1-1-1980

Design and implementation of a Pascal based simulation language.

Robert George Wilder

DESIGN AND IMPLEMENTATION OF A PASCAL BASED SIMULATION LANGUAGE

by

Robert George Wilder

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Industrial Engineering

Lehigh University

1980

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science

4/7/80
(date)

[Louis Plebani]
Professor in Charge

[G. Kane]
Chairman of Department

TABLE OF CONTENTS

ABSTRACT

This thesis investigated the feasibility of simulation languages based on the programming language Pascal. The major effort of this investigation involved the coding and testing of a discrete event language. The specifications of this language closely followed those of the FORTRAN based GASP IV, deviating only to include the constructs and data structures occurring in Pascal but missing in FORTRAN.

Due to the design of the Pascal compiler, the implementation of this language required a preprocessor. This had a considerable impact on the appearance of the language to the user. He must code a pseudo program in which he codes not only the procedure that will process the different events, but also the setting in which the simulation is to occur. This setting first of all defines the global variables and variable types that will be used, such as the names of the different events and the structure of the event records. It is also necessary to define other information which the preprocessor uses to manage the histograms and other statistics and to produce the type of output the user desires. The preprocessor then produces a valid Pascal program which

is compiled and executed.

Further research will, of course, need to be done in the incorporation of more sophisticated types of simulation languages into a Pascal environment and in the more efficient implementation of those constructs already included. Two of the considerations that will guide this future research were identified in this thesis.

The first consideration is the tradeoff between space (the amount of memory that the program needs to execute) and time (how quickly it executes), especially as it is demonstrated in the use of pointers versus the use of arrays. More thought must be given to when dynamic allocation of storage is necessary or advisable. Perhaps two versions of the preprocessor should be created. The first of these would identify the amount of storage actually used but would execute slowly. The second would use this information to allocate the necessary storage and would execute relatively quickly. This would be especially useful if the program is to be used often.

Secondly, careful consideration must be given to the type of statistics that will be collected. This was

discovered to have a significant impact on the implementation of the features of the language. This impact should be identified in the design phase so that effort is not wasted.

INTRODUCTION

The development and use of models and simulations has roughly paralleled the growth of computer technology in the past three decades. This should be no surprise, since the increasing computational power of the computer has enabled more and more problems to be simulated effectively.

The first digital simulations were written in machine or assembly languages, but this was extremely cumbersome, especially if modifications were necessary in the model. Compiler languages were then used, but the translation of a model into a language remained time consuming and expensive. There were two developments from this situation. First, special programs were designed to solve specific classes of problems. In that way much of the design and coding of a program was already completed, as long as the problem was of the correct type. Second, general purpose simulation languages were developed. They could be used on many different problems, but did not eliminate as much of the design or coding as problem specific languages did.

Today there are a great many general purpose simulation languages, and these can be classified by a number of different characteristics. For example,

continuous simulations are those in which the states of the simulation change continuously, whereas in discrete simulations the states change at discrete points in time. Simulation languages can be further differentiated by their point of view - the way a system to be simulated is viewed. The event scheduling approach segments a system into points of time at which the state of the system changes (the events). The activity scanning view segments a system into activities, which are the ways that the system states change. The process interaction view traces the progress of an entity through the many activities and events in a system.

The point of view is an essential part of a simulation language (although the distinctions between the different views appear vague at times). It not only provides a frame of reference to view the problem from, but also determines the building blocks the user will have to construct the model. These, in turn, aid the user in both the decomposition of the system into its functional parts, and in the formulation and translation of the model.

Contrast this with an assembly language or compiler language simulation. The compiler language gives no guidance at all in the formulation of a model, while a

simulation language forces the user to determine the events or activities involved. Much of the coding in the compiler language is concerned with how to keep the list of events correct, or how to keep the bookkeeping for the statistics accurate. The simulation language, on the other hand, allows one to focus his attention on the definition and coding of only the events or activities involved in the model. This provides a great savings in time and effort, and the conceptual guidance offered should make the model more accurate so that less debugging is required.

