PREPROCESSING PASCAL:

A COMPARISON OF TWO APPROACHES FOR EXTENDING PASCAL VIA A PREPROCESSOR

by

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

Computer Sciences Technical Report #460

November 1981

Preprocessing Pascal:

A Comparison of Two Approaches for Extending Pascal via a Preprocessor*

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 implemented via a preprocessor for 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.

Two distinct versions of SIMPAS have been implemented. The first version used an *ad hoc* parsing technique while the second used a table-driven, locally-least-cost error correcting LL(1) parser. Additionally, the first version was not aware of types of variables in the SIMPAS extension statements while the second version maintains a complete symbol table. In this paper, we compare the two implementations of SIMPAS from the standpoints of efficiency, size, maintainability, and ease of extension. In many ways, the first version of SIMPAS is found to be the more cost effective version.

---

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

Author's 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.

1. Introduction

When portability, ease of implementation, and maximal use of existing software are of prime importance, it is common to use a preprocessor to extend the capabilities of an existing programming language. For example, structured preprocessors for FORTRAN (such as RATFOR [11,12]) have been used on many systems to provide FORTRAN-like languages with powerful control and looping structures. For similar reasons, we implemented SIMPAS (a strongly-typed, discrete-system simulation language embedded in Pascal [1,3,4,5,6,7]) via a preprocessor.

Numerous other proposals for extending Pascal to support quasi-parallel programming and simulation have appeared in recent years [8,10,14,16,18,17]. 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.

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 pseudo-random number generators.

A brief summary of these extension statements is provided in Section 2 of this paper. (For a more detailed description ~~see the references cited above.)~~

This paper discusses the general problems of embedding extensions like those of SIMPAS in a strongly-typed language like Pascal. We do this by comparing two distinct versions of SIMPAS. The first implementation (which we will refer to as SIMPAS I), was a "quick and dirty" approach that attempted to examine only the SIMPAS extension statements and pass the rest of the program on to the Pascal compiler. This version used an *ad hoc* parsing technique that can be described as recursive descent applied to the SIMPAS extension statements and the *begin*-*end* structure of the program. SIMPAS II, on the other hand, used a table-driven, locally-least-cost error-correcting LL(1) parser [9]. SIMPAS II parses all the SIMPAS source, although much of the source is passed through to the Pascal compiler unchanged. As can be expected, SIMPAS I runs faster, is a simpler program, and is harder to extend than SIMPAS II. However, because SIMPAS I does not maintain a symbol table, it is unaware of types of variables in the SIMPAS extension statements. This made it impossible to implement certain desired features and complicated implementation of other SIMPAS constructs. This paper compares the implementation of these two versions of SIMPAS.

In the next section of this paper we briefly describe the extensions that SIMPAS provides for the programming language Pascal. We then give an overview of the implementation of SIMPAS I and II and discuss the relative advantages and disadvantages of the two implementations. Performance and size comparisons of the two versions are given. The relative costs of implementation for the two versions are then compared. In terms of cost for features implemented, our conclusion is that SIMPAS I is the more cost effective implementation.

2. Summary of SIMPAS Extensions

Syntax diagrams and brief descriptions for the SIMPAS extension statements are shown below. More detailed descriptions of the SIMPAS statements are available in [5,6,7]. Descriptions of the output Pascal generated for these statements as well as more information on the implementation of SIMPAS II is available in [3].

In this discussion keywords are underlined and angle-brackets ("<" and ">") enclose user supplied portions of SIMPAS statements. Square brackets are indicate optional portions of a statement. Braces ("{" and "}") surround a list of alternatives separated by vertical bars ("|"). One alternatives in the list must be chosen to create a syntactically valid statement.

2.1.1. Type Declarations SIMPAS provides two new type declarations. One is used to declare temporary entity types in the simulation, and the other is used to declare type representing lists of entities. Syntax diagrams for these type declarations are:

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

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

For reasons outlined in [3] these declarations are required to be simple (i. e. of the form <identifier> = <queue declaration>) and only allowed in the global type part of the SIMPAS program.

2.1.2. The include Statement SIMPAS uses a symbolic library to provide a transportable way maintain a library of simulation support routines. The *include* statement indicates which sections of the symbolic library are to be included in the current program. It is found at the start of the procedure, event, and function declaration part of the main program and has the format:

```
include <section-name-list>;
```

2.1.3. Event Declarations An event declaration looks exactly like a procedure declaration except that the keyword *event* replaces the keyword procedure. Event, procedure, and function declarations can be intermixed except that events must be global and cannot be declare local to a procedure, event, or function.

