Proceedings of the 1980 Winter Simulation Conference
T.I. Ören, C.M. Shub, P.F. Roth (eds.)

SIMPAS: A SIMULATION LANGUAGE BASED ON PASCAL

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

SIMPAS is a portable, strongly-typed, event-oriented, discrete system simulation language based on PASCAL. It has most of the useful features of SIMSCRIPT II.5. However, because of the type checking inherent in SIMPAS, a SIMPAS program is easier to debug and maintain than the corresponding SIMSCRIPT II.5 program. This paper briefly describes SIMPAS and discusses the advantages of using SIMPAS instead of SIMSCRIPT II.5 for the rapid construction of reliable simulation programs.

1. INTRODUCTION

As pointed out in (Landwehr 80), in the past few years there has been significant progress in the analysis of simulation results, but techniques used in the efficient construction of simulation programs have received less attention, except that of course software engineering methods are supposed to apply to simulation programs as well. In the field of software engineering, it is now recognized that proper language design can greatly improve the reliability, clarity, and ease of implementation and maintenance of software written in that language (Gannon 75).

Other than structured control statements (e.g. if-then-else or while-do), one valuable feature of languages designed for the implementation of reliable software is "strong typing". Basically this means that every variable must be declared before use, every variable must be declared to be of a specific type, and only expressions of that type can be used to assign new values to the variable.

In spite of evidence that shows the usefulness of strongly-typed languages (Gannon 77), the most popular procedure-oriented simulation languages do not have this feature. At present, the best known simulation language with strong typing is SIMULA (Dahl 69, see also Franta 77) and it exists on only a few machines since a SIMULA compiler is relatively complex and expensive to implement. (We would say that SIMPL/1 (IBM 72) does not have strong typing since the use of pointer variables in PL/1 can lead to serious type violations.) On the other hand, PASCAL (Jensen 74) (while not a simulation language per se) does have the necessary features for construction of reliable software (Wirth 75) and is available on many different machines, since one of PASCAL's design criterion was to allow the compiler to be easily implemented.

Our purpose in developing SIMPAS was to make a strongly-typed simulation language readily available in the simulation community. SIMPAS is implemented as a preprocessor which 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.

We have initially chosen to implement SIMPAS as a preprocessor rather than as a compiler with an easily modifiable code generator in order to simplify implementation and gain use of the language in the simulation community. It is the author's observation that rather than modify a code generator, a person interested in simulation will probably implement his simulation in any convenient high level language. This is because his interest is usually in the simulation results and not in the simulation language itself. From his viewpoint, the effort required to fix the code generator is regarded as better spent in implementing the simulation using a less convenient language rather that the other way around. Thus to be widely used, SIMPAS must be almost trivially transportable between machines. A preprocessor implementation seems the only reasonable way to do this.

We also acknowledge a reluctance of users to learn a new language just for the purpose of writing a simulation. This also leads to a tendency to write simulations only in general purpose languages (most commonly FORTRAN). We hope that this paper will convince the reader that the effort spent in learning SIMPAS (and PASCAL) will be adequately compensated by a reduction in cost of producing the simulation program.

In the next sections of this paper, we describe the SIMPAS extensions to the programming language PASCAL and discuss a sample SIMPAS program. We then compare SIMPAS to SIMSCRIPT II.5, and describe some typical programming errors which the SIMSCRIPT II.5 compiler cannot detect, but which can be detected in the process of expanding and compiling a SIMPAS program.

2. SIMPAS LANGUAGE DESCRIPTION

In this section, we briefly discuss the simulation extensions to PASCAL which have been incorporated into SIMPAS. These extensions are straightforward and very similar to those of SIMSCRIPT II.5 (Kiviat 73), so we will not describe them in complete detail. We merely intend to give the reader the flavor of the language. For a more precise description of the language extensions, see the SIMPAS user manual (Bryant 80).

We assume that the reader is familiar with some modern, block structured programming language such as PL/I, ALGOL, or SIMULA. Where necessary, we will describe those features of PASCAL which are not found in these other languages. For further information about PASCAL see (Jensen 74).

We will 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, Fishman 78).

