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

A TUTORIAL ON SIMULATION PROGRAMMING WITH SIMPAS

R. M. Bryant*
Department of Computer Science
University of Wisconsin-Madison
1210 W. Dayton Street
Madison, Wisconsin 53706

SIMPAS is a portable, strongly-typed, event-oriented, discrete system simulation language embedded in PASCAL. It extends PASCAL by adding statements for event declaration and scheduling, entity declaration, creation and destruction, linked list declaration and manipulation, and statistics collection. A library of standard pseudo-random number generators is also provided. This paper gives a tutorial on simulation programming using SIMPAS. We briefly discuss event-oriented simulation language concepts, give an overview of the programming language PASCAL, and then describe in detail the simulation extensions that SIMPAS provides.

1. INTRODUCTION

Over the past two years, we have been developing a strongly-typed, discrete-system simulation language embedded in PASCAL. SIMPAS is the result of this development effort. Previous papers on SIMPAS have discussed the advantages of using strongly-typed languages for simulation program development (Bryant 1980), and an experimental version of SIMPAS that executed (slowly!) on an LSI-11 microcomputer system (Bryant 1981a). This paper is a tutorial describing the use of the present version of the SIMPAS system for the creation of a simple simulation. A detailed description of this version of SIMPAS is available in the version 5.0 SIMPAS user manual (Bryant 1981b).

Succinctly stated, SIMPAS provides the following extensions to PASCAL:

(1) Event declaration and scheduling statements.

(2) Entity declaration, creation and disposal statements.

(3) Linked list declaration and manipulation statements.

(4) Statistics collection statements.

(5) A predeclared library of psuedo-random number generators.

Furthermore, SIMPAS is a closed system in the sense that even though it is implemented as a preprocessor for PASCAL, the user need not be aware of this. The extension statements can be intermixed with standard PASCAL statements in a natural way. Also, the preprocessor automatically builds, inserts, and initializes all data structures necessary for the simulation. This is to be contrasted with PASCAL simulation packages such as PASSIM (Uyeno 1980), which although they are substantially simpler than SIMPAS, require the user to know many details of the package implementation and to assist in declaration and initialization of the package interface variables.

The preprocessor implementation was chosen to make SIMPAS highly portable without sacrificing execution efficiency. On most systems where it is installed, the preprocessing and compilation phases can be combined under control of a single command procedure so that they are essentially transparent to the user.

*This work was supported in part by NSF grant MCS-800-3341.

Careful attention has been paid to the problems of tracing error messages from the output PASCAL back to the SIMPAS source, as well as reporting run time errors in terms of the original SIMPAS source line whenever possible.

From the standpoint of teaching simulation, SIMPAS has been especially successful. Since more and more students are being exposed to PASCAL, it has become a relatively simple matter to state the SIMPAS extensions and then give the students a simple simulation assignment. Furthermore, since SIMPAS inherits strong typing from PASCAL, the resulting simulation programs are reliable and easy to debug. Thus primary effort can be directed toward understanding the simulation problem itself, rather than tracing down numerous storage exception faults and other hardware detected errors.

In the next sections of this paper, we first introduce the basic concepts of event-oriented, discrete-system simulation, and then outline some of the features of PASCAL that make it especially well suited to the implementation of discrete-system simulation. We then discuss the SIMPAS extensions to the programming language PASCAL and described a sample SIMPAS program.

2. SIMULATION CONCEPTS

In this section we briefly discuss the concepts fundamental to event-oriented, discrete-system simulation. For further details the reader is directed to (Fishman 78).

2.1. Event-oriented Simulation

SIMPAS is an "event-oriented" discrete-system simulation language. This means that changes in the state of the simulated system are modelled by the occurrence of "events". (SIMULA, on the other hand, is "process-oriented." See (Franta 1977) for a description of the process view of simulation.) An event is an idealization of a system state change that is assumed to occur instantaneously. To represent activities in the simulation that occur over an extended period of time (for example, the movement of a box along a conveyor belt) one uses a pair of events. One of the events represents the start of the activity (the beginning of the box's movement) and one of the events represents the end of the activity (the arrival of the box at its destination).

In the simulation program, each event is represented by a procedure that is called when the event occurs; this procedure is called the "event routine" associated with the event. Occurrence of an event is modelled by the execution of the event routine, and changes in the system state are represented by changes in the values of the variables in the simulation program. To continue our previous example, the event routine associated with the start of a box's journey down the conveyor belt would remove the box from its previous location (perhaps merely by decrementing the number of boxes found there) and set a variable to indicate that the conveyor was occupied. The event routine representing the box's arrival event would mark the conveyor empty and see that the box is sent on to its new location. Thus there is a one to one correspondence between execution of an event routine in the simulation program and the occurrence of events in the simulated system. For this reason it is common to refer to the execution of an event routine as the occurrence of an event in spite of the fact that the event itself is part of the simulated system while the event routine is part of the simulation program.

Event routines are called in response to scheduling statements executed by the simulation program. There are various forms of scheduling statements in SIMPAS, but what is essential is that the scheduling statement specifies the simulated time when the event is to occur and the values of any actual parameters (arguments) that the event routine should be called with. One can thus think of a scheduling statement as a "delayed call" on the event routine. It is like a normal procedure call in that values for the actual parameters are provided, but the routine is called not at the present simulated time but at a specified time in the future.

Since the scheduling statement does not actually call the event routine, the schedule statement code records the event time and its parameters in an "event notice". The event notice holds these values until they are needed by the event routine. A separate event notice is created for each occurrence of each event in the simulation. After creation, the event notice is inserted into a list of event notices called the "event set". This set is ranked by increasing simulation time and contains event notices for all events that have been scheduled but that have not yet occurred. We will say that an event notice is scheduled so long as it is in the event set.

