PROGRAM GENERATOR SYSTEMS*

John R. Metzner

Computer Science Department
University of Missouri-Rolla

ABSTRACT

A program generator system is a general software tool for producing program generators. The program generator automates portions of the program synthesis process for members of a particular class of programs, allowing the source forms of programs to be more compact and to reflect the conceptual basis of the applications area more directly. This paper explores the design and workings of program generator systems for simulation programming. A generalization of the macro processor is shown to form a sound design basis, a basis which can be adapted to the simulation programming environment by the inclusion of innovative features. The status of a pilot implementation of a program generator system for GPSS is reported.

PROGRAM GENERATORS

The term program generator is used here to denote a processor for a language more problem-specific than the familiar, procedure-oriented, higher-level languages but less so than the non-procedural problem-oriented languages which have been developed for rather narrow classes of problems. The language accepted by a program generator might properly be called problem-tailored because it seeks to provide accomodation for the working concepts of a particular problem area without the costly burden of a totally descriptive language and its processor for a specific class of problems in that area. It also seeks to provide a programming means more facile than that of a higher-level language by freeing its user from attention to details which are repetitive, error-inducing, and not central to the problem at hand.

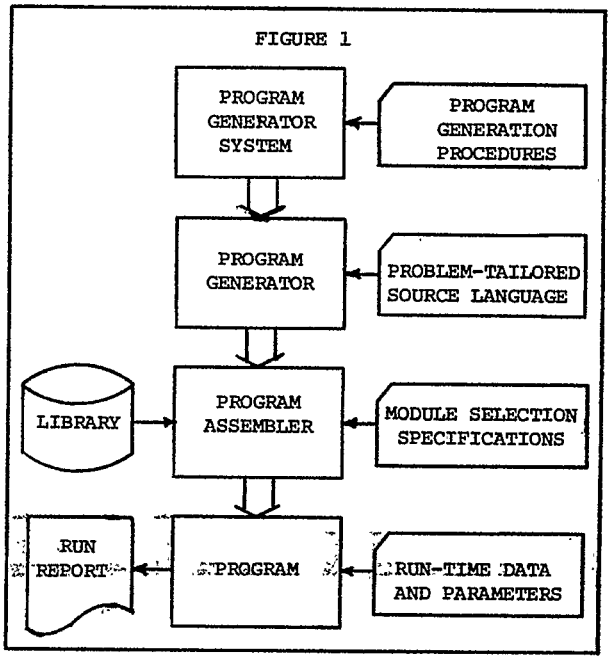
The program generator is a direct descendant of the macro processor. They have the same goals and operate to process programs written in a problem-tailored dialect of an underlying language. Both attempt to produce only portions of programs and allow the user to think in terms of previously defined modules developed expressly for his species of programming. And, both achieve versatility by placing the generated sections of programs within the control framework fashioned by the user in the underlying language. While macro processors exist in great variety, (see Cole [1] for a sampling) they are only rarely applied to higher-level languages in which a modularization feature (the procedure or subprogram and its invocation) already exists.

To be more precise, it is the macro processor plus a set of macro definitions that corresponds to a primitive form of the program generator. As was the case with macros, the primary focus is upon the system within which program generators may be defined and invoked.

PROGRAM GENERATOR SYSTEMS

A program generator system is a general software tool for producing program generators. It is like a re-usuable kit which supplies the building blocks and a general framework in which to place a selection of the blocks. A program generator system enables the efficient creation of problem-specific program generators and thereby a problem-tailored programming dialect. The operational position of the program generator system is illustrated in Figure 1 which shows that it supports the creation of multiple program generators which in turn assist users in the creation of numerous programs. The form of this assistance is the

FIGURE 1



*This work was supported by the National Science Foundation under Grant Number MCS75-15839.

automation of portions of the program synthesis task itself. The benefits of replacing detail-filled tedium by aggregates meaningfull in the context of the problem area are well-known.

