Proceedings of the 1984
Winter Simulation Conference
S. Sheppard, U. Pooch, D. Pegden (eds.)

IMPLICATIONS OF THE ADA[R] ENVIRONMENT FOR SIMULATION STUDIES

Patricia Friel and Sallie Sheppard
Department of Computer Science
Texas A&M University
College Station, Texas 77843

ABSTRACT

Ada packages to support event and process oriented simulation have been developed at Texas A&M University. These packages include standard facilities such as queue handling, random number generation, automatic statistics collection and simulation control. This paper provides an overview of the system facilities followed by an evaluation of the strengths and weaknesses of Ada as an implementation language for simulation software. Based on our experience Ada has appeal as a general purpose language which can be integrated into larger systems. It is portable among various computer architectures, it provides a rich base of constructs in which to implement models, and offers promise in terms of execution on parallel architectures.

INTRODUCTION

The development of new simulation approaches and the development of application programs in the U.S. Department of Defense (DoD) language Ada are parallel research interests at Texas A&M University. One natural outgrowth of these twin threads of interest is a consideration of the implications of the Ada environment for simulation studies. As a pilot project in this area, the Ada packaging concept has been used to develop a library of simulation support tools that can be variously combined to create simulation systems supporting either a process or an event orientation. The event-oriented version supports event simulation in the general style of SIMSCRIPT II.5 [1]. The process-oriented version is based on a design by Bryant [2]. Both versions have been implemented, and sample simulations have been executed on a VAX-11/782 using the ANSI Ada/Ed Translator/Interpreter (Version 1.1.4). The packages that have been developed include facilities for queue handling, random number generation, automatic statistics collection, and simulation control in the two orientation styles. The system development work that has been done is of interest primarily for two reasons:

1) It has provided an elementary prototype for an integrated approach to simulation development.

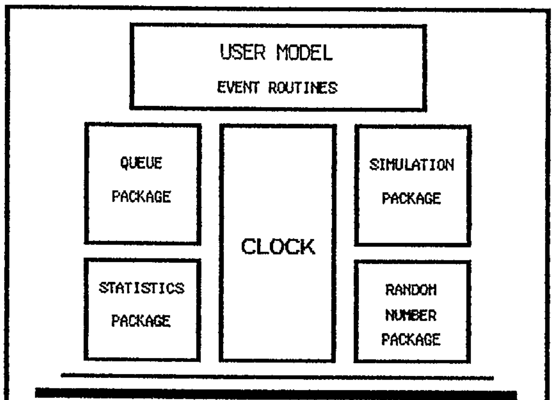
2) It has led to a more general view of the opportunities afforded by the Ada environment for research and development in simulation.

Since an earlier version of the simulation support library that we have developed has been described in detail in a previous paper [3], this paper overviews the entire system and describes in detail only recent additions and revisions. We then conclude with our observations about the possible usefulness of the Ada language to the field of simulation.

[R]Ada is a registered trademark of the U.S. Government (Ada Joint Program Office).

SYSTEM DESCRIPTION

The DoD common high-order language project has emphasized the development of the Ada Programming Support Environment (APSE) as a library of tools available to aid program development. In a well developed Ada environment, packages of the sort that we have designed and implemented for simulation support might be part of the larger APSE library. The library approach to system development implies that a major design objective is to develop modules that are as general as possible. To this end, we have attempted to design packages that are not specific to a particular simulation orientation, but can be used as building blocks for various systems; and, for the two orientations that have been implemented, the concept has generally worked well. The packages initially developed were those fundamental to simulation - a queue handler, a statistics package, a random number generation package, a clock, and packages for simulation control. Of these packages, the only type not used in common by both world views is the control package; although it is reasonable that the simulation control function might be implemented as a generic package able to support more than one viewpoint. Abbreviated listings of the support packages and the two sample simulations are given in Appendices A and B. Figure 1 illustrates the uninstantiated library of packages.



SYSTEM AS CODE PACKAGES
FIGURE 1

Clock Package

The clock was implemented as a separately packaged data structure to make it conveniently accessible by all packages.

Queue Package

The queue package is a modification of the queue package designed by Bryant [2]. The function of the queue package is to manage a queue of temporary entities in a simulation. The fact that it is a generic package allows a simulation writer to design the record format for entities that wait in a queue, so the modeler has complete flexibility to define the number and type of attributes that are associated with a job type. Figure 2 illustrates one possible description for a job that a modeler might define. The instantiation of package QUE makes a tailored version of the queue management package available to handle a queue of type PART_A.

```
type MACHINE is (SAW, DRILL, ROUTER, PLANER);
type INSPECTION_GRADE is (PASS, REJECT, REWORK);
type PART_A is
    record
        ARRIVAL_TIME       : float;
        INSPECT_RESULT     : INSPECTION_GRADE;
        MACHINING_SEQUENCE : array [1..4] of MACHINE;
        JOB_ID             : integer;
    end record;
type PART_A_PTR is access PART_A;

package PART_A_QUE is new QUE
        (ENTITY_REC=> PART_A,
        PTR_ENTITY=> PART_A_PTR);
```

Figure 2 - Definition of a Job Type

The Ada typing rules do pose the constraint that only one type of job (such as PART_A above) can enter a particular queue. If multiple job types enter a common queue in the system to be simulated, the modeler would have to define all jobs as a common type, and make any differentiations within the record.

Queues are structured within the queue package by maintaining a circular linked list of records (type LINK_RECORD) that are defined within the private portion of package QUE. Each LINK_RECORD contains a pointer to a record of the job type defined by the modeler (ENTITY_REC=> PART_A in the example of Figure 2). A record of the simulated time the job entered the queue is also maintained in the LINK_RECORD. Its use in the collection of queue statistics will be explained in the section on the statistics collection facility. The queue is accessed via a header record of type QUEUE which contains queue information (e.g. size). Although only FIFO queues are maintained in this prototype, the package could easily be expanded to offer the modeler a choice of queueing disciplines.

The operations defined for the queue package include queue initialization, job entry and exit, and functions to return the current number in the queue or a boolean value indicating whether the queue is empty. Since the representation of activities as tasks in the process orientation implies the possibility of multiple tasks trying to manipulate the queue at once, the queue operations of job entry and exit are defined within a common task (task Q_OPERATIONS) with exclusive access guaranteed by the select/or construct. To simplify the user's view, the entry points within Q_OPERATIONS are redefined as procedures PUT and TAKE. The queue operations as defined for the process orientation will support an event orientation as well. However, as a practical matter of reducing run times using the Ada/Ed translator, we used a modified version of the queue package that contained only procedures for the event orientation. We hope that this practical consideration will not be necessary when using a production Ada compiler.

Random Number Package

Algorithms for uniform random number generation that depend on high order bits being discarded when an integer multiplication results in an overflow (e.g. FORTRAN integer multiplication) cannot be used in Ada programs since the Ada typing constraints dictate that an unambiguous constraint error message be returned in this situation [4]. An algorithm for a portable FORTRAN Uniform (0,1) generator reported by Marse and Roberts [5], provides one possible alternative. The generator listed in Appendix A is a slight modification of this algorithm rendered in Ada. At present only functions for the Uniform and Exponential distributions have been implemented.

Statistics Collection Package

The statistics collection facility, package STAT, is a generic package that may be instantiated to collect standard statistics on queues or on user defined variables. An instance of package STAT is tailored to the objects on which it will collect observations in the sense that it creates and maintains an array containing one record for each object.

