IMPLEMENTATION OF SIMPAS

by

R. M. Bryant, M. B. Abbott,
J. R. Bugarin, B. S. Rosenburg

Computer Sciences Technical Report #459

November 1981

Implementation of SIMPAS*

R. M. Bryant, M. B. Abbott,
J. R. Bugarin, B. S. Rosenburg

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 is implemented as a preprocessor for Pascal. This paper discusses the design and implementation of SIMPAS, and the difficulties involved in preprocessing a strongly-typed language like Pascal.

keywords: simulation, preprocessor, Pascal

---

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

Authors present addresses: R. M. Bryant is with the IBM T. J. Watson Research Center, Yorktown Heights, N. Y. M. B. Abbott is with Bell Laboratories, Denver, Colorado. J. R. Bugarin is with the Hewlett Packard Desktop Computer Division, Ft. Collins, Colorado. B. S. Rosenburg is with the Computer Sciences Department, University of Wisconsin-Madison, Madison, Wisconsin.

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.

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 (queues).

(4) Automatic statistics collection facilities.

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

A primary design goals of SIMPAS was to make it highly portable while still retaining the full execution speed of the host machine. Implementing SIMPAS by an interpreter (c. f. [17]) could meet the former goal but not the latter. Additionally, we were more interested in the simulation extensions to Pascal instead of the problems of generating code for expressions and Pascal control structures. For these reasons, we decided to implement SIMPAS as a preprocessor for Pascal. Given the existence of Pascal compilers for many machines, and provided we adhered to the "standard" Pascal subset [16] we felt that the resulting program should be trivially transportable from machine to machine.

Numerous other proposals for extending Pascal to support quasi-parallel programming and simulation have appeared

in recent years [8,18,21,22,23,25]. These extensions have been implemented either by constructing new compilers, modifying existing Pascal compilers, or implementing new run-time routines to allow the creation of processes in Pascal. We wished to avoid the machine dependence of these implementations by coding SIMPAS entirely in Pascal. A preprocessor implementation seems to be the only way to do this.

Given the decision to implement SIMPAS as a preprocessor, our second design goal was to make the extension statements appear as natural extensions of Pascal. For example, having the extension statements flagged via special characters in the SIMPAS source was not regarded as an acceptable solution. The result of this design goal is that on most systems where SIMPAS is installed, the preprocessing and compilation phases are called by a unified command procedure, so that the user need not know that translation of a SIMPAS program is a two step process. This decision complicated our implementation effort considerably as we shall discuss in the sequel.

In the next sections of this paper we first describe in a general way the implementation of SIMPAS. We then discuss the simulation extensions that SIMPAS provides for the programming language Pascal. We then illustrate the Pascal translations of some typical SIMPAS statements. Having described the task that the SIMPAS preprocessor must perform, we next describe the implementation of SIMPAS itself. Finally, we discuss the difficulties we have encountered in

creating a preprocessor for a block structured, strongly-typed language such as Pascal and comment on the success of this approach for the implementation of SIMPAS.

2. Overall Structure of SIMPAS

SIMPAS consists of the preprocessor program itself (about 7800 lines of Pascal) and three files that contain the parse table, the error correction table, and a symbolic library. The first two files are used by FMQ [11], the table-driven, locally-least-cost error-correcting parser used by SIMPAS. These files are generated by the FMQ parser generator from the SIMPAS grammar and will not be described here. (For details of the SIMPAS grammar see Section 5.) The symbolic library is necessary because there is (at present) no commonly accepted standard for external compilation in Pascal. Thus the only completely transportable way to build a library is to do it at the symbolic level. A SIMPAS extension statement (include) indicates which portions of the symbolic library are to be included in the program.

Even though Pascal can be compiled in a single pass, SIMPAS must be a two pass processor. This is because the SIMPAS preprocessor builds the declarations for the simulation event set. The structure of this set depends on the of events declared and their arguments, and the latter are not known until the entire source program has been examined.