The setting in which a program generator is used is similar to those of syntax-directed compiling systems and extensible programming languages, but the program generator system differs in several respects. The role of a syntax-directed compiler is to create a translator for an entire language, its voluminous and arcane input must describe every facet of the language and the compiling task for it. It is designed to create a processor for whatever well-formed language its user may care to create and use, offering great freedom in language design at a great cost in translation time. Extensible language systems, in attempting to alter their compilers to offer a degree of problem-tailoring generally have not succeeded in allowing more than a modicum of tailoring and that at the expense of acquiring a mastery over large amounts of technical detail. The program generator system differs markedly from these systems by not trying to offer all things to all users but concentrating upon those few things it can do well for a particular set of users.

APPLICATION TO SIMULATION PROGRAMMING

A large segment of the simulation programming field is suited to the use of program generator systems. Where programs tend to be moderately large, involving several people in their creation and maintenance, where a narrow but detailed class of simulands must be considered, and where model changes due to variation and refinement dominate parameter changes, the program generator system has a likely application.

The use of program generators in simulation programming is not new by any means. Recent literature mentions program generators used as a matter of course, such as for supplying parameters to the DELCAP airport simulation [2]. At the other extreme is the FORTSIM system whose program generators provide the user a simulation dialect of FORTRAN [3]. By and large, program generators are either relatively weak and based upon ordinary macro processors or are robust and highly adapted to their tasks as a result of being expensively programmed "from scratch" for the purpose.

This work attempts to demonstrate that program generator systems for simulation programming can be designed which enable the economical creation of suitable and adept program generators by providing significant assistance from the program generator system. This can be done by generalizing the macro processor, incorporating improvement and innovation into that generally unevolved type of system, and specializing the system to the task of producing program generators for a simulation programming language.

The principal benefit derived from the use of a program generator system is the ability to use alterable submodels in simulation programming the way we use alterable subprograms in other varieties of programming. Other benefits result from the ability to model and program in terms of submodels. The efficiency of having a program generator supply repetitive detail has been noted. Many of the advantages higher-level programming has over assembler language programming accrue here also because program generators can enable the use of a relatively higher-level dialect of the underlying simulation language, a dialect tailored to the workings of a particular type of simuland. Faster programming, more comprehensible and maintainable programs and the ability to deal with larger models are benefits in this category. The submodels also assist in the process of communication with the applications area analysts or the "customers" for which the simulation was developed.

Another group of benefits arises from the modularily aspects of using program generators. (These are not additional benefits when the underlying simulation programming language provides for program modularity, as SIMSCRIPT II.5 does.) When the expression of a program can refer to a module defined out-of-line, a top-down style of programming is encouraged. This leads in our case to a top-down style of model syntheses, usually yielding increased clarity and economy. Modularity permits independent testing and the substitution of modules to effect model changes.

The expected pattern of use for a program generator system in simulation programming proceeds as follows. Initial modeling and perhaps some of the programming are started in the usual manner. As this progresses, recurrent submodels are identified, classified, and then generalized wherever possible. Concise linguistic forms for generator invocations and program generator definitions are then developed for the major submodels and their use is introduced in the developing model documentation and simulation programs. The program generator procedures and their related submodels can be tested at this point. The process iterates as needed to encompass remaining submodels and enhancements or refinements to those already treated. Ultimately, both modeling and programming for the simulation project are done largely in terms of the submodels, and the program generator is used routinely to reduce the size of the simulation programming effort. Variations in the submodels are then handled by changing either the form of the program generator's invocation (if the desired alternative submodels are anticipated by a sufficiently general program generator definition) or by changing the program generators themselves. In either case, the tasks of making alterations throughout the simulation programs and keeping track of all the resulting versions of it are largely avoided.

