SIMPAS User Manual

TR391

R. M. Bryant

Computer Sciences Department
and
Madison Academic Computing Center
University of Wisconsin-Madison
Madison, Wisconsin

SUMMARY

SIMPAS is a portable, strongly-typed, event-oriented, discrete system simulation language based on PASCAL. Facilities for event declaration and scheduling, creation and deletion of temporary entities, declaration and maintenance of linked lists (queues) of entities, and observation of simulation statistics are all provided as natural extensions to PASCAL. In addition, SIMPAS provides a substantial library of support routines that includes random number generators for all of the most common distributions.

This manual describes the SIMPAS extensions to PASCAL and discusses their implementation. Examples of the use of these extensions are given. Directions for transporting SIMPAS to a new computer system are also provided.

Table of Contents

1. Introduction

SIMPAS is an event-oriented, discrete system simulation language based on PASCAL. It is implemented as a preprocessor that accepts an extended version of PASCAL as input and produces a standard PASCAL program as output. The preprocessor itself is written in standard PASCAL, and the language has been designed so that it depends only on the features of standard PASCAL. Thus SIMPAS is extremely portable since it can run on any system which supports standard PASCAL.

Aside from portability, the choice of PASCAL as the target language makes SIMPAS a strongly typed simulation language. SIMPAS is similar in this respect to SIMULA [DAHL69] [FRAN77], although the latter is a process oriented simulation language. Strong typing allows many of the more common programming errors in simulation languages such as SIMSCRIPT II.5 [KIVI74], ASPOL [MACD73], or SIMPL/I [IBM72] to be detected at compilation time when the simulation is written in SIMPAS. (See [BRYA80] for a further comparison of SIMPAS and SIMSCRIPT II.5).

This manual describes the SIMPAS extensions to PASCAL and discusses how to use these language extensions to write powerful and reliable simulation programs. We assume that the reader is familiar with PASCAL; if not, we recommend reading either [JENS74], [WILS79] or some other introductory textbook about PASCAL before reading the rest of this manual. We also assume that the reader is familiar with the concepts fundamental to event-oriented simulation such as "event routine", the "event set", and "event notices". (See, for example, [FISH78] or Section A.1 of this manual).

In discussing use of SIMPAS, we will use lower case letters and the character "_" in identifiers. Since some PASCAL compilers do not support lower case or "_" as a legal identifier character, SIMPAS can easily be reconfigured to use upper case only, and the underbar character can be translated to some other character. (For example, in the UNIX implementation "_" is translated into "Ø" during preprocessing).

This manual is divided into eight major sections and four appendices. The rest of this section gives some general information about SIMPAS. Section 2 discusses the SIMPAS symbolic library and its use. Section 3 discusses event declaration and scheduling and the type declarations and routines provided to deal with the simulation event set. Section 4 discusses the random-number generation routines provided with SIMPAS and describes their use. Section 5 discusses queue memebrs and queue declarations. Section 6 discusses the statistics collection features of SIMPAS.

Section 7 contains an example SIMPAS program. Appendix A describes the SIMPAS implementation and describes the expanded PASCAL code generated by each SIMPAS declaration or statement. Appendix B contains a list of reserved words and restrictions imposed by the SIMPAS implementation. Appendix C discusses the changes that need to be made in moving SIMPAS to a new computer system. Appendix D contains a quick reference guide to SIMPAS.

1.1. Overview of the SIMPAS Preprocessor

SIMPAS consists of a large PASCAL program (about 4,000 lines) and a small file of run-time routines written in PASCAL (the "library" file).

SIMPAS is organized as a two-pass processor. On the first pass, the input program is examined for occurrences of SIMPAS statements; when one is found it is expanded into PASCAL statements. During this pass the output PASCAL is placed in a temporary file. The preprocessor also stores information from the SIMPAS statement for later use. For example, when expanding an event declaration, the preprocessor saves the event name and the names and types of the formal arguments for use in building the event-set declarations.

During pass two, the intermediate code from the temporary file is read and the final output PASCAL is produced. The declarations for the event set are constructed and placed in the global type and variable declaration parts of the program. Support routines are read from the library file and placed at the top of the procedure declaration part of the program. The simulation control routine is created and inserted at the appropriate point, and initialization code for the event set and other global variables is inserted at the start of the main procedure. Other than these insertions, the second pass of the preprocessor merely copies the temporary file to the output.

1.2. Executing a SIMPAS Program

To compile and execute a SIMPAS program requires three steps: (1) Expansion: The SIMPAS preprocessor is invoked and reads your SIMPAS program, producing a PASCAL program as output. (2) Compilation: The PASCAL compiler is called to compile the generated PASCAL program. (3) Execution: The PASCAL program is executed, perhaps after a link edit step to resolve external references.

Errors can occur during any one of these steps. Error messages during the expansion phase refer directly to a SIMPAS statement. Error messages issued during compilation can be traced back to a SIMPAS source line using the line

numbers inserted in the output PASCAL by the preprocessor. (These line numbers appear as comments at the beginning of each PASCAL source line and give the SIMPAS source line number which caused the generation of that line of PASCAL.) Errors during execution are either those caught by SIMPAS run-time routines or by PASCAL run-time routines. The first class of errors indicate directly in which SIMPAS statement the error occurred. The second class of errors can be traced back to the SIMPAS source code by first determining in which PASCAL output line the error occurred, and then using the line number encoded there to find the SIMPAS statement where the error ocurred.

1.3. Notation

Throughout our discussion, we will underline keywords and enclose variable names in quotes. We will use angle-brackets ("<" and ">") to represent portions of SIMPAS statements that 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 an optional portion of a statement. We will use braces ("{" and "}") to enclose a list of alternatives separated by vertical bars ("|"). One of the alternatives in the list must be chosen in order to create a syntactically valid statement. Since statements in PASCAL can extend across card boundaries, we will split SIMPAS statements across lines in order to make them more readable. The statements need not be split across lines as we have indicated.

2. The Library File and the Include Statement

Besides the default routines, which are always included, support routines are loaded from the library file on the user's request. For example, the random-number generation routines that the user needs are loaded. The user specifies which support routines to load using the include statement:

include <name-1> [,<name-2>] . . . ;

The include statement must follow the global var part of the program and precede the first procedure, function , or event declaration of the program. Typically, each <name-i> in the include statement causes a single procedure to be included. The include mechanism can also bring in global constants, variables, or types required by the procedures. (Appendix A contains a description of the library file implementation.)

The library file and the include statement implement a symbolic library of support routines for SIMPAS programs. A symbolic library is necessary because external compilation

is not part of standard PASCAL. If the host PASCAL compiler supports some type of external compilation, much of the library file can be separately compiled. Doing this will reduce the execution time of the preprocessor and will also reduce compile times of the output PASCAL since the library file routines will not have to be repeatedly recompiled.

3. Event Declaration and Scheduling

Events may be declared and scheduled by a set of natural extensions to PASCAL. Facilities are provided to declare a particular event, to create an event notice and schedule it, to reschedule a previously created event notice, and to cancel and/or destroy a particular event notice.

3.1. Events

An event is declared exactly like a PASCAL procedure, except that the keyword procedure is replaced by the word event. An event must be accessible in the main program; an event cannot be declared within a procedure. An event may not have any var parameters; all parameters must be passed by value. This is because the event is called with values of the actual parameters saved in an event notice, and hence all parameters are effectively passed by value.

As an example, the declaration

```
event arrival(machine_id : integer);
begin
    . . .
end;
```

could be used to declare an event called "arrival" which has a single integer-valued argument.

An event whose name is <event> is translated into a procedure whose name is r_<event>. Thus if the host PASCAL compiler only distinguishes identifier names that differ in the first n characters, then event names must be distinct in the first n-2 characters.

3.2. Event Scheduling

An event is scheduled to occur at a particular simulated time by a statement of the form:

```
schedule <event>[(<actual argument list>)]
            at <time-expression>
```

One can specify that an event is to occur after an interval of simulated time by using the keyword delay instead of at.

Thus the following statements are equivalent:

```
schedule arrival(3) at time + 10.0;
schedule arrival(3) delay 10.0;
```

An event must be declared before it can be scheduled, just as PASCAL procedures must be declared before they can be called. An event can be forwarded exactly like a PASCAL procedure; the body of the event is replaced by the word forward. The formal arguments of the event must be specified when the event is forwarded; the body of the event is given after the event heading is repeated without the formal arguments.

The keyword now indicates that the event is to occur next, before any other events scheduled for the current simulated time. The two statements

```
schedule arrival(3) delay 0;
schedule arrival(4) now;
```

are not quite equivalent since the arrival(3) event will occur after any other event also scheduled for the current simulated time; the event arrival(4) will occur before any other event scheduled for the current simulated time. If several events are scheduled by now phrases at the same simulated time, then the last event to be scheduled is executed first.

A particular event notice can be identified by using the named clause:

```
schedule <event>[( . . . )] named <evptr> . . .
```

<evptr> must be a simple or qualified variable or expression[*] of type "ptr_event". (The type "ptr_event" is defined by the SIMPAS preprocessor.) One can use this name to cancel, reschedule, delete or destroy the event notice created by this schedule statement.

Given a name for an event, another event can be scheduled to occur at the same simulated time as the named event by using a before or after clause:

```
schedule arrival(3) after <evptr>;
schedule arrival(4) before <evptr>;
```

[*]The version 2 implementation limits the complexity of this expression. See Appendix B for details.

In each case, the arrival event will occur at the same simulated time as the event described by the event notice pointed to by <evptr>, but in the first case the <evptr> event occurs first while in the second case the arrival event occurs first. Once again <evptr> must be a simple or qualified variable of type "ptr_event" or expression of type "ptr_event".

It is an error to try to schedule an event before or after an event that is not scheduled.

The event notice of the currently executing event is named "current". However, before the event is executed, "current" is removed from the event set, and therefore "current" is not considered to be scheduled while the event is executing. This removal allows the automatic reclamation of the event notice if the notice is not rescheduled during the event routine. Thus one can not normally say

```
schedule arrival(4) after current;
```

However, see Section 3.6.