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

2.1. Program Structure

A SIMPAS program has essentially the same structure as a PASCAL program. That is, it can be divided into seven major parts: the global label declaration part (which will not concern us here), the global constant declaration part, the global type declaration part, the global variable declaration part, the procedure and event declarations, and the main procedure.

Of these sections, the only one which is peculiar to PASCAL is the global type part. We recall that in PASCAL, there is a clear distinction between a variable and its type. Every variable must be declared as a specific type; this type can be one of the base types (integer, real, or boolean) or a defined type built up out of the base types and record or array declarations. One refers to a defined type by giving it a name in the type declaration part of the current program (or procedure). Thus one can create a SIMSCRIPT II.5-like entity with the following declarations:

```
type
    job = record
            arrival_time        : real;
            service_requirement : real;
            memory_size         : integer;
          end;

var
    some_job    : job;
    another_job : job;
```

Note that in this example, "some_job" and "another_job" refer to distinct instances of jobs. One refers to the attributes of such an entity (this is the SIMSCRIPT term, in PASCAL they are called "fields") using the dot notation:

```
some_job.arrival_time
```

The underbar character "_" is not part of the standard PASCAL character set, although it is used in many implementations to improve readability of variable names. To ensure transportability of SIMPAS programs, the SIMPAS preprocessor can be configured to convert "_" to some other legal character.

2.2. The Include Statement

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

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

Each section name specifies a portion of the library to be included.

2.3. Event Declaration

An event declaration has exactly the same form as a PASCAL procedure declaration, except that the reserved word event replaces the reserved word procedure. Thus the general form is:

```
event <name> [ ( <formal argument list> ) ];
    [label declaration part]
    [constant declaration part]
    [type declaration part]
    [variable declaration part]
    [procedure declaration part]
begin
    [statements of the event routine]
end;
```

For example:

```
event done(service_time : real);
begin
    . . .
end;
```

One may intermix procedure and event declarations, except that an event cannot be declared local to a procedure or an event. This is required in order to make the event accessible from the event-set scanning routine.

When the event routine is called, the formal arguments are set to the actual arguments specified on the schedule statement for the event. Thus a value for "service_time" must be specified when event done is scheduled, or the preprocessor will flag this as an error. The only restriction on the arguments is that they must be passed by value. The reason for this is that the event routine is actually called with a copy of the

argument values saved in the event notice, and thus changing an argument passed by reference would result in the contents of the event notice being changed, which is definitely not what the user expects. Thus we have made it illegal to declare a var argument in an event.

2.4. Start Simulation

To start the simulation (i. e. begin execution of events), one uses the statement:

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

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

Statements after the start simulation statement will be executed when the event set becomes empty or when an event notice for event main reaches the head of the event set. The variable <status> can be inspected to determine which of these two cases caused the simulation to terminate.

Event main is predeclared as if it looked like:

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

As a matter of fact, there is no event routine associated with event main; as discussed in the last paragraph an event notice for event main merely informs the event scanning routine to terminate the simulation and resume executing statements following the most recently executed start simulation statement. In this case, the <status> variable in the start simulation statement is set to the argument of event main. By setting this argument to a non-zero number, the user can return a flag to indicate why the simulation terminated.

2.5. Event Scheduling Statements

Event notices are created and inserted into the event set by the schedule statement:

```
schedule <event-name> [ (actual-argument-list) ]
    [ named <event-pointer> ]
    { now | {at | delay} <time-expression> |
            {before | after} <event-pointer> }
```

An event must be declared before it is scheduled. To allow this in general, an event declaration can be forwarded exactly like a PASCAL procedure.

For example, to schedule an instance of event done to occur at time 10.0 with service_time equal to 30.0 one would say:

```
schedule done(30.0) at 10.0;
```

If one wished done to occur immediately one could say:

```
schedule done(30.0) now;
```

The statement

```
schedule done(30.0) delay 5.0;
```

is equivalent to

```
schedule done(30.0) at time + 5.0;
```

and allows the user to schedule an event to occur after an interval of time has elapsed rather than at a specific instant of time.

The named clause is used to record a pointer to the event notice for this event. The <eventpointer> gives the name of a variable of type "pointer to event_notice" (in PASCAL notation, "^event_notice"). A pointer to the event notice is stored in this variable. One can then use this name in a before or after clause to indicate that some other event is to be scheduled at the same simulated time as, but immediately before or after a named event.