SYSTEM DESIGN ISSUES

There is some freedom in the design of a program generator system. The program assembler shown in Figure 1, for example, may be entirely absent. It must be omitted where the simulation programming language does not permit a breakdown into separately processed modules which can be subsequently included from a library. Or, it may be a central component of the system, as in

the AVSIM facility [4]. In any event, a program assembler (even when it is just a Linkage Editor) operates like a primitive program generator by selecting modules in response to commands. So, where a program assembler is used, there is a division of effort between it and the program generator with the former selecting the desired modules for roles in which there is so little variation expected that a set of "canned" routines will suffice.

Figure 1 also fails to show a translator between the program generator and the program assembler, making that diagram represent a compiler-compiler style of implementation. While much more difficult and expensive to implement, program generator systems of this type have been built. Basili [5] describes a system which generates portions of compilers so that somewhat problem-tailored dialects of a base language can be produced by the knowledgable user. This design permits the program generator to be a full compiler, avoiding the processing time costs of preprocessing or interpreting. The drawback is that the problem-tailored dialect must be used long and hard to recover the investment in learning time and compiler creation.

A less ambitious form is based upon the familiar macro processor which, by accepting macro definitions in-stream, combines the two elements at the top of Figure 1. Its workings are depicted in Figure 2. The definitions may be placed in the library previously and may be in the primary input. Under the requirement that the in-stream definitions be entered first, the path through DEFINE is closed down as soon as any other is used. Leaving it open is an obvious convenience. Further, using the convention that a later definition replaces an earlier definition of the same name allows the definition of a macro to be changed during the course of processing. This redefinition could even occur as the result of macro expansion, as Halpern [6] suggests.

The process labeled CLASSIFY in Figure 2 normally operates at the statement level. That is, the problem-tailored dialect must contain only constructs which can be isolated into units like statements in the base language which are to pass through unscathed. The invocation of a macro expansion must therefore be an entire statement. This need not be so. Leavenworth [7] has shown that relatively simple parsing mechanisms can be employed in the CLASSIFY process to permit the invocation of program generators by a construct other than a statement, for example, by an operand or by a block of statements. The utility of this enhancement, however, depends strongly upon how much syntactic variety there is in the structure of the base language.

A more productive improvement of the macro processor is that of giving the EXPAND process other destinations for its generated code. The output path shown in Figure 2 places generated statements back into the system (to possibly detect further, nested macro invocations) and corresponds to the familiar in-place expansion replacing the invocation. Allowing EXPAND to select among several destinations for each generated statement adds a great deal of flexibility. Those possible destinations are shown in Figure 3. The insertions made into the base language program are not rescanned for further invocations (although multi-pass preprocessing would provide rescanning) and the insertions must be made at previously noted syntactic interstices such as "beginning of program", "end of initial declarative block", and the like. Similarly, insertions shown to the input stream cannot be made at the time they are generated but must be stored until their syntactic cue (i.e. "next START card") is recognized by the CLASSIFY process and then inserted as a block