Event notices are removed from the event set and event routines are called by a procedure in the simulation called "simulation control routine". The heart of the simulation control routine is a loop that consists of the following steps:

(1) Remove the next event notice from the event set. If the event set is empty, the simulation control routine returns to its caller (normally the main program). This stops the simula-

tion.

(2) Advance the simulation clock to the simulation time of the event notice.

(3) Call the appropriate event routine with the arguments as saved in the event notice.

These steps are repeated over and over throughout the simulation. So long as the event set is non-empty, execution of this loop causes simulation time to advance in an orderly fashion and events to occur in their scheduled order. Events scheduled before the simulation control routine is called do not occur until after the control routine is called. For this reason, one says that the simulation is "active" from the time when the simulation control routine is called until it returns.

2.2. Entities

Just as an event is an idealization of the state change in a simulated system, an "entity" is a idealization of the objects which move through the the system (the box in the conveyor belt example). Entities can represent jobs in a computer system, customers in a bank, or cars in a car wash. An entity may have distinguishing features such as an arrival time, a color, or a service requirement. Using the SIMSCRIPT II.5 (Kiviat 1974) terminology, we refer to these quantities as "attributes" of the entity.

Entities are normally divided into two classes: temporary and permanent. Temporary entities represent transient objects that are created, move through the simulation and are then destroyed. Permanent entities exist throughout the simulation. Thus while temporary entities could represent the jobs moving through a computer system, permanent entities might be used to represent the system itself. In the next section, we will discuss how the features of PASCAL can be used to represent temporary and permanent entities and their attributes.

3. A VERY BRIEF OVERVIEW OF PASCAL

In this limited space, it is clearly impossible to provide a detailed introduction to PASCAL, let alone the SIMPAS extensions to PASCAL. We will therefore assume that the reader is familiar with some modern, block structured programming language such as PL/I, ALGOL, or SIMULA. In this section, we merely wish describe those features of PASCAL that are not found in these other languages and that are significant to our discussion of SIMPAS. For further information about PASCAL see (Jensen 74).

In the sequel, we will underline PASCAL (and SIMPAS) language keywords. We will use angle-brackets ("<" and ">") to represent portions of statements which are to be replaced by appropriate user constructs. Thus the notation: <identifier> indicates that the user is to insert an identifier at this location. We will use square brackets to indicate a portion of a statement which may be omitted. Finally, we will use an ellipsis (". . .") to indicate zero or more repetitions of the preceding construct.

3.1. PASCAL Program Structure

A PASCAL program can be divided into seven major parts: the label declaration part (which will not concern us here), the constant declaration part, the type declaration part, the variable declaration part, the procedure and function declarations, and the main procedure. Any of the declaration parts may be empty so that only the declaration parts that are actually required need be included. Each procedure and function has the same structure as a PASCAL program in the sense that each procedure and function has the same parts as above, except, of course, that a procedure has a procedure body instead of the main procedure. The declaration parts in the procedures and functions apply only to that procedure or function, while the declaration parts for the main program define identifiers known throughout the program. To distinguish between these two sets of declaration parts of the program, we will refer to the ones at the main procedure level as the "global" constant declaration part, the "global" type declaration part, and so forth.

3.2. User Defined Types

Of these program parts, the only one which is peculiar to PASCAL is the "type" part. In PASCAL, there is a clear distinction between a variable and its type. Every variable must be declared as a specific type; this type can be one of the base types (integer, real, boolean, or character) or a defined type built up out of the base types and record or array declarations. One refers to a defined type by giving it a name in the type declaration part of the program or procedure.

However, there is no mechanism for defining operations on these new types. Instead, user defined types are used to define data structures and to insure that identical objects have the same declarations. For example, suppose that one wishes to declare A and B to be arrays of integers, each 100 elements long. One could declare A and B as follows:

```
var
    A : array [1..100] of integer;
    B : array [1..100] of integer;
```

If the declaration for A is now changed, the programmer that made the change will have to remember to change the declaration for B as well. This can be a demanding task if the declarations for A and B are separated by a few hundred or thousands of lines of code! To avoid this, we could declare a user defined type:

```
type
    intarray = array [1..100] of integer;
```

then declare A and B as instances of this array:

```
var
    A : intarray;
    B : intarray;
```

Now any changes made to the declaration of "intarray" will properly migrate to all instances of the involved arrays.

User defined types also simplify the construction of structured or record types. For example, one can create a SIMSCRIPT II.5-like entity with the following declarations:

```
type
    box = record
        arrival_time : real;
        destination  : destination_id;
        weight       : integer;
    end;

var
    some_box    : box;
    another_box : box;
```

(Note: The underbar character "_" is not part of the standard PASCAL character set, although it is used in many implementations to improve readability of variable names.)

In this example, "some_box" and "another_box" refer to two different variables of type "box". Each has their own copies of the "fields" arrival_time, destination, and weight. One refers to the fields of a record using the PASCAL dot notation:

```
    some_box.arrival_time
    some_box.destination
    some_box.weight
```

The PASCAL with statement can be used to abbreviate references to field variables. For example, consider the following statement:

```
    with some_box do
    begin
        . . .
    end;
```

In between the begin and end in this statement, references to "weight" are interpreted as references to "some_box.weight". Other fields of "some_box" can be referenced in the same way.

