discrete uniform, continuous uniform, exponential, Weibull, normal, Student, etc.

Module **STAT** provides tools for statistical collection. For each kind of statistics to be collected, the user declares a variable of type `Block` (a predefined type in module `STAT`), and calls the procedure `Create` to create the corresponding information block and define its type. There are two types of statistical blocks: the `Tally` type is used to tally a sequence (sample) of real valued observations $X_1, X_2, X_3, \ldots$, while the `Accumulate` type is used in the case of a variable which evolves over time, i.e. $X(t), t \geq 0$, with piecewise constant trajectory. User-defined statistical blocks are not updated automatically by the package. This gives more flexibility. The user calls the procedure `Update` to give each new observation $X_i$, for `Tally` type blocks, and to give the new value of $X(t)$ at every jump, for blocks of type `Accumulate`. Procedure `Init` can be called at any time to reinitialize all the counters associated with a statistical block. Procedure `Report` prints a full statistical report on a block. For both types of blocks, specific functions are available to observe the minimum, maximum, sum (or integral, in the "accumulate" case), and average. For `Tally` type blocks, other functions return the number of observations, the variance and the mean square, while a confidence interval of desired level can be printed with a simple procedure call. Of course, such a confidence interval is valid only for the case of independent observations, like for instance when observations correspond to means of independent runs, and if a normal distribution can be assumed.

Module **SIM** provides a function which returns the current simulation time, and procedures to initialize, start and stop the simulation. This module maintains the clock and the event list, and provides an interface to the simulation "engine".

Module **EVENT** offers tools to schedule an event to occur after a specific time delay, or to cancel an event already scheduled. In defining an event type, one gives the name of the (parameterless) procedure that should be executed when an event of this type occurs. When scheduling an event, one gives the name of the event type, and (optionally) a pointer to a block of parameters that could be recovered during that procedure execution.

Basic facilities for process-oriented programming are supplied by the module **PROCS**. As seen in the example of section 2, each type of process must be associated with a user-defined parameterless procedure that describes its behavior. Like events, processes can be scheduled to start after a specific time delay, possibly with a pointer to a block of parameters which could be recovered during process execution. Normally, processes are anonymous. However, a specific name (identifier) could be given to a process instance at scheduling time. That identifier should be declared by the user as a variable of type `ProcessInstance`, a pre-defined type in `PROCS`. A process can delay itself for a specific period of time, suspend itself and wait to be reactivated from the outside, or terminate. Processes may also be interrupted, resumed or killed from the outside. Procedures permit to start or stop a Poisson arrival process, which generates (anonymous) processes of a given type. A trace of operations on processes could also be printed if desired.

The module **LIST** provides the predefined type `List`, which is a doubly linked list, and a set of list management tools. Lists are managed without knowing the types of objects they contain. In fact, the objects in the lists are viewed by the package as of the `ADDRESS` type, which is compatible with any pointer type. Hence, lists may contain about anything, like for instance processes, statistical blocks, resources, other lists, etc. (all these objects are implemented as (opaque) pointer types). In practice, the objects handled by a list have pointer types declared in the user program. Here, in contrast to what happens in *Simula*, an object can be in many lists at the same time.

Every user-defined list should be declared as a variable of type `List`, and then created (initially empty) before being used. A list can be *ordered* or *unordered*. In an ordered list, the objects are kept in order automatically by the module using an ordering function supplied by the user at list creation. This user-defined function should be a boolean valued function which takes two objects `A` and `B` as its arguments, and returns `TRUE` if and only if object `A` should precede object `B` in the ordered list. In an unordered list, the objects

are also arranged in some way, but their ordering depends on how they have been inserted in the list.

For each list, the system maintains a (hidden) pointer to its last referenced or inserted object, which is called its "current" object. In an unordered list, an object can be inserted at the head, at the tail, just before the current object, or just after. An object can be removed from a list, or viewed (recovered) without being removed from the list. One can remove or view the first object in the list, the last one, the current object, its successor, or its predecessor. This permits running through a list, for instance to find an object with particular features. Procedures are also available to remove a specific object from a list, when the name of the object is known, or to verify if that object is in the list or not. There is a function which returns the size of a list (the number of objects it contains), a procedure to sort an ordered list according to a new ordering function, and other tools to concatenate, merge, split, empty and delete lists. Every user-created list owns two predefined statistical blocks: one to "tally" the times spent in the list by objects, the other one to "accumulate" (i.e. integrate and average) the size of the list with respect to time.

