SIMPAS 5.0 USER MANUAL

by

Raymond M. Bryant

Computer Sciences Technical Report #456

November 1981

SIMPAS 5.0 User Manual

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 embedded in PASCAL. Facilities for event declaration and scheduling, creation and deletion of temporary entities, declaration and maintenance of linked lists (queues) of entities, and automatic collection 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 work was supported in part by the Wisconsin Alumni Research Foundation and through NSF grant MCS-800-3341.

Table of Contents

1. Introduction

SIMPAS is an event-oriented, discrete-system simulation language embedded in 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 [5,8] 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 [11], ASPOL [15], or SIMPL/I [1] to be detected at compilation time when the simulation is written in SIMPAS. (See [2] 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 [10] 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 [7], 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 "0" during preprocessing).

This manual describes the use of SIMPAS version 5.0 and supercedes the previous version of this manual which described SIMPAS version 2.0 [3]. The present version is a complete revision of the SIMPAS preprocessor that provides the following enhancements to SIMPAS:

(1) Version 5.0 uses a table-driven, error-correcting parser [6] to drive the expansion process instead of the ad hoc parser of version 2.0. The result is improved error detection and recovery as well more consistency between the SIMPAS extensions and PASCAL grammar.

(2) Version 5.0 catches most type clash errors associated with SIMPAS extension statements.

(3) Since Version 5.0 knows the types of most variables, code which had to be expanded inline for Version 2.0 can now be converted to a procedure call. This results in a smaller output PASCAL program when there are many insert and remove statements in the SIMPAS source.

(4) The code generated by schedule and reschedule statements has been significanly simplified.

(5) Several utility routine calls have been removed and replaced by new SIMPAS extension statements. For example, the clear procedure to initialize a statistic has been replaced by a clear statement.

(6) Automatic statistics collection features have been added to SIMPAS.

A brief introduction to this version of SIMPAS is also available [4].

This manual is divided into eight major sections and five 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 members and queue declarations. Section 6 discusses the statistics collection features of SIMPAS. Section 7 describes the use of SIMPAS and 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. Appendix E describes the major differences between SIMPAS versions 2.0 and 5.0.

1.1. Overview of the SIMPAS Preprocessor

SIMPAS consists of a large PASCAL program (about 7,200 lines) and a small file of run-time routines written in PASCAL (the "library" file). Other external files contain parse and error correction tables; these files will normally be of no concern to the SIMPAS user.

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 occurred.

On most systems where it is installed, SIMPAS can be called by a single command procedure or macro so that the preprocessing and compilation steps are performed automatically for the user.

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>;

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 destroy 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;
```

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;
```

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 . . .
```

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.

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".

3.7. Event Trace

To simplify simulation debugging, SIMPAS provides routines to dump the event set or to print the contents of an event notice, and provides mechanisms to selectively trace event occurrences. Calling the procedure dmp_evset (no arguments) will print the current contents of the event set. The procedure dmp_event(evptr) will print a description of the event described by the event notice pointer "evptr". In both cases all printable arguments (integer, real, or boolean) will be displayed as part of the dump.

There are three ways to control the event trace. One can set the global variable "trace_all" to true. This causes the scheduling, cancellation, and occurrence of each event in the simulation to be traced. Alternatively, one can cause the same printouts to occur for event "foo" by setting ev_trace[foo] to true. Finally, one can trace a particular event by setting the "trace" field of the event to true:

```
schedule foo named bar at time 30.0;
bar^.trace := true;
```

To do this requires that the event be scheduled with a schedule statement with a named clause so that a pointer to

the event notice can be obtained.

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 [12]

"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 [13]

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 [7].

```
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 [7] 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 [7] 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 [7].

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 [7] 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,mu1,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_gdsetup" or "i_gdsetup" depending on whether you want to generate real- or integer-valued random variables. The calling sequences for the setup routines are:

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

procedure i_gdsetup(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;
```

where "^" is the PASCAL "up-arrow" or pointer dereference operator. This trailing "^" is optional and is included merely to remind the user that the queue member declaration defines a pointer type.

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>;
```

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 : 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.

