EXPERIENCE WITH SIMPAS

by

Raymond M. Bryant

Computer Sciences Technical Report #455

November 1981

Experience with SIMPAS*

R. M. Bryant

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

SIMPAS is a portable, strongly-typed, event-oriented, discrete system simulation language embedded in Pascal. It extends Pascal by adding statements for event declaration and scheduling, entity declaration, creation and destruction, linked list declaration and manipulation, and statistics collection. A library of standard pseudo-random number generators is also provided.

SIMPAS has been in use in the University of Wisconsin-Madison Computer Sciences Department for the past two years. This paper summarizes our experience with simulation programming using SIMPAS. A brief introduction to the simulation extensions SIMPAS provides is also given.

*This work was supported in part by the Wisconsin Alumni Research Foundation and through NSF grant MCS-800-3341.

Author's present address: IBM T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, N. Y. 10598.

1. Introduction

Over the past two years, we have been developing a strongly-typed, discrete-system simulation language embedded in Pascal. SIMPAS is the result of this development effort. Previous papers on SIMPAS have discussed the advantages of using strongly-typed languages for simulation program development [2], and an experimental version of SIMPAS that executed (slowly!) on an LSI-11 microcomputer system [5]. We have used various versions of SIMPAS for the construction of numerous simulations during this period and in this paper we summarize our experience with this type of Pascal based simulation.

Succinctly stated, SIMPAS provides the following extensions to Pascal:

(1) Event declaration and scheduling statements.

(2) Entity declaration, creation and disposal statements.

(3) Linked list declaration and manipulation statements.

(4) Statistics collection statements.

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

Furthermore, SIMPAS is a closed system in the sense that even though it is implemented as a preprocessor for Pascal, the user need not be aware of this. The extension statements can be intermixed with standard Pascal statements in a natural way. Also, the preprocessor automatically builds, inserts, and initializes all data structures necessary for the simulation. This is to be contrasted with

Pascal simulation packages such as PASSIM [14], which although they are substantially simpler than SIMPAS, require the user to know many details of the package implementation and to assist in declaration and initialization of the package interface variables.

The preprocessor implementation was chosen to make SIMPAS highly portable without sacrificing execution efficiency. On most systems where it is installed, the preprocessing and compilation phases can be combined under control of a single command procedure so that they are essentially transparent to the user. Careful attention has been paid to the problems of tracing error messages from the output Pascal back to the SIMPAS source, as well as reporting run time errors in terms of the original SIMPAS source line whenever possible.

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

In the next section of this paper, we discuss the SIMPAS extensions to the programming language Pascal. We then discuss some typical SIMPAS programs. Finally, we summarize the strengths and weaknesses of SIMPAS as seen through the perspective of two years of use.

2. SIMPAS Extensions to Pascal

This section describes the simulation extensions to Pascal which have been incorporated into SIMPAS. We assume that the reader is familiar with both Pascal [10] and the basic concepts of event-oriented discrete-system simulation [9]. For simplicity, this presentation skips some non-essential details. A more precise description of the language extensions is available in the latest version of the SIMPAS user manual [6].

In the discussion that follows, we will underline Pascal and SIMPAS keywords. Portions of statements that are to be replaced by appropriate user constructs will be enclosed in angle brackets ("<" and ">"). Optional portions of statements will be enclosed in square brackets, and ellipses (". . .") will be used to indicate one or more repetitions of the preceding construct.

2.1. SIMPAS Program Structure

A SIMPAS program has essentially the same structure as a Pascal program. The only differences are that an "include" statement has been added to allow transportable way to create a library of Pascal routines. Since implementing a library of pseudo-random number generation routines was necessary for SIMPAS, we implemented a symbolic library. The include statement indicates which portions of the symbolic library are to be included in the program. The include statement is found at the start of the procedure, function, and event declaration part of the program and has the form:

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

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

```
include expo;
```

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

2.2. 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. Events cannot be declared local to an event or procedure, nor can they be declared with var arguments. The first restriction is necessary so that the event routine can be called from the simulation control routine, and the latter is enforced because the event routine is called with a copy of the actual parameters stored in an event notice. Hence all parameters are effectively passed by value.