An alternative approach would have been to insert an "event" declaration part following the label declaration part in a SIMPAS program. The event declaration part would include the event names and the names and types of the event arguments. We did not use this approach because (1) we wished an event declaration to be as similar as possible to a procedure declaration and (2) using the alternative implementation, event arguments would often be declared via type identifiers that had not yet been declared (since the "event" declaration part precedes the type declaration part). While the output Pascal would be properly ordered, the SIMPAS source program would thus violate normal declaration rules in Pascal. Because of these problems we adopted the two-pass approach.

During the first pass, the input program is read and parsed. Comments are removed and SIMPAS extension statements are translated into standard Pascal statements. The include statement is parsed and a list of section names to include from the library file is created. A symbol table containing all identifiers declared in the program is built. Markers are placed in the temporary file at the beginning of the global constant, type, var, procedure, and main procedure parts of the program.

The second pass copies the output of the first pass to the output, stopping at each marker to insert additional declarations. Program dependent declarations are built by the preprocessor and inserted at these points. Program

independent declarations are added from the symbolic library. To simplify this task, the library file is divided into const, type, var, procedure and main (initialization) parts. Within each part are sections that correspond to the section names given on the include statement. When the const-part marker is encountered, the requested sections from the const part of the library file are added at the top of the const part of the output Pascal program. This task is repeated for each of the other markers.

Normally the only place a section is found is in the procedure part of the library file. However, if that procedure requires a global type or variable declaration, all that needs to be done is to include that section name and the appropriate declaration in the type or var part of the library file. These declarations will be included in the output Pascal along with the procedure. Thus the include statement implements a significantly more intelligent source level inclusion mechanism than is supported by most systems.

3. SIMPAS Extensions to Pascal

This section describes the simulation extensions to Pascal that have been incorporated into SIMPAS. These extensions were inspired by the simulation features of SIMISCRIPT II.5 [20], and in many ways SIMPAS can be thought of as a strongly-typed implementation of SIMSCRIPT.

We assume that the reader is familiar with both Pascal [16] and the basic concepts of event-oriented discrete-

system simulation [12]. 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 [2].

3.1. Event Declaration

An event declaration has exactly the same syntax as a Pascal procedure declaration, except that the reserved word event replaces the reserved word procedure. Events must be declared local to the main procedure. They 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.

3.2. 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 that is passed by reference to the simulation control routine. Status is set by the simulation control routine to indicate why the simulation terminated. (See [2] for further details.)

While the simulation is active, the global, real, variable "time" gives the current simulation time.

3.3. Event Scheduling Statements

Event notices are created and inserted into the event set by scheduling statements that are similar to those of SIMSCRIPT II.5 [20]. Typical scheduling statements have the form:

schedule <event-name> at <time-expression>
schedule <event-name> delay <time-expression>
schedule <event-name> now

The <event-name> may be followed by an <actual-argument-list> if so required.

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 schedules an event to occur immediately.

An event must be declared before it is scheduled. This means that any scheduling statement referring to a particular event must lexically 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 arguments and the execution time of the event are stored in the event notice. The event notice contains the information necessary to

execute the event.

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 retain 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> <time-specifier>

where <time-specifier> is one of: now, at <time-expression>, etc. The variable <this_event> must be declared as type "ptr_event" (pointer to event notice).

Given a pointer to an event notice, it can be removed from the event set by using the cancel statement:

cancel <event-pointer>

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 a variable of type pointer to event notice instead of an event name. The actual arguments of the event remain the same as those on the original schedule statement.

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 <time-specifier>

If "current" is not rescheduled by the event routine, the

event notice is automatically destroyed when the event routine returns.

3.4. 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.

3.4.1. Entity and Queue Declarations One declares an entity type in the global type declaration part of the program; the declaration has syntax similar to that of a record declaration:

```
type
    <entity> = queue member ^
                 <attribute_1> : <type_1>;
                 <attribute_2> : <type_2>;
                 . . .
           end;
```

Using the normal simulation terminology, the fields of this record declaration are referred to as the "attributes" of the entity. However, unlike a record declaration, this declaration results in <entity> being a pointer type, since this is the natural declaration for a temporary entity.* The trailing "^" is included to remind the user that <entity> is

*Temporary entities are dynamically created during the simulation. Permanent entities, on the other hand, are static and remain in existence throughout the simulation. Arrays of records are the natural representation of permanent entities in Pascal [3].

a pointer type. It can be omitted if desired.

