Automatic generation of computer programs*

by NOAH S. PRYWES

University of Pennsylvania
Philadelphia, Pennsylvania

ABSTRACT

This is an introduction and summary of research on Automatic Program Generation conducted at the Moore School, University of Pennsylvania. This research culminated in development of a Module Description Language (MODEL) designed for use by management, business, or accounting specialists who are not required to have computer training. MODEL statements describe input, output, and various formulae associated with system specification. No processing or sequencing information is required from the user. A MODEL Processor analyzes the specifications and interacts with the user in resolving inconsistencies, ambiguities and incompleteness. A program for performing the required functions is then generated based on the "complete" specification.

INTRODUCTION

A major aspiration of computer programming language designers has been to make programming so easy that large classes of educated people who have not been exposed to computer training are able to program. For instance, in 1960 the CODASYL committee designed a programming language named COBOL-Common Business Oriented Language. Despite this and many other attempts to reduce the complexity of the programming process, it has continued to require considerable skill and specialization. Currently there are a large number of Application Programmers who handle such tasks. The standard procedure is for a "user," whether manager, accountant or business specialist, to communicate requirements to an Application Programmer who, in turn, composes a program to fit these requirements. The research described in this paper is a continuation of efforts to make feasible the preparation of programs by users (interacting with an automatic program generator) without recourse to middle-men Application Programmers.

In view of the historical elusiveness of this goal, it is approached with considerable trepidation.

Figure 1 illustrates the overall concept of interactions between the automatic computer program generator and the user. (The components in the diagram are referred to by the indicated numbers). The user (1) is viewed as an individual who is proficient in the immediate field in which the programs are to be applied. Namely, he is viewed as being in a management or a technical capacity, such as in accounting, production control, etc. He must not only have had professional training in this specialized area of activity, but also have had a good mathematical background. The user is not required, however to have had any specialized computer training, but he must understand that when a function is properly specified it can then be executed by a computer.

The user composes statements (3) (via a terminal and a Text-Editor (2)) in a language named MODEL (MOdule DEscription Language). Each statement is considered an integral unit and contains a "chunk" of information. A statement may describe an item of data (data description) or an algebraic or logical relation among items of data (assertion). References may be made to statements previously entered in a data base (4) by the user, by others who have specified requirements for similar applications, or by others with whom the user wants to share data. The MODEL Processor (5) analyzes the totality of statements transmitted to it and, as appropriate, solicits from the user additions or changes to these statements. It provides a user with listings, cross-references, and requests for additions or changes necessary to resolve incompleteness, ambiguities or inconsistencies (6). Finally, the analysis by the processor leads to certain logical implications which are also communicated to the user to enable him to clarify and self-check his specifications of the requirements. When all outstanding problems have been resolved in this dialogue, the processor produces a program in a computer language. An Optimizing Compiler (7) produces the object code (for the application program) which is loaded into a digital computer (8) for execution.

The system description in this report is based on an operational MODEL II system which incorporates corrections and improvements consisting of new language, sequencing and iteration analysis, over a previously described language and processor. Work is under way to reprogram.

the processor to reflect additional improvements. The system is programmed in PL/I and produces object programs in PL/I.

In the interest of brevity, the paper describes the use of the MODEL language in the second section of this paper, and the analysis leading to solicitation of additional user information in the third section. The automatic program (and flowchart) generation phases of the MODEL system are omitted and described in the referenced reports. A survey of related research can be found in References 4 and 5.

**Distinctive characteristics of MODEL**

The MODEL language described in this paper incorporates several new characteristics not existing in previous
programming languages and which were adopted because of their distinct advantages over past practices. These characteristics are explained below in the context of the system illustrated in Figure 1 where a user communicates with a processor that generates programs.

The new labor saving characteristics that have been incorporated in this approach are as follows:

1. Non-proceduralness—means that the user need not (and cannot) specify any order of evaluation or memory assignments. The "control logic" parts of the ultimately produced program, which are based on such procedural information, are to be deduced by the MODEL Processor. This feature is considered important, not only because it saves programming labor but also because it reduces the necessary computer training of the user. For instance, the user does not need to have such basic concepts as flowcharting or memory.

2. Independence of statements—means that the user can concentrate on composing a single statement at a time. It is neither required nor possible to indicate any relationships among the statements (except implicitly, such as when specifying relationships among variables). A single statement is required for describing each data name, or each formula. Modification or addition of statements can be carried out independently, one statement at a time, in the same manner.