If an event notice is created using a schedule statement with a named clause, it is assumed that the user will explicitly dispose of the event notice. Otherwise the pointer to the event notice may be invalid when it is used. Therefore, if a notice is created by a schedule statement with a named clause, the automatic reclamation of the event notice is inhibited, and the user must use the destroy statement (see Section 3.5) to dispose of the event notice when it is no longer needed.

3.3. Start Simulation and Event Main

The start simulation statement is used to begin executing scheduled events. Its form is:

```
start simulation(<status>)
```

While events are being executed, the global variable "time" gives the current simulation time.

The statement after start simulation is executed only if the event set becomes empty or an event notice for the event main reaches the front of the event set. <status> is an integer variable whose value can be inspected to determine why the simulation stopped.

If the event set is empty when start simulation is executed, the control routine will return immediately. Thus the proper way to start a simulation is to schedule at least one event before executing the start simulation statement.

This event will then occur immediately and it (presumably) will schedule other events in order to maintain the simulation process.

The event main is predeclared as if it looked like:

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

When event main occurs, execution resumes after the most recently executed start simulation statement. The value of the <status> variable in the start simulation statement is set to the value of the argument to main specified in the schedule main statement. This status variable can be used to flag why the simulation stopped. For example, one can terminate a simulation at time 10.0 and return a status of 3 to the main program by saying

```
schedule main(3) at 10.0;
```

If the event set becomes empty, a schedule main(0) now statement is automatically executed. That is, program execution will resume after the most recently executed start simulation statement, and the status variable will be set to zero.

3.4. Event Notice Utility Functions

Some utility routines have been predefined to simplify inspecting the contents of event notices. In most cases, these routines may be included using the include statement; certain of the routines are always included. These routines return information about the event notice given a pointer to the notice. The same information is available by direct reference to a field of the event notice (if the pointer is not nil). The advantage of the predefined routine is that it checks to make sure the pointer is not nil.

The first utility function is "scheduled". Scheduled is a boolean function that returns true if the event notice pointed to by its argument is scheduled; it returns false if the notice is not scheduled or if the pointer is nil. It is declared as:

```
function scheduled(name : ptr_event) : boolean;
```

and is defined in every SIMPAS program.

The function "etime" returns the time of the event described by the event notice, or -1.0 if its argument is nil. "Etime" is declared as:

```
function etime(name : ptr_event) : real;
```

"Etime" must be explicitly included by the include statement.

The function "etype" returns the type of the event described by the event notice, or the value "no_event" if its argument is nil. "Etype" is declared as:

```
function etype(name : ptr_event) : t_ev_1;
```

Here "t_ev_1" is an enumeration type defined by the preprocessor. It contains the names of the events defined in the SIMPAS program and the identifiers "no_event" and "main". For example, if you have an event "departure" you may check to see if a particular event notice describes a departure event by saying:

```
if etype(evptr) = departure then . . .
```

Also, the following two statements are equivalent:

```
if evptr = nil then . . .
if etype(evptr) = no_event then . . .
```

The function "etype" must be explicitly included via the include statement.

3.5. Cancel, Destroy, Delete and Reschedule

If an event has been scheduled with a named clause, the event notice may be removed from the event set by using the cancel statement:

```
cancel <evptr>
```

Here <evptr> must be a simple or qualified variable or expression of type "ptr_event".

Cancel does not destroy the event notice. The destroy statement disposes of a previously canceled event notice:

```
destroy <evptr>
```

It is an error to try to destroy an event notice that is still scheduled. To destroy a scheduled event notice use delete instead of destroy. Delete first cancels then destroys the event notice.

Reschedule can be used to put an event notice back into the event set. Reschedule has the same form as schedule except that one specifies a pointer to an event notice rather than the name of an event. The event pointer must have been set by a previously executed schedule statement with a named clause. The actual arguments of the event

remain the same as those on the schedule statement. If necessary, the actual arguments can be accessed and modified, but this action requires knowledge of the event notice structure. (See Appendix A for details.)

The reschedule statement has the form:

```
reschedule <evptr> { at <time-expression> |
                     delay <time-expression> |
                     now |
                     after <evptr-1> |
                     before <evptr-1> }
```

Here <evptr> and <evptr-1> must be simple or qualified variables or expressions of type ptr_event.

It is an error to try to reschedule an event that is currently scheduled. To change the time of an event, first cancel and then reschedule the event.

Without examining the event set directly, it is impossible to cancel, delete, destroy or reschedule an event unless it has been given a name through the named clause on a schedule statement. However, one can use a forall statement to scan the event set and obtain pointers to arbitrary event notices. In this way arbitrary event notices can be canceled, deleted, destroyed or rescheduled. (See Section 5.5.)

Care must be taken not to change the status of an event notice that can be referenced by another event pointer. For example:

```
var event1, event2 : ptr_event;
    . . .
schedule arrival(3) named event1 delay 10.0;
event2 := event1;
    . . .
delete event1;
    . . .
reschedule event2 delay 20.0;
```

In this case, when reschedule is executed it is likely that event2 does not point to the event notice for the arrival(3) event that was originally scheduled. In fact, depending on the PASCAL implementation, event2 may still be a valid pointer, but it may point to a different arrival event than the arrival(3) originally scheduled. Needless to say, this can cause unexpected results.

This aliasing problem is especially severe when the PASCAL procedure "dispose" is not implemented and SIMPAS must maintain available lists of event notices, since in

this case the pointers are always valid. (A pointer to a disposed event notice points to an event notice on an available list for that type of event. When this event notice is reused, then the old pointer can point to an event notice for a new event!) To avoid this problem in general, try to not to have more than one copy of a pointer to an event notice. When that event notice is destroyed, the destroy procedure can set the pointer to nil to indicate this fact and this "dangling" pointer problem cannot occur.

3.6. Reschedule and Current

Before the current event is called, a pointer to its event notice is placed in the global variable "current". The notice named "current" is removed from the event set before the event routine is called; thus "current" is not scheduled when the event is started. If when the event terminates, "current" is still not scheduled, the event notice will be destroyed.

If you wish the present event to be rescheduled at a later time (using the same event notice), you can say

```
reschedule current . . .
```

where . . . represents any of the legal forms for reschedule. By doing so, you will have scheduled "current" and the event notice will not be destroyed.

After having rescheduled current, you may now say something like

```
schedule <event> after current;
```

However it is likely that this statement does not have the effect you want. It appears that this statement should be the same as

```
schedule <event> now;
```

or

```
schedule <event> delay 0;
```

But it is not. If you execute the statements:

```
reschedule current at 10.0;
schedule arrival(3) after current;
```

then the last statement is equivalent to

```
schedule arrival(3) at 10.0;
```

since "current" has been scheduled at time 10.0 and the after clause will schedule "arrival" to the same time as "current".

4. Pseudorandom Number Generation in SIMPAS

All (pseudo) random-number generators in SIMPAS depend on the basic uniform (0,1) random-number generator "u_random":

```
function u_random(stream: integer): real;
```

The argument to "u_random" is the stream identifier which indicates which element of the array "seed_v" is to be used to as a seed to generate the random number. The absolute value of "stream" must be between 1 and "n_seed" respectively. (In the distributed version of SIMPAS, n_seed=10). If "stream" is positive it directly indicates which element of the array is to be used; if stream is negative then "seed_v[abs(stream)]" is used, but then the antithetic variate (one minus the generated value) is returned as the value of u_random. Antithetic variates are sometimes useful in variance reduction techniques for the analysis of simulation experiments [KLEI74].

"u_random", in turn, calls a machine-dependent random number generator named "r_random":

```
function r_random(var seed: integer): real;
```

In the distributed versions of SIMPAS, "r_random" is implemented in a more or less machine-independent way using the mod function of PASCAL. The distributed version will not work properly on machines with word sizes smaller than 32 bits. In any case, "r_random" can be replaced by a more efficient, machine-dependent version as necessary. In general we would recommend that you replace "r_random" with a uniform (0,1) pseudo-random number generator in common use at your computer facility or one that has passed a set of statistical tests such as those described in [KNUTH71].

The routines mentioned above ("u_random" and "r_random") are automatically included in every SIMPAS program. The following random number generation routines are included by requesting them in the "include" statement. The "stream" argument always determines which random number stream is used to generate the results:

```
function expo( lambda: real; stream: integer): real;
```

generates an exponentially distributed random variable with parameter "lambda". This procedure uses the inverse transform method.

function poisson(lambda: real; stream: integer): integer;
generates an integer random variable from the Poisson distribution with parameter "lambda". This procedure uses Algorithm P1, page 440 from [FISH78].

function binomial(r:integer; p:real; stream:integer):integer;
generates a binomial random variable. "r" is the number of trials; "p" is the probability of success on any given trial.

function udisc(a, b, stream: integer): integer;
generates a uniform discrete random variable whose value is an integer in the range "a" to "b" (inclusive).

function normal(mu, sigma: real; stream: integer): real;
generates a normally distributed random variable with mean "mu" and variance "sigma". The acceptance-rejection method given as Algorithm N3B on page 414 of [FISH78] is used to generate the random variable.

function lognormal(mu, sigma: real; stream: integer): real;
generates a lognormal random variable. This function uses function "normal" so if "lognormal" is requested, the user must request "normal" as well.

function gamma(alpha, beta: real; stream: integer): real;
generates a random variable whose density is given by:

$$f_X(x) = \frac{x^{\alpha-1} e^{-\beta x} \beta^{\alpha}}{\Gamma(\alpha)} \qquad x \geq 0$$

"Alpha" need not be an integer. Algorithm G3A page 425 of [FISH78] is used. This procedure is not optimal for large values of alpha; instead one should probably use Algorithm G3B page 426.

function erlang(alpha:integer; beta:real; stream:integer):real;
generates an Erlangian random variable as the sum of "alpha" exponential random variables. The resulting random variable has mean 1/"beta". The method is that of Algorithm G1B page 421 of [FISH78].

function beta(a, b: real; stream: integer): real;
generates a random variable with the beta distribution; here "a"-1 is the exponent of x and "b"-1 is the exponent of (1-x). Algorithm Be1, page 430 of [FISH78] is used.

function unif(a,b : real; stream : integer):real;
generates a continuous uniform random number in the range ("a","b").

function choose(a : real; stream : integer) : boolean;
returns true with probability "a".

function hyper(alpha,mul,mu2:real; stream:integer):real;
generates a random variable with the two-stage hyperexponential distribution:

$$F_X(x) = \alpha\left(1-e^{-x\mu_1}\right)+(1-\alpha)\left(1-e^{-x\mu_2}\right) \quad x \geq 0$$

The obvious composition method is used.

gdisc

While not a pseudorandom generation procedure itself, putting this name in the include list causes a collection of general discrete random variable setup and generation routines to be included. To define a general discrete random variable, one declares it to be of type "gdiscvar". The variable is then initialized using one of the two routines "r_gdiscsetup" or "i_gdiscsetup" depending on whether you want to generate real- or integer-valued random variables. The calling sequences for the setup routines are:

```
procedure r_gdiscsetup(var head_rand : gdiscvar;
                       first: boolean;
                       tprob, tvalue : real);

procedure i_gdiscsetup(var head_rand : gdiscvar;
                       first: boolean;
                       tprob : real; tvalue : integer);
```

where:

"head_rand" is the name of the random variable, and must be declared as type "gdiscvar".
"first" is true on the first call to the setup routine.
"tprob" is the probability to be assigned to "tvalue".
"tvalue" is the value (real or integer as appropriate).