(3) Before a queue member can be placed in a queue, it is necessary to initialize the queue. SIMPAS provides an initialization statement to do this. Its format is:

```
initialize <queue> [,<queue>] ;
```

where each <queue> is a variable of type queue of <some_entity>. 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: *)
        initialize job_lists[5];

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

        for i:=1 to 5 do initialize job_lists[i];
```

The purpose of the initialization statment 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 <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 <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 <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 attribute gives the number of members in the queue. Size behaves as if it were "watched" integer, this means that statistics about size are automatically collected. For example, the time-averaged mean queue size is available as size.mean. See Section 6 for a description of watched types.

head- This attribute is of type <entity> and points to the head node of the linked list which represents the queue. This attribute is set when the queue is initialized. The first <entity> in the queue is head^.next; the last entity is head^.prev. If the queue is empty, both of these variables point to the queue head.

5.3. Entity Creation and Disposal

To create a new queue member one uses the statement:

```
create <entity>;
```

where <entity> is a variable of type queue member ^. Similarly, to dispose of an existing queue member one uses the statement:

```
destroy <entity>
```

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

```
type
    job=queue member ^ . . . end;
var
    jobptr : job;
        . . .
        . . .
        create jobptr;
```

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

```
destroy jobptr;
```

Of course, one can always use the PASCAL procedures "new" and "dispose" to do the same thing. However, by using create and destroy 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 of type "<entity>" and <queue> must be of type queue of <entity>. Attempts to insert or remove an entity of the wrong type in a queue will result in semantic errors at preprocessing 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 : job;
    begin
    (* create a job *)
    create 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
    begin
            (* start cpu *)
            schedule departure delay job_pointer^.cpu_time;
            insert job_pointer in cpu_queue;
    end]fR
    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: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 *)
        destroy 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 "<entity>"; <queue> must be a simple or qualified variable of type queue of <entity>. Attempts to use a variable of type <entity1> as a loop index in a forall loop where the queue is of type queue of <entity2> will result in preprocessor detected errors.

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>;
```

Within a forall loop, specific fields of the <entity> can 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 : job;
        cpu_queue : job_queue;

begin
        . . .

        avg_cpu:=0.0;
        forall job_pointer in cpu_queue do
                avg_cpu:=avg_cpu+job_pointer^.cpu_time;

        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
                with job_pointer^ do
                        avg_cpu=avg_cpu+cpu_time;
```

5.6. Forall Loops and the Event Set

To simplify scanning the event set, the event set is declared equivalently to the following:

```
type
    ptr_event = queue member ^
        (* standard event attributes *)
            . . .
    end;
    ev_queue : queue of ptr_event;
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 the size attribute is declared as an integer instead of as an a_integer. 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
        case etype(ev_ptr) of
        no_event : begin . . . end;
        main     : begin . . . end;
        . . .
        end; (* case *)
    . . .
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

Starting with version 5.0, SIMPAS provides automatic statistics collection features similar to those of SIMSCRIPT II.5. Statistics collection is enabled for a particular variable by declaring it to be a special type, which we will refer to as a "watched type". For example, to calculate time averaged statistics for an integer variable, one declares the integer as a "a_integer" (for accumulated integer). A variable of type a_integer can be used in expressions exactly as a normal integer variable can. However, whenever the variable is updated, statistics maintained about the variable are also updated. These statistics are available as predefined attributes of the watched variable:

mean	the mean value of the variable's observed values.
variance	the variance of the variable's observed values.
max	the maximum value this variable has had since it was last cleared or reset. (Not kept for boolean watched types).
min	the minimum value this variable has had since it was last cleared or reset. (Not kept for boolean watched types).
nobs	the number of observations made for this variable to date.

For example, if x is declared as an a_integer, then x.mean is its average, x.max is its maximum and so forth. Other attributes can be added by modifying the SIMPAS source library. The algorithm of [17] is used to stably update the mean and variance.