FIGURE 2

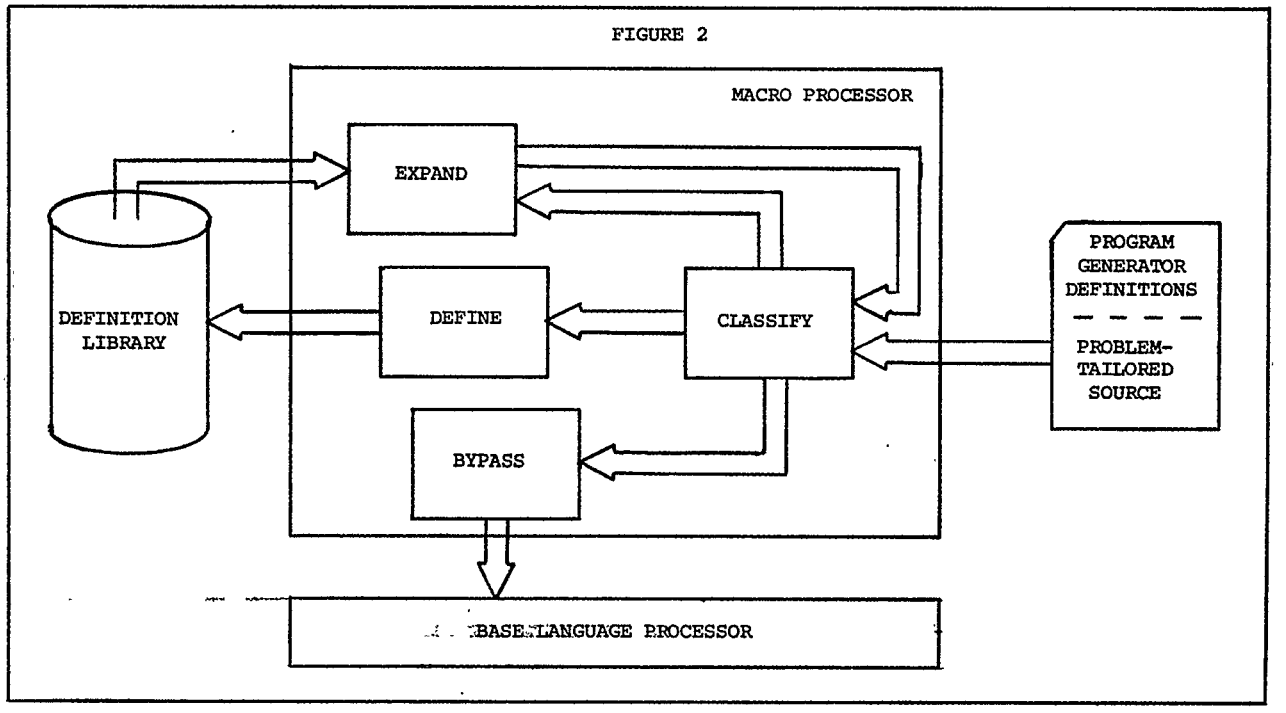
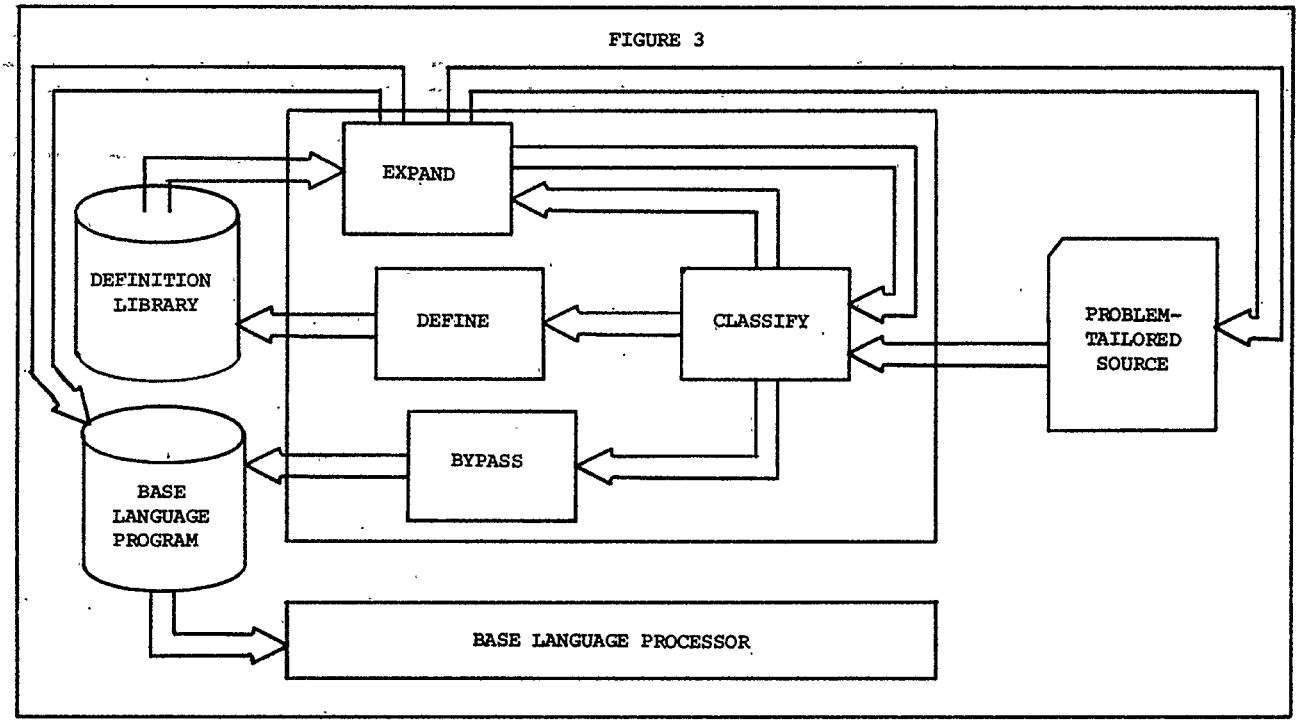