2.3. Start Simulation

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

```
start simulation(status)
```

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

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

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

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

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

2.4. Event Scheduling Statements

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

```
schedule arrival(3) at 10.0;
schedule arrival(4) delay 5.0;
schedule arrival(where) now;
```

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

An event must be declared before it is scheduled. This means that any scheduling statement referring to a particular event must syntactically follow the declaration for that event. To allow this in general, an event declaration can be forwarded exactly like a Pascal procedure.

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

```
schedule <event> named <this_event>
      delay <time_expression>;
```

Here <this_event> must be declared as type "ptr_event" (pointer to event notice).

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

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

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

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

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

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

For example, if one wished to change the time of the event <this_event>, one could use the following code:

```
cancel <this_event>;
reschedule <this_event> at new_time;
```

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

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

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

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

2.5. 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.

2.5.1. 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:

```
<entity> = queue member ^
```

```
            <attribute_1> : <type_1>;
            <attribute_2> : <type_2>;
            . . .
      end;
```

Unlike a record declaration, this declaration results in ~~<entity> being a pointer type, since this is the natural~~ declaration for a temporary entity [7]. The trailing "^" is optional and is included to remind the user that <entity> is a pointer type.

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

```
var
    <an_entity>      : <entity>;
    <another_entity> : <entity>;
```

Then <an_entity> and <another_entity> represent two different <entity>'s. Attributes of each distinct entity are referred to as follows:

```
<an_entity>^.<attribute_1>
<another_entity>^.<attribute_2>
```

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

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

where <entity> must be a previously declared queue member. This declaration may only appear in the global type part of the program. In any var part of the program one can declare a particular queue with a declaration like:

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

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

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

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

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

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

```
create <an_entity>;
destroy <an_entity>;
```

Create will insure that all preprocessor defined attributes are properly initialized. Similarly, destroy will insure that the entity is not presently in any queue, since this could result in dangling pointer errors.

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

```
initialize <queue>;
```

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

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

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

Similarly the preprocessor defines several standard queue attributes:

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

size- This attribute is of type "a_integer" and records the number of members in the queue. For example, the current queue size is "size". The maximum queue size is "size.max" and the average queue size to date is "size.mean". See Section 2.8 for details about the type "a_integer".

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

```
insert <an_entity> last in <queue>;
```

or

```
insert <an_entity> in <queue>;
```

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

```
insert <an_entity> first in <queue>;
```

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

```
remove <an_entity> from <queue>;
```

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

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

In all cases, the inserted (removed) entity must be of the same type as the queue into which it is to be inserted (removed from). Attempts to insert or remove entities in queues of the wrong type are detected during preprocessing. 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.

2.5.6. Forall Loops To simplify searching queues, SIMPAS provides the loop statements:

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

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.

2.6. Pseudo-random Number Generation

A standard collection of pseudo-random number generators are provided in the SIMPAS library and can be incorporated in the user program through the include statement. These routines all depend on a single uniform random number generator which is a portable version of LLRANDOM [9] use on all machines with a word size of 32 bits or larger. A 16 bit version of this generator is also available, but is much less efficient. Given the existence of the basic uniform random number generator, random number generators for the following distributions are provided:

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

The generation algorithms were taken from [9].

SIMPAS provides 10 random number generation streams (numbered 1 to 10). Each random number generator takes as input one of these stream id´s. Distinct streams represent different portions of the LLRANDOM generation sequence. Initially, each stream is separated from its neighbors by at least 100,000 calls.

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

2.7. Statistics Collection

SIMPAS provides automatic statistics collection features similar to those of SIMSCRIPT II.5 [12]. 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 a real variable, one declares the variable as an "a_real" (accumulated real). A variable of type a_real can be used in expressions exactly as a normal real 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.
min	the minimum value this variable has had since it was last cleared or reset.
nobs	the number of observations made for this variable to date.