2.1.4. Simulation Extension Statements These statements can be used in a SIMPAS program anywhere that a Pascal statement would be acceptable.

The start simulation statement begins execution of events under control of the simulation event sequencing routine:

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

Schedule and reschedule statements insert event notices into the simulation event set:

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

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

The cancel, delete, and destroy statements remove a previously scheduled event notice from the sequencing set:

```
cancel <ev_ptr>

destroy <ev_ptr>

delete <ev_ptr>
```

Insert and remove statements put or take entities from a queue:

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

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

A forall statement allows a specific statement S to be repeated for all members of a queue:

```
forall <e_ptr> in <queue> [in reverse] do S
```

Create and destroy statements serve the functions of new and dispose but ensure that preprocessor defined attributes of the entities are properly initialized:

```
create <entity-list>
destroy <entity-list>
```

The initialize statement sets a queue variable to represent an empty queue:

```
initialize <queue-list>
```

2.1.5. Watched Variables

SIMPAS II provides automatic statistics collection features similar to those of SIMSCRIPT II.5 [13]. A variable is marked for statistics collection by declaring it to be of a special type that we refer to as a "watched type". A variable declared with a watched type is called a "watched variable." Watched variables can be used almost everywhere normal variables can be used. The only difference is that whenever the value of a watched variable is changed, statistics associated with the variable are updated as well. For further details see [3].

The following statements are associated with watched variables in SIMPAS:

```
clear <watched_variable-list>
sreset <watched_variable-list>
display <watched_variable-list>
regen <watched_variable-list>
recalc <watched_variable-list>
```

2.1.6. Summary The SIMPAS extensions to Pascal are intended (with minor restrictions) to appear as natural extensions of Pascal and to be usable wherever a standard Pascal statement could appear. The preprocessor´s task is to translate the extension statements into standard Pascal.

3. Overall Structure of SIMPAS

Even though Pascal can be compiled in a single pass, SIMPAS must be a two pass processor. One of the services of the preprocessor is to create the declarations for the simulation event set. The structure of this set depends on the events declared and their arguments, and the latter are not known until the entire source program has been examined. Therefore both SIMPAS I and II consist of two passes.

During the first pass, the input program is read and parsed. Output from this pass is placed in a temporary file. Comments are removed and SIMPAS extension statements are translated into standard Pascal statements. The include statement is parsed and list of section names to be included from the symbolic library file is created. Tables containing the names of events (and their arguments), queues and queue members are constructed. Markers are placed in the temporary file at the beginning of the global constant, type, variable, procedure declaration, and main procedure parts of the program.

The second pass moves the temporary file to the output, stopping at each marker to insert additional declarations. These declarations are either generated by the preprocessor or read in from the library file.

3.1. Implementation of SIMPAS I

In the initial implementation of SIMPAS, our plan was to examine as little of the input program as possible and depend on the host-system Pascal compiler to catch most errors. (We felt that completely parsing the input Pascal would make the preprocessor too slow to be usable). To do this we implemented a scanner that was called with a set of "interesting" token types. The scanner read the tokens of the input program and placed them in the temporary file until a token in the interesting set was found. The scanner then placed the interesting token in a global variable and returned to its caller for appropriate action. The intent was that the scanner could "stream" all uninteresting input into the temporary file, and the preprocessor need only examine the SIMPAS extension statements themselves. The principle loop of the preprocessor could then be coded something like the following:

```
{"parse" body of SIMPAS program}
while not eof(input) do
begin
    scan([schedulewd, insertwd, removedwd, . . .]);
    case currtoken.kind of
        schedulewd:  expandschedule;
        insertwd:    expandinsert;
        removewd:    expandremove;
        . . .
    end; {case}
end; {while}
```

This is possible because each SIMPAS extension statement is identifiable from its first keyword.

The actual SIMPAS I "parser" is more complicated than illustrated above. For example, to insert initialization code, SIMPAS had to be able to find the beginning of the main procedure. This meant that the begin-end structure of the program had to be known to the preprocessor, and this implied that begin, end and case had to become "interesting" tokens. Additionally, the restriction that an event could not be declared local to a procedure meant that procedures and functions as well as nested procedures and functions had to be known to the preprocessor.