To generate a random variable with the general distribution, one calls the general discrete generation routines:

```
function r_gdisc(head_rand : gdiscvar;
                 stream : integer) : real;

function i_gdisc(head_rand : gdiscvar;
                 stream : integer) : integer;
```

A run-time error will occur if when either "r_gdisc" or "i_gdisc" is called for the first time with a particular argument, the random variable is found to be defective, that is, if the "tprob" values saved during the setup process do not add up to one.

The inverse transformation method is used, and the values are stored as a linear linked list. For general discrete random variables with large numbers of values a binary search tree would be more efficient.

5. Queues (Linked Lists) in SIMPAS

SIMPAS provides facilities for the declaration, maintenance and inspection of linked lists or queues of temporary entities (queue members). Summary statistics about the number of elements in a queue are also maintained.

5.1. Queue Declarations

A queue declaration consists of two parts. The first part, which is found in the global type declaration section of the program, specifies the type identifiers for the queue members and the queues. Queue and queue member variables are then declared in var parts of the program.

The queue member type declaration is of the form:

```
<entity> = queue member
               <attribute-1> : <type-1>;
               <attribute-2> : <type-2>;
               . . .
               <attribute-n> : <type-n>;
           end;
```

There need be no user defined attributes; however in this case the end keyword must still be present.

To declare a queue type, one uses a declaration of the form:

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

Queue and queue-member variables can be declared using declarations of the form:

```
var
    <queue-name> : <queue-type>;
    <queue-member> : ^<entity>;
```

One also can declare variables of type <entity>, but since the queue handling statements require a variable of type ^<entity>, a variable of type <entity> would be impossible to insert or remove from a queue.

Note that certain PASCAL implementations use a stronger notion of type equivalence than that of standard PASCAL. In standard PASCAL two types are equivalent if they have the same structure. Thus if one declares two variables a and b as follows:

```
var
    a : record i,j : integer end;
    b : record i,j : integer end;
```

then a and b are of the same type and the statement a:=b is legal. Some PASCAL compilers use a stronger notion of type equivalence called "name equivalence"; the Berkeley PASCAL available on UNIX is an example. Under Berkeley PASCAL, two variables are of the same type only if they have been declared using the same type identifier. In particular, unnamed types such as given in the last example are unique to the variables declared with them. Thus a and b are not considered to be of the same type under Berkeley PASCAL and hence they are not assignable to one another. To get around this, one must either declare all such variables at the same time or declare a type identifier and then declare a and b with the same type name:

```
type
    rec = record i,j : integer end;
var
    a : rec;
    b : rec;
```

or:

```
    a,b : record i,j : integer end;
```

Now a and b are of the same type and b is assignable to a so that a:=b is legal.

The point of this discussion is that if your PASCAL compiler uses name equivalence to determine type equivalence, it will not work to declare variables as ^<entity>, unless all such variables are declared at the same time. For example, if we say:

```
var
```

```
a : ^<entity>;
b : ^<entity>;
```

Then a and b are not assignable! However, if we declare a and b like this:

```
a,b : ^<entity>;
```

Then a and b are assignable.

To simplify declaration of entity pointers, the preprocessor creates the type name "ptr_<entity>" for ^<entity>. One can therefore declare a and b as follows:

```
var
    a : ptr_<entity>;
    b : ptr_<entity>;
```

This declaration will work regardless of whether not the host PASCAL compiler uses name equivalence to determine type equivalence.

For example, to declare a queue of jobs called "cpu_queue" and a variable called "jobptr" to access members of the queue, one would use the following declarations:

```
type    (* global type declarations *)
    job = queue member
            arrival_time:real;
            cpu_time:real;
            memory_size:integer;
    end;

    job_queue = queue of job;

    . . .
var     (* global or local var declarations *)
    cpu_queue:job_queue;
    jobptr : ptr_job;
```

A variable of type queue member cannot be in more than one queue at a time. Furthermore, a queue member must be removed from one queue before it can be placed into another queue. This is necessary to properly maintain the queue occupancy statistics.

5.1.1. Restrictions Certain restrictions have been imposed on the queue member and queue type declarations in order to simplify the preprocessor:

(1) As mentioned above, the type descriptors queue member and queue are only allowed in the global type

declaration part of the program. They will not be recognized anywhere else in the program. Their presence in other parts of the program will cause compilation time errors.

(2) Complex types which include the declaration queue or queue member are not allowed. The preprocessor will not recognize a queue or queue member type declaration unless the keyword queue immediately follows the equals sign in the type declaration. Thus if one wishes to have an array of queues or to include a queue as a field of a record, one must first assign a type identifier to the queue and then include the type identifier in the array declaration. Hence instead of saying

```
type
    job = queue member . . . end;
    job_queues = array [1..5] of queue of job;
```

one must say

```
type
    job = queue member . . . end;
    job_queue = queue of job;
    job_queues = array [1..5] of job_queue;
```

Similarly, one may not directly use a queue declaration as a type in a record.

Similar restrictions apply to the type queue member. However, since one normally needs to have access to the queue member through a pointer variable, declaring a variable to be an array of queue members is not normally useful.

(3) Before a queue member can be placed in a queue, it is necessary to initialize the queue. The SIMPAS preprocessor automatically creates an initialization routine for each type of queue. For a queue of type <queue> the initialization routine is named "i_<queue>." To initialize a particular queue variable, call the initialization routine for that type of queue with the queue variable as the argument. For example:

```
type
    job = queue member . . . end;
    joblist = queue of job;

var
    job_lists : array [1..5] of joblist;

    . . .
    (* to initialize job_lists[5] one would say: *)
```

```
i_joblist(job_lists[5]);

(* to initialize all of job_lists one would say
   something like: *)

for i:=1 to 5 do i_joblist(job_lists[i]);
```

The purpose of the initialization routine is to set the queue head pointer properly and to initialize the queue statistics variable. An attempt to insert a member into a queue which has not been initialized will usually cause a run time error; it is impossible to guarantee this across all PASCAL implementations.

5.2. Standard Queue and Queue Member Attributes

Every queue member has a standard list of attributes defined by the preprocessor. These attributes can be referred to wherever the queue-member variable is accessible. The user may not declare an attribute of the same name as the standard attributes. Doing so will cause a compilation time error. One refers to the attributes using the dot notation of PASCAL; thus to refer to the attribute "size" of queue "job_queue", one would say "job_queue.size". The standard queue member attributes are:

next- This attribute is of type ptr_<entity> and 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 is of type ptr_<entity> and points to the previous member of the queue or to the queue head if this is the first member of the queue.

inqueue- This boolean attribute is true if the queue member is in a queue.

qhead- This attribute is of type ptr_<entity> and points to the head node of the queue, or is nil if the <entity> is not in any queue. Thus one can determine if an <entity> is in <queue> by using an if statement of the form:

```
if <entity>^.qhead = <queue>.head then
    (* yes it is *) . . .
    else
    (* no it isn't *) . . .
```

The standard queue attributes are:

empty- This boolean attribute is true if the queue is empty.

size- This integer attribute gives the number of members in the queue.

head- This attribute is of type ptr_<entity> and points to the head node of the linked list which represents the queue. This attribute is set when the queue is initialized.

stat- This attribute is of type "statistic" and is used to collect statistics about queue occupancy. The user can restart collection of these statistics by calling the routine clear:

```
clear(<queue>.stat,accumulate).
```

Statistics about queue occupancy are available through the statistics attributes. Thus the mean number of customers in a queue is given by <queue>.stat.mean. See Section 6 for further details on the type "statistic", its use, the procedure clear and other procedures associated with statistics collection.

5.3. Entity Creation and Disposal

To create a new queue member one uses the procedure call:

```
c_<entity>(<entity-pointer>);
```

where <entity-pointer> is of type ptr_<entity>. Similarly, to dispose of an existing queue member one uses the procedure call

```
d_<entity>(<entity-pointer>)
```

Thus the following can be used to create a new "job":

```
type
    job=queue member . . . end;
var
    jobptr : ptr_job;
        . . .
        end;
        . . .
        c_job(jobptr);
```

And to dispose of a "job" one can say:

```
d_job(jobptr);
```

Of course, one can always use the PASCAL procedures "new" and "dispose" to do the same thing. Recall, however, that dispose need not be a working procedure in an implementation of standard PASCAL. Instead it may be a dummy procedure which has no real function. To insure portability, a SIMPAS program cannot depend on procedure "dispose". By using the procedures c_<entity> and d_<entity> to create and destroy queue members, a simulation program will be transportable to other implementations of SIMPAS, regardless of whether or not "dispose" is a working procedure in that implementation.

An additional advantage of using c_<entity> is that the standard queue member attributes will be properly initialized when the <entity> is created. (Initialization of fields of records created by "new" statements is not specified in standard PASCAL.)