To simplify printing of statistics, with statements that involve watched variables are handled as a special case. Nominally, the statement "with <watched_var>" is illegal since <watched_var> is supposed to behave like a an integer, real, or boolean variable, respectively, and not like a record. Strict enforcement of this rule would always require the fully qualified name to be used to access the statistics information. This is cumbersome when printing statistics. Thus if x is an a_integer, then

```
with x do
    writeln(mean, min, max, variance);
```

is equivalent to

```
writeln(x.mean, x.min, x.max, x.variance);
```

A watched variable may be declared as either event-averaged (tallied) or time-averaged (accumulated). For a time-averaged variable, values observed are weighted by the length of time the value was held; event-averaged statistics give equal weight to all values. The six watched types are:

t_integer	to obtain the event average (tally) of an integer variable
a_integer	to obtain the time average (accumulate) of an integer variable
t_real	to obtain the event average (tally) of a real variable
a_real	to obtain the time average (accumulate) of a real variable
t_boolean	to obtain the event average (tally) of a boolean variable. (True = 1.0, False = 0.0).
t_boolean	to obtain the time average (accumulate) of a boolean variable. (True = 1.0, False = 0.0).

(The size attribute of a queue may be used as if it were an a_integer).

The clear statement is used to initialize a watched variable so that it can be used. The clear statement has the format:

clear <watched_variable> [,<watched-variable>]

In order to obtain meaningful statistics, a watched variable must be cleared before it is used.

The reset statement (format same as for clear) can be used to reset statistics collection during a run. It performs the same function as clear except that the current value of the watched variable is not set to zero. Thus reset merely clears the statistics associated with the watched variable. For example, it is a common practice to discard statistics during an initial "transient" portion of a simulation. The reset statement can be used to do this.

For a time-averaged, watched variable, one sometimes needs to flush out the last observation when the simulation ends. Otherwise the contribution of the last value the watched variable has will not be incorporated into the variable's statistics. To do this, one uses the statement:

flush <time_avg variable> [,<time_avg variable>]

A watched variable can be used anywhere a simple type can be used except as a for loop variable. Watched variables can be passed by value as simple variables to procedures and functions, or as var parameters that are themselves declared as watched variables, but may not be passed by reference as simple variables or passed by value as watched variables. Passing a watched type by reference as a

simple variable would allow the value of the variable to be changed without updating the associated statistics. Passing a watched type by value as a watched variable serves no useful purpose. The error in this case can be ignored, however, if the user wishes.

The following code is therefore legal:

```
var
    w : t_integer;

procedure foo(i : integer);
begin . . .  end;

procedure glarch(var i : t_integer);
begin . . .  end;

begin {main procedure}
    clear w;
    . . .
    foo(w);
    glarch(w);
    . . .
end.
```

On the other hand, declaring foo and glarch as follows would cause preprocessing errors to occur at the procedure call points:

```
procedure foo(var i : integer);
begin . . .  end;

procedure glarch(i : t_integer);
begin . . .  end;
```

6.1. Regenerative Simulation with SIMPAS

One may also declare a watched variable for confidence interval generation based on the regenerative simulation approach of [8,14] The classical confidence interval estimators are used [9]. The regenerative simulation approach requires that the simulation occasionally reach a state where the simulation "starts over". For example, in an M/G/1 queueing system simulation, every time the server becomes idle, the past behavior of the simulation is forgotten and statistics collected after this point are independent of statistics collected before this point. Such a state is called a regeneration state and the period of time between regeneration states is called a regeneration cycle.

To use these features one declares the watched variables exactly as above, but includes the section name "regen" on the program's include statement. The effect of this include section is to enable all watched variables in the program for use a confidence interval statistics. The

standard watched variable attributes are still accessible and the following new attributes are defined:

mid_point	the mid point of the confidence interval
half_width	the confidence interval half width. Thus the actual interval generated is: mid_point ± half_width.
z_alpha	selects the coverage probability of the confidence interval. Set to 1.96 (95%) by the clear statement. It may be changed by the user at essentially any time. (See below).

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

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

where Z is a N(0,1) random variable.

One initializes the confidence interval variable using the clear statement, exactly as for any watched variable. Mid_point and half_width are set at the end of each regeneration cycle through use of the regen statement:

regen <variable> [,<variable.] . . .

Mid_point and half_width are not usually meaningful until 20 or so regen's have been done. A negative half_width is used to indicate that not enought regen's have been done to make the result meaningful.

Z_alpha can be changed by the user and then the confidence interval can be recalculated using the recalc statement:

recalc <watched-variable> [,<watched-variable>]

Z_alpha can also be changed at any time during a regeneration cycle provided only that the regeneration cycle is not of length zero (no observations occur during the regeneration cycle). After the next regen statement is executed the confidence interval length will be adjusted to that of the new coverage probability.