3.3. Entity Representation

As seen above, PASCAL records can be used to represent entities in a simulation and the record fields can be thought of as the attributes of the entity. For example, to represent a collection of conveyor belts, one might use the following declarations (comments in PASCAL are delimited by { and }):

```
const
  {the total number of belts in the shop}
  number_belts = 10;

type
  {a conveyor belt id is a number between
   1 and number_belts}
  cv_belt_id = 1..number_belts;

  cv_belt = record
    {is belt in use? true or false}
    busy           : boolean;

    {how long does it take to move
     a box down the belt?}
    move_time      : real;

    {counts number of boxes on the belt}
    number_boxes    : integer;

    {counts number of boxes delivered}
    delivered_boxes : integer;
  end;

var
  conveyor_belt : array [cv_belt_id]
                  of cv_belt;
```

These declarations declare a set of 10 conveyor belts, each with a busy flag, a real variable indicating how long it takes to move a box down the conveyor belt, and two count fields. The 3rd conveyor belt is described by the record stored at conveyor_belt[3] and has attributes:

```
    conveyor_belt[3].busy
    conveyor_belt[3].move_time
    conveyor_belt[3].number_boxes
    conveyor_belt[3].delivered_boxes
```

This representation is suitable for permanent entities that exist throughout a simulation, but is not convenient for representation of temporary entities, since the number of conveyor belts is fixed at compilation time. Instead one should use a PASCAL "pointer" variable:

```
var
    boxp : ^box;
```

The variable "boxp" is a pointer at the record "box" (which we are using to represent an entity). One can convert a pointer variable into an object through the use of the dereference operator: "^". Thus boxp^ is a box record and its fields can be referred to as

```
    boxp^.arrival_time
    boxp^.destination
    boxp^.weight
```

The advantage of the pointer representation for temporary entities is that new instances of the entity can be generated using the PASCAL procedure "new". Thus, to create a new box, one can say:

```
    new(boxp);
```

The previous value of boxp (if any) is lost so that the previous record pointed to by boxp is no longer accessible as boxp^. Instead a new record is now available.

Similarly, one can destroy a previously created entity using the PASCAL procedure "dispose":

```
    dispose(boxp);
```

One of the problems in using "new" is that the fields of a record created in this way are undefined and must be explicitly initialized by the user. Similarly, if you wish to insert an entity into a linked list, you must explicitly declare and set the link fields. SIMPAS provides mechanisms for automatically doing both of these tasks through the insert, remove, and create statements, and the queue member and queue declarations. These statements are discussed in the next section.

4. SIMPAS LANGUAGE DESCRIPTION

We now discuss the simulation extensions to PASCAL which have been incorporated into SIMPAS. For simplicity, this presentation skips some non-essential details. A more precise description of the language extensions is available in the latest version of the SIMPAS user manual (Bryant 81).

4.1. SIMPAS Program Structure

A SIMPAS program has essentially the same structure as a PASCAL program. The only differences are that an "include" statement has been added to allow insertion of predefined procedures and types from the SIMPAS source library and that event declarations can appear in the global procedure and function declaration part of the main program.

4.2. The Include Statement

Because external compilation of PASCAL procedures is not part of standard PASCAL, there is no completely transportable way to create a library of PASCAL routines. Since implementing a library of pseudo-random number generation routines was necessary for SIMPAS, we implemented a symbolic library. The include statement indicates which portions of the symbolic library are to be included in the program. The include statement is found at the start of the procedure, function, and event declaration part of the program and has the form:

```
include <section> [, <section> ] . . .;
```

Each section specifies a portion of the library to be included. For example, to include the exponential pseudo-random number generator "expo" in the program, one would use this include statement:

```
include expo;
```

For each section, all global constant, type, and variable declarations required by that section are also included. Thus if "expo" required a special global variable to function properly, the library can be configured to include this variable in the source program whenever expo is included.

4.3. Event Declaration

An event declaration has exactly the same form as a PASCAL procedure declaration, except that the reserved word event replaces the reserved word procedure. To continue our conveyor belt example:

```
event box_moves(belt : cv_belt_id);

    {belt tells which of the conveyor
     belts we are using}

begin

    {mark the belt as being busy and
     move a box onto the belt}

    with conveyor_belt[belt] do
    begin
      busy := true;
      number_boxes := number_boxes - 1
    end;

    {schedule the arrival event }
    schedule box_moved(belt) delay
           conveyor_belt[belt].move_time;

end;

event box_moved(belt : cv_belt_id);

begin

  {move the current box of off the belt
   and mark the belt not busy if it
   is empty}

  with conveyor_belt[belt] do
  begin
    number_boxes := number_boxes - 1;
    if number_boxes = 0 then busy:=false;
    delivered_boxes:=delivered_boxes+1;
  end;

end;
```

This code declares two events, one to represent the start of movement of a box down the conveyor belt, and the latter to represent the arrival of the box at the end of the conveyor belt. The scheduling statement assures that the arrival event occurs at the proper time. (We will discuss the scheduling statements in more detail below).

4.4. Start Simulation

To activate the simulation (i. e. call the simulation control routine), one uses the statement:

```
start simulation(status)
```

Here status is an integer variable. While the simulation is active, the global variable "time" gives the current simulation time.

As described in Section 2.0, the simulation control routine will return if the event set becomes empty. In certain cases, one may want to terminate the simulation prematurely according to some arbitrary stopping criterion. SIMPAS provides this capability by predfining the pseudo-event "main". Event main is predeclared as if it looked like:

```
event main(status : integer);
```

As a matter of fact, there is no event routine associated with event main. When an event notice for event main reaches the front of the event set, the simulation control routine terminates the simulation exactly as if the event set had become empty. In this case, the status variable in the start simulation statement is set to the argument of event main. By setting this argument to a non-zero number, the user can return a flag to indicate why the simulation terminated.