PASCAL AND SOFTWARE DESIGN

The growth of simulation languages has also reflected the development of software design techniques. Ten years ago, the major emphasis of computer programming was on the end product, namely the program output, rather than on a clear and logical path to achieve that end. As a result, programmers would start to write a program by coding some detailed procedure only to discover that their basic foundation was faulty and needed to be redone. They continued this coding and patching process until they had a program that ran, but was often difficult to understand.

This is not possible today. The complexity of today's problems and the time constraints frequently encountered demand that programs or systems be designed and coded by teams of people. Each person's task must therefore be more narrowly defined from the beginning. It is also necessary that programs be easy to maintain. All too often the original problem changes slightly, or an error is discovered, and it is not uncommon for the person making the change to be someone other than the original writer. It is imperative, then, that the program be easy to understand, or the modifier could spend as much time trying to understand the program as

it took to write it.

The techniques and languages in use today have been developed to make a program easier to design and code by allowing one's thought processes to be reflected more easily. They also make programs more easy to maintain because of the simplicity and readability of the code. Some of those techniques most pertinent to the design of programs are the following:

1) Top down design - beginning the design of a program with the highest level of control;

2) Modular design - the design of a program module by module; and

3) Functional decomposition - the process of subdividing a program into smaller, more manageable, modules or functions.

However, the most widely publicized tool, and the one pertaining most to coding, is structured programming. Structured programming is based on the structure theorem which states that any "proper program" (a program with only one entrance and one exit) can be constructed using only three control structures. The simplest of these is the block or sequence. It consists of a block of statements executed one after another. The next structure is the IF-THEN-ELSE. This executes one of two alternatives, depending on whether a specified condition is true. Finally, there is the DOWHILE or DOUNTIL. These

repeatedly execute a sequence of statements while a condition is true (or until it is true). Notice the absence of a GO TO statement. Some people feel that GO TO statements make the flow of the program logic difficult to follow, and in addition do not model the thought process accurately enough. For example, when one codes IF (condition) GO TO (statement), what is actually intended is IF (condition) THEN (execute these statements).

Pascal was designed with structured programming in mind. The basic control structures of Pascal include those mentioned above: the sequence, the IF-THEN-ELSE, the DOWHILE (WHILE condition DO statement), and the DOUNTIL (REPEAT statements UNTIL condition). In addition, Pascal has the GO TO statement, a FOR statement which repeatedly executes a statement while a progression of values is assigned to a control variable, and a CASE statement which executes one of many alternatives depending on the value of a variable. The CASE statement was included so that large nested IF-THEN-ELSE statements may be avoided in some instances.

Other characteristics of Pascal include its block structure (with both variables and procedures being defined either locally to a procedure or globally), its

allowance of recursive calls, and the flexibility of its data structures. Pascal is not restricted to the standard data structures of integer, real, character, boolean, and array. It also has files, sets, pointers (variables that point to the location of an item), records (a collection of named attributes), and scalar types (an ordered sequence of identifiers or keywords).

The proponents of Pascal point out that coding in this language guides one's thought processes in designing a program in much the same way that a simulation language guides one's thoughts in the design of a simulation. It is only natural, then, to combine the two, for simulation languages lend themselves easily to this structured approach. The different functions or procedures are precisely the processing of the different events. The basic control structure is extremely simple, consisting of selecting and processing the events until the simulation is finished (by whatever stopping rule is used). In structured terms the main procedure might look like this:

```
Initialize simulation variables
DOUNTIL finished
   Get next event
   Process event
END-DO
Write output.
```

The design of Pascal lends itself easily to coding and performing this procedure. If Pascal is truly one of the languages of the future, simulations are going to be written in it. This project is a first step in that direction.

DESIGN CONSIDERATIONS

The first step in constructing a simulation language is to identify the type of simulation and its point of view. This project is concerned only with discrete, event scheduling simulations. Instead of simulation time being advanced in fixed increments, time is advanced to the time of the next event. This illustrates more clearly what is actually occurring in the simulation, and simplifies some of the coding. This decision was also motivated by an attempt to follow the specifications of the GASP IV language as closely as possible since GASP IV was straightforward, simple to understand, and the most familiar language to those involved in the project. This had an added advantage of allowing more time to be devoted to studying the problems of implementation.