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 (or procedure) one can declare a particular queue with a declaration like:

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

3.4.2. Entity Creation and Disposal Since a variable of type queue member is a pointer variable, one can use the standard Pascal procedure "new" to create new entity

instances. 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 make sure that the preprocessor defined fields of an entity are properly initialized, SIMPAS provides the create and destroy statements:

create <an_entity>;
destroy <an_entity>;

Create uses "new" to allocate a new entity and then initializes the preprocessor defined fields. Similarly, destroy calls "dispose" but will first insure that the entity is not currently in a queue, since this could result in dangling pointer errors.

3.4.3. Queue Initialization Queues in SIMPAS are represented as doubly linked lists with head nodes. Before any entity may be inserted in a queue, the queue must be initialized by allocating the head node and setting the queue attributes to represent an empty queue. Attempting to place an entity in an uninitialized queue will result in unpredictable behavior. The initialize statement is used to initialize a queue:

initialize <queue>;

3.4.4. 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 a type that matches the specified queue. 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.

3.4.5. 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.

3.5. 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 portable version of LLRANDOM [12] usable 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 [12].

3.6. Statistics Collection

A requirement of a simulation language is that it should provide support for the collection of simulation statistics. 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 of a special (predefined) type, which we will refer to as a "watched type". A variable declared in terms of a watched type will be called a "watched variable." For example, to enable time-averaged statistics collection of a real variable, one declares the variable as an "a_real" (accumulated real). A real-valued, event-averaged watched variable is declared as a "t_real" (tallied real). Accumulated and tallied integer and boolean watched types are also supported.

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

clear <watched_variable>

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 sreset statement can be used to do this:

sreset <watched_variable>

Watched variables can also be used to generate approximate confidence intervals through regenerative simulation. See [2] for details.

4. SIMPAS Output

As previously discussed, a SIMPAS program is translated into standard Pascal by the preprocessor. This section describes the generated Pascal.

4.1. Event Declarations

The declaration

```
event <name> (<formal arguments>);
    <event-body>;
```

is translated to the declaration

```
procedure r_<name> (<formal arguments>);
    <event-body>;
```

Additionally the event name and the names and types of the actual arguments are saved for use in building the event set

data structure and translating the schedule and reschedule statements.

The event set declarations are built as follows: The enumeration type t_ev_l is declared as a list of all event names declared in the current program. For each event with arguments, the type t_<name> is declared as a record type whose field names match the names and types of the formal parameters. Then the event notice declaration is built in the following form:

```
event_notice = record
               prev, next : ^event_notice;
               . . . {some other fields for tracing, etc}
               case eventtype : t_ev_l of
                    { repeat the following for each event <name> }
                   <name> : (a_<name> : t_<name>);
                    . . .
               end;
```

The tag "eventtype" indicates the name of the event described by this event notice. The field a_<name> contains the actual arguments for the event and are stored there by the schedule statement code. The event set itself is constructed as a doubly linked list with head node using the pointers "prev" and "next".

4.2. Scheduling Statements

The statement

schedule <name>(<actual arguments>) <mode> <time-expression>

is translated to

```
begin
    c_notice(g_notice, <name>);
```

```
    with g_notice^.t_<name> do
    begin
        <argument-1> := <actual-arg-1>;
        <argument-2> := <actual-arg-2>;
        . . .
    end;
    e_insert(g_notice, nil, <time_expr>, e_<mode>,
            <line_number>);
end
```

Here g_notice is a global variable of type ^event_notice, <mode> is one of at, delay, or now, and <time_expr> is the (perhaps empty) time expression. C_notice creates an event notice of type <name> and initializes its fields, and e_insert places this notice into the event set. The second argument of e_insert is used in schedule before (after) statements to provide e_insert with the pointer to the event notice to be scheduled before (after). The <line_number> is the SIMPAS source line of the schedule statement and is used in error reporting. Finally, since the schedule statement looks to the user like one SIMPAS statement but expands to several Pascal statements, the output is enclosed in a (perhaps unnecessary) begin-end pair.