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

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

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

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

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

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

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

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

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

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

2.6. Queue Handling Statements

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

2.6.1. Entity and Queue Declarations One declares an entity type in the global type declaration part of the program; the declaration looks like a special record declaration:

```
type
    <entity-type> = queue member
            [ <attribute-name> : <attribute-type> ; ]
            . . .
            end;
```

The preprocessor inserts additional field names to contain links to other members of the queue and to record which queue (if any) this entity is a member of.

After the type declaration, one declares a particular instance of an entity by declaring a variable to be a pointer at that type:

```
var
    <entity> : ^<entity-type>;
```

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

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

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

<entity-type> must be previously declared. In a var part of the program (or procedure) one can declare a particular queue with a declaration like:

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

For each entity type, the preprocessor builds a creation routine (c_<entity-type>) and disposal routine (d_<entity-type>) for entities of this type. Using these routines to create and destroy entities instead of the standard PASCAL storage allocation routines "new" and "dispose" ensures that the queue membership attributes of an entity are properly initialized when the entity is created and that the SIMPAS program will execute using PASCAL compilers which do not implement "dispose".

For each queue type, the preprocessor also builds a queue initialization routine (i_<queue-type>) which must be called for each variable of type <queue-type>. Attempting to place an entity in an uninitialized queue will result in a run time error.

As an example, the following declarations can be used to declare a job entity and queue named waiting_queue which can hold jobs:

```
type
   job        = queue member
                  arrival_time : real;
                end;

   job_queue = queue of job;

var
   waiting_queue          : job_queue;
   a_job                  : ^job;
```

2.6.2. Queue and Entity Standard Attributes The standard queue member attributes defined by the preprocessor are:

next- This attribute is of type ^<entity-type> 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-type> 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 current queue member is in a queue.

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

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

The standard queue attributes are:

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

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

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

stat- This attribute is of type "statistic" and is used to collect statistics about queue occupancy. See Section 2.8 for details about type "statistic".

2.6.3. Queue Manipulation Statements To insert or remove entities from a queue, SIMPAS provides the statements:

```
insert <entity-ptr> [ {first | last} ] in <queue>

insert <entity-ptr-1> {before | after}
                        <entity-ptr-2> in <queue>

remove <entity-ptr> from <queue>

remove [the] { first | last } <entity-ptr> from <queue>
```

In all cases, the variable <entity-ptr> must be of type ^<entity>.

Attempts to insert entities in queues of the wrong type are detected at compile time. Other errors, such as attempting to insert an entity into a queue when it is already in a queue, attempting to remove an entity from a queue it is not in, and so forth are detected at run time.

To simplify loops over queue membership, SIMPAS provides the loop statements:

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

Here S is either a simple or compound statement. If the queue is empty, then S is not executed.

For example, to average all of the waiting times of the jobs in the waiting_queue we declared above, one could use the following code:

```
avg_wait := 0.0;

forall a_job in cpu_queue do
        avg_wait := avg_wait +
                (time - a_job^.arrival_time);

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

2.7. Pseudo-random Number Generation

A standard collection of pseudo-random number generators are provided in the SIMPAS library and can be incorporated in the user program through the include statement. These routines all depend on a single uniform random number generator. A generator suitable for most machines with a word size of 29 bits or larger is included in the standard version of SIMPAS. Given the existence of the basic uniform random number generator, random number generators for the following distributions are available:

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

The generation algorithms were taken from (Fishman 78).

2.8. Statistics Collection

At present, SIMPAS does not provide the automatic statistics collection features of SIMSCRIPT II.5. However, SIMPAS does provide a statistic collection type and a set of observation routines. Thus to allocate a variable for statistic collection, one declares a variable of type "statistic." For example:

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

    tsys      : real;
```

```
tsys_stat : statistic;
```

A statistic can be either time or event-averaged. This selection is made when the statistics variable is initialized with the "clear" routine:

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

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

To observe a value of a variable, one calls an observation routine of the appropriate type. For example, to observe the values of nsys and tsys, one would say:

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

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

Similarly, one calls b_observe to record the value of a boolean variable.