3. Randomness—takes into account that information may originate from a group of users. Also each user's concept of computer requirements is usually not well organized, and a variety of information comes to his mind at different times. The user can describe this information in statements, one at a time, in the order that the information occurs to him. While a certain organization in this approach may be helpful, it is not required.

4. Incrementality—means that once users have provided a certain portion of the totality of the statements, the processor should be able to solicit additions or changes incrementally until a complete specification of the computer requirement is obtained. In this manner, it should be possible to avoid the problems of ambiguities or incompleteness that lead to major misunderstandings between the users and Application Programmers, and which require costly corrections and reprogramming.

5. Self Documentation—is attained as the documentation is generated during the dialogue between a user and the Processor. The additional documentation of the corresponding flowchart can be generated by the Processor automatically when generating the program. These latter types of documentation would normally not be of interest to a user, but rather to a Systems Programmer. The documentation of the user's computer requirement and of the associated program to be produced by MODEL is comprised of the collection of the corresponding statements together with the cross references and summary tables and comments produced by the Processor.

6. Maintenance—involves corrections of the programs based on malfunctions discovered during operation, or on modifications of the specifications to meet new needs. In current practice the modifications must also be performed by a middle-man Application Programmer. It is envisaged, instead, that the user would make the changes in the MODEL statements to reflect either the corrections or the modifications, whereupon the Processor would generate a new program automatically.

7. Sharing—of data or computations can be attained by storing the corresponding MODEL statements in the Processor's data base. A user desirous of sharing the know-how of others who have previously stated requirements in similar application areas needs only to reference these statements in order to incorporate them in the specification of his program. Data bases could be physically shared, while computations would be repeated in each user's program. To make changes in the organization of shared data bases, the data description statements must be modified or added, and previously generated programs based on these statements must be automatically regenerated. In this way changes to shared data bases or programs can be carried out without requiring the users to modify their programs individually.

8. Tolerance—The Processor is tolerant of the user's ambiguities and omissions. To fully specify a requirement would necessitate composing many statements which may appear to a user to be self evident and superfluous. The Processor in synthesizing the MODEL statements into a program must recognize the resulting ambiguities and omissions and generate the necessary additional MODEL statements automatically, thus relieving the user of much tiresome detail.

THE LANGUAGE

Example

An example is used in the following pages to illustrate how a user describes a requirement which he wishes to automate.

The example envisages the environment of a department store with many departments, a large number of charge account customers, and an extensive and diverse stock inventory. Point-of-Sale terminals connected to a network of computers are distributed through the several locations of the department store. The user of the MODEL system is envisaged to be a department store analyst who desires to specify the accounting requirements for purchases by cash and charge account customers.

Figure 2 gives an overview of the accounting requirement. The corresponding program module is named DEPSALE, and is shown at the center of the figure.

The data for DEPSALE comes from three sources. The sales transactions (SALETRAN), come from a Point-of-Sale terminal (POSTERM) sequentially, one at a time, and contain the information provided by the purchasers. The customer data (CUSTMAST) contains records of customers which can be referenced by providing customer numbers. Finally, there is inventory (INVEN) data, where information on stock items can be referenced by providing a stock
number. The data that comes in to DEPSALE is referred to as **SOURCE** data.

The **TARGET** data consists of the records in CUST-MAST and INVEN which are affected by the sales transaction and must be updated. Other TARGET data are the entries made in a sales journal (SALEJOUR) which are ordered sequentially by sales numbers (SALES #). Finally, a sales slip (SALESLIP) is produced on the terminal in cases where the sales transaction has been consummated or, alternatively, an exception notice (EXCEPT) is produced when the transaction does not take place.

**An outline for preparing requirement descriptions**

Figure 3 shows, in outline form, the information that needs to be provided in describing a requirement to be automated. As indicated, the user does not need to follow this outline, but can provide the information in any order. With the aid of a Text Editor, the user can enter statements and organize them into sections, subsections, etc. At the highest level the description is divided into three sections: the header, the data description, and the computation description.

The header contains identification information: module name(1), source(2) and target(3) data names, and references to sections or subsections of computation description statements, (called assertions) that are in a library in the data base(4). The latter may represent standards of data formats and organizations or any previously entered statements. The user is able to specify more complex operations that would be applied to the statement-data-base, to produce new statements to be incorporated in the specifications of a desired program module.