These problems were handled by constructing a simple recursive descent parser for the structural portions of Pascal. An outline of the basic SIMPAS I parser is shown below. For simplicity, we have eliminated most error handling details from this code fragment. The key routines are the procedures "scan", "procscan" and "blockscan". Scan is the scanner procedure, procscan is called at the start of each procedure, function, or event, and blockscan recursively examines begin-end and case-end blocks. Blockscan also calls the appropriate expansion routines when SIMPAS extension statements are encountered.

```
procedure procscan;
{ enter at start of procedure, function, or nested event }
begin  { proc procscan }

   {nested event declaration}
   if currtoken.kind=eventwd then error;

   {output function, event, or procedure keyword}
   putout( currtoken );

   { look for next procedure, function, event, forward,
     or begin }
   scan( [ procwd, funcwd, eventwd, beginwd, forwardwd ] );

   if currtoken.kind = forwardwd then
      { forwarded function or procedure }
      { output the "forward" }
      putout( currtoken )
   else
      begin
         { process nested proc or func }
         while (currtoken.kind <> beginwd) do begin
            procscan;
            scan( [procwd, funcwd, eventwd,
                  forwardwd, beginwd] );
         end;  { while }
         { examine the body of the current procedure }
         blockscan;
      end;  { else }

end;  { proc procscan }

procedure blockscan;
{ process begin-end, case-end blocks recursively
  call expansion routines to expand SIMPAS extension
  statements}
begin

   {output current block opener (either case or begin)}
   putout( currtoken );

   {blockset consists of the reserved words: start, cancel,
        delete, schedule, reschedule, begin, case, destroy,
        forall, insert, remove, function, procedure, event,
          and end }
     scan( blockset );

     while (currtoken.kind <> endwd) do begin
        case currtoken.kind of

           startwd, cancelwd, deletewd,
           destroywd, reschedulewd,
           schedulewd, forallwd, insertwd, removewd:
                  { call appropriate "expansion" routine
                    to expand this SIMPAS statement into
                    standard Pascal }
                    . . .

           { recursively examine nested blocks }
           beginwd,
           casewd : blockscan;
        end; { case }

        scan( blockset );

     end;  { while }

     { output the "end" }
     putout( currtoken );

  end;  { proc blockscan }


  begin { part of main procedure of preprocessor }

     . . .

     { examine global const, dec, var parts }
     decscan;

     { process procs, funcs, events at program level }
     scan( [ procwd, funcwd, eventwd, beginwd ] );

     { mark start of procedure declaration part for second pass }
     writeln( tempfile, flagchar );

     while (currtoken.kind in [ procwd, funcwd, eventwd ] ) do begin

        {all events should be found here unless nested inside a procedure
        if (currtoken.kind = eventwd) then expand_event
            { process function or procedure }
            else if currtoken.kind in [procwd, funcwd] then procscan;

        scan( [procwd, funcwd, eventwd, beginwd ] );
        {finding begin or eof indicates no more procs,
         funcs, events to examine}
```

```
   end; { while }

   { should have found begin of main procedure at this point}

   if currtoken.kind <> beginwd then error
   else
   begin
        { flag beginning of main program }
        write( tempfile, flagchar, mainflag:1 );

        { process body of main }
        blockscan;

        { check end of program }
        scan( alltokenset );
        if ( currtoken.kind <> periodtoken ) then error;
        putout(currtoken);

   end;

   . . .

end.
```

The procedure decscan (not shown here) is responsible for marking the beginning and end of the global constant, type, variable, and procedure declaration parts of the program, as well as expanding the queue and queue member type declarations. All other declarations are ignored. Thus the SIMPAS I preprocessor is unaware of the names and types of user declared variables. The result is that the SIMPAS I preprocessor cannot generate code dependent on the type of a variable in a SIMPAS extension statement.

3.2. Implementation of SIMPAS II

SIMPAS II is based around a locally-least-cost error-correcting LL(1) parser [9] that we modified from [15]. Most Pascal keywords are echoed by the parser directly into the pass 1 temporary file as a "default" semantic action. However, as soon as a SIMPAS extension statement is encountered, appropriate semantic action routines are called. The first of these normally turns off the default semantic action so that the expanded Pascal representing the SIMPAS statement can be output instead. A complete symbol table is maintained. It includes the types, variables, procedures, functions, and events declared in the current program. This version of SIMPAS corresponds in many ways to the front end of a Pascal compiler.