Thus, statements after the start simulation statement will be executed when the event set becomes empty or when event main occurs. Normally, one places code to print simulation statistics at this point in the program.

4.5. Event Scheduling Statements

Event notices are created and inserted into the event set by scheduling statements. Typical scheduling statements are of the form:

```
schedule box_start(3) at 10.0;
schedule box_moved(4) delay 5.0;
schedule box_start(which_belt) now;
```

The difference between schedule at and delay is that the time expression in the first case is an absolute simulation time, while in the second case the time expression gives how long in the future the event should occur. The now phrase is used to schedule an event to occur immediately and is equivalent to scheduling the event to occur at the present time.

An event must be declared before it is scheduled. This means that any scheduling statement referring to a particular event must syntactically follow the declaration for that event. To allow this in general, an event declaration can be forwarded exactly like a PASCAL procedure. This is done by giving the event declaration with the event body replaced by the word forward. Later in the program one repeats the event declaration (without the formal arguments) and follows this declaration with the event body.

Each execution of a scheduling statement causes the generation of an event notice and the insertion of the event notice into the event set. The event notice contains all of the information necessary to execute an event routine. Thus to identify a particular event execution, it is sufficient to identify that event notice. The named clause in a schedule statement can be used to record a pointer to the event notice generated by a scheduling statement. The form of the named phrase is, for example:

```
schedule box_moved(3) named a_box_moved
       delay 20.0;
```

Here "a_box_moved" must be declared as type "ptr_event" (pointer to event notice).

If an event has been scheduled with a named clause so that you can identify a particular event notice, you can remove the event notice from the event set by using the cancel statement:

```
cancel <event-pointer>
```

Here <event-pointer> must be a variable or expression of type ptr_event. A cancel statement does not destroy the event notice. One uses the destroy statement to dispose of a previously canceled event notice:

```
destroy <event-pointer>
```

It is an error to try to destroy an event notice which is still scheduled.

To put an event notice back into the event set, one uses the reschedule statement. The reschedule statement has the same form as a schedule statement except that one specifies an ptr_event variable rather than the name of an event. The actual arguments of the event remain the same as those on the original schedule statement.

For example, if one wished to change the time of the event "a_box_moved", one could use the following code:

```
cancel a_box_moved;
reschedule a_box_moved at new_time;
```

Thus if to change the time of an event, first cancel the event, and then reschedule the event.

When an event routine is called, a pointer to the event notice is placed in the global variable "current". Thus if the user wishes to reschedule the current event at a later time he can say

```
reschedule current at <time-expression>;
```

If "current" is not rescheduled by the event routine, the event notice is automatically destroyed.

4.6. Queue Handling Statements

SIMPAS also provides SIMSCRIPT II.5 like "sets". Since PASCAL already includes "sets" of a different kind, we use the terminology "queue" to describe the SIMPAS structures. A queue consists of a particular type of entity. Only entities of that type can be placed in the queue.

4.6.1. Entity and Queue Declarations One declares an entity type in the global type declaration part of the program; the declaration looks like a special record declaration. The preprocessor inserts additional field names to contain links to other members of the queue and to record which queue (if any) this entity is a member of. Continuing our conveyor belt example, one could change our previous declaration of box to the following:

```
box = queue member
        arrival_time : real;
        destination  : destination_id;
        weight       : integer;
      end;
```

Unlike the record declaration, this declaration results in box being a pointer type, since this is the natural declaration for a temporary entity.

After the type declaration, one declares a particular instance of an entity as follows:

```
var
    this_box : box;
    that_box : box;
```

Then "this_box" and "that_box" represent two different box's. Attributes of each distinct box are referred to using the PASCAL dereference and dot operators:

```
this_box^.arrival_time
this_box^.weight
```

Entities by themselves are not very useful unless they can be stored and accessed easily. In SIMPAS, a collection of entities can be placed in a queue and retrieved in order for later processing. To declare a queue one first declares a queue type:

```
type
    <queue-type> = queue of <entity-type>;
```

where <entity-type> must have been previously declared. This declaration may only appear in the global type part of the program. In any var part of the program (or procedure) one can declare a particular queue with a declaration like:

```
var
    <queue> : <queue-type>;
```

For example, to declare a queue of boxes called box_queue one could proceed as follows:

```
{must be in global type part of program}
type
  box = queue member
          . . . {as before}
        end;

  {declare the box queue type}
  box_q = queue of box;

var
    {declare the box queue itself}
    box_queue : box_q;
```

4.6.2. Entity Creation and Disposal Since a variable of type "box" is actually a pointer variable, one can use the standard PASCAL procedure "new" to create new boxes. However, there is no guarantee that all the fields of an entity created in this way will be consistent, since PASCAL does not require the initialization of variables allocated by "new" (or of variables in general for that matter). To overcome this problem, SIMPAS provides the create and destroy statements:

```
create this_box;
destroy that_box;
```

Create will insure that all preprocessor defined attributes of this_box will be properly initialized. Simlarly, destroy will insure that that_box is not presently in any queue, since this could result in dangling pointer errors.

4.6.3. Queue Initialization Queues in SIMPAS are represented as doubly linked lists with head nodes. Before any entity may be inserted in a queue, it must be initialized so that that the head node can be allocated and the queue attributes properly set. Attempting to place an entity in an uninitialized queue will result in unpredictable behavior. To simplify queue initialization, SIMPAS provides the initialize statement:

```
initialize box_queue;
```

Eventually, we plan to have the preprocessor generate code to automatically initialize all queues for the user. However, in certain cases it is impossible to determine at preprocessing time whether or not a particular variable refers to a queue or not (queues declared in variant parts of records, for example) so that the initialize statement will still be needed.