FIGURE 3



into the input stream. This generalization is not new, the SYMPLE system employed it to good effect [8].

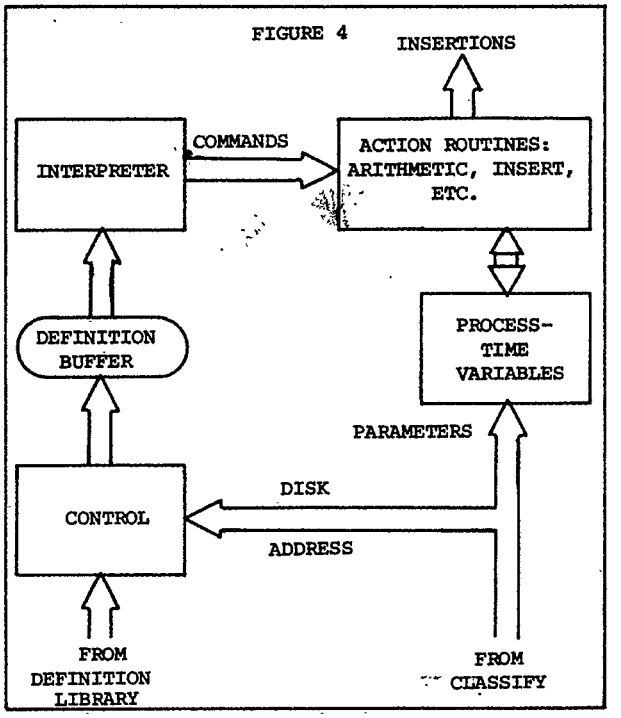
Taking a look into the structure of the process labeled EXPAND reveals other opportunities for design improvement. Figure 4 shows the major data flows within this component. The EXPAND module, in a conventional macro processor, performs its processing by interpreting the definition (one line at a time) and calling action routines in the proper sequence.

Maurer [9] has demonstrated the feasibility of "compiling" heavily used macro definitions and placing them within an augmented EXPAND module as shown in Figure 5. The "compiled" definitions consist of the series of calls of action routines the interpreter would generate in expanding the macro. Both the fetching from the library and the slow process of interpreting are thereby avoided.

This is where the program generator system aspects come in. An EXPAND module altered by the addition of compiled macros becomes a tailored program generator still capable of accepting further generator definitions. It acts like a macro processor for a dialect of the base language enriched by the verbs corresponding to the compiled macros.

The implementation of the program generator system can greatly facilitate the task of creating and adding a compiled macro. Using a higher-level language for EXPAND and sharing its source with the user will enable a systems-oriented member of his programming group to trace through the course of expanding the interpreted version of the definition and create the CALLs to action routines it performs. The resulting routine is then compiled with EXPAND, the control section's table is given the location of the new routine, and the altered version becomes the program generator for those applications needing the compiled macro.