4.3. Queue and Queue Members

The queue and queue member declarations are translated to appropriate record declarations in the obvious way. Entity attributes become fields of the record and the preprocessor inserts extra fields to hold links to the next and previous members of the queue. In response to declaration of a queue member of type <entity>, procedures c_<entity> and d_<entity> are output to create (and destroy)

queue members of that type. Create and destroy statements for entities of type <entity> are translated into calls on c_<entity> or d_<entity> respectively. The type of the variable to be created or destroyed determines which of the c_<entity> (d_<entity>) routines should be called. To make this choice, the preprocessor keeps the types of all variables declared in the program in its symbol table. (We found that it was simpler to keep everything in the symbol table instead of just the variables of type queue member, for example, even though some of this information is not needed.)

In response to declaration of a queue type named <queue>, the SIMPAS preprocessor declares a queue initialization procedure i_<queue> and queue insertion and removal procedures p_<queue> and r_<queue>. The initialize statement translates into a call on i_<queue> and the insert and remove statements translate into calls on p_<queue> and r_<queue> respectively. The type of the variable to be initialized determines which i_<queue> routine is to be called by the initialize statement. Since the types of all variables are kept in the preprocessor's symbol table, the preprocessor can check to make sure that the entity being inserted/removed from the queue is of the correct type for that queue.

Declaration of the routines associated with queues and queue members is the primary reason that these declarations are only allowed at the global type level. Queue and queue

member declarations are required to be simple (i. e. of the form <id>= queue of . . .) so that the correspondence between the queue or queue member type and its associated maintenance procedures can be made. Queues declared with anonymous type names (as a field inside of a record for example) would make this correspondence difficult.