5.4. Queue Manipulation

SIMPAS provides a variety of queue manipulation statements. The simplest forms are the statements:

```
insert <e_ptr> in <queue>;
remove the first <e_ptr> from <queue>;
```

In the first statement the entity is inserted last in the queue; while in the second statement the entity removed is the first entity in the queue. Thus these simple statements enable a straightforward implementation of a FCFS queue.

In these statements, <e_ptr> must be a simple or qualified variable of type "ptr_<entity>" and <queue> must be a simple or qualified variable of type queue of <entity>. Attempts to insert or remove an entity of the wrong type in a queue will result in type clash errors at compile time.

Other variations of the insert statement are:

```
insert <e_ptr> first in <queue>;
insert <e_ptr> last in <queue>;
insert <e_ptr-1> after <e_ptr-2> in <queue>;
```

The second case is equivalent to the same phrase with the word "last" omitted. In the third case, <e_ptr-2> must be in the queue <queue>; if it is not, then a run-time error will occur.

The following variations on the remove statement are supported:

```
remove the first <e_ptr> from <queue>;
remove the last <e_ptr> from <queue>;
```

```
remove <e_ptr> from <queue>;
```

The second statement is the opposite of the remove the first statement. The effect of the third statement is to remove the particular entity pointed at by <e_ptr> from the <queue>. In this case the remove statement does not modify the <e_ptr> while in the other cases the remove statement assigns to <e_ptr> a pointer to the entity which was removed. The keyword the in these statements is optional.

To continue our cpu_queue example, one would normally use the following declarations and statements to insert and remove jobs from the "cpu_queue":

```
var
    cpu_queue:job_queue;

    . . .
event departure; forward;

event arrival;
var job_pointer : ptr_job;
    begin
    (* create a job *)
    c_job(job_pointer);

    (* assign a cpu time to job_pointer^.cpu_time *)
    . . .

    (* we will assume that the job at the head of the
       queue is executing *)
    if cpu_queue.empty then
            (* start cpu *)
            schedule departure delay job_pointer^.cpu_time
    else
            insert job_pointer in cpu_queue;

    (* we will assume that inter_arrival_time has
       been defined *)
    reschedule current delay inter_arrival_time;
end; (* arrival *)

event departure;
var job_pointer:ptr_job;
    begin
        remove the first job_pointer from cpu_queue;

        if not cpu_queue.empty then
            reschedule current delay
                cpu_queue.first^.cpu_time;

        (* dispose of the job *)
        c_job(job_pointer);
```

```
end;
```

5.5. Forall Loops

To simplify searching queues, SIMPAS provides two types of loop statements:

```
forall <e_ptr> in <queue> do S;
forall <e_ptr> in <queue> in reverse do S;
```

As before <e_ptr> must be a simple or qualified variable of type "ptr_<entity>"; <queue> must be a simple or qualified variable of type queue of <entity>. Attempts to use a variable of type ptr_<entity1> as a loop index in a forall loop where the queue is of type queue of <entity2> will result in type clash errors at compile time.

In the current implementation (version 2), S must be a compound statement. This restriction is made to simplify the preprocessor and we expect that it will be lifted in later versions of SIMPAS.

If <queue> is empty then S is not executed.

The statement S must not include a remove <e_ptr> from <queue> statement. Otherwise the link structure used to implement the loop could be destroyed while the loop is executing. To remove all members from a queue, one cannot use a forall loop but instead must say:

```
while not <queue>.empty do
    remove <entity> from <queue>;
```

In a forall loop, the loop variable must be declared as a type "ptr_<entity>", and not of type <entity>. This means that specific fields of the <entity> must be referred to using the dereferenced name: <e_ptr>^. For example, to average all of the cpu times of the queue of jobs in the cpu_queue we declared above, one could use the following declarations and code:

```
var
        avg_cpu : real;
        job_pointer : ptr_job;
        cpu_queue : job_queue;

begin
        . . .

        avg_cpu:=0.0;
```

```
        forall job_pointer in cpu_queue do
                begin
                avg_cpu:=avg_cpu+job_pointer^.cpu_time;
                end;

        if not cpu_queue.empty then
                avg_cpu:=avg_cpu/cpu_queue.size
        else
                avg_cpu:=0.0;
        . . .
end.
```

Alternatively, one could use a PASCAL with statement to make the fields accessible:

```
        forall job_pointer in cpu_queue do
                begin
                with job_pointer^ do
                        avg_cpu=avg_cpu+cpu_time;
                end;
```

5.6. Forall Loops and the Event Set

To simplify scanning the event set, the event set is declared as follows:

```
type
    event_notice = queue member
        (* standard event attributes *)
            . . .
    end;
    ev_queue : queue of event_notice;
var
    ev_set : ev_queue;
```

The event set is thus a queue of event_notice's and is named ev_set; the only difference between the declaration of ev_set and that of a queue of events is that no statistics attribute is defined for ev_set. The result of this is that one can use a forall statement to scan the event set:

```
var
    ev_ptr : ptr_event;
    . . .
begin
    . . .
    forall ev_ptr in ev_set do
    begin
        case etype(ev_ptr) of
        no_event : begin . . . end;
        main     : begin . . . end;
        . . .
```

```
        end; (* case *)
    end;
    . . .
end.
```

However, since the event set is a queue ordered by event time, the user is prohibited from inserting and removing event notices from the event set using the insert and remove statements. Instead, to insert an event notice in the event set, use a reschedule statement; to delete an event notice from the event set, use a cancel or delete statement.

6. Statistics Collection in SIMPAS

At present, SIMPAS does not provide the automatic statistics collection features of SIMSCRIPT II.5. However, SIMPAS does provide a statistic collection type and a set of observation routines which significantly simplify statistics collection. To include the statistics routines one places the section name "statistics" in the include list. Since statistics about queue membership are automatically maintained, the statistics routines are always automatically included whenever a queue type is declared.

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. This selection is made when the statistics variable is initialized with the "clear" routine:

```
    clear(nsys_stat, accumulate);  (* time averaged  *)
    clear(tsys_stat, tally);       (* event averaged *)
```

The procedure call clear(. . ., accum) is a shortened form of the first statement given above. A statistics variable must be cleared before it is used to save observed data. The routine "clear" can also be used to reset statistic collection during a run.

To observe a value of a variable, one calls an observation routine of the appropriate type. The observation routines are:

```
r_observe( value, stat_variable); (* for real values *)
i_observe( value, stat_variable); (* for integer values *)
b_observe( value, stat_variable); (* for boolean values *)
```

As an example, to observe the values of nsys and tsys, one would say:

```
(* nsys is integer, so call i_observe *)
i_observe( nsys, nsys_stat);

(* tsys is real, so call r_observe    *)
r_observe( tsys, tsys_stat);
```

The standard statistics attributes are

max	max value observed
min	min value observed
total	sum of values observed
mean	mean of values observed
variance	variance of values observed

Other attributes can easily be added.

These quantities are always available through the standard statistic attributes. That is, at any time the mean and variance of an observed variable (up to the value recorded by the last observation routine call for the variable) are given by the mean and variance attributes of the associated statistics variable. To include the current value in a statistics attribute, one must call the observation routine.

Some example attribute references are:

nsys_stat.mean	is the time average 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

The observation routine uses the algorithm of [WEST79] to stably update the mean and variance.

A subtle point with respect to time-averaged observations is that the observation routines expect to be called immediately before the value of the observed variable has changed. (This seems more natural than having to call the observation routine after the value changes, since in that case one must make an artificial observation at time zero to initialize the statistics variable.) If the value of the observed variable at the end of the simulation is to be

included, the user must make an extra observation routine call when the simulation completes. In most cases, this extra observation will not significantly change the statistics, but it can be significant if the observed variable has not changed value for a non-trivial fraction of the simulation run time.

7. A Sample SIMPAS Program

The following SIMPAS program simulates an M/M/1 queueing system. Our discussion about this program is contained in comments in the program text:

```
program example_simulation(output);

(* the program reads no input because all parameters
   are declared as compile time constants *)

const
   max_departures = 5000;
   arrival_stream = 1;
   service_stream = 2;
   arrival_rate   = 0.36;
   service_rate   = 0.4;
   normalterm     = 1;

type
   job       = queue member
                 arrival_time : real;
               end;

   job_queue = queue of job;

var

(* departures counts the number of departures *)
(* arrivals   counts the number of arrivals   *)
   departures, arrivals : integer;

(* waiting_queue is the queue of waiting jobs *)
   waiting_queue        : job_queue;

(* status is used in the "start simulation" statement *)
   status               : integer;

(* tsys_stat records the mean time in system etc      *)
   tsys_stat            : statistic;

(* sys_busy  records the amount of time the system is busy *)
   sys_busy             : statistic;

(* fetch exponential random number generator routine
   from library *)
```

```
include expo;

(* we could have declared event departure first,
   but this shows how to forward an event   *)
event departure; forward;

event arrival;

var
    arriving_job : ptr_job;

    begin (* arrival *)
       arrivals:= arrivals + 1;
       c_job(arriving_job);  (* create a new job *)

       (* set the jobs arrival time *)
       arriving_job^.arrival_time  := time;

       (* put the new arrival in the waiting_queue and
          schedule a departure event if necessary *)
       if waiting_queue.empty then
          begin
             (* record end of system idle period *)
             b_observe( false, sys_busy);
             insert arriving_job in waiting_queue;
             schedule departure
                 delay expo(service_rate,service_stream);
          end
       else
          insert arriving_job in waiting_queue;

       (* set up next arrival *)
       reschedule current
           delay expo(arrival_rate, arrival_stream);

    end; (* arrival *)


event departure;

var
    departing_job : ptr_job;

    begin (* departure *)

       departures:= departures + 1;

       remove the first departing_job
                    from waiting_queue;

       (* record this job's time in system *)
       r_observe(time - departing_job^.arrival_time ,
                  tsys_stat);
```

```
        (* stop simulation if requested number jobs have
           departed *)
        if (departures >= max_departures) then
           schedule main(normalterm) now;

        (* otherwise dispose of this job and
           reschedule departure *)
        d_job(departing_job);

        if waiting_queue.empty then
           (* record end of system busy period *)
           b_observe( true, sys_busy)
        else (* schedule next departure *)
            schedule departure
                delay expo(service_rate, service_stream);

    end; (* departure *)


begin (* main procedure *)

    (* initialize waiting_queue *)
    i_job_queue(waiting_queue);

    (* initialize statistics *)
    clear(tsys_stat, tally);
    clear(sys_busy,  accumulate);

    (* set random number generator seeds *)
    seed_v[arrival_stream] := 87654 ;
    seed_v[service_stream] := 67993;

    (* schedule first arrival *)
    schedule arrival now;

    (* run the simulation *)
    start simulation(status);

    (* print results of run *)
    writeln('Simulation Terminated at:', time:10);
    writeln('End of run status       :', status:10);

    (* flush out final busy/idle observation *)
    b_observe( waiting_queue.empty, sys_busy);
    writeln('Server utilization      :',
                          sys_busy.mean:10);
    writeln('Number of jobs serviced :', departures:10);
    writeln('Number of arrivals      :', arrivals:10);

    (* note that time average mean number of jobs in
       waiting_queue is the time average mean number
       of jobs in system -- and this  is recorded
       automatically *)
```

```
   writeln('Mean number in system   :',
                          waiting_queue.stat.mean:10);

   writeln('Max  number in system   :',
                          waiting_queue.stat.max:10);

   writeln('Mean time in system     :', tsys_stat.mean:10);
   writeln('Max  time in system     :', tsys_stat.max:10);

end.
```