On subsequent calls to procedures within the statistics package, observations will be recorded on the appropriate record. The data structure used is shown in Figure 3. STATISTIC, TALLY_VAR, and ACCUM_VAR are parameters to the generic package. The STATISTIC parameter permits the user to instantiate the package with any floating point type; and the TALLY_VAR and ACCUM_VAR parameters provide subscripts into the arrays (T_RECORD and A_RECORD) of records. Records in T_RECORD are used for point collection statistics, and records in A_RECORD are used for collecting time weighted statistics. The declarations of A_RECORD and T_RECORD also serve to initialize the records.

```
private
  type STAT_REC is
    record
      LAST_VALUE    :  STATISTIC;
      NUMBER        :  STATISTIC;
      SUM           :  STATISTIC;
      WEIGHT        :  STATISTIC;
      SUM_OF_SQRS   :  STATISTIC;
    end record;

T_RECORD  :  T_REC := T_TEC'
    (TALLY_VAR'first..TALLY_VAR'last =>
    (WEIGHT => 1.0, others => 0.0));

A_RECORD  :  A_REC := A_REC'
    (ACCUM_VAR'first..ACCUM_VAR'last =>
    (others => 0.0);
```

Figure 3 - Data Structure for Statistics Collection

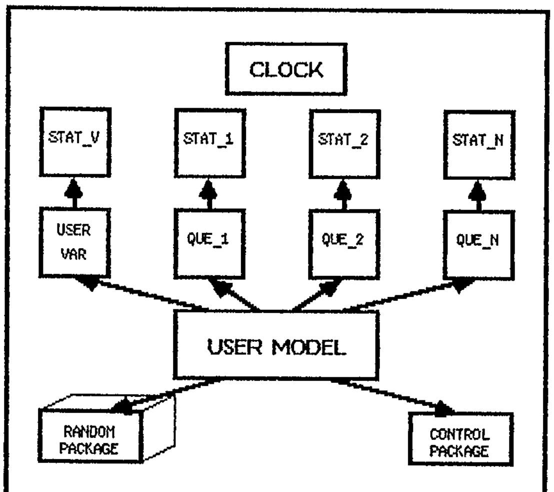
Six operations have been defined for the statistics package:

1) procedure DATA - Called by procedure TALLY or procedure ACCUM, procedure DATA records observations.
2) procedure TALLY - Called from the package that instantiated this statistics package, procedure TALLY collects unweighted (or weight = 1) observations.
3) procedure ACCUM - Called from the instantiating package, procedure ACCUM collects time weighted observations.
4) function MEAN - Computes a mean.

5) function VARIANCE - Computer a variance.
6) procedure PRINT_STATS - Prints summary statistics.

Summary statistics are printed only if the modeler calls for them at the end of the simulation run. Since the method of handling statistics collection varies depending on whether the object of statistical interest is a queue or a variable, each will be described in turn.

Queue Statistics Summary statistics are collected automatically by the simulation system. Whenever the user instantiates a queue package, the instance of package QUE in turn instantiates a statistics collection package for that queue. Figure 4 shows diagramatically the sequence of instantiations that take place at compile time.



SYSTEM IN ACTION
FIGURE 4

When in the course of the simulation, a call is made to the PUT or TAKE procedures in the queue package (to enter or remove a job from a queue), the queue package procedure will call procedure TALLY or ACCUM to collect an observation. The recording of the time the job entered the queue in the LINK_RECORD (see the Queue Package section) is used to collect time weighted statistics. If the modeler wishes to have the statistics on this queue printed at the end of the simulation, he simply calls procedure PRINT_STATS within the queue package. Procedure PRINT_STATS calls a corresponding procedure PRINT_STATS within the statistics package which causes the summary statistics to be calculated and printed. This indirection is necessary to make the statistics collection process transparent to the modeler. As Figure 4 illustrates, the user is one level removed from any statistics package.

Statistics on User Defined Variables The automatic collection of statistics for a queue is easy to implement because the LINK_RECORD provides a ready repository for the time information needed and because one queue package manages one queue, so the number of observation records needed is known. The case of the user defined variables is not quite so simple. Without a preprocessing step, the simulation support system has no way of knowing how many variables a modeler may define, and no way of directly seeing assignments the user will make to those variables. If the user of the system instantiates a statistics package directly, he must also assume responsibility for making the appropriate calls to record each observation. The compromise solution that we chose involves inserting a package between the specific simulation code and the statistics package. Package SPECIAL_STATS parallels package QUE in its function of removing the modeler from the details of statistics collection. When SPECIAL_STATS is instantiated, it in turn instantiates a version of package STAT tailored for user defined variables. In order to have statistics collected on variables, the modeler must declare an enumeration type that contains a list of the variable names. Two subtypes are also declared, one for standard, unweighted averages, the other for variables on which time weighted averages are desired. The two sets need not be mutually exclusive. Package SPECIAL_STATS is then instantiated with the enumeration types as parameters. Assignments to these user defined "variables" are actually made via calls to procedure ASSIGN in package SPECIAL_STATS. Figure 5 illustrates the declarations required, a package instantiation, and a typical assignment statement.

```
type USER_VAR is (TIME_IN_SYSTEM, MY_COUNT,
                  MYSTERY_VAR);
subtype TALLY_VAR is USER_VAR range
        TIME_IN_SYSTEM..MY_COUNT;
subtype ACCUM_VAR is USER_VAR range
        MY_COUNT..MYSTERY_VAR;
package MYSTATS is new SPECIAL_STATS
        (USER_VAR => USER_VAR,
         ACCUM_VAR => ACCUM_VAR,
         TALLY_VAR => TALLY_VAR);

ASSIGN (TIME_IN_SYSTEM, SIM_TIME -
        float(DEPARTING_JOB.ARRIVAL_TIME));
```

Figure 5 - Declarations for User Defined Variables

The data structure used by package SPECIAL_STATS is an array of type float that is subscripted by the enumeration type USER_VAR. This array, named VARIABLE, contains the current values for the variables identified by the enumeration type subscript.

SPECIAL_STATS has two operations. The first, renamed procedure ASSIGN for the convenience of the user, is actually entry PROTECTED_ASSIGN in task ASSIGNMENT. Procedure ASSIGN records the value the user has passed to it in the VARIABLE array of values for user variables in the TALLY_VAR or ACCUM_VAR set and calls the corresponding statistics collection routine in the STAT package. The assignment operation is implemented as a task with an entry in order to guarantee exclusive access to the data structure. Otherwise modeler written tasks might attempt to update variables simultaneously. The second operation is a print procedure identical to procedure PRINT_STATS in package QUE which passes on the modelers call for printout of summary statistics to package STAT.

Simulation Control Packages

Regardless of the world view, the function of the control package is to manage the sequence of events and maintain the clock. The methods used to accomplish this depend on the orientation, so each will be discussed separately.

Event Orientation Conceptually, the simulation manager should cyclically remove the next notice from a sequence of scheduled events, update the system clock, and activate the currently scheduled event. Since the currently scheduled event is a user written procedure for a specific simulation, the manager needs some way of being informed of its name. In general, the scheduler must be able to call modeler written procedures in varying numbers and with user defined names. In order to accomplish this basic task, the control package, EVENT_SIMULATION, is implemented as a generic package with parameters that effectively communicate the number and names of event procedures. The modeler must instantiate EVENT_SIMULATION with an enumeration type, type NOTICE, that contains a list of identifiers for the event routines that he has written. These identifiers are subsequently placed in event notices via calls to procedure CREATE_EVENT_NOTICE in the control package. EVENT_SIMULATION must also be instantiated with the name of a single procedure as a generic parameter. This procedure encapsulates all the modeler written routines and contains a case statement that will activate the event routine indicated by some member of the enumeration type event identifiers. When the scheduler removes an event notice from the chain, it calls the single user routine with the identifier from the event notice as a parameter. The user procedure in turn activates the appropriate event through the case statement.

The data structure used to manage event sequencing is the traditional one of the linked list of event notices. The record type, FUTURE_EVENT_NOTICE, is not visible to the user - as it should not be since he has no need to manipulate it directly.

The operations defined for the package are procedure SCHEDULER and procedure CREATE_EVENT_NOTICE. The function of CREATE_EVENT_NOTICE is to place a FUTURE_EVENT_NOTICE on the chain; and its parameters are the scheduled time for the event and the type of event. The function of SCHEDULER is to remove FUTURE_EVENT_NOTICEs from the chain and call the appropriate procedure. The parameter to SCHEDULER is the specified ending time for the simulation.

Process Orientation In the process orientation, the explicit modeling of time delays (e.g., during a service activity) in the task representing an activity requires a different kind of control mechanism. The task must have some way of suspending itself for a period of simulated time. The control package, PROCESS_SIMULATION, provides the support for the modeler to directly specify time delays. All the modeler need do is call procedure HOLD in package PROCESS_SIMULATION (e.g., HOLD (SERVICE_TIME)). In order to understand how procedure HOLD is able to suspend a modeler defined task in a specific simulation, it is necessary to examine the data structure used by the control package. The structure consists basically of a linked list of event notices similar to the one used in the event orientation. The difference lies in the structure of the records that are placed on the event chain. As shown in Figure 6, each record contains a task of type SYNCH.