Data description is independent of the computation description, so that the data may be shared by several programs. Data and computation descriptions may be in a library, and called for automatically, to facilitate sharing of data and computations. Data description breaks down into

**Data description statements**

**Data statements**

A **data network** concept is employed. Its application to the DEPSALE program is illustrated in Figure 4. First, a user has to name each data structure (to be further explained) and compose a statement for each name used. Each of the source or target files, inputs or reports is organized internally in a hierarchical structure resembling a

---

From the collection of the Computer History Museum (www.computerhistory.org)
tree. Each node of the tree must be given a name. The user must compose a statement corresponding to each data name, in which he provides associated information on the branch connection, data length, number of repetitions and other parameters of source and target media. Network (non-tree) structures are described by use of POINTER type assertions which coordinate instances of repeating data.
For example, consider the sales transaction (SALETRAN) source data at the top left of Figure 4. The STORAGE name is a POINT-of-SALE Terminal—POSTERM, (not shown in Figure 4). The SALETRAN file (an incoming message, referred to here as a FILE) is describable as a tree structure. The name of the FILE is at the apex, emanating from it is the RECORD (SALEREC) node. The branches emanating from the RECORD node lead to GROUP or FIELD nodes. GROUP nodes are not terminal nodes, and the branches emanating from GROUP nodes can again lead to GROUP or FIELD nodes. The terminal nodes are always referred to as FIELDs.

Figure 5 shows the data description statements of SALETRAN. The words, RECORD, STORAGE, GROUP or FIELD are followed by their respective parameters, in parentheses. The STORAGE parameters depend on the devices specified. For POSTERM they are the format, unit number and block size. The parameters of FILE, RECORD or GROUP are the data names of the descendent nodes. Another parameter of FILE is the name of the referencing or sequencing field, if any. Parameters of FIELD are data length and number of repetitions (if more than 1). These parameters can be constants or variables. If they are variables, LENGTH and EXIST type assertions must be provided separately to specify how they can be computed.

Data description assertions

A number of assertion types are needed to describe the data further. They concern length and number of repetitions of data and inter or intra file pointers. Figure 6 illustrates these assertions.

The number of transaction items (ITEM) in a sale transaction is a variable (EXIST type) named EXIST.ITEM.

Assertions specifying the computation of these variables are shown at the top of Figure 6. For instance, number of repetitions can be determined from the position of delimiter characters. The delimiter for end of transaction is the field 10. The delimiter for end of transaction is the field 10.

Figure 5—Data description statements for SALETRAN

ENDTRANS. It is assumed to be a Ready symbol, R. The first assertion in Figure 6 specifies the calculation of the number of repetitions of the group TRITEN. It uses the function INDEX which evaluates a string of characters to determine the position of the R symbol in relation to the beginning of the strings of the SALEREC record.

Figure 6 also shows the assertions which specify that fields CUST# and STOCK# in SALEREC can be used as pointers to the customer master and inventory files as illustrated in Figure 4. Note that POINTER.INVENREC has two subscripts, the first is FOR_EACH_ITEM, and the second is '1'. As will be seen, the pointer value may be derived also from the REPLACE field of INVENREC.

The other source and target data indicated in Figures 2 and 4 can be described in similar manner. In the interest of brevity the discussion of this data is omitted.

In reproducing a listing of the submitted statements, the Processor names all data description statements and assertions and identifies source and target variables, unless already specified by the user. The names assigned to statements are a derivation of the names of the data elements described or a derivation of the names of the dependent (target) variable in assertions.

Computation description statements

Composition of assertions

Following the outline in Figure 3, the computation description consists first of the description of interim variables in a manner similar to data description, except that these variables are stated to be INTERIM.

In addition to describing interim variables, the description of computation consists primarily of statements with logical or/and arithmetic constructs which are also referred to as assertions. Using arithmetic and logical operators, as well as functions, the user composes such statements to specify relationships among variables.

In composing an assertion it is necessary to separate the dependent and independent variables. One common convention is to place the single dependent variable on the left of the equal sign (=). Note that the = sign means algebraic equality and not assignment.

In composing assertions, the user specifies relationships using mathematical, non-procedural notations. Many relationships, not directly expressible using arithmetic and logical operators, must then be expressed using functions that map the SOURCE data into the TARGET data. These
functions are a substitute for established mathematical notation. (an example is the Σ symbol, meaning "summation" which is illustrated further below). Other functions evaluate character strings (such as the INDEX function described above). The use of functions in assertions requires stating the name of the function followed with the specification of the parameters, enclosed in parentheses.