Since the language was based on Pascal rather than the FORTRAN used in GASP IV, a number of changes were introduced. These changes were incorporated into the project to determine the effect that the increased flexibility and naturalness of Pascal would have on a simulation language.

For example, one of the major differences between Pascal and FORTRAN is in the types of data structures

allowed. An event TYPE could be a scalar type of ARRIVAL and ENDOFSERVICE, and an event would then be a record with an event indicator (ARRIVAL or ENDOFSERVICE), time of the event, time it entered the system, the particular server it would have, and whatever else might be needed to describe the event. These events could be linked together in a list with the use of pointers, thereby allowing storage to be allocated dynamically.

Of course, this additional flexibility has some side effects. The different possible types of event records and lists make the list management routines (those that handle the inserting and removing of the events) considerably more complex. A different REMOVE procedure is now needed for each list type, and a different INSERT procedure is needed not only for each list type, but also for each field of the event record that the event list is ordered upon. The requirement that these types be declared before the list management routines means that either the user codes these or that a preprocessor codes them.

The collection of statistics is also made considerably more complex. Separate pointers and lists are needed for each statistic collected. It is also difficult to refer to the lists and statistics unless

tables are generated which assign numbers to the different names and types. These drawbacks must be weighed, though, against the flexibility and ease of writing a self-documenting program that Pascal offers. The clarity of Pascal's control structures, coupled with the multitude of data structures and descriptive names available, enables the user to code a program that much more closely reflects his thought processes and that is easier to read. These advantages are clearly worth the added complexity in implementation of the language.

THE PREPROCESSOR

The necessity of a preprocessor has already been alluded to in the previous section. The early stages of this project attempted to follow GASP IV fairly closely by having the user code an events procedure that would be combined with various subroutines to produce the simulation program. This eventually proved impossible because of the linkage problems involved (How, for example, could the list handling procedures be written beforehand when the attributes of the event are not yet known?).

Even if this were not a problem, it would still not be possible to just code an events procedure. Two methods were investigated to declare an events procedure that had not yet been written, and each one failed. The first method was to declare that procedure to be of type EXTERNAL. The problem then arose that a Pascal procedure, unlike a FORTRAN subroutine, can not be compiled by itself. Several attempts were made at compiling it as part of a dummy program and then extracting the procedure from the program but these attempts failed. An additional problem was created by the fact that the procedures are not known by their procedure names, but

by a compiler generated name such as PROC0002 or PROC0003.

The second method was to write a dummy events procedure in the program, and to override this with the one written by the user. This is the method used by GASP IV, but in Pascal the same problems as before were encountered, and so the whole idea was abandoned.

These considerations led to the concept that the user would code not just a procedure, but also the context of the simulation. This pseudo program would be input to a preprocessor which would extract the necessary information from this program skeleton, code the required supporting procedures, and thus create a valid Pascal program.

The preprocessor is therefore divided into two major parts - a parsing part and a program writing part. The parsing part identifies the standard Pascal constructs and the additions to these that make up the language. This information is used to build the lists and tables that enable the program writing part to assemble the supporting procedures required by the program.

The first set of tables is used to hold information about the way the lists are constructed. LISTTYPES is an array that contains the names of the record types of

the entries that make up a list. The names of the different lists for each particular list type are found in the table LISTS. The entries of a list can, in turn, be ordered on many different fields. The name of each field that a list is ordered on (and the word "DUMMY" when a list is not ranked on a field) is kept in the array LISTFLDS, and the name of the respective insert procedure is stored in LISTNAM.

The second set of tables consists of a few work areas. For example, the user currently has the ability to dump the values of selected fields at the end of the simulation. The names of these fields are stored in the array DUMPS, and the names of their respective lists are stored in DUMPLISTS. TOTALLIST is used to assign a number to each list for the purpose of collecting statistics. LOW, HIGH, INTERVAL, and TITLE pass information from the HISTOGRAMS declaration to the resultant program, and SAVES is used to hold identifiers until it is known if they are list names.