When procedure HOLD is called, it places a record of type FUTURE_EVENT_NOTICE on the event chain, then calls the entry WAIT in SIGNAL. Since task SIGNAL is now activated, but unable to accept the call to WAIT until its entry SEND is called, procedure HOLD is unable to return to its calling task; so the calling task is likewise delayed. Because the code for procedure HOLD is reentrant, many activations of HOLD may coexist. The calling task will remain suspended until task SCHEDULER removes the notice from the chain and calls entry SEND in SIGNAL. Now the calling task is able to complete, followed immediately by HOLD and the calling task. This sequence of calls is the kernel of the original design by Bryant [2].

```
task type SYNCH is          task body SYNCH is
  entry SEND;               begin
  entry WAIT;                 accept SEND;
end SYNCH;                    accept WAIT;
                            end SYNCH;

type FUTURE_EVENT_NOTICE is
  record
    SCHED_TIME     :  TIME;
    SIGNAL         :  SYNCH;
    NEXT           :  F_EVENT_LINK;

type F_EVENT_LINK is access FUTURE_EVENT_NOTICE;
```

Figure 6 - Data Structure Used by Scheduler

Parallel processing implies by nature the possibility of certain rather subtle problems with exclusive access and sequencing; and some modifications to Bryant's basic design had to be made to handle these problems. Since tasks running in parallel may make simultaneous calls to procedure HOLD, simultaneous attempts may be made to place event notices on the chain. To guarantee exclusive access to the event list, the code sections to enter and remove a record from the chain reside in a task, EVENT_CHAIN, and are guarded by the select/or construct.

Sequencing is another problem. Task SCHEDULER, which is responsible for removing notices from the event chain and reactivating the appropriate tasks in the specific simulation code, will resume at one time all tasks which are scheduled for activation at the same simulation time. These tasks in turn will need to schedule themselves for subsequent resumptions, but since both they and the SCHEDULER are running in parallel, the SCHEDULER could resume one out of order. For example, suppose task ALPHA and task BETA are executing. Task ALPHA calls procedure HOLD and schedules itself for resumption at time 900 which is the earliest time on the event chain. Task SCHEDULER promptly resumes task ALPHA. Now task BETA has reached the portion of its code in which it calls procedure HOLD and schedules itself for resumption at time 850. There is the problem. The problem is solved by having the SCHEDULER keep a count of all active tasks. Then no tasks are reactivated until the currently active tasks have completed their scheduling. The NUMBER_ACTIVE variable in tasks SCHEDULER keeps track of the active task count.

One other problem that is latent in parallel processing and is of particular concern in the context of simulation is that of determinancy among consecutive runs. Although the scheme described above prevents the SCHEDULER from reactivating tasks out of order within a particular run, there is nothing to prevent the tasks from scheduling themselves to resume at different times in different runs because they have called the random number generator in a different order. (The Ada/Ed translator quite properly emulates true parallelism by varying the interleaving of code for tasks sections in different program executions.) While one execution may model the subject of the simulation as well as another, subsequent executions must yield identical results in order for any sensitivity analysis to be meaningful. The solution that we chose - not necessarily the best one - was to require each simulation task to instantiate its own random number package, each with a different seed value. In this way, each task in multiple program executions will draw random numbers in the same order, and the simulation will be determinant from one run to the next. What effect this solution may have on the charactistics of the actual stream of numbers selected in a simulation is an interesting question. It should also be noted that although this solution causes scheduling to occur in a determinant way, it in no way guarantees that statistics gathered on user defined variables will be consistent from one execution to the next. If dependencies should exist between tasks in assigning values to such variables, the possibility exists of different assignments or different orderings of assignments being made on subsequent runs. We have not discovered a way to protect a user of the simulation packages from this kind of potential error.

IMPLICATIONS OF THE ADA ENVIRONMENT

Few would dispute the contention that simulation is an extremely powerful and flexible tool for planning in a wide variety of contexts. From manufacturing systems to street and highway design to prison facilities planning to the ubiquitous checkout stand, simulation can support better decision making. Yet it remains an underutilized tool for several reasons.

Perhaps the greatest of the these is that simulation languages have traditionally been developed as specialized, stand alone systems that do not accommodate themselves well to the context in which they must function. The language is different from that used for, say, process control in a factory; it is not designed to interface with the resident database system; it is one more language that a potential user must learn; and it takes considerable time and expertise to develop a model for simulation even when the need for one has been identified. In short, it does not integrate efficiently into the larger system it serves. The IBM publication, Communications Oriented Production Information and Control System (COPICS) [6], gives a vivid description of the kind of manufacturing situation in which simulation could be particularly useful.

> "The problems of lead time, work-in-process, and capacity are interrelated; in fact; they can be self-perpetuating...for example, as a company's business improves, more orders are released to the plant. If the work input is greater than the work capacity, and queues build up at the work centers, jobs have to wait longer. As a result, customer orders are late. The foreman feels that given two weeks more lead time they could complete orders on schedule, so the manufacturing lead times are accordingly increased by two weeks. New shop orders are now released two weeks earlier, generating an additional two weeks' volume of work on the plant floor. This increases both work-in-process and queue length some more. Lead times become longer than ever, and orders are still finished behind schedule."

This familiar scenario constitutes a vicious cycle stoked by myopic decision making. The need for readily accessible planning tools is evident. If the foreman above must be relied upon to identify the need for forecasting tools, collect the necessary data to determine appropriate parameters, and initiate the development of a simulation, it is not difficult to imagine that the simulation may never be done. If, on the other hand, the developing vicious cycle may be flagged by a computerized moitoring system; if both shop floor data and management planning information to which the foreman may not have access may be channelled directly to a decision support system of which a simulation system is an integrated component, the tool may be utilized and the vicious cycle broken. Several design features of Ada seem particularly applicable to the development of such integrated systems. It is designed for real time processing. The APSE or library approach to system development is intended to supply both the flexibility and the capability for coordinatation implicit in the design of large systems of functionally different but integrated components. And the syntax of the language (in particular the specification part) prescribes the clean documentation and carefully defined interfaces that are needed for large and complex systems.

Another impediment to more widespread use of simulation is the lack of portability of models between computers. When a firm invests in the development of a large model, there is often a need to excute the model on different computer systems, either because of hardware differences between sites or because of changes in the hardware during the life of the model. One of the main design goals of Ada is portability not only over different computer architectures. Such promise of portability makes the model development investment more attractive since it expands the opportunities for model usage beyond that of the original host computer.

A third reason for underutilization of the simulation tool is simply that any one simulation language chosen may not have the constructs necessary to model naturally many different subsystems within the same environment. Can it be used to produce a conceptually straightforward model of a production line? Of a job shop? Of a building (as opposed to manufacturing) process? Of the control systems in a plant? Of the planning process itself? Of some combination of the above in the same simulation? Does it support various world views? Again the packaging concept, the "building blocks" approach, of Ada may prove useful in developing simulation systems that can be tailored to an environment. There is at least the opportunity to experiment.

The time it takes to execute a simulation of any size is a fourth impediment to the extensive use of simulation in industry. One obvious way to reduce overall run times is to use parallel processors. However, procedures for subdividing simulation systems into components that can run effectively in parallel are not yet well understood, and provide a fertile area for research (such research currently is underway at Texas A&M University [7,8,9]). Both the tasking feature of Ada and the capability of the ANSI Ada/Ed Translator/Interpreter to emulate actual parallel execution offer opportunities to explore the possibilities of concurrent processing in simulation systems. An interesting approach to the design of a distributed simulation system might be to first model the proposed design on a single processor system utilizing an Ada task to model the activities on each processor. Since the translator will emulate parallel execution among the tasks (subsequent runs will not yield identical results), the translator itself could be used to aid in modeling the systems and allowing one the opportunity to collect data on communication among the processors. Various configurations might be modeled in order to study the effects of design changes prior to actually building the system. This approach seems at least to embody the spirit of simulation.

We are not by any means touting Ada as the final answer to effective simulation development. In fact, we have found both strengths and weaknesses of the language in the course of our learning experience with Ada.

