Division of Research
Graduate School of Business Administration
The Univeristy of Michigan

Written Fall 1976
Revised Summer 1977

A NON-PROCEDURAL LANGUAGE AND A SYSTEM FOR AUTOMATIC GENERATION OF DATA PROCESSING PROGRAMS

Working Paper No. 156

by

N. Adam Rin

The University of Michigan

A Non-Procedural Language and a System for Automatic Generation of Data Processing Programs

N. Adam Bin *

Abstract
The rising cost of software development has motivated research on the automation of parts of the software development process. Towards that objective, a non-procedural language called MODEL has been developed to describe desired programs of an information processing system. This paper provides an overview of MODEL and describes a software system that automatically generates conventional business data processing programs from specifications in that language.

1. Background and Overview

This paper is concerned with research done on the automatic generation of conventional data processing programs. It provides an overview of a non-procedural specification language called MODEL and of an automatic methodology that produces application programs from specifications expressed in the non-procedural language. The research reported in this paper is directed toward reducing the costs of software development by automatic generation of application programs from non-procedural specifications of

---

* The author, currently Assistant Professor at the Graduate School of Business Administration, University of Michigan, performed the research reported in this paper in 1974-1975 while a graduate student at the University of Pennsylvania Computer Science Department. This project was supervised by Prof. N.S. Prywes and supported by the Office of Naval Research under contract N-00014-67-A-0216-0014. See [1] for greater detail on this project.

user requirements. Another goal of this work is to make the computer usable for a broader range of people. By developing a language and methodology capable of automatically analyzing specifications and generating programs, it is hoped that it would be possible for systems analysts to build an information processing system without the intervention of application programmers as middlemen.

In order to pinpoint the area of automation covered by this research, it is useful first to review briefly research by others in automating the stages in the development of an information processing system. Various authors [2,3] have enumerated these stages differently but they are generally as follows:

1. Perception of automation need and determination of overall requirements;
2. Production of functional specifications (user requirements specification or logical design);
3. Physical system design;
4. Programming (program design, coding, debugging, and testing);
5. Installation and operation;
6. Maintenance and modification.

The systematic automation of each of these phases has been proposed several years ago [3] in the ISDOS project, which has primarily made progress in the areas of the recording, documentation, and analysis of user requirements. The

potential benefits of automating each of the above stages have been subsequently evaluated [2,4].

Production of user requirements specification above requires knowledge of the application area and problem-solving capabilities. It has so far been automated only very minimally by limiting narrowly the scope of the problem domain in application-specific packages (e.g., [5]). Balzer [6] has suggested that the computer could acquire a problem domain through an interactive session in a natural language. Yet success with such an approach probably will not be realized until major advances are made in artificial intelligence. Therefore in MODEL the user is assumed to have knowledge of the application area while the automatic programming system has design and programming knowledge.

Some progress has been made in the partial automation of expressing problem requirements formally, automatically analyzing system functional specifications, and performing some physical design. These have been reviewed elsewhere [2,3], but a few are mentioned here. There are a number of so-called problem statement languages for expressing system functional specifications formally [7]. The importance of such languages was recognized at least theoretically quite early in works such as those of Young and Kent [8]. A notable example is the Problem Statement Language (PSL) to express system functional specifications [7,9]. The impact of such problem statement languages has been shown to be of

great benefit [4] in aiding the software development effort by giving the system designer a formal way to record user requirements that can be documented and analyzed automatically. Once functional specifications have been expressed formally, a further degree of automation has resulted from automatically checking the specifications and producing some of the physical design. Many projects (see for example [10,11,12,13,14,15,16]) have dealt with various aspects of automating the physical design process through formal models and structured systems development techniques. More limited aids to the systems analyst have actually been available previously in forms-oriented and tabular languages (see for example [17,18]).

Yet not much effort has gone into the automation of the program generation process (stage 4) based on analysis of user requirements; i.e., the above systems do not generate programs based on previous non-procedural specification, analysis, and design. It is with this area that this research has dealt. Other work in automatic program generation is mentioned in the end of Section 3 of this paper and compared to MODEL.

The programming phase has been recognized as a very time-consuming and tedious task. Even with recent generalized data base management systems (which have to some extent relieved their users of physical knowledge of the data base and details of access and have provided for some

data independence and central control, among other services), programming is extremely tedious. They generally require programming and procedural knowledge to write application programs.

The research reported here helps to bridge the automation gap by including the automation of phase 4 above -- the program design, coding, and debugging phase.

Section 2 of this paper provides a general overview of the MODEL language and Processor. Section 3 describes the main features of the MODEL specification language and provides an example of its use. Section 4 outlines the automatic program generation methodology used. Section 5 makes some concluding remarks and comments on further research.