The preprocessor reads the input program character by character, combines these characters into symbols (operators, numbers, reserved words, or identifiers), and syntactically evaluates them by the parser. The procedures which handle the construction of a symbol

are NEXTCHAR (which reads the next character from the input stream), GETCHAR (which writes the previous character to the buffer and calls NEXTCHAR), PUTBUF (which handles the placing of a character into the buffer), WRITEBUF (which writes the contents of the buffer onto the output file), and GETSYM (which contains the logic to get the next valid symbol).

The parser needs two output files, AAA and BBB. As the parser examines and formats the syntax of the input program, it writes the PROGRAM, LABEL, CONST, TYPE, and VAR declarations to AAA, and writes the procedure and function declarations to BBB, which acts as a temporary holding area until the preprocessor is able to construct the proper procedures. The program writing portion will eventually combine the two parts onto AAA.

The parser closely follows the syntax charts found in the Pascal User Manual and Report.[1] Most of the purposes for each procedure are obvious. For example, FORMATLABEL parses a LABEL declaration, and CONSTANT parses a constant. Other procedures have been added to manage the additions and modifications to Pascal that make up this simulation language. They will now be

---

[1] Kathleen Jensen and Niklaus Wirth, Pascal User Manual and Report, pp. 116 - 118.

described in further detail.

A number of procedures have been added to assist in the collection and reporting of statistics. FORMAT-HISTO extracts the information from the HISTOGRAMS declaration to fill in the tables LOW, HIGH, INTERVAL, and TITLE. It will pass this information to the output program by means of the initialization procedure, ZZZZZINITL, and by means of the declarations produced by FORMATVAR. FORMATDUMP extracts information from the DUMPS declaration to fill in the arrays DUMPLIST and DUMPS. This information details the list record fields that the user wishes to be printed for each record left in the list when the simulation is over. ASSIGN is used to assign a place in TOTALLIST for the list names declared in FORMATVAR and GETTYPE. This also assigns to them a number which is used in the collection of statistics. RECOVER returns the number that was assigned by ASSIGN. This is used when the INSERT and REMOVE commands are editted in the user's program.

The other procedures incorporate modifications to Pascal. GETTYPE parses a data structure type (found on the right side of a TYPE or VAR declaration). In addition to the standard Pascal types, a new type LIST

is defined. This type is used to declare the names of the lists and the record types that are linked together. It is allowed only in the VAR declaration. An example is - QUEUE: LIST OF EVENTTYPE. FORMATTYPE parses the TYPE declaration. A major modification of the standard declaration is the addition of a subblock, LISTTYPES ARE (record definitions) END. This is the way in which the record types of the entries in a list are declared. It is necessary to set them apart in this manner because pointers to these records and an array used to identify the first and last records of a list must be defined in terms of this record type. A boolean variable PUTPTRS must also be set so that when GETTYPE is called the NEXT and PRIOR pointer declarations may be inserted into the record declaration. FORMATTYPE also adds the TYPE declarations that the preprocessor needs. FORMAT-VAR parses the VAR declaration and inserts needed variables. The array SAVES is used to hold identifiers until the type has been determined. If the variables are lists the identifiers are stored in the LISTS array. STATEMENT parses a Pascal statement. However, special action must be taken when an INSERT or REMOVE command is encountered. A unique procedure name must be assigned and the proper procedure must be written. The

name is constructed by adding the first five characters of the list name to a unique two character combination. The last three characters are "PUT" if the procedure is an INSERT, "GET" if it is a REMOVE. This process is managed by the procedures INSERTING and REMOVING.