7.1. Execution Output

```
simulation terminated at:  1.407e+04
end of run status       :         1
server utilization      :  8.564e-01
number of jobs serviced :      5000
number of arrivals      :      5013
mean number in system   :  7.946e+00
max  number in system   :  5.800e+01
mean time in system     :  2.232e+01
max  time in system     :  1.390e+02
```

8. Acknowledgements

Mr. Mark Abbott was responsible for the implementation of the SIMPAS preprocessor itself, and without his assistance the project would never have been completed. This project was supported in part by the Wisconsin Alumni Research Foundation. 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. Finally, I would especially like to thank Dr. Raphael Finkel for his assistance in debugging the early versions of SIMPAS and his editorial assistance in writing this manual.

Appendix A

SIMPAS Implementation Notes

A.1: The Event Set and the Simulation Control Routine

Event routines are called and the simulation clock is advanced by the "simulation control routine" in conjunction with the "event set". The event set is a linked list of records ("event notices"), each of which describes the execution of an event. An event notice contains the actual arguments for the event, the simulation time when the event is to be executed and a link to the next member of the event set. To schedule an event, an event notice for the event is created, the actual arguments of the event are stored in the event notice, the time of the event is saved in the event notice, and the event notice is inserted in the event set. The event set is sorted by increasing simulation time; the next event to occur is always described by the event notice at the front of the event set.

The simulation control routine proceeds by (1) advancing the clock to the time of the first event in the event set, (2) removing that event notice from the event set, and (3) calling the appropriate event routine. When the event routine returns, this process is repeated.

The simulation control routine is itself called by the start simulation statement. The control routine continues to execute as described above until (1) the event set becomes empty, or (2) an event notice for event "main" reaches the head of the event set. In either of these cases the control routine returns to its caller and the simulation is stopped.

A.2: Event Set Structure

The event set is maintained as a doubly linked list with head node. It thus has the same structure as a queue, except that there is no statistics attribute for the event set. The event set is declared as:

```
ev_set : record
     head : ptr_event;
     size : integer;
     empty : boolean;
end;
```

These attributes have the same meaning as for queues; see Section 5.2 for more details. "ptr_event" is declared as

```
ptr_event = ^event_notice;
```

and an event notice is declared as a record with variants:

```
event_notice = record
      next, prev, qhead: ptr_event;
      inqueue, named: boolean;
      evtime: real;
      case eventtype: t_ev_1 of
            no_event    : ();
            . . .
            <event>     : (a_<event> : t_<event>);
            . . .
            main        : (a_main : t_main);
end;
```

"next" and "prev" point to the next and previous event notices in the event set; "qhead" points to the head of the event set. "evtime" contains the time of the event and "eventtype" gives the name of the event (e. g. arrival, departure, etc.). "inqueue" is true as long as the event notice is in any queue. The boolean variable "named" is used to override the automatic reclamation of the event notice after the event is executed when the notice was created via a schedule statement with a named clause.

The type "t_ev_1" is an enumeration type each of whose values is either the name of an event or the names "no_event" or "main". The preprocessor inserts one line in this case statement for each event declared by the user. If the event has arguments, the preprocessor declares a record to hold them. The record is named "a_<event>" and is of type "t_<event>". The fields of "a_<event>" are set to the values of the actual arguments during the execution of a schedule statement.

Some other global variables associated with the event data structure are:

```
time    : real;
g_notice: ptr_event;
current : ptr_event;
```

"time" contains the current simulation time. "g_notice" is a global temporary used by the schedule statement code to hold a pointer to the event notice being scheduled. "Current" contains a pointer to the current event notice. This is so the user can say reschedule current . . . .

A.3: Event Notices and Event Scheduling

The schedule statement

```
schedule arrival delay expo( lambda, 3);
```

expands to

```
begin
    c_notice( g_notice, arrival );
    g_notice^.evtime:= time + expo( lambda, 3);
    e_insert( g_notice, <line-no> );
    g_notice:= nil;
end;
```

The routine c_notice creates an event notice of the specified type (in this case of type "arrival") and returns a pointer to the new notice in the variable "g_notice". The "evtime" of "g_notice" is then set and the routine e_insert is called to insert the notice at the appropriate point in the event set. The second argument of e_insert is the SIMPAS line number of the schedule statement and is used for reporting run-time errors.

A somewhat more complicated case is the statement

```
schedule terminate(leastjob) named death delay run_time;
```

We will assume that terminate is declared as

```
event terminate( jobp : ptr_job);
```

This statement expands to:

```
begin
    c_notice( g_notice, terminate );
    with g_notice^ do begin
        a_terminate .jobp := leastjob ;
    end;
    death:= g_notice;
    g_notice^.named:= true;
    g_notice^.evtime:= time + run_time ;
    e_insert( g_notice, <line-no> );
    g_notice:= nil;
end;
```

The primary differences between this and the last statement are that the argument to the event is saved in the field "jobp" of the record "a_terminate" and that a pointer to the generated event notice is saved in the user-declared variable "death".

A.4: Queue and Queue Member Declarations

The queue-member type declaration is expanded into a record type as follows:

```
<entity> = queue member
    <attribute-1> : <type-1>;
```

```
    <attribute-2> : <type-2>;
    . . .
    <attribute-3> : <type-n>;
    end;
```

becomes:

```
ptr_<entity> = ^<entity>;
<entity> = record
    next, prev, qhead: ptr_<entity>;
    inqueue : boolean;
    <attribute-1> : <type-1>;
    <attribute-2> : <type-2>;
    . . .
    <attribute-3> : <type-n>;
    end;
```

The three pointer fields are initialized to nil and "inqueue" is initialized to false when the <entity> is created. The qhead pointer is nil unless the <entity> is currently in a queue, and otherwise the head pointer points to the head node of the queue. This is used to check that <entity> is indeed in a particular queue.

In the "no dispose" case (See Appendix C), for each queue-member type declared, the preprocessor declares the global variable "f_<entity>" as type "ptr_<entity>". This global variable is used to maintain an free list of entities of this type. The procedures c_<entity> and d_<entity> use this global variable. f_<entity> is initialized to nil at the start of the main procedure.

The <entity> creation and deletion routines c_<entity> and d_<entity> are inserted at the top of the program after the global var part declarations. In the "no_dispose" case, the procedure c_<entity> works by first looking to see if f_<entity> is nil. If it is then c_<entity> uses "new" to dynamically create another <entity>. Otherwise it unlinks the next <entity> from the free list. d_<entity> links the <entity> pointed its argument onto the free list. In the "usedispose" case, d_<entity> uses "dispose" instead of using the free list, and c_<entity> uses "new" to create a new entity. Since "dispose" is not a working procedure in all versions of PASCAL, the free list mechanism is provided to allow dynamic entity creation and disposal in any SIMPAS implementation.

The queue head type declaration

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

is expanded to

```
<queue-type> = record
                    head : ptr_<entity>;
                    size : integer;
                    empty : boolean;
                    stat : statistic;
               end;
```

Note that the queues are maintained as doubly-linked lists with head nodes. <queue-type>.head points at the head node of the queue once the queue has been initialized. Thus, the first member of the queue (assuming it is not empty) is given by <queue-type>.head^.next; the last member is <queue-type>.head^.prev.

For every queue type declared in the program, the preprocessor creates and inserts an initialization procedure. For the queue type named <queue> the initialization routine is named i_<queue>. The initialization routine initializes the stat pointer and resets the statistics. The size of the queue is set to zero and empty is set to true. A new entity of type <entity> is generated by calling the procedure c_<entity>; head is set to point at this entity. The next and prev fields of head^ are set to head to represent an empty, doubly-linked list, head^.inqueue is set to false and head^.qhead is set to nil. These special settings of the fields of the head node are used in the remove statements to detect attempts to remove a member from an empty queue.

Note that attempting to insert a member in an uninitialized queue will probably result in a reference through an uninitialized pointer and a corresponding run-time error.

A.5: Insert

The statements

```
insert <entity> in <queue>
insert <entity> first in <queue>
insert <entity> last in <queue>
insert <entity> before <entity-2> in <queue>
```

are all translated to an equivalent insert after statement, and then the insert after statement is expanded. For example,

```
insert <entity> first in <queue>
```

is translated to

```
insert <entity> after <queue>.head in <queue>
```

and then the later statement is expanded. Thus we need only

describe the expanded PASCAL produced by an insert after statement.