The functions return a value (which may be a single variable, or components of a vector or an array).

SUBSET assertions

Examples of SUBSET assertions are shown in Figure 7. For example, the first assertion specifies that only transactions from terminals SALE2 through SALE5 and from clerks C5 through C7 are to be processed. As shown, it is applied to the SALETRAN source data.

The second assertion in Figure 7 applies to target data EXCEPT. It specifies that entries in this target file be limited to cases where the balance (BALANCE) would exceed the credit limit (CREDLIM).

Illustration of computational assertions

As indicated in Figure 3, assertions can be used to specify relations exemplified by accounting rules or business decisions. Figure 8 gives four examples of such assertions (the total number in DEPSALE is 20) and illustrates several features of assertions:

The first assertion in Figure 8 specifies the evaluation of the EXTENSION field which is the dollar value of purchased stock items of one type. The subscript notation (FOR_EACH_ITEM) after a variable means that it can have several components and by implication, that the process will be repeated a variable number of times corresponding to the number of repetitions of ITEM.

The second assertion in Figure 8 illustrates the use of the SUM function to specify the evaluation of the total charge made on a purchase sales slip (TOTCHARGE). The SUM function has a parameter of the variable to be summed (EXTENSION). (Another assertion not shown in Figure 8 must specify the calculation of TAX).

The third and fourth assertions in Figure 8 illustrate business decision rules. For instance, if an item in the sale transaction is out of stock, namely the Quantity On Hand (QOH) field is smaller than the quantity specified in the sales transaction (QUANTIT) then it is desired that a suitable substitute item, if any, should be sold. The stock number of a suitable substitute item is stored in the inventory record (INVENREC) in the field named REPLACE (see Figure 4).

To represent decisions, the user can use a variable with the name of the decision prefixed by the word CHOICE. All such variables can have only two values, SELECTED and NOT-SELECTED. They will be described automatically by the Processor (see a later section). The third assertion in Figure 8 shows the expression that specifies when the substitution is to take place.

The last assertion in Figure 8 specifies the implementation of the decision, namely the stock number in the REPLACE field is treated in the same way as if it were the stock number in the sales transaction. This requires a new value for the POINTER.INVENREC field (see Figure 8). This operation is expressed by use of subscript.

The above assertions are representative of some of the relationships that can be expressed by assertions. The library of functions is open-ended and additions can be made easily to accommodate special needs. However, it is important to restrict the number of functions and to have their operations similar to common mathematical notation in order to assure ease in user familiarization with them.

Reporting formatting assertions

The description of messages or reports in MODEL is similar to that of information stored in computer storage media. The user always views the information as a string of information divisible logically into records, groups, and fields. However, in the specification of messages or reports he also must consider the continuity and availability of physical space. Additionally, the internal order of data substructures can be specified by the sequence of submission of the corresponding data statements to the Processor.

In describing the format of a report, the user must consider tab, carriage return or new page symbols as if they were data fields. In source data, these formatting symbols would already have the desired values. In target data, the obtaining of the values of these data must be specified by assertions. These values are frequently data dependent. Figure 9 illustrates this by showing two assertions that

```
IF CREDLIM > BALANCE THEN SUBSET.EXCEPT = SELECTED
ELSE SUBSET.EXCEPT = NOT SELECTED;
ELSE POINTER.INVENREC(ITEM, 2) = REPLACE
ELSE POINTER.INVENREC(ITEM, 2) = NULL;
```

Figure 8—Examples of computational assertions for the DEPSALE example

```
ENDITEM (FOR_EACH_ITEM) = STRING (CR, 1);
END-TAX = STRING (CR, 12 - EITEM);
```

Figure 9—Example of report formatting assertions
compute the number of carriage returns used in the saleslip (SALESLIPREC, see Figure 4) after printing ITEM groups. Normally, one carriage return symbol after each item line suffices, except after the TAX line when it is desired to advance to the 12th line of the saleslip, where the total charge (TOTCHARG) is printed. The function STRING generates a string, consisting of substrings (CR) specified in the first parameter, which are repeated a number of times specified in the second parameter (12-EXIST-ITEM).

This task of describing a report appears laborious. However, it can become easier by use of picture data types and certain operations (definable in MODEL) which will specify automatically report standards, such as for instance including "end of record," "end of group" and "end of field" fields following the respective data description statements.