6.2. Printing Statistics

The display statement is provided to simplify printing the contents of a watched variable. The format is:

display <watched_variable>

It is intended to be used as shown below:

write('Number of Jobs:'); display jobs_stat;

and produces output of the form shown below:

```
Number of Jobs:nobs=10 max=7.55e+00 mean=2.36e+00 var=5.15e+00
```

6.3. User Defined Statistics

The user can declare his own watched types if he wishes. The syntax to do so is:

```
<watched_type> = watched[<proc_name>,<suffix>] of <type>
```

This declaration may only appear in the global type part of the SIMPAS program. The effect of this declaration is that every time a variable of type <watched_type> is detected as the target of an assignment statement, SIMPAS converts the assignment statement to a call on <proc_name>. The procedure <proc_name> must be declared as follows:

```
<proc_name>(flag : t_asg_flag; var LHS : <t_type>; RHS : <t_type>;
                   line : integer; progunit : mod_name);
```

where

flag — indicates to the procedure why it was called. flag=assign indicates that a variable of type <watched_type> has been assigned to; flag=clear indicates that a variable of type <watched_type> has been cleared. In general, the statement <id> <var> with <var> a <watched_type> is translated to a procedure call with flag=<id>. This is how the clear, regen, recalc etc. statements are implemented.

LHS — is the left-hand-side of the assignment statement. If flag=assign then procedure <proc_name> MUST assign the value LHS to the appropriate part of parameter RHS before returning.

RHS — is the right-hand-side of the assignment statement.

<t_type> — is the true type of the watched variable. If the <type> in the <watched_type> declaration is a simple type, then the true type is this type. In this case the <suffix> must be empty. Normally, <type> will be a record, and suffix will be one of the field names (including the ".") of this record. The true type of the <watched_type> in this case is the type of that field in the record.

line , — is the current SIMPAS source line.

progunit — is the current program/module name (only useful in the case of separate compilation). Line and progunit are provided primarily for printing error messages.

Thus, the assignment statement

```
LHS := RHS;
```

where LHS is of type <watched_type> is converted to the procedure call:

<proc_name>(assign, LHS, RHS, line, progunit)

The clear statement:

clear <var>

where <var> is a variable of type <watched_type>, is converted to the procedure call:

<proc_name>(clear, <var>, <var>, line, progunit)

Everywhere a variable of type <watched_type> appears in a place that requires a value, <suffix> will be appended to the variable name. (with are an exception to this rule). Normally, <type> will be a record and <suffix> should be a field name in the record; the type of <suffix> will be the true type of any variable declared as an instance of <watched_type>. Suffix is textually added to the name of the variable of type <watched_type>. If <type> in the <watched_type> declaration is indeed a record, then <suffix> must include the initial ".". The only requirement on <suffix> is that for any variable of type <watched_type>, the result of appending <suffix> to the variable name must be a legal PASCAL construct. The SIMPAS preprocessor does not check the legality of this construction in any way.

The type of .<suffix> can itself be a record. The only complication here is that if x is an instance of such a watched type, then the statement

with x do

is NOT converted to

with x.<suffix> do

If this is indeed what the user wants, the following form of the with statement must be used:

with x, <suffix> do

or

with x do with <suffix> do

This restriction is enforced to allow the user to use a with statement to gain access to the statistics data in a watched variable.

7. Use of the Preprocessor

In general the preprocessor is given the SIMPAS program as input and produces a PASCAL program as output. The PASCAL program is then compiled and executed in the normal fashion. Since SIMPAS is designed to be portable from machine to machine and system to system, it is difficult to

describe in more detail how it is to be used at your particular installation. For SIMPAS versions that run on the VAX UNIX system and the UNIVAC 1100 system, more detailed information is available in the machine specific manuals for those systems. This information is normally supplied as part of the SIMPAS distribution package.

7.1. Preprocessor Control Options

Several actions of the preprocessor can be controlled by placing option flags in the SIMPAS source. These options are specified by letters that are enclosed in a pair of matching $ signs. Several letters can appear between the $ signs and their case is not significant. The options useful to the user are:

L	Turn on SIMPAS source level listing.
N	Turn off SIMPAS source level listing.
_	Turn off translation of _ to 0.
C	Turn off translation of upper to lower case.

Other option letters that are defined are S, T, D, and A. These options turn on or off SIMPAS debugging printouts.