For example, if x is declared as an a_real, then x.mean is its average, x.max is its maximum and so forth. 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>
```

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

The clear statement sets the value of the watched variable to zero. During a simulation, it is sometimes useful to clear only the statistics portion of a watched variable without changing the variable´s current value. (e. g. at the end of the transient interval in a steady state simulation). The reset statement can be used to do this:

```
reset <watched_variable>
```

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 the output:

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

By including the section name "regen" on the _include_ statement of the SIMPAS program, watched variables can be used to generate approximate confidence intervals through regenerative simulation. See [6] for details.

3. Experience with SIMPAS

As discussed in the introduction, SIMPAS has been used in the Computer Sciences department at the University of Wisconsin-Madison from Spring 1980 until the present (Fall 1981). During that period of time, it has been used for teaching simulation and used as a tool for computer system simulation by several researchers in the department. To evaluate the use of the SIMPAS extension statements, we selected five example simulations from those available on our VAX-11/780 system during the fall of 1981. These simulations were:

DISTCC	a model of concurrency control in a distributed database system using a shared bus architecture [15]
PROTOC	a model of the low level communications facility in a distributed operating system being developed at the University of Wisconsin
NETPAC	a general network of queues simulator with many of the features of the simulation portion of RESQ [13]
P5	a simulation of distributed scheduling on a multicomputer system organized as a rectangular grid of processors [3]

MM1SIM	a simulation of an M/M/1 queueing system [2]
TRIVIAL	the trivial simulation:

```
program trivial(output);

event foo;
begin
end;