4.4. Forall Loops

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;
        <entity-ptr> := <entity-ptr>^.next;
    end;
end
```

Statement S may not contain a remove <entity-ptr> from <queue> statement since this destroys <entity-ptr>^.next and makes it impossible to advance the loop index. 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." This test catches some, but not all occurrences of the error.

The restriction on statement S could be avoided by declaring a temporary variable of type <entity> for each forall loop. But this is a complex task for the

preprocessor, since the variable declaration must be placed in the var part of the current function or procedure. Since we have already passed that point when the forall is encountered, this insertion would have to be done during Pass 2. Our judgement was that this convenience was not worth the added complexity. (However, see [4]) .

4.5. Watched Variables

Each watched type is actually a record type with a field named "val" of the true base type of the variable (e. g. val is a real field for variables of type t_real). Other fields are used to maintain the statistics associated with the variable.

Wherever a variable appears in a place where a value is required (on the right hand side of an assignment statement, for example) the preprocessor checks in its symbol table to see if the identifier is a watched variable. If it is, the preprocessor appends ".val" to the variable name. This has the effect of converting the watched variable to its corresponding simple type.

In general, the user is free to use a watched variable as if it were a simple type. One can declare arrays of watched variables and fields of a record can be declared as watched types. The only places watched variables cannot be used are where Pascal prohibits the use of a field value in place of a simple variable, such as the index in a for loop.

For each assignment statement in the SIMPAS program, the preprocessor checks to see if the variable being assigned to is a watched variable. If it is then the assignment statement is translated to a call on a statistics update routine. For example, the assignment statement:

<watched> := RHS;

is translated to a procedure call of the form:

obs_<type>(assign, <watched>, RHS);

where <type> is the simple type of <watched>. The procedure obs_<type> is declared as

obs_<type>(action : s_mode; var wv : watched_var; RHS : <type>);

Procedure obs_<type> takes the action indicated by the first argument. For example, when action=assign, obs_<type> updates the statistics in watched_var in accordance with the new value RHS, then assigns the value of RHS to watched_var.val.

Other possible values for action are "clear", which is generated in response to a clear statement, and "regen" which is used in regenerative simulations to indicate the end of a regeneration cycle. See [2] for details.

The problem of watched variables being passed to a procedure and modified there is handled the following way. If a watched variable is used as a value argument to a procedure or function, and that argument is declared as a simple type instead of a watched type, then the watched vari-

able instance is changed to a simple type by appending ".val" to the identifier. This corresponds to the occurrence of the watched variable on the right hand side of an assignment statement. On the other hand, if the argument type is a watched type, then the preprocessor requires this argument to be declared as a var parameter. Any changes to the argument in the procedure will therefore be properly updated according to the rules described above.

A special case occurs when a watched variable is used in a with statement. On the one hand, the watched variable is supposed to appear as if it is an instance of its simple type, and the target of a with statement must be a record type. On the other hand, it is clear to the user that the watched type looks like a record (the mean of a watched variable named "w" is "w.mean") so that the user will naturally try to say "with w do . . .". Translating this to "with w.val do . . ." has the effect of prohibiting watched variables in with statements. This is handled as a special case so that the target of the with is always a record type and ".val" is not appended to the watched identifier. (Note that watched variables can be present in subscript expressions in the with target, for example, and must still be expanded in this case.)

5. Implementation of SIMPAS

For a variety of reasons, SIMPAS has turned out to be a much more complex program than we had originally envisioned.

Much of this complexity is a direct result of our design decisions to make the SIMPAS extensions appear as natural parts of the language. This implied that SIMPAS had to parse all the input Pascal, even though it was primarily interested only in a small subset of the statements. SIMPAS had to be aware of Pascal grammar because:

(1) The types of variables had to be known to the preprocessor to correctly expand certain SIMPAS statements. Determining this information without completely parsing the type declarations appears impossible.