4.6.4. Queue and Entity Standard Attributes The preprocessor inserts additional attributes into each queue member declaration to allow the entity to be inserted in queues, to make it easy to determine if an entity is in a queue and so forth. The most useful of these are:

next-	This attribute points to the next member of the queue or to the queue head if this is the last member of the queue.
prev-	This attribute points to the previous member of the queue or to the queue head if this is the first member of the queue.

Similarly the preprocessor defines several standard queue attributes, some of which are:

empty-	This boolean attribute is true if the queue is empty.
size-	This integer attribute gives the number of members in the queue.
stat-	This attribute is of type "statistic" and is used to collect statistics about queue occupancy. See Section 4.8 for details about type "statistic".

4.6.5. Queue Manipulation Statements To insert or remove entities from a queue, SIMPAS provides insert and remove statements. To insert an entity last in a queue one can say either:

```
insert this_box last in box_queue;
```

or

```
insert this_box in box_queue;
```

Similary, one can place the entity at the front of the queue by

```
insert that_box first in box_queue;
```

To remove a particular entity from a queue one uses the statement:

```
remove this_box from box_queue;
```

Corresponding to insert first and insert last statements are the statements:

```
remove the first new_box from box_queue;
remove the last  new_box from box_queue;
```

These statements differ from the first example of the remove statement in that the variable "new_box" is set to point at the specified entity while in the first case, "this_box" already points at a particular entity and the execution of the statement merely removes it from the queue.

In all cases, the inserted (removed) entity must be of the same type as the queue into which it is to be inserted (removed from). Attempts to insert or remove entities in queues of the wrong type are detected either at preprocessing or compile time. Other errors, such as attempting to insert an entity into a queue when it is already in a queue, attempting to remove an entity from a queue it is not in, and so forth are detected at run time.

4.7. Pseudo-random Number Generation

A standard collection of pseudo-random number generators are provided in the SIMPAS library and can be incorporated in the user program through the include statement. These routines all depend on a single uniform random number generator which is a portable version of LLRANDOM (Fishman 1978) suitable for use on all machines with a word size of 32 bits or larger. A 16 bit version of this generator is also available, but is much less efficient. Given the existence of the basic uniform random number generator, random number generators for the following distributions are provided:

exponential	poisson
binomial	discrete uniform
general discrete	normal
lognormal	gamma
erlang	continuous uniform
beta	hyperexponential

The generation algorithms were taken from (Fishman 78).

SIMPAS provides 10 random number generation streams (numbered 1 to 10). Each random number generator takes as input one of these stream id's. Distinct streams represent different portions of the LLRANDOM base random number generation sequence. Initially, each stream is separated from its neighbors by at least

100,000 calls.

Distinct streams can be used to reduce the possibility of any dependence between successively generated random variables, or to keep a sequence of random variables in the simulation fixed while varying another.

4.8. Statistics Collection

At present, SIMPAS does not provide the automatic statistics collection features of SIMSCRIPT II.5. However, SIMPAS does provide a statistic collection type and an observation statement that simplifes the collection of simulation statistics.

To allocate a variable for statistic collection, one declares a variable of type "statistic." For example:

```
var
    nsys      : integer;
    nsys_stat : statistic;

    tsys      : real;
    tsys_stat : statistic;
```

A statistic can be either time or event-averaged. For a time averaged statistic, values observed are weighted by the length of time the value was held; event averaged statistics give equal weight to all values.

The distinction as to type of statistic is made when when the statistics variable is initialized with the "clear" routine:

```
{ time averaged }
clear(nsys_stat, accumulate);

{ event averaged }
clear(tsys_stat, tally);
```

The routine "clear" can also be used to reset statistic collection during a run.

To observe a value of a variable, one uses the observation statement:

```
observe nsys in nsys_stat;

observe tsys in tsys_stat;
```

The max, min, and mean value over all observations of a variable are available through the standard statistic attributes. At any time, the values of these attributes reflect the values of the observed variable up to the last time it was "observed". Thus:

nsys_stat.mean	is the time averaged mean of nsys
tsys_stat.variance	is the event averaged variance of tsys
nsys_stat.max	is the maximum of nsys
tsys_stat.min	is the minimum of tsys

Other attributes are easily added. The observation routine uses the algorithm of (West 79) to stably update the mean and variance.

A subtle point here deals with the time-averaged observations. A convention must be adopted as to when to do an observation; the convention can be to do it immediately before changing the value of the observed variable or immediately after changing the value. We have adopted the convention that one must observe the value before changing the variable.

5. AN EXAMPLE SIMULATION

In this section we combine the examples from the previous sections to illustrate their use in a simple simulation. The system we are going to simulate can be described as follows:

> Trucks arrive a loading dock every 10 to 20 minutes (uniformly distributed) and deliver from 1 to 20 boxes (again, let us say, uniformly distributed). When a truck arrives, a worker unloads the boxes and places them on one of 5 conveyor belts to be delivered to various parts of the plant. It takes 1 minute to unload each box and place it on the conveyor. Ten percent of all boxes go onto conveyor belt 1, 20% go on belt 2, 30% go onto 3 and 4, and 10% go onto 5. It takes 5 minutes for a box to traverse each of the conveyor belts. On the average, how many boxes are on each conveyor belt, and how many are waiting at the loading dock to be placed on a conveyor? Finally, what is the average transit time from the loading dock to the box's final destination?