The program writing part writes all the supporting procedures onto file AAA, and then adds the procedures that were editted and written onto file BBB. The procedures it uses are rather straightforward in their function. FIRSTPROCEDURES writes out the standard procedures that need to be declared first onto file AAA. These include ZZZZZTIMST, ZZZZZADDTO, and ZZZZZSUBFM. WRITEREMOVES writes the necessary remove procedures that have been constructed. SIMILARPROCEDURES then writes out other procedures common to all simulations: COLCT, HISTO, WRITEHISTO, ZZZZZDUPDT, ZZZZZINITL, OUTPUTT, RANDOM, and the statistical distribution functions. WRITEINSERTS writes the constructed insert procedures. GETNXTEVNTPROC writes the procedure that will get the next event from the event list. Finally, COMBINEFILES copies the final parts of the program from BBB to AAA.

THE RESULTANT PROGRAM

The basic structure of the program produced by the preprocessor can be seen in its main procedure:

```
BEGIN
   ZZZZZINITL;          (simulation initialization)
   INIT;                (user's initialization)
   REPEAT
      NEXTEVENT;        (gets the next event)
      EVENTS;           (processes the event)
   UNTIL STOP;          (STOP must be set to TRUE
                         when the user wishes to
                         stop)
   OUTPUTT;             (simulation output)
END.
```

The core of this is the user coded events procedure, EVENTS. In EVENTS, the user specifies what is to be done with each event, and how other events are to be scheduled. The most important procedures he will need are those that handle the list maintenance. To understand them it is necessary to see how lists are treated by the program.

A user creates a list by first declaring a record type to be used in the list. For example:

```
EVENTTYPE = RECORD
               EVENTTIME,
               QUEUETIME,
               SERVICETIME,
               ENTRYTIME : REAL;
               EVENTINDICATOR : EVENTCHOICES;
            END.
```

When this is declared in the TYPE declaration, the

preprocessor defines two other types and modifies the record type. PTRxxTYPE is declared as a pointer to EVENTYPE where the "xx" are two characters assigned by the preprocessor to make the type unique. ARRAYxxTYPE is defined as an array (with two indices: FIRST and LAST) of PTRxxTYPE, and the record declaration is modified by inserting a declaration which declares NEXT and PRIOR as PTRxxTYPE. When the user then declares QUEUE: LIST OF EVENTTYPE, the preprocessor translates it to QUEUE: ARRAYxxTYPE. The first and last entries of ARRAYxxTYPE are initialized to NIL (the pointer that points to nothing), but when there are entries in the list they point to their respective entries, and the entries themselves are linked together by means of the NEXT and PRIOR pointers. The NEXT pointer of the last entry and the PRIOR pointer of the first entry are NIL.

To insert an event called EVENT into EVENTLIST (the list of events), the user codes the function call INSERT(EVENTLIST, EVENT, EVENTTYPE, list discipline, field) where "list discipline" is the manner in which the entry should be inserted, and "field" refers to the field which is used to order the list (if any). Current choices of the list discipline are LIFO (last in, first out), FIFO (first in, first out), and NEXTTIME (rank in

ascending order, on field). The preprocessor translates this to yyEVENTPUT(EVENTLIST, EVENT, list discipline, list number) where "yy" make the name unique, and the list number is generated by ASSIGN. The list type and the field are used to write the procedure yyEVENTPUT.

To remove an event from EVENTLIST and assign it to EVENT, the user codes REMOVE(EVENTLIST, EVENT, EVENT-TYPE, pointer) where pointer is a pointer to the record which the user wishes removed (at the present there is no way to assign this), or NIL for the first entry. This is translated to yyEVENTGET(EVENTLIST, EVENT, pointer, list number) with the list number assigned as before. Note that the "EVENT" in the name is from EVENTTYPE; in the insert command it was from EVENTLIST.

The list number is used as an argument in a call to ZZZZZADDTO (if an insert) or ZZZZZSUBFM (if a remove), which keep track of the number of entries in the list. These procedures modify the list ZZZZZSNTRY which links together records containing the list number, the number of entries in that list, the list name, and a pointer to the next record. These records are of type NUMTYPE.

The procedure ZZZZZTIMST is called to keep statistics on the average number in the list. This procedure works exactly like the subroutine TIMST of

of GASP IV, integrating the step function of the statistic being collected with respect to time, and dividing out by the total time when a report is generated. The insert routines also call ZZZZZDUPDT to compute the mean and standard deviation of the fields the user has specified to be dumped at the end of the simulation.