7.2. SIMPAS and PASCAL Line Numbers

When compiling the output PASCAL, one will get errors (as it sometimes happens!) in terms of the output PASCAL line numbers. These numbers can be related back to the SIMPAS input line numbers by looking for a lines that start with a comment of the form (*nnnn*). This output PASCAL line number was generated from SIMPAS input line number nnnn. If the line where the error occurred does not begin with (*nnnn*), then examine the previous few lines looking for such a comment. If no such line is found, the section of code in error probably was brought in from the library file (these lines are not numbered). Since in general this code should not contain errors, it is likely that some error you made previous to that point has caused these error messages to be generated.

7.3. 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 is declared as a tallied integer          }
   tsys_stat            : t_real;


{  sys_busy  records the amount of time the system is busy  }
   sys_busy             : a_boolean;

{  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 : job;

    begin {  arrival  }
       arrivals:= arrivals + 1;
       create 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  }
               sys_busy := true;
               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 : job;

      begin {  departure  }

         departures:= departures + 1;

         remove the first departing_job from waiting_queue;

         {  record this job's time in system  }
         tsys_stat := time - departing_job^.arrival_time;

         {  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  }
         destroy departing_job;

         if waiting_queue.empty then
                {  record end of system busy period  }
                sys_busy := false
         else {  schedule next departure  }
                schedule departure delay
                         expo(service_rate, service_stream);

      end; {  departure  }


   begin {  main procedure  }

      initialize waiting_queue;
```

```
   { initialize statistics }
   clear tsys_stat, sys_busy;

   { 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 }
   sys_busy := waiting_queue.empty;

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

end.
```

7.4. 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

Mark Abbott, John Bugarin, and Bryan Rosenburg have worked on various phases of the SIMPAS implementation and without their assistance the project would never have been completed. This project was supported in part by the Wisconsin Alumni Research Foundation and by NSF Grant MCS-

800-3341. I would also like to thank Dr. Raphael Finkel for his assistance in debugging the early versions of SIMPAS and his editorial assistance in writing this manual. Finally, 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.

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 size is declared as an integer instead of as an a_integer. 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;
        trace     : boolean;
        schedtime : real;
        schedline : integer;
        schedunit : mod_name;
        id        : integer;
        case eventtype: t_ev_1 of
              no_event    : ();
              main        : (a_main : t_main);
              . . .
              <event>     : (a_<event> : t_<event>);
              . . .
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 scheduled. "trace" is the tracing flag for this event notice. 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. "schedtime", "schedline" and "schedunit" contain the time, line, and program unit where the event notice was last scheduled/rescheduled and are used by the event set tracing routines. "id" is a unique integer assigned to this event notice for use in tracing.

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 );
    e_insert( g_notice, nil, expo(lambda,3),
              e_delay, line, progunit);
    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 routine e_insert takes g_notice and inserts it at the correct place in the event set. "line" and "progunit" are the current SIMPAS source line and program unit names; these names are used for error reporting.

The second through fourth arguments to e_insert control the type of insertion to be made. For example, the second argument is nil unless the schedule (or reschedule) statement uses a before or after phrase. In this case the second argument gives a pointer to the event notice that is to be inserted before or after, respectively. The third argument gives the time expression of the schedule/reschedule statement. The fourth argument is one of: e_at, e_delay, e_now, e_before, or e_after depending on the form of the schedule/reschedule statement.

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 : 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;
    e_insert(g_notice, nil, run_time, e_delay, line, progunit);
    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:

```
<entity> = ^r_<entity>;
r_<entity> = record
    next, prev, qhead: <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.

The queue head type declaration

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

is expanded to

```
<queue-type> = record
                   head : <entity>;
                   size : a_integer;
                   empty : boolean;
               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 member type declared in the program, the preprocessor creates procedures c_<entity> and d_<entity> to create and destroy entities of that type. These procedures are called by the create and destroy statements respectively. For every queue type declared in the program, the preprocessor creates and inserts an

initialization procedure named i_<queue>. For the queue type named <queue> the initialization routine is named i_<queue>. The initialization routine is called by the initialize statement. The initialization routine sets the size of the queue is to zero and sets empty to true. The statistics portion of a integer is cleared. 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 converted to an equivalent insert after statement, and then the insert after statement is translated to a call on the procedure p_<queue_type> ("p" stands for "put"):