If it were not for the presence of "watched variables", most expressions could be passed through SIMPAS II unchanged. However, each time a watched variable is used, the preprocessor must output a modified version of the expression containing the watched variable. Since the only way to determine if an identifier is a watched variable or not is to look it up in the symbol table, semantic processing of each expression in the SIMPAS program is necessary. The result is that echoing of tokens into the temporary file by the parser is more often off than on. Further details of the SIMPAS II implementation are given in [3].

4. Comparison of SIMPAS I and II

4.1. Features Supported SIMPAS I is limited in two primary respects: (1) It only understands a little Pascal syntax and (2) It is unaware of the types of user variables found in SIMPAS extension statements. The first limitation meant that the body of a *forall* loop could not be an arbitrary Pascal statement, since the SIMPAS I preprocessor had no way of recognizing the end of an arbitrarily complex Pascal statement. (Code to update the *forall* loop index needs to be placed at the end of the *forall* loop body.) To solve this problem, the SIMPAS I preprocessor required the body of a *forall* loop to be surrounded by a *begin*-*end* pair, even if the body consisted of only a single statement.

The second restriction is more severe because it meant that many of the statements described in Section 2 could not be implemented or had to be implemented in space inefficient ways. For example, the *initialize* q statement is translated by SIMPAS II to a call on a procedure i_<queue> where <queue> is the type of q. (This procedure is generated by the preprocessor as a side effect of processing the *queue* declaration.) In the SIMPAS I version, the type of q was unknown during preprocessing, so the preprocessor could not determine which i_<queue> call to generate. Instead the user had to call i_<queue> directly. For similar reasons, SIMPAS I did not support the create and destroy <entity> statements, nor was SIMPAS I capable of supporting watched variables. These services were provided by subroutines called by the user.

Another problem with SIMPAS I occurred in expansion of *insert* and *remove* statements. Since neither the type of the <entity> to be inserted nor of the <queue> to be inserted in were known to the preprocessor, no compatibility checking could be done at preprocessing time. Worse yet, since the preprocessor did not know the type of <queue>, it could not call a procedure to do the insertion or removal.* Instead the insert and remove code had to be generated inline at an expense of about 10 Pascal statements per insert or remove statement.

SIMPAS I also had trouble recognizing the ends of expressions in SIMPAS statements. For example, consider the expression between "at" and "else" in the statement:

```
if exponential_arrivals then
    schedule arrival
        at expo(arrival_rate[udisc(1,3,2)],stream)+3.0
else
    schedule arrival delay 50;
```

This expression must be picked up and saved so that it can be passed to the event set insertion routine. But how does the SIMPAS I preprocessor recognize the end of this expression? The basic rule we eventually came up with was "pick up all tokens until finding a semi-colon, end, or else token". This was hardly a satisfying solution.

A final problem with SIMPAS I was error correction and recovery. If an error was detected while expanding a SIMPAS extension statement, the expansion routine terminated and returned control to the parser, which in turn called the

Because of the strong typing in Pascal, each insertion or removal procedure had to be dedicated to a particular queue and entity type pair. Use of records with variants as a representation for queues and queue members was discarded because this would not supply a sufficiently secure implementation method [3]. 1] scanner with the current set of interesting tokens. Misplacement of critical tokens (e. g. the begin of the main program) was treated as a fatal error.

These problems were corrected in SIMPAS II. For example, the forall loop was added as a new statement type in the SIMPAS grammar:

```
<STMT> ::=    . . . /* other statements */ |
              forall <VAR> in <VAR> do <STMT> |
              . . .
```

Similarly, since SIMPAS II maintains a complete symbol table, the types of all variables in an insert or remove statement are known and can be checked for compatibility. Type specific insertion and removal routines for each queue type can be generated and called when queues of that type are referenced in an insert or remove statement. This avoids the inline expansion of these statements as in SIMPAS I. Since expression syntax is known, the problem of recognizing the end of an expression disappears. Finally, error correction is done by the parser so that the semantic routines need never deal with a syntactically erroneous program.

4.2. Program Complexity

SIMPAS I is about 4900 lines of Pascal, while SIMPAS II is about 7200 lines of Pascal. The difference in size is made more apparent when examining the size of the object programs (see Table I). One can see that SIMPAS II is nearly twice as large as SIMPAS I.

version	code	initialized data	uninitialized data	total
I	47104	22528	7708	77340
II	75776	23552	47060	146388

Table I
Size of SIMPAS I and II Preprocessors
(in bytes, for a VAX/11-780)

SIMPAS II is also significantly slower than SIMPAS I (see Table II). Our original intuition that SIMPAS I would be faster than a grammar based version is seen to be correct. However, SIMPAS II clearly performs much more work than SIMPAS I so that this comparison is not completely fair. Each version spends about 80% of its time in the scanner and the primary execution time differences between the two versions is mostly due to the more complex scanner used by the FMQ parser.