The dumps that the user wishes to see require a separate procedure for each list, and these are given names in a fashion similar to that used for the remove procedures. Calls to each of these procedures are part of the simulation generated output procedure OUTPUTT.

Other statistics the user can collect are means and standard deviations through the procedure COLCT (exactly like GASP IV's procedure), and histograms through the procedure HISTO (which also generate calls to COLCT). Reports from these procedures are also generated from OUTPUTT.

In addition to these procedures, the preprocessor also creates ZZZZZINITL, which initializes all the variables and lists in the program created by the preprocessor. It is also necessary for the user to write an initialization procedure called INIT to schedule at least the first event. The distribution sampling functions and NEXTEVENT are then written. NEXTEVENT

removes the next event from the event list, calculates the time since the last event, and sets NOW equal to the current time, as determined by the new event. NEXTEVENT has no arguments, but automatically places the values of the event into the variable selected by the user in the USE statement.

The output produced by the simulation is, for the most part, self explanatory. A few lines are written noting that the simulation is over and displaying the current time. The title, mean, standard deviation, maximum, minimum, and number of observations for each statistic collected by the user are displayed. The histograms are then displayed with their corresponding statistics. For each histogram, the observed frequency of values in that range is recorded, along with the relative frequency, the cumulative frequency up to that point, the value of the upper limit of that cell, and a bar graph showing the relative frequency as a row of asterisks and the cumulative frequency as a "C". There are 76 print positions, each representing one and one third percent of the total. The statistics collected for the values are then labelled and listed below the histogram. Finally, the statistics on the fields that were dumped are labelled and listed, with the dumps following.

RECOMMENDATIONS

A major goal of this project was to identify potential problems that would need to be examined more thoroughly in the future. The project began by developing a simple program, and proceeded by adding on other desirable features. As one might expect, this led to quite a few awkward constructions and others that were redundant or inefficient. Therefore, the first recommendation concerns the program itself. While the major points of the preprocessor are good and should be used, further design work is necessary. This should center mainly on the type of statistics and output reports that should be provided to the user, and how they are to be generated. Further discussion of some of these problems follow.

Since a major reason for the existence of simulation languages is to provide ease in writing simulations, procedures to assist in the collection of statistics will be a significant feature of the language. In fact, it would be quite useful to have certain statistics concerning list utilization collected automatically. Careful consideration must, however, be given from the beginning to the way in which this will be implemented.

For maximum flexibility there should be practically no limit to the number of statistics possible. This would require either dynamically allocated storage (pointers and lists), or an array whose size would have to be declared by the preprocessor. An array would also be faster if the indices could be managed efficiently.

The decisions about statistics will impact the types of data structures in two ways. One obvious decision is whether the user may collect statistics on real numbers only, or also integers and, if possible, scalar types. If a field is being dumped at the end, must it also be a real number? A relatively minor problem would be how to keep track of the statistics. The simplest solution would appear to be to have the preprocessor assign different numbers to different statistics. It was hoped in the beginning that this somewhat inelegant solution could be avoided, but there seems to be no other manageable method.

Attention must also be given to the effect the type of statistics collected will have on the other procedures. For example, it is really only necessary to have an insert procedure for each list type (plus an extra one for each field the user orders the list on). However, if the user is allowed to specify that certain

statistics be collected on fields being dumped at the end, the user will either have to collect those statistics himself, or an insert procedure will be needed for each list instead of each list type (to handle the special processing necessary for each list).

The basic structure of the preprocessor would most likely be very similar to what it is now. Minor differences would, of course, be necessitated by changes in the design. It should have a parsing part that collects information (and perhaps format the program), and a program writing part. The lists would be kept track of in much the same way (pointers to first and last entries), but most of the statistics would perhaps be updated in arrays. Even the histograms, although the idea of a list of cells is very appealing, would probably be more efficiently generated if indexed in an array. This would suggest that maybe a list of arrays should be used to combine the best features of both. In any case, whenever the arrays are incorporated instead of lists, a savings in execution time will probably result.