One of the goals of the designers of Ada was to create a language that would in its syntax enforce the development of reliable code. The result was the strong typing constraints that unquestionably serve the reliability goal, but that also result in some cost in flexibility. Even in the limited development work that we have done, the language has more than once forced us to choose a less elegant solution than the one we would have liked as the following cases illustrate.

In the design of the SCHEDULER for the event oriented control package, the problem arose of how to inform the SCHEDULER of the names of the user written event routines so that the procedures could be activated by the SCHEDULER during execution of a simulation. Without a preprocessor step, the procedure names must somehow be passed as parameters to the EVENT_SIMULATION package or to procedure CREATE_EVENT_NOTICE. The nice solution would have the model routines pass a pointer to the appropriate procedure as a parameter to CREATE_EVENT_NOTICE which would then place the pointer in the notice on the event chain. The SCHEDULER would simply activate the scheduled event via the pointer. Ada does permit pointers to tasks (though not procedures), and a procedure could be disguised within the begin/end block of a task rendezvous. However, for CREATE_EVENT_NOTICE to accept a task pointer as a parameter, it must know the type of the task to expect. The need to know task type names has been substituted for the need to know procedure names. In this case, the strong typing of the language has both protected us from passing meaningless pointers and frustrated our design goal of isolating the modeler from the implementation. Another possibility would have been to pass the procedure names as parameters to the generic package EVENT_SIMULATION, but the limitation to a one to one relationship between formal and actual procedure parameters prevents this solution from being a viable one. It is hardly practical to place an arbitrary limit on the number of procedures a user may write.

Another instance that could directly affect a user of the packages is the limitation of one job type to one job queue mentioned above in the queue package section.

Task termination proved somewhat problematic within the context of simulation. To be useful in a simulation, tasks must be allowed to loop "forever" or at least for variable lengths of time, so they cannot terminate by completing. The modeler must abort any tasks that he has written before the simulation can terminate. We used a procedure END_SIMULATION within package PROCESS_SIMULATION to abort the remaining tasks in PROCESS_SIMULATION. However the tasks within package QUE and SPECIAL_STATS can only be aborted from the modeler package since they are generic packages and specific instances of them can only be seen by the instantiating package.

In general, the drawbacks to simulation system development imposed by the language that we have encountered fall into two categories - typing constraints and "vision" problems. The SCHEDULER's problem with being able to "see" user written event routines and the problem with task termination fall within the vision classification. However, from the point of view of a potential user of the system, there are actually few details outside of the logic of his event routines with which he must concern himself. There are a number of declarations and instantiations that a modeler must make. The obvious extension to the system that would most simplify the modeler's task would be the development of an interactive development tool that would query the user and write the declarations and instantiations for him. It could be a much more general tool within the APSE - including perhaps a syntax directed editor, a library manager, and access to a database or knowledge base. Such a tool would also imply the opportunity for using a preprocessor step to solve the vision problems. We have not chosen the preprocessor approach heretofore partly because we wished to test the flexibility of the language itself. Although we have not found Ada to provide an elegant solution to every situation, we do believe that the language design extends the realm of the possible.

On the positive side, the generic packaging concept clearly redefines the meaning of reusable software. In the case of this work, it was the single most useful feature of the language. A comparison of figures 1 and 4 should make the power of the concept evident. With respect to further research, the tasking feature and the emulation of parallelism provided by the translator offer the most promise. Tasking provided the key to the design of the process oriented system originally proposed by Bryant and extended by us; and the emulation by the interpreter of parallel execution facilitated our discovery and understanding of the problems of exclusive access and determinancy latent in the system. Certainly there is a need for the development of debugging tools specifically designed for parallel processing, and the translator could aid such development work on a single processor system. A third feature that we will classify as very positive is the specification part of packages. The clarity in terms of interfaces among components provided by this feature might be rated nearly indispensable if one were to undertake the kind of large scale integrated system development that we have suggested.

We suggested earlier that one of the primary problems in the field of simulation was that we have not yet learned how to smoothly interface a simulation system to the larger system it serves. Whether or not Ada ever becomes a widely used language in industry, it nevertheless offers the opportunity to explore new approaches to the development of integrated systems. If Ada and simulation are indeed a matched pair, it may be because the environmental approach encouraged by Ada is precisely what simulation needs to further move it into the world of business and industry.

ACKNOWLEDGEMENT

The authors would like to acknowledge the work of Ms. Donna Reese on both the design and implementation of the initial system. Her continued interest and suggestions have been valuable to the project.

This material is based on work supported in part by the National Science Foundation under Grant NO. ECS-8215550.