(2) Finding the start of the main procedure (to allow insertion of event set initialization code) required that the structure of a Pascal program be known to the preprocessor.

(3) A forall loop is expanded by inserting code before and after the statement comprising the body of the loop. Allowing the body of a forall loop be an arbitrary Pascal statement requires knowledge of Pascal grammar to recognize when the statement ends.

Another reason for deciding to completely parse the input Pascal was that a previous version of SIMPAS attempted to examine only the extension statements themselves and pass the rest of the input through essentially unchanged [5]. The following features of the language as described in Section 2 were not implementable:

(1) watched variables

(2) clear, initialize, etc statements (These were replaced by subroutine calls.)

(3) forall loops with simple statement bodies (The body of the loop was required to be surrounded by a begin-end pair.)

(4) insert, remove statements translated to subroutine calls (Inline code was generated instead.)

These problems occurred either because this version of the preprocessor did not parse the Pascal portions of the input program or was not aware of the types of variables. Additionally, this version of the preprocessor (while a simpler program than the present version) was difficult to extend because of its hand-coded parser. We therefore decided to use a table-driven parser to analyze all the source input and to build a complete symbol table. This would overcome the problems of the previous version and allow us to easily extend the preprocessor.

The SIMPAS grammar was constructed from a grammar for Pascal by adding productions for the SIMPAS extensions and simplifying the grammar where possible. The original FMQ grammar for Pascal had about 250 productions. This included extensions for external compilation and is somewhat larger than a grammar for "standard" Pascal. The UNIX* version of the SIMPAS grammar has about 220 productions of which 40 are directly related to SIMPAS constructs. There are about 65 reserved words in the SIMPAS grammar; 25 of these are associated with the SIMPAS extensions. The SIMPAS grammar was simplified by eliminating some of the productions associated with code generation (for example, productions that help maintain operator precedence while parsing expressions) and by collapsing some of the productions for type declarations. The resulting grammar is an extension of the Pascal grammar

*UNIX is a trademark of Bell Laboratories.

in the sense that any legal Pascal program will be accepted by the parser (provided that there are no SIMPAS reserved words in the program). Statements such as clear, create, destroy, etc. were all handled by a single production in the grammar that introduced

<identifier> <identifier-list>

as a new statement type in the grammar.

Semantic actions are associated primarily with SIMPAS extension statements. The most common semantic action turned echoing of parsed tokens either on or off. The parser normally echoes all tokens into the Pass 1 temporary file. Pascal control statements therefore cause no semantic processing. When a SIMPAS statement is encountered, the first semantic action turns off echoing so that the associated semantic routine can output the Pascal version of the SIMPAS statement. The last semantic action of the production turns echoing back on. However, handling of watched variables requires semantic processing of all expressions so that echoing is more often off than on.

5.1. SIMPAS 'Run Time' Routines

The SIMPAS 'run time' routines consist primarily of the event set maintenance routines and the simulation control routine. Currently, SIMPAS uses a linear, doubly-linked list of event notices to represent the simulation event set. Alternative organizations based on heaps [14], the Franta-Maly TL mechanism [13], and the binary search algorithm of

Henriksen [15] have been investigated. Thus far, we have found that other activities associated with scheduling an event (e. g. allocating a new event notice, recording trace information) dominate the scheduling time and the differences between these algorithms until the event set becomes extremely large. In such cases we surmise that the simulation itself should be reorganized instead of expecting the event set insertion routine to cope with an unreasonably large event set.

The automatic reclamation of unused event notices by the simulation control routine is complicated by the lack of a function in Pascal to determine if a pointer is valid or not. When an event routine returns to the simulation control routine, the event notice "current" should be destroyed if it was neither rescheduled nor destroyed by the user. Since there is no way to check whether a pointer is valid or not, we adopted the conventions that an event-notice pointer is set to nil whenever it is destroyed and that "current" is removed from the event set before the event routine is called. The following if statement determines whether or not to destroy "current":