From module **RES**, one can import the `Resource` type, and a set of associated routines for synchronization through *capacity-constrained resources*. A `Resource` represents a service facility with a limited (integer valued) capacity (e.g. number of servers), and a single waiting queue, with service policy fixed at creation time. A process must request a number of units of a resource before using it, and releases that number of units when it has finished with it (it is not necessary to request or release all the units at the same time). Requests can also be made with (real valued) priorities. Simple procedures permit to modify part of the capacity of a resource (for instance, when a machine breaks down). Each resource owns two lists, one for its waiting queue and one for the processes in service, and both lists have their associated statistical blocks.

Module **BIN** provides the `Bin` facility, which allows a user to establish *producer/consumer* relationships between processes. A `Bin` corresponds essentially to a pile of tokens (or a buffer), and a waiting queue of processes. A (producer) process may add any number of tokens to the `Bin` by calling procedure `Give`, while a (consumer) process retrieves tokens by calling `Take`. In the latter case, if enough tokens are available, the consumer process is allowed to continue after the `Bin` count is decremented by the number taken. Otherwise, the consumer is blocked and placed into the waiting queue for this `Bin`. It is reactivated when it is his turn for service and enough tokens are available. Processes may call functions that return the number of tokens in a bin, or the `List` of processes in its queue.

Sometimes, a process should be delayed until a (often complex) boolean *condition* becomes true. The module **COND** offers the type `Condition`, which consists of a boolean flag and a waiting queue (`List`) of processes. Procedure calls permit to set the `Condition` to true or false. A process who calls `Wait` on a `Condition` is allowed to continue if the `Condition` is true. Otherwise, it must wait in the queue until the `Condition` is reset to true (from the outside).

Module **MASLA** implements the *master/slave* synchronization paradigm between processes. A `MasterSlave` (a predefined type) has two queues: one for the waiting masters and one for the waiting slaves. A (master) process asks for a slave by calling the function `Request`, which returns a process willing to act as a slave. If no slave is presently available, the calling process is blocked and placed into the master's queue until one becomes available. A process becomes a slave by calling `Wait` for a given `MasterSlave`. If no master is waiting, it is blocked and placed into the slave's queue, otherwise a master is waken up and both may continue.

For any given `List` or `Resource` the user can ask for automatic statistical collection, by simple procedure calls.

## 4. A JOB SHOP MODEL

In this section, we use *SIMOD* to simulate a job shop model, taken from section 2.6 of Law and Kelton (1982). The shop contains $M$ groups (types) of machines, with $N_m$ machines in group $m$, for $m = 1, \ldots, M$. It is modelized by an open queueing network, with one FIFO queue for each group of machines. $J$ types of jobs arrive to this shop. For $j = 1, \ldots, J$, jobs of type $j$ arrive according to a Poisson process, with rate $\lambda_j$. Each job follows a sequence of tasks that must be executed in a specific order, on specific machine groups. A job of type $j$ has $T_j$ tasks, to be executed on machine groups $m_{j,1}, \ldots, m_{j,T_j}$ and whose durations are $d_{j,1}, \ldots, d_{j,T_j}$ respectively. The purpose of the model is to evaluate the performance of the shop for a particular workload. We simulate the shop operations for $T_F$ hours, starting with an empty shop. To reduce the initial biais, we collect statistics only over the time interval $[T_W, T_F]$, where $T_W$ is an initial warm-up time. Statistics are collected on (i) the times spent in the shop by jobs of different types, and (ii) the utilization rate, waiting times, service times, queue length, etc. for each machine group.