APPENDIX A

PROCESS ORIENTATION

```
package CLOCK is
  SIM_TIME       : float := 0.0;
end CLOCK;
package body CLOCK is
begin
  null;
end CLOCK;
----------------------------------------------
with CLOCK, text_io;    use CLOCK, text_io;
generic
 type STATISTIC is digits <>;
 type TALLY_VAR is (<>);
 type ACCUM_VAR is (<>);
package STAT is
 type STAT_REC is private;
 type T_REC is array (TALLY_VAR) of STAT_REC;
 type A_REC is array (ACCUM_VAR) of STAT_REC;

 procedure TALLY (ID     : in TALLY_VAR;
                  VALUE  : in STATISTIC);
 procedure ACCUM (ID     : in ACCUM_VAR;
                  VALUE  : in STATISTIC);
 procedure DATA (STATS   : in out STAT_REC;
                  DATA   : in STATISTIC);
 function MEAN(STATS:in STAT_REC)
                         return STATISTIC;
 function VARIANCE (STATS: in STAT_REC)
                         return STATISTIC;

 procedure PRINT_STATS;
 package OUT_STAT is new float_io (STATISTIC);
 package OUT_TALLY is
          new enumeration_io (TALLY_VAR);
 package OUT_ACCUM is
          new enumeration_io (ACCUM_VAR);
 private
  type STAT_REC is
        record
          LAST_VALUE     : STATISTIC;
          NUMBER         : STATISTIC;
          SUM            : STATISTIC;
          WEIGHT         : STATISTIC;
          SUM_OF_SQRS    : STATISTIC;
        end record;
 T_RECORD : T_REC :=
   T_REC'(TALLY_VAR'first..TALLY_VAR'last =>
        (WEIGHT => 1.0,
         others => 0.0));
 A_RECORD : A_REC := A_REC'(ACCUM_VAR'first..
   ACCUM_VAR'last => (others => 0.0));
end STAT;

package body STAT is
----------------------------------------------
procedure DATA (STATS   : in out STAT_REC;
                DATA    : in STATISTIC) is
begin
 STATS.NUMBER := STATS.NUMBER + STATS.WEIGHT;
 STATS.SUM := STATS.SUM + STATS.WEIGHT * DATA;
 STATS.SUM_OF_SQRS := STATS.SUM_OF_SQRS +
          STATS.WEIGHT * DATA * DATA;
end DATA;
----------------------------------------------
procedure TALLY (ID     : in TALLY_VAR;
                VALUE   : in STATISTIC) is
begin
 DATA (T_RECORD (ID), VALUE);
end TALLY;
----------------------------------------------
```

```
procedure ACCUM (ID    : in ACCUM_VAR;
                 VALUE : in STATISTIC) is
begin
 A_RECORD (ID).WEIGHT := STATISTIC(SIM_TIME)
              - A_RECORD (ID).WEIGHT;
 DATA(A_RECORD(ID), A_RECORD(ID).LAST_VALUE);
 A_RECORD (ID).LAST_VALUE := VALUE;
 A_RECORD (ID).WEIGHT := STATISTIC(SIM_TIME);
end ACCUM;
----------------------------------------------
function MEAN (STATS    : in STAT_REC)
                        return STATISTIC is
begin
 if STATS.NUMBER /= 0.0 then
      return STATS.SUM / STATS.NUMBER;
 else
      return 0.0;
 end if;
end MEAN;
----------------------------------------------
function VARIANCE (STATS: in STAT_REC)
                    return STATISTIC is
begin
 if STATS.NUMBER /= 0.0 then
  return (STATS.NUMBER * STATS.SUM_OF_SQRS -
     STATS.SUM * STATS.SUM) / (STATS.NUMBER *
              (STATS.NUMBER - 1.0));
 else return 0.0;
 end if;
end VARIANCE;
----------------------------------------------
procedure PRINT_STATS is
 begin
  <header statements>
  for INDEX in TALLY_VAR
    loop
     OUT_TALLY.put (INDEX);
     OUT_STAT.PUT (MEAN(T_RECORD(INDEX)));
     OUT_STAT.PUT(VARIANCE(T_RECORD(INDEX)));
    end loop;
  for INDEX in ACCUM_VAR
    loop
     ACCUM (INDEX, 0.0);
      OUT_ACCUM.put (INDEX);
      OUT_STAT.PUT (MEAN(A_RECORD(INDEX)));
      OUT_STAT.PUT(VARIANCE(A_RECORD(INDEX)));
     end loop;
 end PRINT_STATS;
end STAT;
----------------------------------------------
with STAT, CLOCK; use CLOCK;
generic
 type USER_VAR is (<>);
 type TALLY_VAR is (<>);
 type ACCUM_VAR is (<>);
package SPECIAL_STATS is
 type USER_VAR_RECORD is array
                   (USER_VAR) of float;
 VARIABLE: USER_VAR_RECORD := USER_VAR_RECORD'
      (USER_VAR'first..USER_VAR'last => 0.0);
 task ASSIGNMENT is
  entry PROTECTED_ASSIGN (SUB : in USER_VAR;
                        VALUE : in float);
 end ASSIGNMENT;
 procedure ASSIGN (SUB: in USER_VAR; VALUE:in
  float) renames ASSIGNMENT.PROTECTED_ASSIGN;
 procedure PRINT_STATS;
end SPECIAL_STATS;

package body SPECIAL_STATS is
 package MODELER_STATS is new STAT
  (STATISTIC => float, TALLY_VAR => TALLY_VAR,
                 ACCUM_VAR => ACCUM_VAR);
----------------------------------------------
```

```
 task body ASSIGNMENT is
  begin
   loop
    accept PROTECTED_ASSIGN(SUB : in USER_VAR;
                            VALUE : in float) do
    VARIABLE (SUB) := VALUE;
    begin
     if TALLY_VAR (SUB) in TALLY_VAR then
        TALLY (TALLY_VAR (SUB), VALUE);
     end if;
     exception
       when constraint_error => null;
    end;
    begin
     if ACCUM_VAR (SUB) in ACCUM_VAR then
        ACCUM (ACCUM_VAR (SUB), VALUE);
     end if;
     exception
        when constraint_error => null;
    end;
   end PROTECTED_ASSIGN;
  end loop;
end ASSIGNMENT;
----------------------------------------------
 procedure PRINT_STATS is
  begin
   MODELER_STATS.PRINT_STATS;
  end PRINT_STATS;
 end SPECIAL_STATS;
----------------------------------------------
with STAT, CLOCK;       use CLOCK;
generic
 type ENTITY_REC is private;
 type PTR_ENTITY is access ENTITY_REC;
 package QUE is
  type QUEUE  is private;
  procedure PRINT_STATS;
   -- Initialize a queue.
  procedure INIT       (Q: in out QUEUE);
   -- Operations on queue
  task Q_OPERATIONS is
   entry PUT_IN (ENTITY: in PTR_ENTITY;
                      Q: in out QUEUE);
   entry TAKE_OUT (Q: in out QUEUE;
            ENTITY: in out PTR_ENTITY);
  end Q_OPERATIONS;
  --Reference Q_OPERATIONS entries as procs
  procedure PUT (ENTITY: in PTR_ENTITY;
   Q: in out QUEUE) renames
                   Q_OPERATIONS.PUT_IN;
  procedure TAKE (Q: in out QUEUE; ENTITY:
    in out PTR_ENTITY)
            renames Q_OPERATIONS.TAKE_OUT;
  function IS_EMPTY(Q:in QUEUE)return BOOLEAN;
  function NUMBER_IN (Q: in QUEUE)
                            return integer;
  private
  -- A queue has records of type link_record.
  --Each link_record holds a ptr to an entity.
   type LINK_RECORD;
   type LINK is access LINK_RECORD;
   type LINK_RECORD is
    record
      NEXT, PREV: LINK;
      ENTITY      : PTR_ENTITY;
      TIME_Q_ENTERED : float;
    end record;
   type QUEUE is
    record
      HEAD       : LINK;
      SIZE       : INTEGER;
      EMPTY      : BOOLEAN;
    end record;
   end QUE;
   package body QUE is
```

```
  type TALLY_Q is (QUEUE_TIME);
  type ACCUM_Q is (QUEUE_LENGTH);
  --Instantiate a STAT package for queue
  package QUEUE_STATS is new STAT
   (STATISTIC => float,TALLY_VAR => TALLY_Q,
              ACCUM_VAR => ACCUM_Q);
  use QUEUE_STATS;
----------------------------------------------
procedure INIT (Q: in out QUEUE) is
begin                  -- initialize q
 --Allocate head node of queue and set ptrs
 -- to represent an empty queue
 Q.HEAD          := new LINK_RECORD;
 Q.HEAD.NEXT     := Q.HEAD;
 Q.HEAD.PREV     := Q.HEAD;