begin
end.
```

To give an estimate of program size and complexity, the sizes of these programs and the number of events declared in each are given in Table I. DISTCC and PROTOC were actually constructed as several separate compilation units, but for purposes of comparison here they have been recombined into single compilation units.

As can be seen, none of these simulations are in the 10-15,000 line range sometimes encountered in industrial applications. In an academic environment, however, DISTCC would be regarded as a large simulation program and each of the next three would be considered typical of simulations

Program	Lines	Events Declared
DISTCC	2989	11
PROTOC	1555	9
NETPAC	1609	1
P5	1309	8
MM1SIM	159	2
TRIVIAL	8	1

Table I: Sizes of Sample SIMPAS Simulations

designed to explore computer architecture or operating systems performance questions.

3.1. SIMPAS Statement Usage

Table II gives counts of the number of extension statements used by the example programs.

Clearly, the most commonly used statements are event, schedule, insert, remove, create, and destroy. The count of <command> statements is high because a statement like:

clear x, y, z;

gets counted as three statements. Thus the large number of

Statement	DISTCC	PROTOC	NETPAC	P5	MM1SIM
event	22	18	2	8	3
schedule	13	23	11	8	4
reschedule	0	5	0	4	1
cancel	0	0	0	3	0
create	4	8	3	3	1
destroy	3	4	1	3	1
insert	6	11	4	6	2
remove	5	8	6	3	1
forall	3	0	33	6	0
initialize	2	6	16	3	1
<command>	42	0	11	15	6

Table II: SIMPAS Statement Usage

NOTES:

(1) The number of event statements may be more than the number of events declared due to the presence of forwarded events.

(2) A <command> is a statement of the form <identifier> <watched-variable>. Examples are clear or display.

(3) We will skip consideration of the TRIVIAL simulation until Section 3.4.

<command> statments in DISTCC merely reflects the large number of watched variables declared in that program.

Note that most programs that use reschedule statements do not use cancel statements. A reschedule statement can be used even without cancel, for example as in the statement:

reschedule current delay . . .

As seen from the table, this is apparently the most common use of reschedule.

The cancel statement is used only by the P5 simulation. P5 uses cancel as part of its simulation of a processor sharing scheduler [8]. We would expect cancel to be required in any simulation that involves preemption, asynchronous interrupts or the like. Thus in spite of the fact that only one of the five simulations uses cancel, we regard it as a necessary feature of SIMPAS.

In general the use of forall is limited by the fact that one may not remove the loop variable from its queue. Since this is the most common case (i. e. search until finding a queue member with particular characteristics and then remove it from the queue), the forall statement is used only for the search operation. For example, NETPAC uses queue's and forall statements to maintain lists of customers and passive tokens [13] in the queueing network simulation. This limitation is reflected in the relatively infrequent use of this statement in the other example simulations.

A correct implementation of forall would require that a local variable be declared for each forall statement. The preprocessor implementation makes this very difficult, since the local variables must be declared in the var part of the current procedure, and the forall statement is encountered after this point.

PROTOC is an unusual simulation in that it was written primarily to test a protocol implementation and not to study its performance. This is why there are no <command>'s in PROTOC; there are no watched variables declared in the program.

3.2. Schedule and Queue Mode Usage

SIMPAS supports several variations of the schedule statement. An event can be scheduled according to now, at, delay, before, or after clauses. These scheduling "modes" were used with the frequencies shown in Table III. As expected, the delay mode is the most common event scheduling mode.

Schedule Mode	DISTCC	PROTOC	NETPAC	P5	MM1SIM
now	1	16	2	1	2
at	0	1	0	2	0
delay	12	11	9	9	3

Table III: Counts of Scheduling Modes

Examination of the source code shows that the at mode is most commonly used to schedule a simulation end of run event. Such events are normally used to force the simulation to terminate after running for a specific simulation interval. Similarly, the now mode is most commonly used to schedule event main (i. e. cause the simulation to terminate), or to initialize the simulation by scheduling some events before the start simulation statement is executed.

The before and after clauses of the schedule statement were not used in any of the example simulations and thus could probably be discarded.

Corresponding to the schedule modes are the "queue" modes used in insert and remove statements. The possible queue modes are: first (as in insert first), last, before, after, and <specific> (as in remove this_job from queue). These queue modes were used with the frequencies shown in Table IV. Comparing these frequencies with the counts of insert and remove statements in each program indicates that

Queue Mode	DISTCC	PROTOC	NETPAC	P5	MM1SIM
first	3	8	4	2	1
last	4	7	3	4	2
before	1	0	0	0	0
after	1	0	0	1	0
<specific>	2	4	3	2	0

Table IV: Insert/Remove Mode Use

the most common form if *insert* is an *insert last* (the default mode) and the most common form of *remove* is *remove first* (once again the default mode). The *after* mode in P5 is used to implement a SIMSCRIPT II.5 like "insert ranked" and could be eliminated if SIMPAS supported an "insert ranked". However, the *before* and *after* modes used in DISTCC appear to be necessary, so even with the inclusion of the "insert ranked" statement in SIMPAS, one would still need insert *before* and *after*.

3.3. Statistics, Queue and Queue Member Declarations