The *SIMOD* program appears in figure 3. Each group of machines is viewed as a `Resource`, whose capacity is equal to

```
MODULE JobShop;

IMPORT UTIL, RES, STAT, SIM, PROCS, EVENT;
FROM Storage IMPORT ALLOCATE;
FROM RES     IMPORT Resource, Request, Release, ServicePolicy;
FROM InOut   IMPORT ReadCard, ReadReal, ReadString, OpenInput, CloseInput;
TYPE
   NumTypMachine = [1..5];                    (* A machine group number.          *)
   NumTypJob     = [1..5];                    (* A job type number.               *)
   NumTask       = [1..5];                    (* A task number.                   *)
   InfoTypJob = RECORD                        (* Information on a type of job.    *)
      ArrivalRate  : REAL;                    (* Arrival rate.                    *)
      SojournTimes : STAT.Block;              (* Stats. on times spent in shop.   *)
      NTask        : CARDINAL;                (* Nb. of tasks for this job type.  *)
      DurationTask : ARRAY NumTask OF REAL;   (* Durations of the tasks.          *)
      MachTask     : ARRAY NumTask OF Resource; (* Mach. required for tasks.      *)
      END;
VAR
   NTypMachine : NumTypMachine;               (* Number of machine groups.        *)
   NTypJob     : NumTypJob;                   (* Number of job types.             *)
   TypMachine  : ARRAY NumTypMachine OF Resource;
                                              (* A table of all machine groups.   *)
   TypJob      : ARRAY NumTypJob OF POINTER TO InfoTypJob;
                                              (* A table of all job types.        *)
   m           : NumTypMachine;               (* Index of a machine group.        *)
   j           : NumTypJob;                   (* Index of a job type.             *)
   Job             : PROCS.ProcessType;
   EndSim, EndWarmUp : EVENT.EventType;
   WarmUpTime, FinishTime : REAL;

PROCEDURE ReadAndCreate;
   VAR
      Name     : UTIL.String20;              (* Name of a machine or job type.   *)
      Capacity : CARDINAL;                   (* Number of machines in the group.*)
      Task     : NumTask;
   BEGIN
   OpenInput (".DAT");
   ReadReal (WarmUpTime);   ReadReal (FinishTime);
   ReadCard (NTypMachine);  ReadCard (NTypJob);
   FOR m := 1 TO NTypMachine DO
      ReadString (Name);   ReadCard (Capacity);
      RES.Create (TypMachine [m], Fifo, Capacity, Name);
      RES.CollectStat (TypMachine [m]);
      END;
   FOR j := 1 TO NTypJob DO
      NEW (TypJob [j]);
      WITH TypJob [j]^ DO
         ReadString (Name);  ReadReal (ArrivalRate);  ReadCard (NTask);
         FOR Task := 1 TO NTask DO
            ReadCard (m);  ReadReal (DurationTask [Task]);
            MachTask [Task] := TypMachine [m];
            END;
         STAT.Create (SojournTimes, STAT.Tally, Name);
         END;
      END;
   CloseInput;
   END ReadAndCreate;
```

**Figure 3.** Simulation of a job shop model.

```
PROCEDURE ProcJob;
   VAR
      ArrivalTime : REAL;                  (* Arrival time of this job.       *)
      Typ         : POINTER TO InfoTypJob; (* Type of this job.               *)
      Task        : NumTask;               (* Current task number.            *)
   BEGIN
   ArrivalTime := SIM.Time();
   Typ         := PROCS.Attrib();
   WITH Typ^ DO
      FOR Task := 1 TO NTask DO
         Request (1, MachTask [Task]);
         PROCS.Delay (DurationTask [Task]);
         Release (1, MachTask [Task]);
         END;
      STAT.Update (SojournTimes, SIM.Time() - ArrivalTime);
      END;
   PROCS.Terminate;
   END ProcJob;

PROCEDURE ProcEndWarmUp;
   BEGIN
   FOR m:=1 TO NTypMachine DO  RES.InitStat (TypMachine [m]);  END;
   FOR j:=1 TO NTypJob     DO  STAT.Init (TypJob [j]^.SojournTimes);  END;
   END ProcEndWarmUp;

PROCEDURE ProcEndSim;
   VAR
   BEGIN
   FOR m:=1 TO NTypMachine DO  RES.Report (TypMachine [m]);  END;
   FOR j:=1 TO NTypJob     DO  STAT.Report (TypJob[j]^.SojournTimes);  END;
   SIM.Stop;
   END ProcEndSim;

BEGIN
ReadAndCreate;
EVENT.Create ( EndSim, ProcEndSim);
EVENT.Create ( EndWarmUp, ProcEndWarmUp );
PROCS.Create (Job, ProcJob, 2000);
SIM.Init;
EVENT.Schedule (EndWarmUp, WarmUpTime, NIL);
EVENT.Schedule (EndSim, FinishTime, NIL);
FOR j := 1 TO NTypJob DO
   PROCS.StartPoissonArrivals (Job, TypJob[j] ,(TypJob[j]^.ArrivalRate), 1, 0);
   END;
SIM.Start;
END JobShop.
```

**Figure 3.** Simulation of a job shop model (continuation).

the number of (identical) machines in the group. All waiting queues are FIFO. Each type of job has an associated "RECORD" holding all its related informations. A job type has an arrival rate, a number of tasks to be performed, two tables giving the duration of each task and the machine group on which it should be performed, and a statistical block that gathers statistics on the overall time spent in the system by jobs of this type. The procedure `ReadAndCreate` reads all data from a file and creates the machine types and job types. The numbers 5 in the type definitions are upper bounds on the numbers of machine groups, job types and tasks per job, respectively.