GRAPH REPRESENTATION AND COMPLETENESS OF A MODEL SPECIFICATION

Organization of precedence information

Each statement in MODEL is an integral unit identified by a name. The existence of a precedence relationship between two statements indicates that a statement must be evaluated prior to initiating the evaluation of its successor statement. The entire collection of statements is envisaged as a directed graph where the statements are represented by correspondingly labeled nodes and where directed arcs or pointers connect the nodes, each representing a precedence relationship between the statements at the pointer origin and termination nodes. Figure 10 illustrates this view, showing the statements (represented as □) which form nodes of a graph for the above example. Each pointer is labeled, with the corresponding precedence type. Thus, each pointer has a direction and a type. The nodes in Figure 10 are shown to have a number of pointers exits and entries. The exits correspond to precedence pointers emanating from a node and pointing to descendant or target data or assertion nodes; the entries correspond to pointers originating at parent or source data or assertion nodes and terminating at a node.

The pointer finding process examines statements pairwise, using rules for determining the precedence type which will become the type of the precedence pointer. The types of the found pointers are entered in a precedence matrix, illustrated in Figure 11. Assume that a specification consists

---

Figure 10—Partial graph for depsale, showing data(.) of Figure 5 and assertions ( ) of Figures 7 and 9.
of $n$ statements, then there are $n(n-1)$ pairs of statements that need to be considered for finding pointers.

The different precedence types indicate corresponding methods of interpreting the respective statements in the subsequent phase of code generation, not discussed in this report. The pointer type recognition rules are summarized below. These precedence types are extensible. Precedence types can be added provided that they can be stated in terms of pointer selection rules applied to statements, pairwise. The definition of pointer selection rules involves analysis of data and function names in predecessor and successor statements. Once the rules have been applied to pairs of statements and existence of pointers has been determined, these pointers are labeled with the appropriate precedence type. The labels of the respective pointers are then entered in the precedence matrix table, shown in Figure 11, at the intersection of a predecessor statement row and the successor column. Thus, once this process has been completed, a row will contain the types of all the exit pointers of a statement and a column will contain all the entry pointers types.

There are basically two main types of pointers:

1. **Data Tree Hierarchy**: between data description statements (data names) within a FILE, organized in a tree structure. For source data the node closest to the apex is the predecessor and the node at the end of the branch is the successor, (precedence type 1) and vice versa for target data (precedence type 2).
(2) Data Determinancy: between assertions and data descriptions statements. When the data name is the source of an assertion, a data node is the predecessor and an assertion node is the successor (type 3). When a data name is the target of an assertion, an assertion node is the predecessor, and a data node is the successor (type 4).

Additionally, there are several miscellaneous precedence types requiring special interpretations in the code generation, as follows: Media (storage) statements can have entries from source file statements (type 6) or entries from target files (type 7). LENGTH and EXIST type variables can have pointers to source (type 8) or target data (type 9) respectively. The POINTER type variables have pointers only to source data (type 10).

Figure 11 shows the Precedence Matrix with the rows and columns ordered by respective types of statements. The possible precedence types are indicated at the appropriate intersections. The ordering of statements in Figure 11 is not in fact required, but is purely for illustration and suggests useful reports to the user (as described below).

Analysis of precedence information

As indicated in Figure 1, a most important aspect of the concept of the MODEL system is the MODEL Processor's solicitation of new or modified statements. A number of reports are produced for review by the user. First, the Processor produces a listing of MODEL statements and a report which cross references each data name with the corresponding data and assertion statements. In addition, the Processor requests the user to resolve problems that it encounters. These requests have been divided into four classes: Incompletenesses, Ambiguities, Inconsistencies, and other Implications. Analysis of the information in the precedence matrix (Figure 11) can provide most of this information, as discussed below.

Incompleteness and Inconsistency problems are similar to "errors" and resolution of such problems is prerequisite for completion of the processing. They normally terminate processing. Ambiguity and Implications problems are similar to "warnings," and the Processor continues to complete the subsequent generation of code in the object language. The user may wish to examine these comments of the Processor, and, if necessary, make appropriate modifications, and resubmit them to the Processor. Otherwise these comments should be incorporated in the documentation of the program module.

The messages reporting results of the analysis which are sent to the user must be phrased in a manner that will make it easy for him to make modifications. The messages should therefore preferably address only one statement which needs to be added or modified. The exception to this rule would be where problems arise from Inconsistencies or Implications which are based on more than one statement.