Tables III and IV show a division of the 4900 lines of SIMPAS I and the 7200 lines of SIMPAS II into modules.

SIMPAS Program Length	Execution times (VAX 11/780) SIMPAS I	SIMPAS II
166 lines	7.7s	10.3s
1310 lines	35.2s	54.2s
2989 lines	64.0s	100.8s

Table II
Execution times of SIMPAS I and II

function	lines	percent of total lines
global declarations	414	8%
scanner	752	15%
parser	554	11%
semantic and code generation routines	1339	27%
pass 1&2 & main program	1222	25%
utilities	627	13%
Total	4908	
Source library file	906	

Table III
SIMPAS I Module Sizes

function	lines	percent of total lines
declarations	610	8%
scanner	501	7%
FMQ parser, error corrector, etc.	1562	22%
semantic and code generation routines	2575	35%
pass 1&2 & main program	958	14%
utilities (symbol table, input and output, dump routines)	1030	14%
Total	7236	
Other supporting routines:		
SIMPAS Grammar	316	
FMQ Parser Generator	4341	
Source library file	1495	

Table VI
SIMPAS II Module Sizes

The semantic and code generation routines of SIMPAS II are more complex because of the additional semantic actions required for watched variables, symbol table maintenance and the like.

Since we did not implement the FMQ parser and parser generator (these were essentially unmodified from that of [15]), the lengths of these pieces of code should be subtracted from the total length to compare the implementation effort of each version. When these 1800 lines (code plus declarations) are subtracted, it appears that the amount of new code implemented for each version is about the same (4900 lines versus 5400 lines). However, SIMPAS II is a more complex program than SIMPAS I. For example, a typical SIMPAS statement is parsed in the following way. Several intermediate semantic routines are called that pick up portions of the SIMPAS statement and record information on the semantic stack. Toward the end of statement parsing, a final semantic routine is called that examines the semantic stack and outputs the appropriate Pascal code. If an error is now detected, it can be a non-trivial task to determine where the error occurred. While this organization is typical of most syntax directed compilation techniques, it is more complex than the SIMPAS I parser.

On the other hand, for obvious reasons, it is easier to extend SIMPAS II than it is to extend SIMPAS I. In general, all that is required to add a new statement to the language accepted by SIMPAS II is to update the grammar, rebuild the parse tables, add the new semantic routines and proceed. Extending SIMPAS I requires that the extension statements fit within the confines of the SIMPAS I parser. One of the reasons for implementing SIMPAS II was that SIMPAS I had reached the limits of the language that it could process. To continue our investigations into simulation and queueing network modelling software implementation, it was clear that we needed a more powerful preprocessor than SIMPAS I.

5. Use of SIMPAS

SIMPAS I has been use in the Computer Sciences Department at the University of Wisconsin-Madison from Spring 1979 to Summer 1981. It has been used for a variety of computer system related simulation projects (cf. [2,19]) as well as for teaching a graduate level computer science course in simulation.

SIMPAS II was completed during the Fall of 1981, so we have had less experience with its use. However, we have converted five SIMPAS I simulations to SIMPAS II and counted the number and type of SIMPAS extension statements that were used in each simulation. The lengths and number of events declared in each of these simulations is given in Table V. (For a discussion of the simulations themselves see [5]).

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

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

Table V
Sizes of Example Simulations

would be regarded as a large simulation program and each of the next three would be considered typical of simulations designed to explore computer architecture or operating systems performance questions.

5.1. SIMPAS Statement Usage Counts of the number of SIMPAS extension statements used in each of the simulations are given in Table VI.

The table shows that the number of insert and remove statements is small. Inline expansion of these statements would not increase program length significantly.

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 VI
SIMPAS Extension Statement Use

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> <identifier> that is not a create, destroy or initialize statement. Examples are clear or display.

Preprocessing Pascal

All the programs (except for PROTOC) use a substantial number of <command>´s. In SIMPAS I, these commands are implemented by procedure calls, while in SIMPAS II, the <command>´s look like SIMPAS extension statements. The primary advantage of the latter approach is that the user need not remember the type of the variable to be cleared, initialized etc. SIMPAS II remembers this information for the user. To do so, SIMPAS II must parse and remember all variable declarations in its symbol table. This is a non-trivial task.