Statements of the form

insert <entity-1> after <entity-2> in <queue>

are expanded to

```
begin
    if <entity-1>^.inqueue then error_p(9,<line-no>);
    if <entity-2>^.qhead <> <queue>.head then
                                    error_p(10,<line-no>);
    with <entity-1>^ do
    with <queue> do
    begin
        qhead := <entity-2>;
        next := qhead^.next;
        prev := qhead;
        qhead^.next := <entity-1>;
        next^.prev := <entity-2>;
        inqueue := true;
        qhead := head;
        observe(size,stat,accumulate);
        size := size + 1;
        empty := false;
    end;
end
```

If <entity-1> is currently in a queue, then the error message: "tried to insert a member already in a queue at line nnn" is printed. If <entity-2> is not in the queue <queue>, the error message: "tried to insert after a member not in the queue at line nnn" will be printed. Note that the head node is not considered to be in the queue. Thus attempting to insert an entity first in a queue by a statement of the form:

insert <entity> after <queue>.head in <queue>

will also cause this execution time error.

Note that each insert statement is expanded in-line to several lines of PASCAL. An alternative approach would be for the preprocessor to create an "insert" routine for each queue type. This is not done in the version 2 preprocessor because the preprocessor does not know the types of the variables and hence if the insert statement were to be converted to a subroutine call the preprocessor could not determine which insert subroutine should be called. Remove statements are expanded in-line for exactly the same reason. Thus if a program contains many insert and remove statements, significant savings can be made by isolating these

statements in subroutines and calling the subroutines instead of using the insert and remove statements repeatedly throughout the program.

A.6: Remove

The statement

remove <entity> from <queue>

is translated to

```
begin
    with <queue> do
    begin
        if <entity>.qhead <> head then
                    error_p(err_id, <line-no>);
        observe(size,stat,accumulate);
        size:=size-1;
        empty:=size=0;
    end;
    with <entity> do
    begin
        inqueue:=false;
        qhead := nil;
        prev^.next:=next;
        next^.prev:=prev;
    end;
end
```

Statements of the form

remove the first <entity> from <queue>
remove the last <entity> from <queue>

are implemented by first setting <entity> to <queue>.head^.next or <queue>.head^.prev (respectively) and then executing the above code. Attempting to remove the first or last member from an empty queue will be caught because <queue>.head^.next will then point at the queue head node which has qhead set to nil. Thus attempting to either remove a member from a queue it is not in, or attempting to remove the first or last member from an empty queue causes the same execution time error message: "tried to remove a member from a queue it is not in or attempted to remove the first or last member of an empty queue at line nnn."

A.7: Forall

Statements of the form

forall <entity-ptr> in <queue> do S

are translated to

```
begin
    <entity-ptr> := <queue>.head^.next;
    while <entity-ptr> <> <queue>.head do
    begin
        S;
        if <entity-ptr>^.qhead <> <queue>.head then
                                    error(11,<line-no>);
        <entity-ptr> := <entity-ptr>^.next;
    end;
end
```

The test after the statement S is to ensure that <entity-ptr> is still in the queue. The error message in this case is: "user removed the loop variable in a forall loop."

Statements of the form

forall <entity-ptr> in <queue> in reverse do S

are translated similarly, except that "prev" is used instead of "next".

A.8: Libfile Organization

The libfile consists of a header and five parts. The header is a single line which identifies whether the libfile is one which is designed to be used with a PASCAL compiler which supports the procedure dispose or a libfile to be used with a PASCAL compiler in which dispose is not a working procedure. The header is checked to make sure the libfile present matches the setting of the constant "usedispose" in the preprocessor. (See Appendix C for details.)

The parts of the libfile are indicated by a line which begins with a dollar sign and they correspond to the parts of a PASCAL program:

$const	insertions for the const part
$type	insertions for the type part
$var	insertions for the var part
$procedures	insertions for the procedure part
$main	insertions for the start of the main program (initialization code)

Within each of these parts of the libfile are the sections which the user requests on the "include" statement. The start of a section is flagged by a line which begins with an asterisk. The rest of the line gives the section name. One section is always included; its name is "(default)". The other sections are included only if the

user requests them or, in the case of the statistics section, if the user declares a queue.

The algorithm for including sections from the libfile is the following. During the second pass of the preprocessor, the beginning of the global const, type, var, procedure, and main program parts of the SIMPAS program are detected by recognizing flags put at the appropriate places during the first pass. When the global const part of the program is found, for example, the const part of the library file is read. Every section whose name is on the include list (created by the first pass) is inserted in the program. This process is repeated for each of the other sections.

A section name normally appears in several parts of the library file. For example, the section name "statistics" appears in both the type part and the procedure part of the library file. The statistics section in the type part contains the declarations for the types stat_type and statistic. The statistics section in the procedure part of the library file contains the procedures r_observe, i_observe, and b_observe. In this way the library file can be used to include constants, types, and initialization code associated with each procedure included from the library file. There is no requirement that an included section appear in any particular part of the library file. A preprocessor error will occur if a requested section name is not found in any part of the library file.

Each section of the library file is uninterpreted by the preprocessor. When the appropriate section is found all lines up to the end of the section are inserted in the output program. Thus, while the most common case is for a section to contain a single procedure, it may contain several. If the user has access to the library file, he may easily customize it to satisfy his own requirements.

Appendix B

Reserved Words and Implementation Restrictions

B.1: Reserved Words

All of the reserved words in PASCAL are clearly reserved words in SIMPAS as well. In addition, the following words are reserved due to their use in SIMPAS extensions to the language PASCAL:

after	last
at	member
before	named
cancel	now
delay	of
delete	queue
destroy	reschedule
event	remove
first	start
forall	simulation
from	the
insert	

The following identifiers are reserved for use by the preprocessor:

accum	n_seed
accumulate	no_event
b_observe	no_stat
c_notice	r_observe
current	r_random
d_notice	s_control
e_insert	sched_n
error_p	scheduled
error_x	seed_v
ev_set	stat_type
event_notice	statistic
f_evnotice	t_ev_l
g_notice	t_main
i_i	tally
i_ev	time
i_observe	u_random
main	

The user should also avoid using identifiers which are the same as names of routines in the libfile.

Finally, the preprocessor defines some identifiers in response to declarations made by the user:

For an event named <event> the following names are generated and used by the preprocessor:

r_<event>	is the event routine name
a_<event>	used to hold actual arguments of event
t_<event>	is the type of a_<event>

For a queue member of type <entity> the following names are generated and used by the preprocessor:

c_<entity>	entity creation routine
d_<entity>	entity destruction routine
f_<entity>	head of free list or entities (only used in "no dispose" case)
ptr_<entity>	pointer type for name equivalence

For a queue named <queue> the name i_<queue> gives the name of the queue initialization routine.

B.2: Implementation Restrictions

To simplify preprocessor implementation, we enforce certain restrictions on a SIMPAS program:

All identifiers are truncated to 12 characters during expansion. This is controlled by the compile-time constants "maxidlength" and "maxreslength" and can be changed to some other length if necessary. If these constants are redefined then several string constants and literals in the preprocessor must be changed to have the appropriate length.

No string literal in the SIMPAS program can be longer than 80 characters. If necessary, this can be increased by changing the value of the compile-time constant "maxlitlength".

No numeric constant in the SIMPAS program can contain more than 20 digits. This can be increased by changing the value of the compile-time constant "maxnumlength".

There can be at most 49 names in the include statement list. The constant "maxincl" can be increased to overcome this if necessary. The name "(default)" is always in the include list and consumes one of the slots allocated by this constant.

If the host PASCAL compiler supports both upper and lower case, the entire program is translated to one of these

cases to simplify identifier comparisons. The procedure "getchar" in the preprocessor is responsible for this translation. Since this procedure must usually be modified in the process of transporting SIMPAS (See Appendix C), the choice of which case to use as the "standard" case can easily be made at that time.

Names generated by the preprocessor must be unique. Thus if the host PASCAL only distinguishes between identifiers which differ in the first 8 characters, declaring events with names "aaaaaaaa" and "aaaaaabb" will cause compile time errors. The reason is that these events will become procedures named "r_aaaaaaaa" and "r_aaaaaabb" and the compiler regards these two names as identical. This problem is most critical with queue members, since the ptr_<event> type generated as part of the queue member declaration allows only four characters to uniquely describe a queue member. Since most PASCAL compilers distinguish identifiers in more than 8 characters, this is not as bad a problem as it may appear.

Qualified variables or expressions which evaluate to ptr_<entity>, ptr_event, or <queue> in schedule, reschedule, cancel, delete, destroy, insert, remove, or forall statements can not be arbitrarily complicated. A maximum of twenty "tokens" is allowed in each expression. Each identifier, string constant, or special character counts as one token. This limit can be changed by increasing the value of the compile time constant "maxexprlength" at the expense of some wasted storage.

An event may not have var arguments.

Events with names "main" or "no_event" are not allowed.

A schedule or reschedule statement uses the global variable "g_notice" to hold a pointer to the event notice being scheduled. Therefore if during a schedule or reschedule statement a user-defined procedure or function is called, that procedure or function cannot itself contain or cause the execution of another schedule or reschedule statement.

In the version 2 preprocessor, the body of a forall loop must be a begin-end, case-end, or repeat-until statement. If it is not, the error message "block expected to begin here" will be issued.

The loop variable must not be removed from the <queue> in a forall <entity> in <queue> statement. In most cases this will cause an execution time error, but the error can not always be detected.

A maximum of ten queue member types and ten queue types may be declared in any one SIMPAS program. These limits can be overcome by increasing the values of the compile-time constants "maxmembers" and "maxqueues".

The queue member and queue declarations may only appear in the global type declaration part of the SIMPAS program. See also the restrictions given in Section 4.1.1.

Extremely long input lines in a SIMPAS program can cause lines of output to be created which cannot be compiled. Without discussing specific systems, it is difficult to quantify the maximum length an input line can have; extremely long expressions with few blanks per line are the most common culprit. Blanks are squeezed from the source input before expansion so that blanks are not significant when discussing line length.

Appendix C

Transporting SIMPAS

This Appendix gives those details of the implementation which are machine specific and which must be modified when transporting SIMPAS to another machine.