 Q.SIZE          := 0;
 Q.EMPTY         := TRUE;
end INIT;
----------------------------------------------
task body Q_OPERATIONS is
 HOLDER: LINK;
begin           -- Put entity at end of q
 loop
  select
   accept PUT_IN (ENTITY: in PTR_ENTITY;
                       Q: in out QUEUE) do
    Q.SIZE     := Q.SIZE + 1;
    Q.EMPTY    := FALSE;
    --Collect statistics on queue length.
    ACCUM (QUEUE_LENGTH, float (Q.SIZE));
    -- The list that represents the queue
    -- consists of holder records that contain
    -- pointers to the actual queue members.
    HOLDER              := new LINK_RECORD;
    -- Set the pointer to the job and record
    --the time in queue.
    HOLDER.ENTITY        := ENTITY;
    HOLDER.TIME_Q_ENTERED := SIM_TIME;
    -- Set linked list pointers to add this
       holder to the end of the queue.
    HOLDER.PREV                 := Q.HEAD.PREV;
    HOLDER.PREV.NEXT            := HOLDER;
    HOLDER.NEXT                 := Q.HEAD;
    Q.HEAD.PREV                 := HOLDER;
   end PUT_IN;
   or
   accept TAKE_OUT (Q: in out QUEUE;
            ENTITY: in out PTR_ENTITY) do
    if not Q.EMPTY then
     Q.SIZE    := Q.SIZE - 1;
     Q.EMPTY   := Q.SIZE = 0;
     --Unlink the first holder record.
     HOLDER              := Q.HEAD.NEXT;
     Q.HEAD.NEXT         := HOLDER.NEXT;
     Q.HEAD.NEXT.PREV := Q.HEAD;
     --Collect statistics on queue length
     --and time in queue.
     ACCUM (QUEUE_LENGTH, float (Q.SIZE));
     TALLY (QUEUE_TIME, SIM_TIME -
              HOLDER.TIME_Q_ENTERED);
     --Return a ptr to the entity removed
     ENTITY := HOLDER.ENTITY;
    else
     ENTITY := NULL;
    end if;
   end TAKE_OUT;
  end select;
 end loop;
end Q_OPERATIONS;
----------------------------------------------
function IS_EMPTY (Q: in QUEUE)
                       return BOOLEAN is
begin
 if Q.EMPTY then
```

```
      return true;
   else
      return false;
   end if;
 end IS_EMPTY;
-----------------------------------------------
 function NUMBER_IN(Q:in QUEUE)
                        return integer is
 begin
    return Q.SIZE;
 end NUMBER_IN;
end QUE;
-----------------------------------------------
generic
 SEED   : in integer := 1;
package RANDOM is
 function LN (X : in float;
         N : in integer := 10) return float;
 function UNIFORM (LOW  : in float := 0.0;
       HIGH  : in float := 1.0) return float;
 function EXPONENTIAL
               (LAMBDA:in float) return float;
end RANDOM;
-----------------------------------------------
package body RANDOM is
 SEED_VAL         : integer := SEED;
 function LN (X : in float;
      N : in integer := 10) return float is
  LN_X             : float;
  X_POINT                 : float := 1.0;
  INCREMENT       : float := (X - 1.0)/float(N);
  ACCUMULATOR     : float := 1.0;
  MULTIPLIER      : float := 4.0;
  DIV_3N          : float := (3.0 * float (N));
 begin
  for INDEX in 1..(N - 1) loop
   X_POINT := X_POINT + INCREMENT;
   ACCUMULATOR := (1.0 / X_POINT) *
               MULTIPLIER + ACCUMULATOR;
   if MULTIPLIER = 4.0 then
               MULTIPLIER := 2.0;
   else MULTIPLIER := 4.0;
   end if;
  end loop;
  ACCUMULATOR := ACCUMULATOR +
       (1.0 / (X_POINT + INCREMENT));
  LN_X := ((X - 1.0) / DIV_3N) * ACCUMULATOR;
       return LN_X;
 end LN;
-----------------------------------------------
 function UNIFORM (LOW   : in float := 0.0;
      HIGH  : in float := 1.0) return float is
   SEED          : integer := 2096730329;
   B2E15         : integer := 32768;
   B2E16         : integer := 65536;
   MODULUS       : integer := 2147483647;
   MULTIPLIER    : integer := 24112;
   HIGH_15, HIGH_31, LOW_15, LOW_PRODUCT,
   OVERFLOW      : integer;
   RAND          : float;
 begin
  for INDEX in 1..2
   loop
    HIGH_15      := SEED / B2E16;
    LOW_PRODUCT := (SEED - HIGH_15 * B2E16)
                              * MULTIPLIER;
    LOW_15 := LOW_PRODUCT / B2E16;
    HIGH_31 := HIGH_15 * MULTIPLIER + LOW_15;
    OVERFLOW := HIGH_31 / B2E15;
    SEED := (((LOW_PRODUCT - LOW_15 * B2E16)
        - MODULUS) + (HIGH_31 - OVERFLOW *
        B2E15) * B2E16) + OVERFLOW;
    if SEED < 0 then
      SEED := SEED + MODULUS;
    end if;
    MULTIPLIER := 26143;
   end loop;
   MULTIPLIER := 24112;
   RAND := float (2 * (SEED / 256) + 1)
                          / 16777216.0;
    return ((HIGH - LOW) * RAND + LOW);
 end UNIFORM;
-----------------------------------------------
 function EXPONENTIAL (LAMBDA : in float)
                         return float is
  RAND   : float;
 begin
  for INDEX in 1..3
   loop
    begin
     return -LAMBDA * LN (UNIFORM);
     exit;
     exception
      when NUMERIC_ERROR   =>
             null;
     end;
   end loop;
 end EXPONENTIAL;
end RANDOM;
-----------------------------------------------
with CLOCK, text_io; use CLOCK, text_io;
package PROCESS_SIMULATION is
 type TIME is digits 5 range -1.0..float'large;
 task type SYNCH is
  entry WAIT;
  entry SEND;
 end SYNCH;
 --future_event_notice describes when a held
 --process should be restarted. It contains
 --a variable of type synch used to delav
 --a held process, the time the process should
 --be resumed and a ptr to the next notice;
 type FUTURE_EVENT_NOTICE;
 type F_EVENT_LINK is access FUTURE_EVENT_NOTICE;
 type FUTURE_EVENT_NOTICE is
  record
   SCHED_TIME : TIME; --simulation time of task
   SIGNAL     : SYNCH;  -- used to delay task
   NEXT : F_EVENT_LINK; --points to next notice
  end record;
 procedure HOLD (DELAY_TIME: in TIME);
 --Scheduler is the simulation control routine.
 --scheduler.start is called to begin simulation.
 --scheduler.next  communicates between
 --procedure schedule and the scheduler, and to
 --allow a simulation task that is going to
 --terminate without calling hold to inform the
 --scheduler to proceed with the next task.
 task SCHEDULER is
  entry START(N: in integer);
  entry NEXT;
  entry SPAWN;
  entry WAIT;
 end SCHEDULER;
 --END_SIMULATION terminates active tasks.
 procedure END_SIMULATION;
 --task EVENT_CHAIN manages future events list
 task EVENT_CHAIN is
  entry INSERT (NEW_NOTICE: in F_EVENT_LINK);
  entry REMOVE;
 end EVENT_CHAIN;
end PROCESS_SIMULATION;
-----------------------------------------------
package body PROCESS_SIMULATION is
 --first_f_event - head of the events chain.
 FIRST_F_EVENT: F_EVENT_LINK := NULL;
 type ACTIVE_TASKS is range 0..integer'last;
 NUMBER_ACTIVE: ACTIVE_TASKS;
 INDEX:       ACTIVE_TASKS;
 task body SYNCH is
```

```
  begin
   accept SEND;
   accept WAIT;
  end SYNCH;
 task body EVENT_CHAIN is
  TRAILING_PTR   : F_EVENT_LINK;
  LOOP_PTR       : F_EVENT_LINK;
 begin
  loop
   select
    accept INSERT(NEW_NOTICE:in F_EVENT_LINK)do
     if FIRST_F_EVENT = null then
      FIRST_F_EVENT         := NEW_NOTICE;
      NEW_NOTICE.NEXT       := null;
     else
      TRAILING_PTR          := null;
      LOOP_PTR              := FIRST_F_EVENT;
      loop
      if NEW_NOTICE.SCHED_TIME <