```
procedure p_<queue_type> (var <queue> : <queue_type>;
                          m1, m2 : <entity>;
                          line : integer; progunit : mod_name);
```

Here <queue_type> is the type of <queue>. This procedure inserts m1 after m2 in <queue>.

For example,

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

is converted to

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

and then the later statement becomes:

```
p_<queue_type>(<queue>, <entity>, <queue>.head, line, progunit)
```

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

is expanded to

```
p_<queue_type>(<queue>, <entity-1>, <entity-2>, line, progunit)
```

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.

Attempts to insert <entity> in a <queue> that is not a queue of <entity> will be flagged by the preprocessor as an error.

A.6: Remove

The statement

remove <entity> from <queue>

is translated to a call on the procedure r_<queue_type> ("r" stands for "remove"):

r_<queue_type> (<queue>, <entity>, line, progunit)

The procedure r_<queue_type> is declared as:

```
procedure r_<queue_type> (var <queue> : <queue_type>;
                          mem : <entity>;
                          line : integer; progunit : mod_name);
```

Statements of the form

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

are implemented as

```
r_<queue_type>(<queue>, <queue>.head^.next, line, progunit);
r_<queue_type>(<queue>, <queue>.head^.prev, line, progunit);
```

respectively. 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 six parts. 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:

$dependencies	tells which sections depend on other sections so that all sections that are needed are brought in
$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.

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.

The include list consists of the sections explicitly included by the user, plus all of the sections that those sections depend on, plus certain sections included in response to declarations in the SIMPAS source. For example, the section "events" is included whenever the user declares an event.

A section name normally appears in several parts of the library file. For example, the section name "events" appears in both the type part and the procedure part of 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.

A:9 Watched Variable Implementation

An a_integer is implemented as if it were declared as

```
a_integer = watched[ai_observe,.val] of record
                                          val : integer;
                                          mean: real;
                                          variance: real;
                                          min,max : integer;
                                          nobs: integer;
                                          . . .
                                          end;
```

(See Section 6.2). Thus every time a variable of type a_integer is assigned to, the procedure ai_observe is called. This procedure updates the other fields of the record so that the statistics stored there remain correct. Before returning, ai_observe assigns the RHS parameter to LHS.val. Similarly, if x is of type a_integer, then the assignment statement

```
y := x;
```

is expanded to

```
y := x.val;
```

The other watched types are implemented in a similar fashion. The names of the routines are:

```
ti_observe      for t_integer
ai_observe      for a_integer
tr_observe      for t_real
ar_observe      for a_real
tb_observe      for t_boolean
```

ab_observe for a_boolean

Including the name "regen" on the include statement causes a different set of definitions for the watched types and the procedures given above to be loaded. This also enables use of the regen, recalc statements.

Appendix B

Reserved Words and Implementation Restrictions

B.1: Reserved Words

The following words are reserved in SIMPAS. These include all of the reserved words of PASCAL plus those of the SIMPAS extensions:

after	forward	program
and	from	queue
array	function	record
at	goto	remove
before	if	repeat
begin	in	reschedule
cancel	include	reverse
case	initialize	schedule
const	insert	set
create	label	simulation
delay	last	start
delete	member	system
destroy	mod	the
div	named	then
do	nil	to
downto	not	type
else	now	until
end	observe	var
event	of	watched
file	or	while
first	otherwise	with
for	packed	
forall	procedure	
flush		

In addition the identifiers: system, external, and module are reserved words in the VAX UNIX implementation.

The following identifiers are reserved for use by the preprocessor:

assign	e_insert	regen
a_boolean	e_mode	recalc
a_integer	e_now	reset
a_real	error_p	s_control
assign	error_x	scheduled
c_notice	ev_set	seed_v
clear	ev_trace	t_boolean
current	ev_name	t_ev_l
d_notice	event_notice	t_integer
display	flush	t_real
dmp_event	g_notice	t_asg_flag
dmp_evset	i_ev	t_main

dmp_evnotice	main	time
do_event	n_seed	trace_all
e_after	no_event	true_false
e_before	no_stat	u_random
e_delay	r_random	

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
r_<entity>	record type for entity

For a queue of type <queue_type> the following names are used:

i_<queue_type>	queue initialization routine
p_<queue_type>	put in queue
r_<queue_type>	remove from queue

B.2: Implementation Restrictions

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

If the host PASCAL compiler supports both upper and lower case, the entire program is translated to lower case to simplify identifier comparisons. The boolean "tranupper" enables this translation.

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. Since most PASCAL compilers distinguish identifiers in more than 8 characters, this is not as bad a problem as it may appear.

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.

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.

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 5.1.1.

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

Installing SIMPAS

This Appendix gives those details of the implementation which are machine specific and which must be modified when installing SIMPAS on a new machine. This Appendix assumes that the target machine is not one of the machines for which "standard" versions of SIMPAS are available from the Program Librarian, Madison Academic Computing Center, University of Wisconsin-Madison, 1210 W. Dayton Street, Madison, Wisconsin, 53706. Standard versions presently exist for DEC VAX's running the Berkeley version of UNIX and Univac 1100 systems using UW-PASCAL.

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 80 characters and contains one card image. The blocks contain no control information and no special characters are used to compress out blanks.

There are five files on the tape; the files are separated by hardware end-of-file marks. The files contain a total of about 10,000 card images. The files contain the following:

file	Contents
1	simpas preprocessor
2	fmq parser table
3	fmq error correction table
4	simpas library file
5	test program
6	simpas bnf

Files 2, 3, and 4 are auxillary files required by SIMPAS. Files 2 and 3 are generated by the FMQ parser generator.

FMQ [6,16] is the error correcting parser used in version 5.0 of SIMPAS. The FMQ parser generator is separately licensed and is not distributed as part of SIMPAS. File 6 is useful only if you have purchased FMQ as well. For details about FMQ, contact Prof. C. N. Fischer, Department of Computer Science, 1210 W. Dayton Street, University of Wisconsin--Madison, Madison, Wisconsin 53706, Phone 608-262-1204.

To install the preprocessor copy the first five files on the tape onto disk. File 2 must be made to correspond to the internal PASCAL file "ptableout". File 3 must be made to correspond to the internal PASCAL file "etableout". File 4 must be made to correspond to the internal PASCAL file "libfile". Modify the source of the preprocessor so that

this correspondence is correct.

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.

Version 5.0 of the preprocessor, however, does require that "dispose" be a working procedure (and not a dummy) on your system. Version 2.0 will function on systems that do not support 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.) Now go into the SIMPAS source and find procedure init. (It's at line 6280 or thereabouts. The word "procedure" is in column 1.) Now go about 30 lines further. You are looking for the pair of statements:

```
tranunbar := false;
ub        := '_';
```

To eliminate '_' in the PASCAL output by SIMPAS, change these statements to

```
tranunbar := true;
ub        := '0';
```

This will cause all occurrences of '_' to be translated to '0'.

The standard version of SIMPAS assumes that the input will not contain tab characters. If it is possible for tabs to appear in the SIMPAS source, the preprocessor must be reconfigured to handle this case. To do this, look for "procedure scan;" (its at about line 4590) and then look for "scan - main body". Shortly thereafter is the statement

```
until (ch <> blank) or endfile;
```

Change this to

```
until (ch <> blank) or (ch <> tab) or endfile;
```

Then in the const part of the program, find the declaration of constant "tab" and define it as necessary. (Its presently defined as a "blank").

The preprocessor also translates all input to lower case. If you don't like this feature, find the statement "tranupper := true;" in procedure init and change it to "tranupper := false;".

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 SIMPAS files.

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 abort and error_x respectively. Procedure abort 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 a portable implementation of LLRANDOM [7] that will properly function on any system with a word size of 32 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 [13].

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 modify the read and readln statements in procedure readline as necessary.

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 OpenFiles and Pass2. 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>;

<watched_type> = watched[<proc_name>,<suffix>] of <type>;

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
                    <statement>

create <entity>  [,<entity>]
destroy <entity> [,<entity>]
initialize <queue> [,<queue>]

clear <watched_variable> [,<watched_variable>]
reset <watched_variable> [,<watched_variable>]
display <watched_variable> [,<watched_variable>]
regen <confidence_interval> [,<confidence_interval>]
recalc <confidence_interval> [,<confidence_interval>]
```

D.2: Identifier Glossary

a_<event>	A record of type t_<event> used in event_notice to hold the actual parameters for event <event>.
c_<entity>	This is the creation routine for queue members of type <entity>.
c_notice	An internal routine called to generate an event notice.
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.
do_event	A preprocessor generated routine which actually calls the event routines when the next event is determined. Called by s_control.
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.
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_ev An internal variable used during initialization.