It would also be a good idea to incorporate error checking with error messages in the preprocessor so that the user would not have to understand the output program to determine where a problem had arisen. It is

the intention of the project to keep the user from coding as much as possible. This effect would be negated if he were required to debug a program he did not write.

As mentioned earlier, list searching is slower than array indexing. It might, then, be worthwhile to have a flag in the preprocessor to generate routines using either lists or arrays. A user could then write his program, execute it using the list option to get an idea of the size of the lists involved, and then create a permanent version of the program using the faster array routines. This would still be a considerable improvement over the waste of space in GASP IV which uses only one array, and reserves room for records whose size corresponds to the largest records used in the entire simulation.

Many extensions to the current capabilities of the simulation language are also envisioned. The list discipline of the INSERT command could include the ability to rank entries in descending order on a field. A pointer could also be specified so that an entry could be inserted at a particular place in a list. More parameters could also be included in the dumps. The user may just wish to see the first so many entries, or only those with a field value greater than some specified number. Another useful feature would provide NEXTEVENT

with the capability of selecting the next event from one of several lists. One useful command would be a FIND command. This procedure would find an event in a list with certain required attributes without removing that event. It would return a pointer, and thus could be used with either a REMOVE command or an INSERT command. It would also be beneficial to have the opportunity to provide a secondary method of ranking the event list, as GASP IV does.

As is evident from the above list, a great many features could be conceivably included in a second phase of this project, producing an extremely useful and versatile language. The most important recommendation is, then, to consider the design carefully - especially in reference to the desired statistics collection facilities. They will effect a great many of the problems faced during the design and implementation, and careful consideration beforehand will save a multitude of frustration.

CONCLUSIONS

In attempting to judge the success or failure of a project that only scratched the surface of a problem, it is necessary to remember the objectives of the project. Recall that the major objective was to determine the feasibility of a Pascal based simulation language. This was satisfactorily demonstrated by the preprocessor which, although implementing only the bare essentials of a language, successfully solved the basic problems of event list maintenance and statistics collection.

Another objective was to construct a language that provided guidance to the user in modelling a problem. This approach not only provided the guidance available in most event scheduling languages, but also contributed the additional structure inherited from Pascal. Once the user has identified the events, lists, and statistics, the program will virtually write itself, even if the programmer knows little about Pascal. This illustrates a major advantage of simulation languages - the often confusing details are automatically handled for the user. In this case, although the program is based on list processing, the user need not have any

knowledge of pointers at all.

Along with the guidance provided, the language will also be fairly legible or self documenting. Pascal's design encourages the decomposition of a problem into short, single action, easy to follow procedures. This will, in turn, provide the user with a program that is easier to debug and easier to modify in the future.

A substantial amount of guidance has also been obtained for the next step in developing a Pascal simulation language. Problem areas have been identified, a basic foundational plan has been formulated, and the areas upon which a designer should focus his attention have been discussed.

Further tests need to be run to determine exactly how much slower the language would be in a large simulation, due to the list processing, when it is compared to GASP IV. It is obvious, however, that an extremely versatile and powerful language is possible following this approach.

BIBLIOGRAPHY

Fishman, George S., Concepts and Methods in Discrete Event Digital Simulation, New York, John Wiley & Sons, 1973.

Jensen, Kathleen and Wirth, Niklaus, Pascal User Manual and Report, 2nd ed., New York, Springer-Verlag, 1974.

Kiviat, Philip J., "Development of Discrete Digital Simulation Languages", Simulation, 8:63-70, February 1967.

Pritsker, A. Alan B., The GASP IV Simulation Language, New York, John Wiley & Sons, 1974.

Shannon, Robert E., Systems Simulation: the Art and Science, Englewood Cliffs, Prentice-Hall, Inc., 1975.