                LOOP_PTR.SCHED_TIME then
       if TRAILING_PTR = NULL then
        NEW_NOTICE.NEXT := FIRST_F_EVENT;
        FIRST_F_EVENT   := NEW_NOTICE;
        EXIT;  -- leave enclosing loop
       else
         -- Insert after trailing_ptr
         NEW_NOTICE.NEXT     := LOOP_PTR;
         TRAILING_PTR.NEXT   := NEW_NOTICE;
         EXIT;
        end if;
       end if;
       -- No. Advance down list.
       TRAILING_PTR     := LOOP_PTR;
       LOOP_PTR         := LOOP_PTR.NEXT;
       -- Check for insert at end of list.
       if LOOP_PTR = null then
         TRAILING_PTR.NEXT := NEW_NOTICE;
         NEW_NOTICE.NEXT   := null;
         EXIT;
       end if;
      end loop;
     end if;
    end INSERT;
    or
    accept REMOVE do
     if FIRST_F_EVENT /= null then
      --Advance the simulation time to
      --when the delayed task should be resumed.
      SIM_TIME := float(FIRST_F_EVENT.SCHED_TIME)
      NUMBER_ACTIVE := 0;
      loop
        --Task delayed by future_event_notice
        --has executed a signal.wait call. Start
        --task up by executing a signal.send.
        FIRST_F_EVENT.SIGNAL.SEND;
        NUMBER_ACTIVE := NUMBER_ACTIVE + 1;
        --Advance the head of the future event
        --set to the next future event notice.
        FIRST_F_EVENT := FIRST_F_EVENT.NEXT;
        if float (FIRST_F_EVENT.SCHED_TIME) >
              SIM_TIME then exit; end if;
      end loop;
     end if;
    end REMOVE;
   end select;
  end loop;
 end EVENT_CHAIN;
---------------------------------------------
task body SCHEDULER is
begin
 accept START(N: in integer) do
  NUMBER_ACTIVE := ACTIVE_TASKS(N);
 end START;
 loop  -- forever
  -- Wait until either the task we started
  -- calls scheduler.next or the task
  -- informs us it is not going to continue
  -- by calling scheduler.next.
  INDEX := 0;
   loop
    select
     accept NEXT;
       INDEX := INDEX + 1;
     or
     accept WAIT;
       NUMBER_ACTIVE := NUMBER_ACTIVE - 1;
     or
     accept SPAWN;
       NUMBER_ACTIVE := NUMBER_ACTIVE + 1;
    end select;
    if INDEX=NUMBER_ACTIVE then exit;end if;
   end loop;
   --If future event set is non-empty, pick
   --the next future event notice and
   --resume the task described by that notice.
   EVENT_CHAIN.REMOVE;
  end loop;
 end SCHEDULER;
---------------------------------------------
 procedure HOLD (DELAY_TIME     : in TIME) is
  NEW_NOTICE     : F_EVENT_LINK;
 begin
  -- Create a new future event notice
  NEW_NOTICE     := new FUTURE_EVENT_NOTICE;
  NEW_NOTICE.SCHED_TIME := TIME
              (SIM_TIME) + DELAY_TIME;
  --Insert the event notice on the chain
  EVENT_CHAIN.INSERT (NEW_NOTICE);
  SCHEDULER.NEXT;
  NEW_NOTICE.SIGNAL.WAIT;
 end HOLD;
---------------------------------------------
 procedure END_SIMULATION is
 begin
  abort SCHEDULER;
  while FIRST_F_EVENT /= null
   loop
     abort FIRST_F_EVENT.SIGNAL;
     FIRST_F_EVENT := FIRST_F_EVENT.NEXT;
   end loop;
  abort EVENT_CHAIN;
 end END_SIMULATION;
end PROCESS_SIMULATION;
---------------------------------------------
with CLOCK,PROCESS_SIMULATION,QUE,RANDOM,STAT,
 SPECIAL_STATS, text_io;
use CLOCK, PROCESS_SIMULATION, text_io;
procedure EXAMPLE_SIMULATION is
 RUN_TIME: constant := 25.0; -- run for time 10
 LAMBDA : constant  :=  2.0; -- interarrival
 MU     : constant  :=  1.5; -- service time
 -- Customers in the system are represented by
 --variables of type job.
 type JOB is
 record
   ARRIVAL_TIME : simulation.TIME;
   JOB_ID       : NATURAL;
 end record;
type JOB_PTR is access JOB;
package JOB_QUEUE is new
             QUE (ENTITY_REC=> JOB,
                  PTR_ENTITY=> JOB_PTR);
WAITING_QUEUE : JOB_QUEUE.QUEUE;
JOB_IN_SERVICE : boolean := false;
type USER_VAR is (TIME_IN_SYSTEM,
                  MY_COUNT, MYSTERY_VAR);
subtype TALLY_VAR is USER_VAR
       range TIME_IN_SYSTEM..MY_COUNT;
subtype ACCUM_VAR is USER_VAR
       range MY_COUNT..MYSTERY_VAR;
```

```
package MYSTATS is new SPECIAL_STATS
       (USER_VAR  => USER_VAR,
        ACCUM_VAR => ACCUM_VAR,
        TALLY_VAR => TALLY_VAR);
use MYSTATS;
task ARRIVAL is
 entry START;
 entry STOP;
end ARRIVAL;
task SERVICE is
 entry START;
 entry WAKEUP;
end SERVICE;
--Entry start is used to initiate simulation.
--service.wakeup is used to restart service
--process at the end of an idle period.
----------------------------------------------
task body ARRIVAL is
 NEW_JOB              : JOB_PTR;
 ARRIVAL_COUNT        : INTEGER := 0;
 INTER_ARRIVAL_TIME   : TIME;
 ARRIVAL_NOTICE       : F_EVENT_LINK;
 package A_RAND is new RANDOM (SEED => 1);
 use A_RAND;
begin
 accept START;  --Wait for simulation start.
 INTER_ARRIVAL_TIME := TIME
              (A_RAND.EXPONENTIAL(LAMBDA));
 HOLD(INTER_ARRIVAL_TIME);
 loop          -- forever
  -- Create a new job.
  NEW_JOB := new JOB;
  -- Set attributes of the new job.
  NEW_JOB.ARRIVAL_TIME := TIME (SIM_TIME);
  ARRIVAL_COUNT := ARRIVAL_COUNT + 1;
  NEW_JOB.JOB_ID := ARRIVAL_COUNT;
  --Place new job in waiting queue.
  --If system is idle, wake up service task.
  if ((JOB_QUEUE.IS_EMPTY(WAITING_QUEUE))
          and (not JOB_IN_SERVICE)) then
   JOB_QUEUE.PUT(NEW_JOB, WAITING_QUEUE);
   SCHEDULER.SPAWN;
   SERVICE.WAKEUP;
  else
   JOB_QUEUE.PUT(NEW_JOB, WAITING_QUEUE);
  end if;
  -- Wait for time of next_arrival.
  INTER_ARRIVAL_TIME :=
          TIME(A_RAND.EXPONENTIAL(LAMBDA));
  HOLD(INTER_ARRIVAL_TIME);
 end loop;
end ARRIVAL;
----------------------------------------------
task body SERVICE is
 DEPARTING_JOB : JOB_PTR;
 SERVICE_NOTICE: F_EVENT_LINK;
 SERVICE_TIME  : TIME;
 package S_RAND is new RANDOM (SEED => 3);
begin
 accept START; -- Wait for simulation start.
 loop         -- forever
  if JOB_QUEUE.IS_EMPTY(WAITING_QUEUE) then
      SCHEDULER.WAIT;
      accept WAKEUP;
  end if;
  -- Take the first job out of waiting queue.
  JOB_QUEUE.TAKE(WAITING_QUEUE,DEPARTING_JOB);
  SERVICE_TIME :=TIME(S_RAND.EXPONENTIAL(MU));
  JOB_IN_SERVICE := true;
  HOLD(SERVICE_TIME);
  JOB_IN_SERVICE := false;
  --Observe time in system -record statistics.
  ASSIGN (TIME_IN_SYSTEM, SIM_TIME -
      float (DEPARTING_JOB.ARRIVAL_TIME));
  ASSIGN(MY_COUNT, VARIABLE(MY_COUNT) + 3.0);
 end loop;
end SERVICE;
----------------------------------------------
--The main procedure initializes the
--waiting_queue, starts the simulation tasks,
--and then blocks itself until the simulation
--run time has expired:
begin --main program
 -- Initialize the waiting queue
 JOB_QUEUE.INIT(WAITING_QUEUE);
 --Scheduler, arrival, service blocked now;
 -- start them and the simulation.
 SCHEDULER.START(3);
 ARRIVAL.START;
 SERVICE.START;
 HOLD(RUN_TIME);
 --All that is left  is stop the simulation
 --tasks and print statistics:
 ABORT ARRIVAL;
 ABORT SERVICE;
 ABORT JOB_QUEUE.Q_OPERATIONS;
 ABORT MYSTATS.ASSIGNMENT;
 END_SIMULATION;
 -- Print statistics
 MYSTATS.PRINT_STATS;
 JOB_QUEUE.PRINT_STATS;
end EXAMPLE_SIMULATION;
```