Every job in the system is a process whose lifetime is described by the `ProcJob` procedure. Jobs are generated according to Poisson processes, one per type of job, started off from the main program. The procedure `StartPoissonArrivals` requires five parameters representing respectively the process type to be activated, the address of the process attributes, the mean time between activations of process, the random number generator to be used, and the maximum number of processes to activate (a value of zero for this last parameter means that this maximum number is infinite). Hence, each job is created with a parameter (or attribute) indicating its job type, and whose value could be recovered by calling the `Attrib()` function, inside the `ProcJob` procedure. For each task to be performed, the job requests one machine of the appropriate type, keeps it for the specified duration, and releases it. When the job terminates, its time spent in the shop is computed and added as a new observation to the `SojournTimes` statistical block for this type of job.

Before starting the simulation executive, two events are scheduled. The first one (`EndWarmUp`) marks the end of the warming-up period and reinitializes all the statistical blocks, while the second one (`EndSim`) prints statistical reports and stops the simulation.

## 5. POSITIVE AND NEGATIVE ASPECTS OF MODULA-2

As mentioned in the introduction, *Modula-2* has many strong points and is gaining widespread attention. Compilers and libraries are easily available on most machines. The language is relatively small and simple, enables strong typing and modular design, separates the definition of an abstraction from its implementation, provides procedure-valued variables, and supports concurrency. Current implementations are also reasonably fast. *Modula-2* is a good language for applications for which the speed is important. For instance, the examples given in figures 2 and 3 were also programmed in *SIMSCRIPT II.5*, and executed with the same data than the *SIMOD* programs, on the same machine. The *SIMSCRIPT II.5* versions took between 20% and 60% more CPU time, depending on the data.

Like any other language, it also has its weaknesses. Moffat (1984) has much to say about that; his arguments raise serious concerns and some of them are summarized below.

One bad aspect of *Modula-2* is its treatment of I/O. The I/O facilities are low level and are not really part of the language. They must be imported from a large set of library modules (usually provided with the compiler), which contain hundreds of I/O procedures, all with different names. This name space cluttering problem is a consequence of the fact that I/O has been removed from the language, combined with a rather annoying language restriction: *Modula-2* does not allow procedure parameters that are optional (default valued), or varying in number, or generic (like in *ADA*, for instance). A different procedure name must be used for each number and combination of types of parameters. This often makes many many identifiers. These identifiers may also differ between implementations, giving rise to portability problems.

Another irritating matter is the obligation to give, everytime a new coroutine is created, the size of memory in which that coroutine would execute. Every coroutine has its own execution stack, which is used to store procedure call information, local variables, and any other procedures called by this coroutine. The memory area allocated at coroutine creation should be large enough, otherwise stack overflow and program termination occur. Additional space cannot be allocated when overflow is to occur. In the context of process-oriented simulation, where thousands of processes are created dynamically during execution, and where processes may call procedures from precompiled modules whose implementation is hidden, determining the memory requirements of a process is not always trivial.

Other useful features that are absent from *Modula-2* include facilities for exception handling, automatic storage management (a garbage collector), varying-length strings and string operators, and coroutines whose associated procedures have parameters.

## 6. CONCLUSION

We have shown how the *Modula-2* language can be used to implement a process-viewed simulation package. We also pointed out some important or useful features which lack from *Modula-2*. Adding these features to the language could bring some benefits, but would also add to the complexity and difficulty of implementation. In short, we should say

that *Modula-2* is an experimental language, not a finished product. But it is widely disseminated and has been adopted as the base language for a number of experimental systems. *SIMOD* is one of these. We can expect that these experiments will give valuable insights for future versions of the language, or for other new languages.

## ACKNOWLEDGMENTS

This work has been supported by NSERC-Canada grant # A5463, FCAR-Quebec grant # EQ2831, and a development grant from the Faculté de Sciences et Génie de l'Université Laval, to the first author. Denis Alain, Jean Bélanger, Michel Duclos and Gaétan Perron also took part in the design and development of SIMOD. We wish to thank Jules Desharnais, Olivier Roux and Thien Vo-Dai for helpful comments and suggestions.