Incompleteness is defined as an instance where the graph is incomplete, where entire statements are missing or where statements are duplicated. Such cases can be recognized by searching the precedence matrix of Figure 11 to verify the following conditions:

(a) Each row and column must have at least one pointer (in a column) and one exit pointer (in a row), with the following exceptions: Source and Target file statements have only an exit pointer or an entry pointer, respectively, while field statements may have no exit pointer (where the field is not used in deriving the target data). Absence of expected pointers, as above, indicates that a statement must be added by the user.

(b) The number of pointers in rows and columns must also be checked. Source data and target data statements can have only one pointer in the corresponding column and row, respectively. Also, Assertion statements can have only one pointer in the corresponding row. The existence of more pointers indicates either an Ambiguity which must be resolved by the user adding qualifying names to the names of similarly named data, or a logical Inconsistency due to duplicate statements.

(c) Each source data file statement which has an index sequential (ISAM) organization, must also have a POINTER type statement as a predecessor.

(d) Pointers in each row and column, should not originate or terminate, respectively, in statements having the same data or assertion name, otherwise an Ambiguity or an Inconsistency is indicated.

(e) The number of pointers in an assertion column must equal the number of source variables of the assertion.

(f) All pointers must conform with the allowable types indicated in Figure 11.

Before reporting the Incompletenesses, an attempt is made to resolve these problems automatically. If such resolution is possible, the suggested additions or modifications of statements are reported. If this process is not successful, appropriate messages with an indication of the missing or inconsistent statements are sent to the user. This supplementing of MODEL statements by the Processor is essential to relieve the user of providing much tiresome detail which may appear evident to the user. Rules for making such judgments may be added or modified based on experience with the Processor. Examples of such automatic additions and modifications to MODEL statements include:

(a) Modifying an assertion statement by preceding the names of an ambiguous data used in assertions with the names of their respective files (or other higher level data names). This would resolve the ambiguity where the same data name is used in a number of data statements.

(b) Naming of statements and identifying the SOURCE (independent) and TARGET (dependent) variables of assertions (where the user omitted this information).

(c) Providing assertions that will indicate equality of similarly named source and target data in the absence of
other assertions expressing relationships between such data.

Inconsistencies are conflicts which require the user to conduct a logical analysis of more than one of the submitted statements. Some Inconsistencies are simple to determine. Examples were shown in the discussion of incompletenesses above, where a statement node has more than one exit or entry pointer, but only one is permissible. An Inconsistency message must then be produced which includes the offending statements. A more complex type of Inconsistency arises from the existence of “cycles” which are closed paths in the directed graph, each with a number of nodes and interconnecting pointers. The process for finding cycles is discussed in the references. Cycles in the directed graph denote faulty circular logic. They do not indicate iterations in the resulting program. Iterations in the program originate from a number of other features of the language such as the use of subscripts (FOR_EACH_X) following the name of a variable in an assertion and from the use of repeating data. It is required of the user to “open” the loops found by the processor through a modification of some of the statements corresponding to the nodes of the loops before the Processor can continue with the code generation.

Implications are classes of logical conclusions based on submitted statements that are considered to be potentially of interest to the user. The Processor, while capable of determining such conclusions, cannot further evaluate the implication, either because of limitation of the analysis methods that are employed, or because of lack of information of the area where the program is to be employed. Therefore the cooperation of the user is requested to check and verify the reported conclusions. Implication can be effectively reported in a form similar to “decision tables” which have been widely used in the past. Two such tables can be extracted from the matrix of Figure 11, consisting only of the data name rows and assertion columns where pointers of Type 3 exist and assertion rows and data columns where Type 4 pointers exist. Such tables are considerably smaller than the matrix of Figure 11 and therefore they can include the entire row and column statements.

CONCLUSION

The restrictions on space have limited the scope of this article to the extent that only a small illustrative example was presented with the objective of familiarizing the reader with the general use and operation of the system. The reader is referred to Reference 4 for a more comprehensive survey of the field of automatic generation of programs. References 1, 2, 3 and 4 provide detailed description of syntax and semantics of MODEL, and on the methods of generating programs automatically.

ACKNOWLEDGMENT

This paper is based on MODEL II. The implementation of this system was the responsibility of: S. Shastry and Y. Chang—Syntax analysis, N. A. Rin—precedence matrix and code generation, A. Pnueli—sequence and iteration analysis.

REFERENCES