5.1. Entity Declarations

To model this system, we need a queue of boxes to represent the collection of boxes at the loading dock. For simplicity, we are also going to use a queue of boxes to hold the boxes present on each conveyor belt in the factory. Boxes will be declared as queue members with attributes defining the box's destination (for convenience we will number the destinations the same way we number the conveyor belts) and the box's arrival time (to allow us to compute its time from arrival at the loading dock until it is delivered at its final destination). The following SIMPAS declarations allow us to do this:

```
const
  {the total number of conveyor belts};
  number_belts = 5
```

```
type

  {a conveyor belt id is a number
   between 1 and number_belts}
  cv_belt_id = 1..number_belts;

  box = queue member
          destination : cv_belt_id;
          arrival_time: real;
        end;

  { box_q is the type which represents a
    queue of boxes }
  box_q = queue of box;

  { cv_belt describes one conveyor belt }
  cv_belt = record
        {is belt in use? true or false}
        busy            : boolean;

        {how long does it take to move a
         box down the belt?}
        move_time       : real;

        {queue of boxes}
        boxes           : box_q;

        {counts the number of boxes
         delivered}
        delivered_boxes : integer;
  end;

var

  { the set of conveyor belts is an array
    of records of type cv_belt indexed by
    belt id }
  conveyor_belt : array [cv_belt_id]
                    of cv_belt;

  { loading_queue contains the set of
    boxes delivered but not yet
    placed on a conveyor belt }
  loading_queue : box_q;
```

Note that conveyor_belt[i].boxes is the queue of boxes on conveyor belt "i".

To represent the worker we will use the following record declaration:

```
worker : record
            idle : boolean;
            boxes_moved : integer;
         end;
```

Here "idle" will be used to represent the worker's status and "boxes_moved" will be used to count the number of boxes the worker has moved.

5.2. Event Declarations

We also need three events in the simulation; one event to model arrivals of trucks at the loading dock, one to model movement of boxes to the conveyor belt, and one to model the arrival of a box at its final destination. In this simulation, the names we have chosen for these three events are "truck_arrives", "box_moves" and "box_delivered" respectively.

Let's first consider what event "truck_arrives" must do. Every time a truck arrives, we must generate a number of boxes for that particular truck to deliver. To do this we use the SIMPAS library function "udisc". This function is called as

```
   udisc(a, b, k);
```

and returns an integer uniformly chosen between a and b (inclusive) according to random number stream k. This number of boxes are then generated and placed in the loading_queue. For each box, we must chose a destination according to the percentages given above. To do this we use the SIMPAS library function "i_gdisc" which returns an integer valued random variable with a general distribution. An associated routine, "i_gdsetup" is used to establish the values and associated probabilities for the random variable. Next, if the worker is idle, we then start the movement of boxes to the conveyor belt. The event "box_moves" will only mark the worker as idle when all boxes have been loaded onto the appropriate conveyor belt. Hence if the worker is presently busy, we need not awaken him when new boxes arrive. Finally, we must arrange for the next truck_arrives event to occur. The SIMPAS code for this event is:

```
event truck_arrives;

var
  new_box       : box;
  number_boxes  : integer;
  i             : integer;
begin

  number_boxes :=
    udisc(min_boxes,max_boxes,box_stream);

  for i := 1 to number_boxes do
  begin

    create new_box;

    with new_box^ do
    begin
      arrival_time := time;
      { set the destination of the box }
      destination  :=
          i_gdisc(dest_rv, dest_stream);
    end;

    insert new_box in loading_queue;

  end;

  { if the worker is idle, then start
    moving boxes }
  if worker.idle then worker_moves_box;

  { finally, schedule the next truck
    arrival }
  reschedule current delay
     unif(min_ia_time,max_ia_time,
        arrival_stream);

end; {event truck_arrives}
```

Recall that the variable "current" points to the event notice of the currently executing event routine. Thus the reschedule current statement above causes a "truck_arrives" event to occur after a delay of between "min_ia_time" and "max_ia_time" minutes.

The procedure "worker_moves_box" marks the worker as "not idle", removes the next box from the loading_queue, and schedules a box_moves event for one minute later. Event "box_moves" increments the number of boxes the worker has moved, places the box in the queue of boxes representing the conveyor belt and schedules a "box_delivered" event to remove the box from the conveyor belt. If a sufficient number of boxes have been moved, "box_moves" will terminate the simulation by scheduling an occurrence of event "main". Procedure "worker_moves_box" and event "box_moves" are declared as:

```
procedure worker_moves_box;
var
    carried_box : box;
begin

    { mark the worker as being busy}
    worker.idle := false;

    {get the first box from the loading
     queue}
    remove the first carried_box
      from loading_queue;

    {schedule the delivery of the box to
     the conveyor}
    schedule box_moves(carried_box)
      delay box_move_time;

end; {procedure worker_moves_box}

event box_moves(b : box);
var
    belt : cv_belt_id;
begin

  with worker do
  begin
    {increment the number of boxes
     the worker has moved}
    boxes_moved := boxes_moved + 1;
    {stop the simulation if more than
     max_boxes_moved }
    if boxes_moved > max_boxes_moved
      then
        schedule main(2) now;
  end;

  {go back and move another box unless
   no more boxes to move}
  if loading_queue.empty then
          worker.idle := true
      else
          worker_moves_box;

  {place the box in the appropriate
   conveyor belt queue}
  belt := b^.destination;
  insert b
    in conveyor_belt[belt].boxes;

  {schedule the box delivery event}
  schedule box_delivered(belt)
    delay
      conveyor_belt[belt].move_time;

end; {event box_moves}
```