```
if current <> nil then
    if (not scheduled (current)) and (not current^.named) then
        d_notice (current);
```

The and clause of the second if is necessary to keep the simulation control routine from disposing of "current" in the case that "current" was created in a schedule statement

with a named clause. Since in this case the user has a pointer to the current notice, destroying "current" may result in dangling pointer errors. In spite of these precautions, this if statement is usually where Pascal heap integrity checks fail, often long after the error occurred. Such errors are, in general, extremely difficult to locate.

5.2. Pascal Implementation Difficulties

Pascal I/O inefficiency has been a persistent problem. The speed of the VAX UNIX version was nearly doubled when instead of using Pascal input, the underlying UNIX routines were called directly to obtain input in 1,024 character blocks instead of one character at a time. A similar speedup occurred in an implementation of SIMPAS for an LSI-11 microcomputer system when the Pascal input routines were replaced by direct file I/O [6].

The lack of random access I/O in Pascal has meant that the FMQ parser is less effective in general than we had hoped. The FMQ error correction file is large (65K bytes on a VAX 11/780), and sequentially searching through this file during error correction can be slow. In the UNIX implementation of FMQ (and hence SIMPAS) random access I/O is simulated by using operating system routines to seek to the appropriate place in the error correction file. To avoid long error correction runs by FMQ when running under Pascal compilers that do not allow random access I/O, we have limited the number of errors it attempts to correct to 30 in

the portable versions of SIMPAS.

In generating the output Pascal, we needed to use numerous string constants of varying length. The only reasonable way to do this in Pascal was to use write statements to directly output the generated Pascal (as opposed to placing the output in a buffer and writing the output when the buffer fills). This made it impossible to determine if the output line is potentially too long to be read by the host Pascal compiler. (Some Pascal compilers will not accept input lines longer than 80 characters [10]) . This was overcome by examining long lines during Pass 2 of the preprocessor and splitting them at appropriate places. However, this reparsing of the output is duplicated effort that we should have been able to avoid. Ada* [24] provides a variable associated with each file indicating the current column number in the file; this would have solved our line-length problem.

6. Difficulties in Preprocessing Pascal

One of the primary advantages of SIMPAS over SIMSCRIPT II.5 is that SIMPAS inherits the strong typing of Pascal. Thus a simulation written in SIMPAS will often be easier to debug and maintain than the corresponding simulation written in SIMSCRIPT II.5 [1]. With respect to implementing SIMPAS, on the other hand, strong typing complicates matters considerably. For example, since each queue member

*Ada is a trademark of the Department of Defense

type and queue type become distinct Pascal types, the maintenance routines (c_<entity>, d_<entity>, etc.) must be duplicated for each type.

This could have been avoided by having the preprocessor represent all queue member types as pointers to a single variant record type. However, this would give the user no compile-time error checking against referencing attributes not possessed by a particular queue member, and also would require that distinct queue members have distinct attribute names. Our judgement was that the compile-time error protection was important enough to justify the duplication of queue and queue member maintenance routines. Experience has also shown that only a few queue and queue member types are declared in most SIMPAS simulations so that the number of maintenance routines is small [4].

Similarly, the watched variable observation routines need to be more or less duplicated for six different watched variable types (one for time-averaged and one for event-averaged observations for each of the integer, real, and boolean types). This costs about 250 lines of output code in each SIMPAS simulation.

The lack of external compilation in Pascal also increases the size of the programs output by SIMPAS. Table I gives the sizes of some SIMPAS programs and the corresponding output Pascal programs.

DISTCC and PROTOC were written as several distinct compilation units; for ease of comparison here they were

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

combined into a single compilation unit. DISTCC is a model of concurrency control in a distributed database system [26], PROTOC is a model of a simulation protocol, NETPAC is a general network of queues simulator, and P5 is a model of distributed scheduling in a multicomputer system [7]. Each of these are non-trivial simulations. On the other hand, MM1SIM is an M/M/1 simulator and TRIVIAL is the trivial simulation:

```
program trivial(output);

event foo;
begin
end;

begin
end.
```

For a more detailed analysis of these simulations and the SIMPAS features that they use, see [4].

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 compilation, can be removed from the source output and placed in an object library file. The VAX version of SIMPAS can output Pascal for either case, using the external compilation conventions of the UW UNIX Pascal compiler (these conventions are similar to those of C [19]). Using the external compilation feature reduces the sizes of the output Pascal to the lengths shown in Table II. Even in this case, however, it is clear that the output programs can be significantly longer than the SIMPAS source.

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 II
Sizes of Output programs for SIMPAS
In Lines (and Characters)
Using External Compilation

7. Concluding Remarks

SIMPAS can be thought of as a strongly-typed implementation of SIMSCRIPT II.5 [20]. As such, we have found SIMPAS superior to SIMSCRIPT for the rapid and reliable construction of discrete-system simulation programs [1]. The implementation of SIMPAS as a preprocessor has resulted in a portable simulation system and has allowed us to explore alternate simulation language features [9]. But as a practical matter, modification of the preprocessor to allow external compilation and to improve on the efficiency of standard Pascal I/O is normally required in order to produce a usable system on a new target machine. Also, random access I/O is nearly necessary for the SIMPAS parser to function properly during its error correction phase. These factors greatly increase the complexity of the SIMPAS installation task. Finally, the size of the preprocessor itself means that it will only run on a medium to large CPU and is too large and slow to be used on small machines. The result is that the preprocessor is not nearly as portable as we would like.

While strong-typing is useful in program construction, it complicates translation of a SIMPAS program. Our experience has shown that it is considerably more complex to write a preprocessor for Pascal than it is to write a preprocessor for a less strongly-typed language such as FORTRAN or PL/I. A successful preprocessor for Pascal must have many of the functions of the front end of a Pascal compiler.

Strong typing also means that nearly identical service routines in SIMPAS must be replicated to handle distinct queue and queue member types. External compilation allows some redundant code to be placed in an object library, but certain routines must always be included at the source level since they depend on types declared in the SIMPAS source.

The primary justification for implementing SIMPAS as a preprocessor was to increase portability of the language. The implementation of SIMPAS as a preprocessor has been difficult. Many of the event scheduling primitives could be implemented in a few weeks by using external compilation in Pascal and constructing run-time routines to create coroutines (for example, as in [21]). Even though these routines would not be portable, we could have probably implemented several machine dependent versions of SIMPAS using this method in the same amount of time required to implement the preprocessor version.

In summary, while the present implementation of SIMPAS has been a useful experimental vehicle, a production version of SIMPAS should be implemented as a compiler. This could be done, perhaps, by including the SIMPAS extension statements in an existing Pascal compiler.

8. Acknowledgements

M. Abbott, J. Bugarin, and B. Rosenburg are primarily responsible for the implementation of SIMPAS and without their assistance the project would never have been com-

pleted. D. Wasserman and V. James implemented several different event set maintenance routines. Prof. R. Finkel was one of the earliest users of SIMPAS, and his perseverance in dealing with the early versions helped us to produce a usable preprocessor. I also would like to acknowledge the support of the Madison Academic Computing Center, and in particular the assistance provided by its director, Dr. T. B. Pinkerton.

REFERENCES

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

[2] Bryant, R. M., "SIMPAS 5.0 User Manual," Computer Sciences Department Technical Report #456, University of Wisconsin--Madison (November 1981).

[3] 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.

[4] Bryant, R. M., "Experience with SIMPAS," Computer Sciences Department Technical Report #455, University of Wisconsin--Madison (November 1981). Submitted for publication.

[5] Bryant, R. M., M. B. Abbott, J. R. Bugarin, and B. S. Rosenburg, "Preprocessing PASCAL: A Comparison of Two Approaches for Extending PASCAL via a Preprocessor," Computer Sciences Department Technical Report, University of Wisconsin--Madison (in preparation, 1981).

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

[7] 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).

[8] Deminet, J. and J. Wisiniewska, "Simpascal," Pascal News, pp. 66-68 (March 1980). Newsletter of the Pascal User's Group.

[9] Finkel, R. A. and R. M. Bryant, "Time Lines -- A Process Like Simulation Construct for Event Oriented Languages," Computer Sciences Department Technical Report, University of Wisconsin--Madison (in preparation).

[10] Fischer, C. N., "UW-PASCAL Reference Manual for the Univac 1100," Madison Academic Computing Center, The University of Wisconsin--Madison (October 1977).

[11] 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).

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

[13] Franta, W. R. and K. Maly, "An Efficient Data Structure for the Simulation Event Set," Communications of the ACM 20, 8, pp. 596-601 (August 1977).

[14] Gonnet, G. H., "Heaps Applied to Event Driven Mechanisms," Communications of the ACM 19, 7, pp. 417-418 (July 1976).

[15] Henriksen, J. O., "An Improved Events List Algorithm," Proceedings of the 1977 Winter Simulation Conference, pp. 547-556 .

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

[17] Johnson, G. F., "A Portable Discrete Event Simulation Package for Microcomputers," Proceedings of the 12th Annual Simulation Symposium, pp. 22-27 .

[18] Kaubish, W. H., R. H. Perrot, and C. A. R. Hoare, "Quasiparallel Programming," Software--Practice and Experience 6, pp. 341-356 (1976).

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

[20] 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).

[21] Kriz, J. and H. Sandmayr, "Extension of Pascal by Coroutines and its Application to Quasi-Parallel Programming and Simulation," Software--Practice and Experience 10, pp. 773-789 (1980).

[22] Noodt, Terje and Dag Belsnes, "A Simple Extension of Pascal for Quasi-Parallel Processing," Sigplan Notices 15, 5, pp. 56-65 (1980).

[23] Rooda, J. E., N. G. M. Blokhuis, and C. Bron, "Discrete Event Simulation in Pascal," Department of Mecahnical Engineering Technical Report #7, Twente University of Technology, Enshede, The Netherlands (April 1981).

[24] U. S. Department of Defense,, Reference Manual for the Ada Programming Language, U. S. Government Printing Office, Washington, D. C. 20802. (July 1980). Proposed Standard Document, GPO 008-000-00354-8.

[25] Welsh, J. and M. McKeag, Structured System Programming, Prentice-Hall (1980).

[26] 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.