C.1: Distribution Format

SIMPAS is distributed as a 9-track, 1600 bpi, unlabeled, fixed-block, ASCII tape. Each record on the tape consists of 800 characters and contains 10 card images. The blocks contain no control information and no special characters are used to compress out blanks.

There are four files on the tape; the files are separated by hardware end-of-file marks. The first file contains the SIMPAS preprocessor. The second and third files contain two different versions of the library file: one for the "usedispose" case and one for the "no dispose" case. The last file contains a copy of the example program described in Section 7.

The program as distributed is entirely in lower case and assumes that the character "_" (underbar) is a legal identifier character in PASCAL. The circumflex "^" character is used as the PASCAL pointer dereference operator.

C.2: PASCAL Compiler Requirements

While we have implemented the SIMPAS preprocessor using only the features of standard PASCAL, the implementor of a "standard" PASCAL compiler may enforce size limitations which make it impossible to compile the SIMPAS preprocessor. For example, the maximum size of a set in PASCAL can be as small as the number of bits in one machine word, although most implementations allow sets much larger than this. The SIMPAS preprocessor depends on being able to construct sets of "tokens"; there are 60 elements in this set. PASCAL compilers which enforce smaller maximum size limitations on sets will not be able to compile the SIMPAS preprocessor. Another, less important set (fatalerrors) has about 80 members; this set can be eliminated if necessary.

Proper execution of the SIMPAS preprocessor does not depend on the following features of PASCAL. These features, while part of the standard, are often not implemented:

(1) Global goto's.

(2) Procedures as parameters to functions and procedures.

(3) A working version of "dispose".

C.3: Character Set Differences

Since character sets differ from machine to machine, some adjustment of the PASCAL code will be necessary in order to run SIMPAS on your system. The most complicated case occurs if your computer system only supports upper case since you will have difficulty even reading the distribution tape. We will assume that somehow you get the entire thing translated to upper case and read into a disk file on your system. You can then do the rest of the changes described below using a text editor.

The most common character set problems deal with the characters "_", "^", and "horizontal-tab". "_" is special so let's discuss it first. We will assume for the moment that "_" is part of the character set supported by your system, but that your PASCAL compiler does not allow "_" as a character in identifiers. (If this is not the case, then you will have to translate this character to something else when you read the distribution tape.) Using your text editor, translate every occurrence of "_" in the preprocessor and the appropriate library file to some character which is legal in identifiers ("Ø" is a common choice). If you don't mind using names like ptrØevent instead of ptr_event you can skip the next step and do this same change on the test program as well. We have found the "_" character to be very useful for readability of SIMPAS programs so the next step is to configure the preprocessor to translate the underbar character for you.

To configure the preprocessor, get back into your text editor and look for the identifier "underbar" in the preprocessor source code (it should be at about line 3Ø). If you changed all occurrences of "_" to "Ø" you will have found a line that looks like

```
underbar = 'Ø';
```

This is clearly silly, so change the "Ø" back to "_" on this line only. Now look for "procedure getchar". The word procedure starts in column 1 if that will help you find it. It should be around line 7ØØ. All the source program input comes in through this procedure so we are going to translate "_" to something else here. At about line 73Ø there is a comment describing what needs to be done; you need to insert a line like:

```
if inchar=underbar then inchar:='Ø';
```

Getchar also is responsible for translating the input program to lower case. You can change this if you wish as follows. Use your editor to find a line containing

```
begin  (* proc getchar *)
```

This is the first line of the body of getchar (the "begin" is in column 1). A few lines later is the statement:

```
if inchar in ['A' .. 'Z'] then
    inchar := chr( ord(inchar) - ord('A') + ord('a'));
```

This translates the input to lower case. If you want to translate the input to upper case replace this with:

```
if inchar in ['a' .. 'z'] then
    inchar := chr( ord(inchar) - ord('a') + ord('A'));
```

You must do the case translation one way or the other.

Another difficulty with character sets is the "horizontal-tab" character, hereafter known as "HT". On some systems (UNIX for example), HT's are stored in the source and interpreted by the terminal or the terminal driver when the source is listed. This is fine, except that the preprocessor will then see the HT's and try to classify them as "miscellaneous-symbol tokens" which usually results in puzzling preprocessor error messages. To get around this define a character constant whose value is HT:

```
tab = 'HT';  (* put your HT character in there! *)
```

Now at about line 740, in procedure getchar, is the following case statement:

```
case inchar of
        lparen:  kind:= lparensym;
        rparen:  kind:= rparensym;
        lcurl:   kind:= lcurlsym;
        quote:   kind:= quotesym;
        comma:   kind:= commasym;
        semic:   kind:= semicsym;
        period:  kind:= periodsym;
        colon:   kind:= colonsym;
        space:   kind:= blanksym;
        equals:  kind:= equalssym;
        lsquare: kind:= lsquaresym;
        rsquare: kind:= rsquaresym;
      end; (* case *)
```

Insert the following case in this statement:

```
tab:    kind:= blanksym;
```

This will make all of the tab characters look like blanks to the preprocessor.

One last comment about character sets deals with the pointer dereference operator "^". On some systems this is represented by the two character graphic "->". Whatever the character is on your system, go through and change all occurrences of "^" to the appropriate symbol throughout all of the necessary SIMPAS files.

C.4: Usedispose and No Dispose

One of the stickiest problems of writing portable PASCAL software deals with the procedure "dispose". The procedure "new" is part of standard PASCAL, but dispose need not be a working procedure in standard PASCAL. Since dynamic creation and destruction of entities is an integral part of a discrete system simulation, SIMPAS has to provide some type of dynamic storage allocation facilities even if dispose is not a working procedure. This is done by isolating the user from the actual entity creation and destruction mechanism (the user is supposed to call c_<entity> and d_<entity>). In the case where dispose works (we will call this the "usedispose" case), d_<entity> and d_notice call dispose to destroy <entity>'s and event notices. In the case where dispose doesn't work (this is the "no dispose" case), d_<entity> and d_notice maintain free lists of <entity>'s and event notices respectively, and the procedures c_<entity> and c_notice first look to see if they can use a member of these lists to satisfy their request. If not then they call "new" to generate a new <entity> or notice.

Two things control which type of code is generated. The first is the boolean constant "usedispose" in the preprocessor. If this is set to true, then the preprocessor will generate code that uses dispose. If usedispose is set to false, then the code to handle the free-list case is generated. The second thing which controls the type of code that is generated is the library file which is present. Needless to say, setting usedispose to true and then using the "no dispose" case library file is not going to work. This is why the first line of the library file is a header which is either "usedispose" or "no dispose". Attempting to use a library file which does not match the setting of usedispose in the preprocessor results in a preprocessor error message.

The distributed version of SIMPAS has "usedispose=true;". This constant should be changed as is appropriate for your system. It is defined at about line 80 of the preprocessor.

C.5: Program Termination

Another problem with standard PASCAL is that there is no standard way to terminate program execution. Some PASCAL compilers require every program to terminate by falling off the end of the main program; this usually means using a global goto in order to terminate a program from inside of an arbitrary procedure. Most PASCAL compilers supply a routine named "halt" or "abort" which causes program termination.

These variations are handled in the SIMPAS preprocessor and run-time by calling the procedures terminate and error_x respectively. Procedure terminate is part of the preprocessor and is called when a catastrophic error is encountered. Fix this procedure to do whatever is necessary to terminate a PASCAL program on your system. Error_x is the error routine inserted in the output PASCAL produced by the preprocessor. It is declared in the library file. Change this procedure the same way you changed procedure terminate.

C.6: Random Number Generators

As discussed in Section 4, all random number generators depend on the basic random number generator r_random. r_random is contained in the "(default)" section of the $procedure part of the library file. (See Appendix A for a discussion of the organization of the library file.) The distributed version of SIMPAS contains an r_random which should work on any computer system with a word size of 29 bits or larger. For other systems you will have to supply a suitable r_random. Even if the standard r_random will work on your system, you may wish to replace the default r_random with a procedure tailored to your machine. In general we would recommend that you replace r_random with a uniform [0,1) pseudo-random number generator which is in common use at your computer facility or which has passed a set of statistical tests such as those described in [KNUTH71].

C.7: Source Input and Output

The distributed version SIMPAS reads the input source program from the standard input and directs the output source program to the file whose internal name is "outfile". If you wish the input to come from a file you should declare a new text file as appropriate and then change the statements:

```
inchar := input^;
```

```
get(input);
```

in procedure getchar to reference the appropriate input file name. You also should insert a "reset" statement for the file in procedure passlinit.

The expanded PASCAL output by the preprocessor will be placed in the file which corresponds to the internal file name "outfile". Since the method of establishing this correspondence is system dependent, we will not discuss it here. We will point out, however, that convenient places for establishing this correspondence (assuming this can be done from inside of a PASCAL program) can be found in the procedures passlinit and pass2init. This is where the input files, the temporary output file used by pass one, and the final output file are reset and rewritten.