The last set of extensions whose use we examined were those associated with new types defined by SIMPAS. Table V shows how many *member* and *queue* types, how many variables were declared in terms of a *member* or *queue* type, how many watched variables were declared, and how many assignments to watched variables there were in each program.

These counts do not include variables declared as pointers to objects that contain queue or watched variables

	DISTCC	PROTOC	NETPAC	P5	MM1SIM
member types	3	1	2	2	1
queue types	3	1	2	2	1
member vars	34	16	35	14	2
queue vars	4	5	1	0	1
watched vars	21	0	0	11	2
assign to watched var	26	0	10	16	3

Table V: SIMPAS Extension Type Usage

since these are static counts and the only reasonable way to count variables declared as pointers to objects would be dynamically (i. e. at run time). This explains why the count of watched variables declared in NETPAC is zero but there are still 10 assignments to watched variables. Additionally, the count of assignments to watched variables does not count those assignments done by SIMPAS routines. For example, every time an *insert* or *remove* statement is executed, the watched integer "queue.size" is updated by the queue insertion routines. This means that the number of watched variables declared and used in the simulations is actually higher than the counts indicate since there is one watched integer associated with each queue variable declared.

It is clear from the table that having two distinct types associated with a *queue* and *queue member* is uneccessary, since these types always occur in pairs. It might be more reasonable for SIMPAS to automatically declare a *queue* whenever the user declares a *queue member*.

Finally, the counts of watched variables and assignments to watched variables indicates that these features are heavily used throughout the simulations (except for PROTOC, which is a special case). We would argue that any reasonable simulation language should implement some type of automatic or semi-automatic statistics collection features in order to reduce the user's statistics collection burden.

One statistics collection tool that SIMPAS does not support (and should) is a histogram or table facility similar to that provided by GPSS and SIMSCRIPT II.5. In order to write a reasonable histogram facility, one needs to be able to declare a single routine and pass it different length arrays when processing different histograms. This cannot be done in Pascal, since all parameters must be declared as instances of the same type name, and differences in the array lengths of actual parameters are not allowed. (The alternative of having all histograms in the simulation be the same length seems unreasonable.) The ability to pass generic array parameters as in MODULA [16] or Ada* [1] is necessary to support the histogram feature.

3.4. Output Pascal Program Size

The lack of external compilation in Pascal does mean that PASCAL programs output by SIMPAS can be quite long. Table VI gives the sizes of the Pascal programs generated for the simulations we have been discussing.

The 611 lines of output generated from TRIVIAL consist of about 250 lines of event set declaration and maintenance routines, about 240 lines of watched type declaration and associated routines, 80 lines of event tracing and event set dump routines, and 30 lines of error handling routines. Much of this code is unchanged from simulation to simulation and if the host Pascal compiler supports external

*Ada is a trademark of the Department of Defense. 1]

Program	Input Length	Output Length
DISTCC	2989 (77328)	4310 (118248)
PROTOC	1555 (47338)	2868 (79356)
NETPAC	1609 (38204)	3453 (97397)
P5	1309 (34469)	2545 (70241)
MM1SIM	159 (3976)	970 (24082)
TRIVIAL	8 (75)	611 (15067)

Table VI
Sizes of Input and Output programs for SIMPAS
In Lines (and Characters)

compilation, can be removed from the source output and placed in an object library file. The VAX version of SIMPAS can output code for either case, using the external compilation conventions of the UW UNIX* Pascal compiler (these conventions are essentially the same as those of C [11]). Using the external compilation feature reduces the sizes of the output Pascal to the lengths shown in Table VII. Even in this case, however, it is clear that the output programs can be significantly longer than the SIMPAS source. The result is that the process of expanding and compiling a SIMPAS program can be a lengthy task. Typical times for expanding the simulations considered in this section are given in Table VIII. The expansion time is the expense that one must pay for making SIMPAS portable.

* UNIX is a trademark of Bell Laboratories. 1]

Program	Output Length without External Compilation	Output Length with External Compilation
DISTCC	4319 (118248)	3847 (107152)
PROTOC	2868 (79356)	2399 (68118)
NETPAC	3452 (97397)	2661 (77620)
P5	2545 (70241)	1992 (56942)
MM1SIM	970 (24082)	544 (13954)
TRIVIAL	611 (15067)	246 (6436)

Table VII
Sizes of Input and Output programs for SIMPAS
In Lines (and Characters)
Using External Compilation

Program	Expansion Time	Compilation Time
DISTCC	96.1	609.3s
PROTOC	55.2	219.7s
NETPAC	66.2	155.8s
P5	48.0	112.9s
MM1SIM	9.6	43.0s
TRIVIAL	5.5	40.0s

Table VIII: Expansion and Compilation Times

Note: Times are for a DEC VAX 11/780 running UNIX and using the UW Pascal Compiler.

3.5. Portability of SIMPAS

We have routinely maintained two versions of the SIMPAS preprocessor: one for VAX UNIX, and one for a UNIVAC 1100 system. Simulations written for one version can be moved to the other version with no significant changes. (Tabs must be removed when going from UNIX to the 1100 system.)