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

```
nsys_stat.mean        is the time average mean of nsys
tsys_stat.variance    is the event averaged variance of tsys

nsys_stat.max         is the maximum of nsys
tsys_stat.min         is the minimum of tsys
```

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

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

3. A SAMPLE SIMPAS PROGRAM

In this section we combine the examples from the previous sections to illustrate their use in a simple M/M/1 queueing system simulation. Our discussion of this program is included as comments in the program text:

```
(* program heading indicates that it reads no input, only creates output *)
program example_simulation(output);

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

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

type
   job       = queue member
                 arrival_time : real;
               end;

   job_queue = queue of job;

var

(* departures counts the number of departures *)
```

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

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

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

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

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

(* fetch exponential random number generator routine from library *)
include expo;

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

event arrival;

var
    arriving_job : ptr_job;

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

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

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

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

    end; (* arrival *)


event departure;

var
    departing_job : ptr_job;

    begin (* departure *)

       departures:= departures + 1;

       remove the first departing_job from waiting_queue;

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

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

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

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

    end; (* departure *)


begin (* main procedure *)

    (* initialize waiting_queue *)
    i_job_queue(waiting_queue);

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

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

    (* schedule first arrival *)
    schedule arrival now;

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

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

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

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

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

end.
```

The execution output produced by the above program is:

```
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
```

The expanded PASCAL generated by SIMPAS for this program is about 500 lines long and is too lengthy to include here. Indeed the intent is that the user never examine the generated PASCAL. The SIMPAS preprocessor has been designed to generate PASCAL in the most direct way; unfortunately this means that the output from the SIMPAS preprocessor is not particularly readable. PASCAL compilation or run-time errors can be easily traced back from the generated PASCAL to the SIMPAS source through the SIMPAS source line number which is encoded on each line of the output. Run time errors detected by the SIMPAS run time routines refer directly to the SIMPAS source line number.

4. A COMPARISON OF SIMPAS AND SIMSCRIPT II.5

Since the event scheduling and queue handling statements of SIMPAS are much like those of SIMSCRIPT II.5, one may ask what advantage SIMPAS has over SIMSCRIPT II.5. In this section we discuss some common programming errors in SIMSCRIPT II.5 which will be caught at preprocessing or compilation time in SIMPAS.

In SIMSCRIPT II.5, any variable can represent a pointer to an entity. Thus the following declarations and code are perfectly legal (We have included line numbers here as a convenience for discussion of the programs.):

```
 1    PREAMBLE
 2
 3        THE SYSTEM OWNS THE CPU.QUEUE
 4              AND THE CUSTOMER.QUEUE
 5
 6        TEMPORARY ENTITIES...
 7        EVERY JOB HAS A CPU.TIME, A JOB.ID, A SIZE,
 8              AND BELONGS TO THE CPU.QUEUE
 9        EVERY CUSTOMER HAS A NUMBER.OF.CARDS
10              AND BELONGS TO THE CUSTOMER.QUEUE
11
12        DEFINE CPU.TIME AS A REAL VARIABLE
13        DEFINE JOB.ID, SIZE, NUMBER.OF.CARDS
14     AS INTEGER VARIABLES
15
16    END
17
18    MAIN
19
20        DEFINE I, J AS INTEGER VARIABLES
21        . . .
22
23        CREATE A JOB CALLED I
24        LET CPU.TIME(I) = 30.0
25        . . .
26
27        CREATE A CUSTOMER CALLED I
28        LET NUMBER.OF.CARDS(I) = 100
29        . . .
30    END
```

We note that I is used both as a pointer to a JOB (line 23) and a CUSTOMER (line 27).

So far we have not run into any problems. Let us suppose however, that we forget and do the following:

```
 1  MAIN
 2      DEFINE I, J AS INTEGER VARIABLES
 3      . . .
 4
 5      CREATE A CUSTOMER CALLED I
 6      LET NUMBER.OF.CARDS(I) = 20
 7      FILE I LAST IN CUSTOMER.QUEUE
 8      . . .
 9      '' I STILL POINTS TO A CUSTOMER, AND NOT A JOB!
10      LET CPU.TIME(I) = 34.0
11      LET JOB.ID(I)   = 25
12      . . .
13
14      FOR EVERY J IN CUSTOMER.QUEUE DO
15      PRINT 1 LINE WITH NUMBER.OF.CARDS(J) THUS...
16  NUMBER.OF.CARDS = ***
17      LOOP
18
19      LET J = SIZE(I)
20      . . .
21  END
```

The errors here are that at lines 10 and 11 we have set the CPU.TIME and JOB.ID attributes of the entity pointed to by I; unfortunately, I points at an entity of type

CUSTOMER which does not have any such attributes.

The exact consequences of these errors depend on the SIMSCRIPT II.5 implementation. In the C. A. C. I. version for UNIVAC 1100 series systems (C. A. C. I.) the successor link for CUSTOMER.QUEUE in the entity pointed to by I is destroyed. This then causes the loop at line 14 to fail. Even if the FOR EVERY loop did not fail, the effect of line 19 is unpredictable, because I points to a CUSTOMER, and SIZE is not an attribute of CUSTOMER. The problems in this case are of course trivial to find and correct; but this might not be true if the simulation were several hundred lines long.

Note that we are not criticizing the C. A. C. I. implementation. In fact, without extensive error checking at run-time, it appears that no SIMSCRIPT II.5 compiler could detect errors of this type. The point is that since this is a relatively common programming error, it should be easy to find and correct. If possible, the compiler should do this type of error detection for you.

In SIMPAS, on the other hand, a variable can only point at entities of the appropriate type, and references to attributes not defined for that entity result in compilation time errors. The SIMPAS program corresponding to the above would look like:

```
 1  program test(output);
 2
 3  type
 4       job = queue member
 5           cpu_time : real;
 6           job_id   : integer;
 7           size     : integer;
 8           end;
 9
10       customer = queue member
11           number_of_cards : integer;
12           end;
13
14       job_q      = queue of job;
15       customer_q = queue of customer;
16
17  var
18       job_queue      : job_q;
19       customer_queue : customer_q;
20
21       a_job          : ^job;
22       a_customer     : ^customer;
23
24  begin (* main procedure *)
25       (* initialize the queues *)
26       i_job_q(job_queue);
27       i_customer_q(customer_queue);
28
29       (* create a job and customer *)
30       c_job(a_job);
31       c_customer(a_customer);
32
33       (* the following cause compilation errors *)
34       a_customer^.cpu_time := 10;
35       a_customer^.job_id   := 4;
36       j := a_customer^.size;
37       . . .
38  end.
```

Another type of programming error can occur when an implicitly declared variable has a different type or scope from the one the user expected. This often happens in SIMSCRIPT II.5 since any undeclared variable is implicitly declared as a local variable in the current procedure. It is given a type according to the "background mode" then in effect. Thus if the user has the background mode set to INTEGER and forgets to explicitly declare a particular variable as REAL, the SIMSCRIPT II.5 compiler will implicitly declare the variable as integer.

SIMSCRIPT II.5 also has the feature that in order to declare a global variable, one must include the variable declaration in the PREAMBLE. If the programmer omits or misspells the global declaration, all references to what the user thinks is a global variable are converted to references to an implicitly defined local variable. As an

example of these types of errors, consider the following program fragment:

```
 1    PREAMBLE
 2
 3        '' THE USER MEANT TO DECLARE A VARIABLE
 4        ''  NAMED "GLOBAL"  BUT MIS-SPELLED IT:
 5        DEFINE GOLBAL AS AN INTEGER VARIABLE
 6        . . .
 7
 8        '' SET THE BACKGROUND MODE
 9        NORMALLY, MODE IS INTEGER
10
11    END ''PREAMBLE
12
13    ROUTINE FROM;
14
15        ''STORE A VALUE IN GLOBAL
16        LET GLOBAL = VALUE
17        . . .
18        ''STORE A REAL NUMBER IN X
19        LET X = 34.33452
20        . . .
21    END ''FROM
22
23    ROUTINE TO;
24
25        . . .
26        ''GET THE VALUE FROM GLOBAL
27        LET VALUE = GLOBAL
28        . . .
29
30    END ''TO
31        . . .
```

Because GLOBAL was misspelled in the preamble, routines FROM and TO are given local variables named GLOBAL and no communication between the routines actually occurs. No compile time error occurs. Similarly, because X defaults to type integer, the fractional part of the real number stored in X in line 19 is lost.

The only way the user is going to find these errors is when he notices that values assigned to the variable GLOBAL in routine FROM are not being communicated to routine TO, or that the fractional part of a real number he created disappeared someplace. Often this is detected in a much different part of the program than where the error occurred, and thus can require extensive debugging to locate. This observation is supported by (Gannon 77) who notes that the kinds of errors which occur in languages without strong typing tend to remain in the program longer than those that occur in languages with strong typing.

The cross reference feature of SIMSCRIPT II.5 helps to find errors of the type given in the last example, but a much better solution is to enable the compiler to find such errors, and this is certainly the type of mundane and trivial task that we should be requiring the computer to do for us. This is especially true when the simulation becomes more than a few hundred lines long; in such cases there can be hundreds of variables and it appears unlikely that the programmer is going to check all of them in a cross reference listing to make sure their scope and type is what he thinks they should be.

In SIMPAS on the other hand, the following program results in compilation errors at lines 11, 13, and 22:

```
 1  program error(output);
 2
 3  var
 4      golbal : integer;
 5
 6  procedure from;
 7  var
 8      value : integer;
 9  begin
10      . . .
11      global := value;
12      . . .
```