Appendix D

SIMPAS Reference Guide

D.1: SIMPAS Statement Summary

```
include <name-1> [,<name-2>] . . .;

start simulation(<status>);

event <event-name>[<formal parameter list>];
    <label-part>
    <type-part>
    <var-part>
    <procedure and function decl part>
    begin
    <statement-list>
    end;

schedule <event-name>[<actual parameters>]
        [named <ev_ptr>]
        { now |
          at <time-expression> |
          delay <time-expression> |
          before <ev_ptr> |
          after <ev_ptr> }

cancel <ev_ptr>

destroy <ev_ptr>

delete <evptr>

reschedule <ev_ptr> { at <time-expression> |
                      delay <time-expression> |
                      before <ev_ptr> |
                      after <ev_ptr>  |
                      now }
<entity> = queue member
        <attribute-1> : <type-1>;
        <attribute-2> : <type-2>;
        . . .
        end;

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

insert <e_ptr> [{first | last |
                 before <e_ptr> |
                 after <e_ptr> }]
                 in <queue>
```

```
remove [the] [{first | last}]
        <e_ptr> from <queue>

forall <e_ptr> in <queue> [in reverse] do
                begin
                <statement list>
                end
```

D.2: Identifier Glossary

a_<event>	A record of type t_<event> used in event_notice to hold the actual parameters for event <event>.
accum	short form of "accumulate"
accumulate	Used in calling sequence to procedure clear to indicate that this statistic is to a time-averaged statistic.
b_observe	Boolean variable observation routine.
c_<entity>	This is the creation routine for queue members of type <entity>.
c_notice	An internal routine called to generate an event notice.
clear	This procedure resets a statistics variable and sets its type to either accumulate or tally.
current	Contains a pointer to the current event notice.
d_<entity>	This is the destruction routine for queue members of type <entity>.
d_notice	An internal routine called to destroy an event notice.
e_insert	An internal routine called to insert an event notice in the event set.
error_p	An internal routine called to print execution time errors.
error_x	Standard error exit routine.
ev_set	the event set

event_notice — A record type created by the preprocessor to hold event notices.

f_<entity> — A pointer to an free list of queue members of type <entity>. Used only in "no dispose" case.

f_evnotice — An array which holds pointers to the heads of free lists of event notices. Only used in "no dispose" case.

g_notice — A global, temporary variable of type ptr_event which is used to hold a pointer to the event notice being scheduled in the schedule or reschedule statement.

i_<queue> — This is the initialization routine for a queue of type <queue>.

i_i — An internal variable used during initialization.

i_ev — An internal variable used during initialization.

i_observe — integer variable observation routine

main — A pseudo-event corresponding to the main program.

n_seed — The number of elements of seed_v. Normally n_seed=10.

no_event — A dummy constant name in the enumeration type t_ev_1. Returned by procedure etype if the argument to etype is nil.

no_stat — A dummy constant name in the enumeration type stat_type. Used to attempt to discover use of uninitialized statistics variables.

ptr_event — A type name defined as ^event_notice.

ptr_<entity> — A type identifier defined as ^<entity>.

r_<event> — The name of the event routine (proceudure) for the event named <event>.

r_observe — real variable observation routine

r_random — A machine dependent, uniform (0,1) random number generator.

s_control	the simulation control routine
sched_n	An internal procedure called during schedule (or reschedule) before/after statements to make sure the event notice being scheduled before or after is itself scheduled.
scheduled	Returns true if its argument points to a scheduled event notice.
seed_v	seed_v[i] contains the current seed for random number stream "i". seed_v is declared as: array [1..n_seed] of integer.
stat_type	An enumeration type indicating what type of statistic variable this is.
statistic	A record type defined in the library file and included as part of the section "statistics". Used to declare statistics observation variables.
t_ev_l	An enumeration type containing the names of all the events in the program as constant values.
t_<event>	A type identifier used to declare the record named a_<event> in the record event_notice.
t_main	A record type used to declare the record which holds the <status> variable for event main. Needed for uniform treatment of all event arguments.
tally	A possible value of a variable of type stat_type, this value indicates that the statistics variable is an event-averaged variable.
time	The current simulation time. Time is a real variable.
u_random	The basic uniform (Ø,1) random number generator. It knows about random number streams and antithetics, while r_random does not.

References

[BRYA8Ø] Bryant, R. M. "SIMPAS -- A Simulation Language Based on PASCAL." University of Wisconsin-Madison Computer Sciences Department Technical Report No. 39Ø, June 198Ø.

[BRYA81] Bryant, R. M. "Micro-SIMPAS: A Microprocessor Based Simulation Language." To be presented at the Fourteenth Annual Simulation Symposium, Tampa, Florida, March 17-2Ø, 1981.

[DAHL69] Dahl, O. J., K. Nygaard, and B. Myhrhaug. "The Simula 67 Common Base Language." Pub S-22, Norwegian Computing Center, Oslo (1969).

[FISH78] Fishman, G. Principles of Discrete Event Simulation. John Wiley and Sons, New York (1978).

[FRAN77] Franta, W. R. The Process View of Simulation. Elsevier North-Holland, Inc., New York (1977).

[IBM72] "SIMPL/1 (Simulation Language Based on PL/1): Program Reference Manual." SH19-5Ø6Ø-Ø, IBM Corporation, Data Processing Division, White Plains, New York (1972).

[JENS74] Jensen, K. and N. Wirth. PASCAL User Manual and Report. Springer-Verlag, New York (1974).

[KIVI74] Kiviat, P. J., R. Villanueva, H. M. Markowitz. SIMSCRIPT II.5 Programming Language. C. A. C. I., Inc, 12Ø11 San Vicente Boulevard, Los Angeles, California (1974).

[KLEI74] Kleijnen, J. P. C. Statistical Techniques in Simulation: Part I. Marcel Dekker, Inc. (1974).

[KNUTH71] Knuth, D. E. The Art of Computer Programming Vol. II : Seminumerical Algorithms. Addison-Wesley Publishing Co. (1971).

[MACD73] MacDougal, M. H., MacAlpine, J. S., "Computer System Simulation with Aspol," Proceedings Symposium on the Simulation of Computer Systems, June 19-2Ø, 1973, pp. 92-1Ø3.

[WILS79] Wilson, I. R. and Addyman, A. M. A Practical Introduction to PASCAL. Springer-Verlag, New

York, (1979).

[WEST79] West, D. H. D. "Updating the Mean and Variance Estimates: An Improved Method." Communications of the ACM. Vol. 22, No. 9 (September 1979), pp. 532-535.

Update to the SIMPAS USER MANUAL
December 19, 1980

R. M. Bryant

This note decribes changes to the SIMPAS preprocessor which have been made since the user manual was written. This note describes changes incorporated into version 4.2 of the preprocessor.

Changes are keyed to the corresponding section of the manual:

5.3. Entity Creation and Disposal

On the last line of page 21, "c_job" should be changed to "d_job."

6. Statistics Collection in SIMPAS

The standard statistics attributes have been extended to include the number of observations:

nobs number of observations

6.1. Regenerative Simulation in SIMPAS

The observation routines and the statistics type have been augmented to allow use of the regenerative simulation approach [2,4] in SIMPAS. To use these features requires two statistics variables. One is used to record statistics during each regeneration interval; the second statistic variable summarizes the statistics gathered over each regeneration interval.

To initialize the second statistics variable, one uses the following procedure call:

```
clear(ci_stat, interval);
```

The word interval signifies that this statistic will be used to generate confidence intervals using the regenerative simulation technique. At the end of each regeneration interval, one uses the routine c_observe to transfer the statistics gathered during the interval to the summary statistic variable:

```
c_observe(stat, ci_stat);
```

Here "stat" is the per-regeneration-interval statistics variable. The call to c_observe clears the "stat" variable so that it may be used to record statistics during the next regeneration interval. The max and min attributes of the "stat" variable give the max and min during the current regeneration interval only; the max and min of "ci_stat" give the true max and min.

To calculate confidence intervals, one uses the procedure call

```
c_calc(ci_stat, zalpha, mean, hwidth);
```

Zalpha is the critical point chosen from a normal distribution table. For a 100(1-alpha)% confidence interval zalpha should be chosen so that

$$Pr \{ Z \leq zalpha \} = 100(1-alpha/2)\%$$

where Z is a N(0,1) random variable. Mean is the midpoint of the confidence interval and hwidth is the confidence interval half-width. Thus the resulting confidence interval is mean $\pm$ hwidth. The classical confidence interval estimators are used [3].

B.1: Reserved Words

The identifier "interval" should be added to the reserved identifier list.

C.1: Distribution Format

SIMPAS can be distributed in any of the tape formats specified at the bottom of the MACC software order form. For most non-UNIVAC systems, this means that SIMPAS will be distributed as either an ASCII or EBCDIC card image tape. For VAX UNIX we will write a UNIX compatible tape.

C.2: PASCAL Compiler Requirements

The name "fatalerrors" should be "fatalerrs", the set "nonexerrs" should also be mentioned. These sets can be eliminated or reduced in size without too much trouble.

C.3: Character Set Differences

The character set translation process has been moved from procedure getchar to procedure handleletter. Both the case of translating upper to lower case (or vice-versa) and the translation of underbar to some other character is done there.

C.6: Random Number Generation

The current version of r_random is a portable version of LLRANDOM [1] written in PASCAL which should work properly on any machine with a word size of 32 bits or larger. We also can supply a special version of LLRANDOM suitable for use on a 16 bit machine. However, these routines can be made much more efficient by rewriting them in assembly language and the PASCAL version has been included only for portability.

C.7: Source Input and Output

Some PASCAL compilers only accept input lines up to a certain maximum length. Since SIMPAS generates lines of varying length depending on the input, it is possible for SIMPAS to generate a line that is too long to be compiled. If this happens, the SIMPAS source must be split across lines in such a way as to shorten the output PASCAL. Since comments and multiple blanks are removed during the expansion process, inserting or removing comments or extra blanks does not change the output line length.

Because SIMPAS writes its output directly rather than first placing the output in a PASCAL array of char, it is impossible to detect line overflow during pass one. Instead, this is checked for during pass two. The constant maxlinelength gives the maximum line length which should be generated. It is set to 80 in the distributed version. If your PASCAL compiler does not enforce a maximum line length limitation, set maxlinelength to 200 or so. There is an array of char of dimension maxlinelength in procedure moveline, so it is unwise to set maxlinelength to infinity.

If a line longer than maxlinelength is detected during the second pass, SIMPAS prints the line and gives an appropriate error message. Note that during the second pass, the preprocessor is reading the expanded PASCAL and thus the error printed will refer to a PASCAL statement which the preprocessor created. Use the SIMPAS source line number (which is encoded in the output PASCAL as a comment at the start of the line) to determine which SIMPAS source line needs to be shortened.

REFERENCES

[1] Fishman, G., Principles of Discrete Event Simulation,, John Wiley and Sons, New York (1978).

[2] Franta, W. R., The Process View of Simulation, Elsevier North-Holland, Inc., New York (1977).

[3] Iglehart, D. L., "Simulating Stable Stochastic Systems, V: Comparison of Ratio Estimators," Naval Research Logistic Quarterly 22, 3, (September 1975).

[4] Lavenberg, S. S. and D. R. Slutz, "Introduction to Regenerative Simulation," IBM Journal of Research and Development, pp. 458-462 (September 1975).