## REFERENCES

Birtwistle, G. M. (1979). *Demos — A System for Discrete Event Modelling on Simula.* MacMillan.

Birtwistle, G.M., Dahl, O.J., Myhrhaug, B., and Nygaard, K. (1979). *Simula begin.* Lund:Studentlitteratur.

Bratley, P., Fox, B. L. and Schrage, L. E. (1983). *A Guide to Simulation.* Springer-Verlag, New York.

Eckhardt, H., Koch, J. and Mall, M. (1985). *Logitech Modula-2 Users Manual*, Logitech Inc., Redwood City, California.

Ford, G. A. and Wiener, R. S. (1985). *Modula-2 : A Software Development Approach.* Wiley, New-York.

Gleaves, R. (1984). *Modula-2 for Pascal Programmers.* Springer-Verlag, New-York.

Kaubisch, W. H., Perrot, R. H. and Hoare, C. A. R. (1976). Quasi-parallel programming. *Software Practice and Experience*, **6**, 341–356.

Knapp, V. (1986). The Smalltalk Simulation Environment. *1986 Winter Simulation Conference Proceedings*, 125–128.

Kreutzer W. (1986). *System Simulation — Programming Styles and Languages.* Addison Wesley.

Law, A. M. and Kelton, W. D. (1982). *Simulation Modeling and Analysis.* McGraw-Hill.

L'Ecuyer, P. (1987). Efficient and Portable Combined Random Number Generators. To appear in *Communications of the ACM.*

L'Ecuyer, P. and Côté, S. (1987). A Random Number Package with Splitting Facilities. Report no. DIUL-RR-8705, Département d'informatique, Université Laval.

Malloy, B. and Soffa, M. L. (1986). SIMCAL : The Merger of Simula and Pascal. *1986 Winter Simulation Conference Proceedings*, 397–403.

Moffat, D. (1984). Some Concerns about Modula-2. *SIGPLAN Notice* **19**, 41–47.

Muller, C. (1986). Modula–Prolog: A software Development Tool. *IEEE Software*, **3**, 6, 39–45.

Russel, E. C. (1983). *Building Simulation Models with SIMSCRIPT II.5.* C. A. C. I., Los Angeles.

Schwetman H. (1986). SIMCAL : A C-Based, Process-oriented simulation language. *1986 Winter Simulation Conference Proceedings*, 387–396.

Unger, B. W., Lomow, G. A. and Birtwistle, G. M. (1984). *Simulation Software and ADA*, The Society for Computer Simulation, La Jolla, California.

Vaucher, J. (1984). Process-oriented Simulation in Standard Pascal. in *Simulation in Strongly Typed Languages : Ada, Pascal, Simula, ....* Ed. by R. Bryant and B. W. Unger, SCS Simulation Series, vol. 13, no. 2.

Wirth, N. (1985). *Programming in Modula-2.* Third ed., Springer-Verlag, New-York.

## AUTHOR'S BIOGRAPHIES

PIERRE L'ECUYER is an associate professor in the Computer Science Department at Laval University, Ste-Foy, Québec, Canada. He received the B.Sc. degree in mathematics in 1972, and was a college teacher in mathematics from 1973 to 1978. He then received the M.Sc. degree in operations research and the Ph.D. degree in computer science, in 1980 and 1983 respectively, both from the University of Montreal. From 1980 to 1983, he was also a research assistant at l'Ecole des Hautes Etudes Commerciales, in Montreal. His research interests are in Markov renewal decision processes, approximation methods in dynamic programming, optimization in stochastic processes, random number generation, and discrete-event simulation software. He is a member of ACM, IEEE, ORSA and SCS.

Pierre L'Ecuyer
Département d'informatique
Pavillon Pouliot
Université Laval
Ste-Foy, Qué., Canada
G1K 7P4
(418) 656-3226

NATALY GIROUX is a master's student in computer science at Laval University. She is also working as a consultant for Somapro inc. in the scientific group. As research assistant, she has worked in different research centers, particularly for the National Defence where she worked on a pattern recognition algorithm. She received a B.Sc. in computer science from Laval University in 1987.

Nataly Giroux
699 Dalquier
Ste-Foy Qué., Canada
G1V 3H4
(418) 653-8585