APPENDIX I - SAMPLE USER PROGRAM

```
PROGRAM CONTROL( OUTPUT );
USE EVENTLIST, EVENTTIME, ENTRY;
HISTOGRAMS
   1: TIME-IN-SYSTEM, 0.0, 6.0, 0.25;
   2: TIME-IN-QUEUE,  0.0, 6.0, 0.25;
DUMP
   EVENTLIST: QTIME;
   QUEUE: EVENTTIME, INTIME;
CONST
   LASTEVENT = 1000;
TYPE
   EVENT = (SERVICE, ARRIVAL);
   LISTTYPES ARE
      ENTRYTYPE = RECORD
                     EVENTTIME: REAL;
                     QTIME: REAL;
                     INTIME: REAL;
                     CODE: EVENT;
                  END;
   END;
VAR
   QUEUE, EVENTLIST: LIST OF ENTRYTYPE;
   ENTRY: ENTRYTYPE;
   NUMOFEVENTS: INTEGER;
   BUSY, STOP: BOOLEAN;

PROCEDURE INIT;
   BEGIN;
      NUMOFEVENTS:=0;
      BUSY := FALSE;
      STOP := FALSE;
      WITH ENTRY DO
         BEGIN
            EVENTTIME := 0;
            QTIME := 0.0;
            INTIME := 0.0;
            CODE := ARRIVAL;
            NEXT := NIL;
            PRIOR := NIL;
         END;
      INSERT(EVENTLIST, ENTRY:ENTRYTYPE, NEXTTIME,
         EVENTTIME);
   END;
```

```
PROCEDURE EVENTS;
   VAR
      ATRIB: ENTRYTYPE;
   BEGIN
      CASE ENTRY.CODE OF
         ARRIVAL: BEGIN
            ATRIB := ENTRY;
            IF BUSY THEN
               BEGIN
                  ATRIB.QTIME := NOW;
                  INSERT(QUEUE, ATRIB:ENTRYTYPE, FIFO);
               END
            ELSE
               BEGIN
                  BUSY := TRUE;
                  ATRIB.EVENTTIME :=
                     ATRIB.EVENTTIME + EXPO(5.0);
                  ATRIB.CODE := SERVICE;
                  INSERT(EVENTLIST, ATRIB:ENTRYTYPE,
                     NEXTTIME, EVENTTIME);
               END;
            END;
         SERVICE: BEGIN
            NUMOFEVENTS := NUMOFEVENTS + 1;
            IF NUMOFEVENTS = LASTEVENT THEN
               STOP := TRUE;
            HISTO( NOW - ENTRY.INTIME, 1);
            IF NOT(QUEUE(FIRST) = NIL) THEN
               BEGIN
                  REMOVE(QUEUE, ENTRY:ENTRYTYPE);
                  HISTO( NOW - ENTRY.QTIME, 2);
                  ENTRY.EVENTTIME := NOW + EXPO(5.0);
                  ENTRY.CODE := SERVICE;
                  INSERT(EVENTLIST, ENTRY:ENTRYTYPE,
                     NEXTTIME, EVENTTIME);
               END
            ELSE
               BUSY := FALSE;
            END;
      END;
   END;
```

```
BEGIN
   ZZZZZINITL;
   INIT;
   REPEAT
      NEXTEVENT;
      EVENTS;
   UNTIL STOP;
   OUTPUTT;
END.
```

VITA

Robert George Wilder, son of George A. and Jacquelyn L. Wilder, was born on November 24, 1953 in Baltimore, Maryland. After graduation from a Wappingers Falls, New York high school in 1971, he entered Lehigh University with a National Merit Scholarship. During his third year he was elected to Phi Beta Kappa, and in June, 1974 graduated summa cum laude with a B. A. degree in mathematics. He was a recipient of the Thornburg Mathematics Prize, and continued at Lehigh to pursue graduate studies with a teaching assistantship. In June, 1976 he received an M. S. degree in mathematics. After one year of study at Concordia Theological Seminary in Fort Wayne, Indiana he returned to Lehigh University to pursue a master's degree in Industrial Engineering. In June, 1979 he married Debra Marie Rumfield of Bethlehem. He is now employed by E. I. du Pont de Nemours & Co., Inc. as a programmer analyst.