The majority of the preprocessor in the two versions is exactly the same, but there are enough minute differences between the versions to make transporting the SIMPAS preprocessor an irritating task. On the one hand, all non-standard Pascal can be removed from SIMPAS, but the resulting preprocessor is not completely usable. For example, correspondence between the internal and external files of SIMPAS must be established. This is necessary not only to allow the source program to be read from a file, but also for the preprocessor to find its external files (e. g. files containing the parse table and the symbolic source library). Conventions for doing this vary from system to system.

A second problem deals with inefficiency of Pascal I/O. On both the VAX UNIX implementations and the LSI-11 implementation of SIMPAS [5] it was necessary to replace the Pascal input routines with system routines that read input an entire disk block at a time. This change almost halved the execution time of the VAX UNIX version.

While these changes are irritating, they can be overcome through the prudent use of conditional compilation flags supported by a suitable preprocessor. The source of all versions of SIMPAS can then be kept on one machine and appropriate UNIX or 1100 OS versions can be generated as necessary. A portable (non UNIX or 1100 OS) version of SIMPAS is also derivable from this source. While the portable version lacks some of the niceties of the other versions, it is usable on essentially any system that supports "standard"

Pascal.

The primary problem with transporting the present version of SIMPAS is its size. The preprocessor is about 7,200 lines of Pascal. This means that SIMPAS will simply not function on very small machines (e. g. PDP-11´s). This limits the overall success of SIMPAS, since small machines often have a reasonable Pascal compiler, but lack a large scale simulation language such as SIMULA or GPSS. In retrospect it might have been more reasonable to aim for a less pleasant language description in order to obtain wider applicability.

4. Concluding Remarks

SIMPAS can be thought of as a strongly-typed implementation of SIMSCRIPT II.5 [12]. As such, we have found SIMPAS superior to SIMSCRIPT for the rapid and reliable construction of discrete-system simulation programs [2]. The implementation of SIMPAS as a preprocessor has resulted in a portable simulation system and has allowed us to explore alternate simulation language features relatively easily. But as a practical matter, SIMPAS is most usable with PASCAL compilers that support external compilation. Modification of the preprocessor to allow external compilation and to improve on the efficiency of standard PASCAL I/O is normally required. These factors greatly increase the complexity of the SIMPAS installation task. Finally, the size of the preprocessor itself (more than 7,000 lines of PASCAL) means that it will only run on a medium to large CPU and is not in general suitable for small machines.

While strong-typing is useful from the standpoint of reliable program construction, it complicates the output PASCAL generated from a SIMPAS program. The result is that the output PASCAL can be quite lengthy (presently, a 200 line SIMPAS simulation produces an output PASCAL program of over 1,000 lines). This also makes SIMPAS less usable on smaller machines, since PASCAL compilers for such machines can be quite slow themselves.

Many of the problems with SIMPAS can be traced directly or indirectly back to deficiencies in PASCAL (e. g. no external compilation, inability to pass arbitrary length arrays to a PASCAL routine). The primary advantage of SIMPAS is the strong typing it inherits from PASCAL. As new languages are developed that improve on PASCAL, it should be possible to create modern simulation systems or packages that have the program reliability and maintainability features of SIMPAS. We are presently considering the problems of developing such a system for Ada [4].

5. 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. 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. I would also like to thank Kevin Wilkinson, Raphael Finkel, and Bryan Rosenburg for providing the simulations used in Section 3.

REFERENCES

[1] Barnes, J. G. P., "An Overview of Ada," Software--Practice and Experience 10, pp. 851-887 (1980).

[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. and R. A. Finkel, "A Stable Distributed Scheduling Algorithm," Proceedings of the 2nd International Conference on Distributed Computing Systems, (April 8-10, 1981).

[4] Bryant, R. M., "Discrete System Simulation with Ada," Computer Sciences Department Technical Report, University of Wisconsin--Madison (In preparation, 1981).

[5] Bryant, R. M., "Micro-SIMPAS: A Microprocessor Based Simulation Language," Proceedings of the Fourteenth Annual Simulation Symposium, pp. 35-55 (March 17-20, 1981).

[6] Bryant, R. M., "SIMPAS 5.0 User Manual," Computer Sciences Department Technical Report, University of Wisconsin--Madison (in preparation, 1981).

[7] 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.

[8] Coffman, E. G. and P. J. Denning, Operating Systems Theory, Prentice-Hall (1973).

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

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

[11] Kernighan, B. W. and D. M. Ritchie, The C Programming Language, Prentice-Hall (1978).

[12] 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).

[13] Reiser, M. and C. H. Sauer, "Queueing Network Models: Methods of Solution and Their Program Implementation," pp. 115-167 in Current Trends in Programming Methodology Volume III: Software Modeling, ed. K. Mani Chandy and Raymond T. Yeh, Prentice-Hall, Inc. (1978).

[14] Uyeno, D. H. and W. Vaessen, "PASSIM: A Discrete-event Simulation Package for PASCAL," Simulation 35, 6, pp. 183-190 (December 1980).

[15] Wilkinson, W. K., "Database Concurrency Control and Recovery in Local Broadcast Networks," Computer Sciences Technical Report #448, University of Wisconsin--Madison, Madison, Wisconsin (September 1981). Ph. D. Thesis.

[16] Wirth, N., "Modula: A language for Modular Multiprogramming," Software Practice and Experience 7, 1, pp. 3-35 (1977).