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.

ptr_event A type name defined as ^event_notice.

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

r_random The basic, uniform (0,1) random number generator. May be machine dependent.

s_control the simulation control routine

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.

t_ev_1 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.

time The current simulation time. Time is a real variable.

<t>_observe A routine called when an assignment to a watched variable is made. The following table gives the correspondence between the type of the watched variable and what <t> is:

type	<t>
t_integer	ti
a_integer	ai
t_real	tr

a_real	ar
t_boolean	tb
a_boolean	ab

u_random — The basic uniform (0,1) random number generator. It knows about random number streams and antithetics, while r_random does not.

Appendix E

Differences Between version 2.0 and 5.0*

The syntax for declaring and using a queue member has changed. A queue member type is declared as:

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

The trailing ^ is optional. Specific queue members are declared directly in terms of the type <entity> instead of using the type ptr_<entity> as with SIMPAS 3.5.

Several new statements have been introduced to replace functions supported in SIMPAS 3.5 via procedure calls:

```
clear <watched-var>      replaces clear(<statistic>, <type>)
initialize queue           replaces i_<queue>(queue)
regen <watched-var>        replaces c_observe(<watched-var> . . .
```

Other new statements are flush, display, recalc. These statements relieve the user of the necessity of remembering the type of the target variable in order to clear a variable, intialiaze a queue and the like.

The r_observe, i_observe, b_observe procedure calls and the statistic type have been replaced by the concept of "watched variables". The result is that the observation routines are called automatically for the user whenever a watched variable is updated. See Section 6 for details.

The preprocessor automatically generates procedures to trace event execution and to dump the event set in a readable form.

The body of a forall loop can be any statement and is not restricted to being a begin-end pair as it was in SIMPAS 3.5.

The preprocessor now checks for type compatibility between the items inserted in a queue and the type of the queue itself.

---

*Version 2.0 corresponds to VAX UNIX version 3.5.

REFERENCES

[1] SIMPL/1 (Simulation Language Based on PL/1): Program Reference Manual SH19-5060-0. 1972 .

[2] Bryant, R. M., "SIMPAS -- A Simulation Language Based on PASCAL," Proceedings of the 1980 Winter Simulation Conference, pp. 25-40 (December 3-5, 1980).

[3] Bryant, R. M., "SIMPAS User Manual," Computer Sciences Department Technical Report #391, University of Wisconsin--Madison (June 1980).

[4] Bryant, R. M., "A Tutorial for PASCAL Users on Simulation Programming with SIMPAS," Computer Sciences Technical Report #454, University of Wisconsin--Madison (October 1981). Also Proceedings of the 1981 Winter Simulation Conference, Atlanta, Georgia, December 9-11, 1981.

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

[6] Fischer, C. N., D. R. Milton, and S. B. Quiring, "Efficient LL(1) error correction and recovery using only insertions," Acta Informatica 13, 2, pp. 141-154 (1980).

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

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

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

[10] Jensen, K. and N. Wirth, "Pascal: User Manual and Report," Lecture Notes in Computer Science 18, Springer-Verlag Berlin, New York, (1974).

[11] Kiviat, P. J., R. Villanueva, and H. M. Markowitz, SIMSCRIPT II.5 Programming Language, C. A. C. I., Inc., 12011 San Vicente Boulevard, Los Angeles, California (1974).

[12] Kleijnen, J. P. C., "," in Statistical Techniques in Simulation, Marcel Dekker, New York (1974/75). in two parts

[13] Knuth, D. E., The Art of Computer Programming Volume 2:

Seminumerical Algorithms, Addison-Wesley Publishing Company, Reading, Massachusetts (1971).

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

[15] MacDougal, M. H. and J. S. MacAlpine, "Computer System Simulation with Aspol," Proceedings Symposium on the Simulation of Computer Systems, pp. 92-103 (June 19-20, 1973).

[16] Mauney, J., "FMQ User's Guide," Computer Sciences Department Technical Report, University of Wisconsin--Madison (in preparation, 1981).

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