FIGURE 4



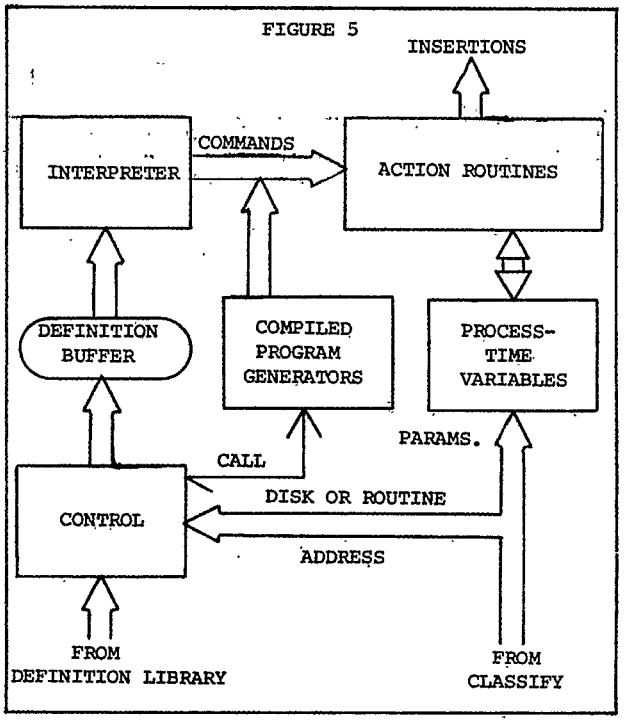


FIGURE 5

Most macro processors have a set of features collectively termed conditional assembly features. These are the mechanisms that enable macros to be much more than shorthand notations for blocks of source code. Process-time variables can be given values which act like data to the program generation process, affecting the characteristics of code generated by program generator invocations located throughout the submitted program. Conditionals (e.g. IF) in the generator definitions can act to adapt the code generated to global variables in addition to invocation parameters. This process-time conditioning can be more important in simulation than the execute-time conditioning based upon test-and-branch coding in the simulation program executed because it can avoid space and time penalties when the same path is to be taken throughout a simulation run. Therefore conditionals can expect to receive close attention in the design of the language in which program generator definitions are to be expressed.

This definition language should be similar to the conceptually familiar macro definition language, but must be much more robust in several respects. The storage and retrieval by name of process-time values must be largely implicit and significant ability to perform computation with such values must be present. Arithmetic capabilities alone are not sufficient, the ability to manupulate text must be present in good measure also. The sophisticated generation and processing of text characterizes the program generator and sets it apart from the macro expander which has only primitive capabilities in this respect. Large amounts of program text must be produced and produced in ways far more articulate than inserting user-supplied argument strings into model statements. The combination of this aspect with the previous one requires that process-time variables may be string-valued and that a rather rich set of string-manipulation verbs be available.

A noted characteristic of the program generator is that while it must perform a lot of text manipulation, that manipulation cannot be the same for every activation of a program generation procedure -- it must be made to depend strongly upon parameters of that activation and process-time "data" parameterizing as a whole large portions of the system to be simulated. Therefore, the language used to specify program generation procedures must contain significant constructs for conditioning, constructs more adroit than the pedestrian IF...THEN..., GO TO..., and REPEAT OVER LIST... normally found in advanced assembler and macro definition languages.

PILOT SYSTEM

As a pilot project to explore the capabilities of this software production assistance mechanism, a macro-style program generator system for GPSS has been designed and a large portion of it has been implemented. The pilot system is intended to be a strikingly adept form of macro processor incorporating improvements and generalizations upon macro systems as we know them and attuning the system to the task of generating significant portions of GPSS programs. Its overall design is necessarily somewhat open-ended so that desirable features can be freely included and evaluated. For example, the initial system will permit invocation of program generators by constructs conforming to the syntax of a GPSS <statement>, but the design anticipates the eventual inclusion of invocation at other levels like <operand> or <label>.

Four remote destinations for generated GPSS source lines have been provided:

1) At the beginning of the program,
2) Just before the first block statement,
3) Before the next START statement,
4) After the next START statement, and
5) Before the next JOB or END statement.