Finally, the event "box_delivered" handles delivery of a box to its final destination. The box is removed from the conveyor belt queue, the number of delivered boxes on that conveyor belt is incremented and the transit time for the box is calculated. The transit time is then observed in a statistics variable declared for this purpose and the box entity is destroyed:

```
event box_delivered(belt : cv_belt_id);

{belt gives the belt upon which the
 box will be delivered}

var
    moved_box : box;
    transit_time : real;
begin

  remove the first moved_box
    from conveyor_belt[belt].boxes;

  with conveyor_belt[belt] do
       delivered_boxes:=delivered_boxes+1;

  transit_time:=
     time-moved_box^.arrival_time;

  observe transit_time in t_time_stat;

  destroy moved_box;

end; {event box_delivered}
```

5.3. Initialization, Execution, and Statistics Reporting

All that is left is to initialize everything, properly start the simulation, and print the statistics. Three things must be initialized: the queues, statistics variables, and the general discrete random variable used to choose a box's destination. The queues are easy to initialize:

```
for belt := 1 to number_belts do
  with conveyor_belt[belt] do
  begin
      {initialize the conveyor belt
        queue}
      initialize boxes;

      {initialize other conveyor belt
       attributes}
      move_time := 5;
      delivered_boxes := 0;
```

```
    end;

{initialize the loading dock queue}
initialize loading_queue;
```

The only explicit statistics variable we need is for the box transit time. The other statistics of interest (mean number of boxes at the loading dock and on each conveyor belt) are automatically maintained in the "stat" attribute of each queue. To declare and initialize the box transit time statistics variable we use the following code:

```
var
    . . .
    t_time_stat       : statistic;
    . . .

begin {main procedure}
    . . .

    clear(t_time_stat, tally);
    . . .
```

Finally, to initialize the general discrete random variable used to assign box destinations, we need the procedure i_gdsetup. This procedure takes as its arguments a pointer to a list of probability and value pairs, a flag indicating whether or not this is the first time that i_gdsetup has been called for this list, and the probability and value associated with this call. Each new probability and value pair is appended to the end of the list of pairs. The list of pairs is passed to i_gdisc in order to generate a random integer. The declarations and code to do this are:

```
var
   . . .
   {this type brought in from library
    file by "include gdisc;" }
   dest_rv  : gdiscvar;
   . . .
begin {main procedure}
   . . .
   {initialize dest_rv}
   i_gdsetup(dest_rv, true, 0.10, 1);
   i_gdsetup(dest_rv, false,0.20, 2);
   i_gdsetup(dest_rv, false,0.30, 3);
   i_gdsetup(dest_rv, false,0.30, 4);
   i_gdsetup(dest_rv, false,0.10, 5);
   . . .
```

A call of the form

```
dest := i_gdisc(dest_rv, dest_stream);
```

will then assign to dest an integer chosen according to the specified distribution.

After initializing the "worker" record so that the worker starts out idle and having moved zero boxes, the first events are scheduled and the simulation control routine is called:

```
schedule truck_arrives now;

schedule main(1) at sim_run_time;

start simulation(status);
```

The second schedule statement is used to guarantee termination of the simulation at a specified maximum run time.

Statements after the start simulation statement can be used to print simulation statistics, since they will be executed only after the end of the simulation. For example, "time" will be the time that the simulation stopped. The status variable can be printed to determine which of the schedule main statements caused the simulation to terminate (status=1 or 2) or if the simulation terminated because the event set became empty (status=0). Similarly, the mean loading_queue size is available as

```
loading_queue.stat.mean
```

and the maximum number of boxes on conveyor number 3 is given by

```
conveyor_belt[3].boxes.stat.max;
```

5.4. The Conveyor Belt Simulation

To give a concise summary of the conveyor belt simulation, here is a skeleton of the entire simulation, with the event declarations we have already discussed removed. The primary additions here are forward declarations necessary since events and procedures must be declared prior to their being scheduled or called. Spacing of the program has been abbreviated in order to fit in the two column format of this paper.

```
program conveyor(output);

const
  {the total number of belts in the shop}
  number_belts = 5;

  {parameters to control the number}
  {of boxes each truck delivers    }
  min_boxes    = 1;
  max_boxes    = 20;

  {minimum truck inter-arrival time}
  min_ia_time = 10.0;

  {maximum truck inter-arrival time}
  max_ia_time = 20.0;

  {how long it takes the worker to
   move a box from the truck to the
   conveyor belt it belongs to }
  box_move_time = 1.0;

  {constants to control simulation run
   length}
  sim_run_time    = 200.0;
  max_boxes_moved= 500;

  {constants that define which streams
   are used to generate the truck
   inter-arrival times, the number of
   boxes delivered per truck and the
   box destination random variables  }
  arrival_stream = 1;
  box_stream     = 2;
  dest_stream    = 3;

type

  {a conveyor belt id is a number
   between 1 and number_belts}
  cv_belt_id = 1..number_belts;

  box = queue member
    destination  : cv_belt_id;
    arrival_time : real;
  end;

  box_q = queue of box;

  cv_belt = record
    busy : boolean;
    move_time : real;
    boxes : box_q;
    delivered_boxes : integer;
  end;

var
  conveyor_belt : array [cv_belt_id]
                    of cv_belt;

  loading_queue : box_q;

  worker : record
          idle         : boolean;
          boxes_moved  : integer;
  end;

  dest_rv : gdiscvar;

  t_time_stat   : statistic;

  belt : cv_belt_id;

include udisc, unif, gdisc, statistics;

procedure worker_moves_box;
  forward;

event truck_arrives;
var
  new_box       : box;
  number_boxes  : integer;
  i             : integer;
begin
      . . .
end; {event truck_arrives}

event box_moves(which_box : box);
  forward;

procedure worker_moves_box;
var
  carried_box : box;
begin
  . . .
end; {procedure worker_moves_box}

event box_delivered (belt : cv_belt_id);
   forward;

event box_moves;
var
  which_belt : cv_belt_id;
begin
  . . .
end; {event box_moves}

event box_delivered;
var
  moved_box : box;
  transit_time : real;
begin
  . . .
end; {event box_delivered}

begin {-----> main procedure <-----}
  {-----> initialize <-----}
  for belt := 1 to number_belts do
      with conveyor_belt[belt] do
      begin
          initialize boxes;
          move_time := 5;
          delivered_boxes := 0;
      end;

  initialize loading_queue;

  clear(t_time_stat, tally);

  i_gdsetup(dest_rv, true, 0.10, 1);
  i_gdsetup(dest_rv, false,0.20, 2);
  i_gdsetup(dest_rv, false,0.30, 3);
  i_gdsetup(dest_rv, false,0.30, 4);
  i_gdsetup(dest_rv, false,0.10, 5);

  with worker do
  begin
      idle := true;
      boxes_moved := 0;
  end;

  {------> schedule initial events <-----}
  schedule truck_arrives now;
  schedule main(1) at sim_run_time;

  {-----> run the simulation <------}
  start simulation(status);

  {-----> print statistics <-----}
  writeln('simulation terminated at ',
           time:10);
  writeln;
  writeln(' status=',status:2);
  writeln;

  writeln('mean boxes at loading dock: ',
          loading_queue.stat.mean:7);
  writeln('max  boxes at loading dock: ',
          loading_queue.stat.max:7);
  writeln;

  for belt := 1 to number_belts do
   with conveyor_belt[belt].boxes
   do
        writeln('belt: ',belt:1,
          ' contains ', stat.mean:10,
          ' boxes (average)');

  writeln;
  writeln('average box transit time: ',
          t_time_stat.mean:10);
  writeln;

  writeln('worker moved ',
           worker.boxes_moved:3,' boxes');
  writeln;

  for belt := 1 to
    number_belts do
    with conveyor_belt[belt] do
    begin
        writeln('belt: ',belt:1,
          ' delivered        ',
          delivered_boxes:3, ' boxes');
        writeln('       ',
                  ' currently contains ',
              boxes.size:3,' boxes');
        writeln;
    end;
end.
```