```
13      value     := 34.33452;
14      . . .
15  end;
16
17  procedure to;
18  var
19      value : integer;
20  begin
21      . . .
22      value := global;
23  end;
24      . . .
25  end.
```

Undeclared-variable errors occur at lines 11 and 22 since "global" is not declared. A type-clash error occurs at line 13 because we are trying to store a real value in a variable of type integer. If we really wish to do this we can truncate the real number to an integer and store it in "value", but this requires an explicit request on our part to throw away the fractional part of the number. Thus no data will be lost by accident.

4.1. A Comparison of Code Efficiency

We have discussed the improved compile-time error detection of SIMPAS over that of SIMSCRIPT II.5. However, one common complaint about PASCAL is that "Most PASCAL compilers are quite inefficient, particularly when compared to a good optimizing FORTRAN compiler." We agree that this is the case. But a good optimizing FORTRAN compiler produces less efficient code than a good assembly language programmer. Except for certain compute bound applications, however, the increase in efficiency gained by moving critical code into assembly language is more than offset by the enormous increase in implementation and debugging cost. Thus there must be a balance between efficiency and implementation cost.

A similar situation occurs when one constructs a simulation program. If the simulation is to be run on a daily basis for a long period, or if if must satisfy execution time constraints, then it should probably be written in a very efficient language. However, the author feels that most simulations are not of this type. Instead a simulation is constructed to study a particular situation, run a few times and then modified or discarded. In such situations, programmer time dominates the cost of the simulation, and code efficiency is not critically important. As the cost of computing power continues to decline, this fact will become more and more evident in all areas of programming.

Therefore, one should definitely not compare SIMPAS to an optimizing FORTRAN compiler for the purpose of estimating code efficiency. Instead one should compare it to a language like SIMSCRIPT II.5. To do this, we recoded the example from Section 3 in SIMSCRIPT II.5 and compared the execution and compilation times for both versions of the program. (It is interesting to note that even though the SIMSCRIPT II.5 program was constructed by creating a line-by-line copy of the SIMPAS program, a global variable error of the type discussed above crept into the SIMSCRIPT program.) The compilation and execution times for the two programs on a UNIVAC 1100/82 were as follows (We changed max_departures to 50,000 in each program in order to provide a more significant execution time comparison.):

	Compilation time	Execution time
SIMSCRIPT II.5	3.0s	37.6s
SIMPAS	0.8s preprocess 1.7s compile 2.5s total	45.8s

All times are in CPU seconds. We see that the PASCAL code is about 20% slower than the SIMSCRIPT II.5 code. Given the simplicity of the program, it is unlikely that an optimizing FORTRAN compiler could have done much better than the SIMSCRIPT compiler, and we suspect that the majority of the difference between the SIMPAS and SIMSCRIPT execution times is due to the run time type checking of PASCAL. In our view, except for the most time critical simulations, this difference in execution time is more than compensated for by the ease of constructing correct simulation programs in SIMPAS.

5. CONCLUDING REMARKS

We have attempted to point out the advantages using a language with strong typing in the construction of simulation programs. Since a simulation program can be an extremely complex piece of software, we should use all of the tools at our disposal to make that software as reliable and as inexpensive to implement as possible. While far from perfect, we believe that PASCAL is a natural base upon which to build a widely available simulation language for the construction of reliable simulation programs. The current SIMPAS preprocessor is one step in that direction. Using the modularity features of Ada (Ichbiah 79) or MODULA II (Wirth 80) would also be another way to reach this same goal. However, for the foreseeable future, neither of these languages will be as widely available as PASCAL, and hence SIMPAS should still be a viable alternative.

To date about half a dozen simulations have been written and debugged using SIMPAS. While it is difficult to quantify, the author feels that debugging time has been reduced from that of similar simulations written in SIMSCRIPT II.5.

The current version of the preprocessor has some restrictions because it was based on an ad hoc rather than syntax-directed parsing method. A newer version which removes many of these restrictions is being considered. In fact if there is sufficient interest, a compiler version of SIMPAS designed around a very portable code generator may be developed.

We feel that SIMPAS should be trivially transportable among many systems which support PASCAL. At present versions exist for UNIVAC 1100 series (using UW-PASCAL) and UNIX (using Berkeley PASCAL). The two versions differ by less than 20 lines of code; the most significant differences are in the basic uniform random number generation routines.

We note that the two languages most commonly available on a microprocessor system are BASIC and PASCAL. This fact has led us to develop a microprocessor version of SIMPAS (Bryant 81).

A copy of the current version and its documentation can be obtained for a distribution fee of $100.00. For details write the author at: Department of Computer Science, University of Wisconsin-Madison, 1210 West Dayton Street, Madison, Wisconsin 53706.

ACKNOWLEDGEMENTS

Mr. Mark Abbott was responsible for the implementation of the SIMPAS preprocessor itself, and without his assistance the project would never have been completed. This project was supported in part by the Wisconsin Alumni Research Foundation. I also would like to acknowledge the support of the Madison Academic Computing Center, and in particular the assistance provided by its director, Dr. Tad B. Pinkerton. The SIMSCRIPT II.5 compiler we used to test our examples in Section 6 is a product of C. A. C. I., Inc. and has been supplied to the University of Wisconsin free of charge.

REFERENCES

Bryant, R. M. (1980), SIMPAS User Manual, Technical Report No. 391, Computer Sciences Department, University of Wisconsin-Madison, 54 p.

Bryant, R. M. (1981), Micro-SIMPAS: A Microprocessor Based Simulation Language, To be presented at the Fourteenth Annual Simulation Symposium, Tampa, Florida, March 17-20, 1981.

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

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

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

Gannon, J. D. and J. J. Horning. (1975), "Language Design for Programming Reliability", IEEE Transactions on Software Engineering, SE-1, 2, pp. 179-191.

Gannon, J. D. (1977), "An Experimental Evaluation of Data Type Conventions," Communications of the ACM, 20, 8, pp. 584-595.

IBM Corporation (1972), SIMPL/1 (Simulation Language Based on PL/1): Program Reference Manual, Publication SH19-5060-0, Data Processing Division, White Plains, New York.

Ichbiah, J. D. (1979), et al, "Preliminary Ada Reference Manual", SIGPLAN NOTICES, June 1979.

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

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

Landwehr, C. E. (1980), Software Engineering Techniques Applied to Protocol Simulation, NRL Report #8385, Naval Research Laboratory, Washington, D. C.

C. A. C. I., SIMSCRIPT II.5 User's Manual: Univac 1100 Series Computer Systems, C. A. C. I., Inc, 12011 San Vicente Boulevard, Los Angeles, California.

Wirth, N. (1975) "An Assessment of the Programming Language PASCAL," IEEE Transactions on Software Engineering, SE-1, 2, pp. 192-198.

Wirth, N. (1980) MODULA-2, Technical Report Nr. 36, ETH Institut fur Informatik, Zurich, Switzerland.

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