The initial version of the program generator language for this system is a verb-keyword language. This gives a BASIC flavor to the language but keeps the interpreter simple and easy to change.

An example generator definition is shown in Figure 6. The purpose of this generator is

FIGURE 6

```
 PROGEN  EXP2  NOM,IND,SEGS,ALPHA,BETA
PUT1  #NOM FUNCTION #IND,C#SEGS
SET   INDEX = #SEGS
SET   STEP = 10./INDEX
SET   DEL = STEP*#BETA
SET   WORK = #ALPHA
SET   VAR = 0.
PUT1 0,VAR
TOP:SET VAR = VAR - STEP
SET   WORK = WORK + DEL
SET   FIRST = 1. - EXP(VAR)
PUT1 FIRST,WORK
GIF   ((INDEX=INDEX-1).GT.1) GO TO TOP
 ENDGEN
```

to supply (from an invocation anywhere in the program before START) the definition of a two-parameter exponential function. The function name, the identity of the independent variable, the number of linear segments to employ, and the two parameters are parameters of the invocation. Using a random number code for the independent variable yields the definition of a function for supplying, without extra arithmetic, a random variate having the exponential distribution with the two parameters specified. PUT1 is the verb indicating formation of a statement and insertion (FIFO) at the start of the program. SET signals a replacement statement for process-time variables (note the assignment in sub-expressions) and GIF is the generator-time IF statement. This generator is very inelegant, producing a follower card for every pair of points in the function's definition.

The process-time variables have their mode stored in the symbol table and need not be declared or initialized before use. Vectors and arrays are provided at program generation time and the lookup type of operation for process-time storage permits subscripting by quantities of any of the four modes; integer, real, logical and character string. The SET statement has all the power of FORTRAN arithmetic, allowing articulate function generation algorithms like Kisko's [10] to be executed in-stream as program generator procedures. In fact, a FORTRAN implementation of the program generator system was deliberately chosen so that such procedures could easily be compiled and made part of a tailored version of the system. Portability was a factor in the language choice also, ANS FORTRAN is used and assembler language action routines for machine-dependent (IBM 370) functions have been kept to a minimum.

The pilot system is still months away from distributable form. Some aspects of its design, especially in the generator definition language where some innovative conditioning mechanisms such as decision tables for text generation will be attempted, are still evolving. It is hoped that this paper will elicit the ideas and suggestions of practitioners.

REFERENCES

1. Cole, A. J., Macro Processors, Cambridge University Press, Cambridge, England, 1976.
2. Gilsinn, J. F., "Validation of an Airport Simulation Model", Proc. 1976 Winter Simulation Conference, 273-277, 1976.
3. Crumm, R. D., Wang, S-L, and Cooper, E. H., "FORTSIM--Simulation Using Structured FORTRAN Plus Table Management", Proc. 1975 Winter Simulation Conference, 15-19, 1975.
4. Scarpino, F., and Clema, J., "A General Purpose Tool for Interactive Simulations", Proc. 1976 Winter Simulation Conference, 475-484, 1976.
5. Basili, V. R., "The Design and Implementation of a Family of Application-Oriented Languages", Proc. 5-th Texas Conf. on Computing Systems, 6-12, 1976.
6. Halpern, M. I., "Towards a General Processor for Programming Languages", Comm. ACM, 11:15-25, (January, 1968).
7. Leavenworth, B. M., "Syntax Macros and Extended Translation", Comm. ACM, 11:790-793, (November, 1966).
8. Vander Mey, J. E., Varney, R. C., and Patchen, R. E., "SYMPLE - A General Syntax Directed Macro Preprecessor", Proc. AFIPS 1969 FJCC, 157-167, 1969.
9. Maurer, W. D., "The Compiled Macro Assembler", Proc. AFIPS 1969 SJCC, 89-93, 1969.
10. Kisko, T. M., "An Automated Method of Creating Piecewise Linear Cumulative Probability Distributions", Proc. 1976 Winter Simulation Conference, 487-494, 1976.