The output produced by this simulation is:

```
simulation terminated at 1.99E+02

status= 1

mean boxes at loading dock: 2.59E+00
max  boxes at loading dock: 1.50E+01

belt: 1 contains  7.97E-02 boxes (average)
belt: 2 contains  5.15E-01 boxes (average)
belt: 3 contains  1.01E+00 boxes (average)
belt: 4 contains  6.84E-01 boxes (average)
belt: 5 contains  3.28E-01 boxes (average)

average box transit time:   1.08E+01

worker moved 106 boxes

belt: 1 delivered           3 boxes
        currently contains  0 boxes

belt: 2 delivered          20 boxes
        currently contains  2 boxes

belt: 3 delivered          40 boxes
        currently contains  1 boxes

belt: 4 delivered          27 boxes
        currently contains  0 boxes

belt: 5 delivered          13 boxes
        currently contains  0 boxes
```

6. CONCLUDING REMARKS

SIMPAS has been in use in the Computer Sciences Department at the University of Wisconsin since Spring 1980. Versions of SIMPAS tailored for execution under VAX VMS, VAX UNIX, and Univac 1100 OS, as well as a portable version designed to run under standard PASCAL are available from the author for a standard distribution fee. It is presently installed at several different sites across the country. For further information, feel free to contact the author at the mailing address specified on the title page, or at 608-262-5386 or 262-1204.

ACKNOWLEDGEMENTS

Mark Abbott, John Bugarin, and Bryan Rosenburg have worked on various phases of the SIMPAS implementation and without their assistance the project would never have been completed. This project was supported in part by the Wisconsin Alumni Research Foundation and by NSF Grant MCS-800-3341. I also would like to acknowledge the support of the Madison Academic Computing Center, and in particular the assistance provided by its director, Dr. Tad B. Pinkerton.

REFERENCES

Bryant, R. M. (1980), SIMPAS -- A Simulation Language Based on PASCAL, Proceedings of the 1980 Winter Simulation Conference, T. I. Oren, C. M. Shub, and P. F. Roth (eds)., Orlando, Florida, December 3-5, 1980, pp. 25-40.

Bryant, R. M. (1981), <u>Micro-SIMPAS: A Microprocessor Based Simulation Language</u>, Proceedings of the Fourteenth Annual Simulation Symposium, R. M. Huhn, E. R. Comer, F. O. Simons, Jr. (eds)., Tampa, Florida, March 17-20, 1981, pp. 35-54.

Bryant, R. M. (1981b), <u>SIMPAS Version 5.0 User Manual</u>, Computer Sciences Department Technical Report, University of Wisconsin-Madison, in preparation.

Fishman, G. (1978), <u>Principles of Discrete Event Simulation</u>, John Wiley and Sons, New York.

Franta, W. R. (1977), <u>The Process View of Simulation</u>, Elsevier North-Holland, Inc., New York.

Jensen, K. and N. Wirth (1974), <u>PASCAL User Manual and Report</u>. Springer-Verlag, New York.

Kiviat, P. J., R. Villanueva, H. M. Markowitz (1974) <u>SIMSCRIPT II.5 Programming Language</u>. C. A. C. I., Inc, 12011 San Vicente Boulevard, Los Angeles, California.

Uyeno, D. H. and W. Vaessen (1980), "PASSIM: A Discrete-event Simulation Package for PASCAL," <u>Simulation</u>, <u>35</u>, 6, pp. 183-190.

West, D. H. D. (1979), "Updating the Mean and Variance Estimates: An Improved Method," Communications of the ACM. 22, 9, pp. 532-535.