5.2. Statistics, Queue and Queue Member Declarations The extension types declared and number of variables declared of each type in the five example simulations are given in Table VII. The counts of declared variables do not include declarations of pointers to objects that contain queues or watched variables. Thus there are watched variables declared in NETPAC and there are queue variables declared in

	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 VII
SIMPAS Extension Types Use

P5 but these are declared within records that are only accessed by pointer variables.

We note the small number of assignments to watched variables in these programs. This is in part due to the straightforward transliteration of the programs from SIMPAS I to SIMPAS II, but it seems to imply that statistics are collected and then assigned to a watched variable. For example, the number of preemptions a job in service at a CPU might be counted during the execution time of the job and then this total assigned to a watched variable when the job departs. The explicit assignment mechanism for watched variables in SIMPAS I (a procedure call) seems as well suited for this type of statistics collection as does the automatic statistics collection feature of SIMPAS II. The latter, as we have pointed out, requires significant implementation effort.

5.3. Summary

The SIMPAS I implementation has been used for more than 100 simulation programs. Error recovery is not as good as in SIMPAS II, but most users report that they rarely encountered a SIMPAS error; most errors were those found by the host system's Pascal compiler. The primary advantage of SIMPAS II is the implementation of watched variables. These can be replaced by the explicit statistics collection routines of SIMPAS I. The user, therefore, is likely to be unwilling to pay the increased preprocessing costs associated with SIMPAS II.

From the implementor's point of view, the SIMPAS II implementation is much simpler to extend than SIMPAS I. However, SIMPAS II is a more complicated program and it is fair to say that it is not currently as reliable as SIMPAS I. From these considerations, we would say that SIMPAS I met its implementation goals and at a much lower cost for benefit derived than does SIMPAS II.

Both implementations are substantially more complicated than the implementations of coroutines or quasi-parallel programming in Pascal (e. g. [8,14]). Some of these packages required only a few man-weeks to implement. The difference is the cost of making SIMPAS portable.

6. Concluding Remarks

In general, the preprocessing of a highly-structured language such as Pascal is more complicated than preprocessing a language such as FORTRAN. Unless one is willing to create inline code for the extension statements, it appears that a successful preprocessor for Pascal must be aware of the types of variables present in the extension statements. Otherwise, the preprocessor will often be unable to determine the correct translation of a statement. If the extension statement is at all complex (such as the insert and remove statements of SIMPAS) it will be advantageous to generate a procedure call instead of the inline code. Additionally, if the extension statements include new control structures (such as the forall statement in SIMPAS), the preprocessor must be aware of Pascal statement syntax. Thus, a successful preprocessor for Pascal must be nearly a complete front end for Pascal.

If one is willing to accept a less than perfect implementation of the extension constructs, simpler preprocessors can be created. SIMPAS I does provide a usable service, at the expense of some "syntactic sugar" (e. g. *clear*, *initialize*) and increased output program size. From the user's standpoint, the primary advantage of SIMPAS II over SIMPAS I is the facility for automatic collection of simulation statistics. Given the difference in program complexity between the two versions, it is not clear that this advantage has completely justified the implementation effort. Finally, when comparing the implementation effort required to make SIMPAS as portable as it is to the effort required to implement coroutines or quasi-parallel programming extensions to Pascal, it appears that the cost of making SIMPAS portable has been very high in relation to the results obtained.

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

[3] Bryant, R. M., M. B. Abbott, J. R. Bugarin, and B. S. Rosenberg, "Implementation of SIMPAS," Computer Sciences Department Technical Report, University of Wisconsin--Madison (in preparation, 1981).

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

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

[6] Bryant, R. M., "SIMPAS 5.0 User Manual," Computer Sciences Department Technical Report #456, University of Wisconsin--Madison (November 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] Deminet, J. and J. Wisiniewska, "Simpascal," Pascal News, pp. 66-68 (March 1980). Newsletter of the Pascal User´s Group.

[9] 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).

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

[11] Kernigham, B. and P. Plaugher, Software Tools, Addison-Wesley (1976).

[12] Kernigham, B. W., "RATFOR--A Preprocessor for a Rational FORTRAN," Software--Practice and Experience 5, 4, pp. 395-406 (Oct.-Dec. 1975).

[13] 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).

[14] 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).

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

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

[17] 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).

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

[19] 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.