APPENDIX B

*

EVENT ORIENTATION

```
with CLOCK;      use CLOCK;
generic
 type NOTICE is (<>);
 with procedure USER_ROUTINES (EVENT_CHOICE :
                                   in NOTICE);
package EVENT_SIMULATION is
 procedure SCHEDULER (SIM_END : in float);
  --SCHEDULER scans the event chain, updates
    the clock, and calls event procedures.
 procedure CREATE_EVENT_NOTICE (EVENT_TIME :
         in float; EVENT_TYPE : in NOTICE);
  --CREATE_EVENT_NOTICE creates a new event
    notice and files it in the event notices
    queue by its scheduled time for execution.
end SIMULATION;
----------------------------------------------
package body EVENT_SIMULATION is
 type FUTURE_EVENT_NOTICE;
 type F_EVENT_LINK is access
               FUTURE_EVENT_NOTICE;
 type FUTURE_EVENT_NOTICE is
  record
   SCHED_TIME  : float;
   NOTICE_TYPE : NOTICE;
   NEXT        : F_EVENT_LINK;--next event ptr
  end record;
  --The record FUTURE_EVENT_NOTICE is entered
  --on the chain by schedule time.
  --The NOTICE_TYPE entry in the record names
  --the event routine that should be called.
 FIRST_F_EVENT  : F_EVENT_LINK := null;
  --FIRST_F_EVENT is a pointer to the head of
  --the future events chain.
----------------------------------------------
procedure SCHEDULER (SIM_END : in float) is
begin
 loop
  if SIM_TIME >= SIM_END then
   exit;--terminate simulation when time ends.
   elsif FIRST_F_EVENT = null then
    exit; --or terminate simulation when no
          --more events are on events chain
   else
    --Update system clock
    SIM_TIME := FIRST_F_EVENT.SCHED_TIME;
    --Call the event routine indicated
    --by the event notice.
    USER_ROUTINES (EVENT_CHOICE =>
                   FIRST_F_EVENT.NOTICE_TYPE);
    --Remove the first event from the chain
    FIRST_F_EVENT := FIRST_F_EVENT.NEXT;
   end if;
  end loop;
end SCHEDULER;
--------------------------------------------
procedure CREATE_EVENT_NOTICE (EVENT_TIME :
         in float; EVENT_TYPE : in NOTICE) is
  NEW_NOTICE     : F_EVENT_LINK;
  TRAILING_PTR   : F_EVENT_LINK;
  LOOP_PTR       : F_EVENT_LINK;
begin
 --Create a new future event notice;
 NEW_NOTICE      := new FUTURE_EVENT_NOTICE;
 --Enter scheduled event time in the notice.
 NEW_NOTICE.SCHED_TIME := SIM_TIME+EVENT_TIME;

 <Rest of procedure is a procedure version
  of task EVENT_CHAIN in process version>

end CREATE_EVENT_NOTICE;
end SIMULATION;
--------------------------------------------
with CLOCK, QUE,EVENT_SIMULATION,RANDOM,STAT,
     SPECIAL_STATS, TEXT_IO;
use  CLOCK, RANDOM,TEXT_IO;
procedure EXAMPLE_SIMULATION is
 RUN_TIME  : constant := 25.0;
 LAMBDA    : constant := 2.0;--interarrival time
 MU        : constant := 1.5;--Service time.
 SERVER_BUSY               : boolean := false;
 ARRIVAL_COUNT             : integer := 0;
 type JOB;
 type JOB_PTR is access JOB;
 type JOB is
  record
   ARRIVAL_TIME : float;
   JOB_ID       : natural;
  end record;
 JOB_IN_SERVICE                      : JOB_PTR;
 package TIME_IO is new FLOAT_IO (float);
 --An instance of the generic package QUEUE
 --is used to represent a queue of jobs.
 package JOB_QUEUE is new QUE(ENTITY_REC => JOB
                         PTR_ENTITY => JOB_PTR);
WAITING_QUEUE : JOB_QUEUE.QUEUE;
type USER_VAR is (TIME_IN_SYSTEM,
                  MY_COUNT, MYSTERY_VAR);
subtype TALLY_VAR is USER_VAR range
                    TIME_IN_SYSTEM..MY_COUNT;
subtype ACCUM_VAR is USER_VAR range
                  MY_COUNT..MYSTERY_VAR;
package MYSTATS is new SPECIAL_STATS
               (USER_VAR  => USER_VAR,
                TALLY_VAR => TALLY_VAR,
                ACCUM_VAR => ACCUM_VAR);
use MYSTATS;
--The modeler-defined event routine types are
--passed as a parameter to EVENT_SIMULATION.
--This allows the package to generalize to a
--variable number and type of event routines
--specified by the modeler.
type EVENT_FORM is (ARRIVAL,
                    SERVICE_BEGIN, SERVICE_END);
--For an M/M/1 system, an arrival event,a begin
--service event, and an end service event may be
--used to model the system. These events are
--encapsulated in the MODEL_ROUTINES procedure.
procedure MODEL_ROUTINES (SELECT_EVENT :
                            in EVENT_FORM);
--An instance of the generic package
--EVENT_SIMULATION controls the simulation.
package SYSTEM_SIMULATION is new
                      EVENT_SIMULATION
         (NOTICE => EVENT_FORM,
          USER_ROUTINES => MODEL_ROUTINES);
--------------------------------------------
procedure MODEL_ROUTINES (SELECT_EVENT :
                     in EVENT_FORM) is
 procedure ARRIVAL_EVENT is
  NEW_JOB               : JOB_PTR;
 begin
  --Schedule next arrival.
  CREATE_EVENT_NOTICE
              (EXPONENTIAL(LAMBDA),ARRIVAL);
  --Create a new job.
  NEW_JOB := new JOB;
  --Set attributes of new job.
  NEW_JOB.ARRIVAL_TIME := SIM_TIME;
  ARRIVAL_COUNT       := ARRIVAL_COUNT + 1;
  NEW_JOB.JOB_ID      := ARRIVAL_COUNT;
  --If server not busy,schedule service begin
  if not SERVER_BUSY then
   CREATE_EVENT_NOTICE (0.0, SERVICE_BEGIN);
  end if;
  --Place new job in waiting queue.
  JOB_QUEUE.PUT (NEW_JOB, WAITING_QUEUE);
 end ARRIVAL_EVENT;
--------------------------------------------
 procedure BEGIN_SERVICE_EVENT is
 begin
  --Remove first job from the waiting queue.
  JOB_QUEUE.TAKE(WAITING_QUEUE,JOB_IN_SERVICE);
  SERVER_BUSY := true;
  --Schedule an end of service event;
  CREATE_EVENT_NOTICE
              (EXPONENTIAL(MU),SERVICE_END);
 end BEGIN_SERVICE_EVENT;
--------------------------------------------
 procedure END_SERVICE_EVENT is
   TIME_IN_SERVER        : float;
 begin
  SERVER_BUSY := false;
  --Collect statistics.
  ASSIGN (TIME_IN_SYSTEM,SIM_TIME -
              JOB_IN_SERVICE.ARRIVAL_TIME);
  ASSIGN(MY_COUNT, VARIABLE(MY_COUNT) + 3.0);
  if not(JOB_QUEUE.IS_EMPTY(WAITING_QUEUE))then
   --Schedule a beginning of service.
   CREATE_EVENT_NOTICE (0.0, SERVICE_BEGIN);
  end if;
  end END_SERVICE_EVENT;
--------------------------------------------
 begin  --MODEL_ROUTINES
  case SELECT_EVENT is
   when ARRIVAL        => ARRIVAL_EVENT;
   when SERVICE_BEGIN => BEGIN_SERVICE_EVENT;
   when SERVICE_END    => END_SERVICE_EVENT;
  end case;
 end MODEL_ROUTINES;
--------------------------------------------
--------------------------------------------
begin            --Main program
 --Initialize waiting queue
 JOB_QUEUE.INIT (WAITING_QUEUE);
 --Schedule first arrival to start simulation.
 CREATE_EVENT_NOTICE
              (EXPONENTIAL(LAMBDA),ARRIVAL);
 --Call SCHEDULER to run simulation.
 SYSTEM_SIMULATION.SCHEDULER (RUN_TIME);
 --Print statistics.
 MYSTATS.PRINT_STATS;
 JOB_QUEUE.PRINT_STATS;
end EXAMPLE_SIMULATION;
```

*Packages common to both orientations are ommited.

REFERENCES

1. Kiviat, P.J., R. Villanueva, H. Markowitz, (edited by E.C. Russell) *SIMSCRIPT II.5 Programming Language*, CACI, Inc., Los Angeles, 1983

2. Bryant, Raymond M., "Discrete System Simulation in Ada", *SIMULATION*, October 1982, pp. 111-121.

3. Sheppard, Sallie, Patricia Friel and Donna Reese, "Simulation in Ada: an implementation of two world views", *Simulation in Strongly Typed Languages: Ada, Pascal, Simula*, Vol. 13 No. 2, (February 1984) pp. 3-9.

4. *Reference Manual for the Ada Programming Language*, US Department of Defense, Ada Joint Program Office (MIL-STD 1815A) July 1982.

5. Maise, Kien and Stephen D. Roberts, "Implementing a portable FORTRAN Uniform (0,1) generator", *SIMULATION*, October 1983, pp. 135-139.

6. *Communications Oriented Production Information and Control System*, Vol. V., IBM Corporation, New York, 1972.

7. Sheppard, Sallie, Don T. Phillips and Robert E. Young, "The Design and Implementation of a Microprocessor-Based Distributed Digital Simulation System", Grant No ECS-8215550, June 1983.

8. Wyatt, Dana L., Sallie Sheppard and Robert E. Young, "An Experiment in Microprocessor-Based Distributed Digital Simulation", *Proceeding of the 1983 Winter Simulation Conference*, December 1983 pp. 270-277.

9. Krishnamurthi, Murali and Robert E. Young, "Multitasking Implementation of System Simulation: The Emulation of an Asynchronous Parallel Processor Using a Single Processor", *Proceedings of the 1984 Winter Simulation Conference*.