2. Overview of the MODEL Language and Processor

The approach taken here is that the user has all the knowledge pertaining to the application area, while the automatic program generator will possess only and all the design and programming knowledge. The focus of the research reported here is on the automation of the programming phase whose input is a functional specification in a very high level, non-procedural language and whose output is an application program in a programming language. It deals primarily with the automatic generation of transaction

processing systems. Figure 1 depicts the roles of the user and the software system which accomplishes this. The letter references below are to the figure. In order to express such specifications, a formal non-procedural language called a "Module Description Language" (MODEL) has been developed in which the user describes desired programs [a]. The language is used to describe non-procedurally objects in various categories:

1. The input data to be processed
2. The output and updates to be produced
3. The computation and decision rules to be used.

Unlike programming languages, the user submits to the Processor descriptive statements in this language in <u>any order</u> that he may desire, and can do so in increments over a period of time. There are no control structures or other control flow statements in the user's language, because the language is concerned with data description, data flow, and information relationships. Each statement is used to describe a unit of data, a computation or decision rule, and is independent of other statements. The user is able to concentrate on one unit of information or one computational requirement at a time and compose statements as they are conceived without regard to their location in the rest of the specification. The Processor puts the statements into the proper sequence later. The MODEL language is described in more detail in section 3 of this paper with an

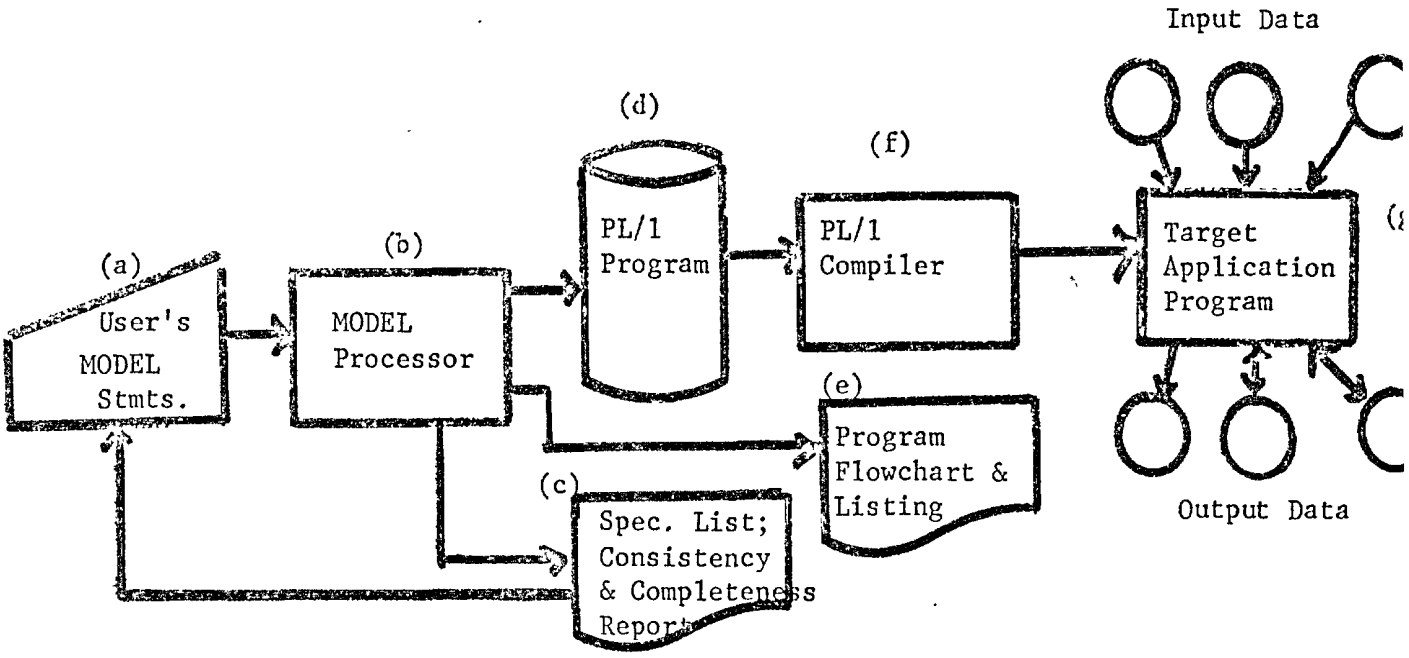

Figure 1

Overview of the MODEL System

illustrative example.

The MODEL Processor [b] (hereafter referred to as the Processor) analyzes the set of statements obtained from the user after storing them in an internal associative form. It analyzes the user statements for their consistency, completeness, and checks for ambiguities, informing the user of these in a report [c]. Thus, a complete and consistent specification of a program would be achieved through several interactions with the Processor. If analysis of the MODEL statements results in a complete and consistent program specification, the Processor designs the program and generates a complete application program in the PL/1 language [d]. It also generates various reports to the user about the specification and the generated program [e]. The generated program can then be submitted to a compiler [f], and subsequent execution of the target program proceeds [g].

A more detailed description of the MODEL Processor and its methodology is outlined in section 4 of this paper. A preliminary prototype processor embodying the system concepts has been developed.* It was designed and implemented in PL/1 on an IBM 370/168.

---

* The version of MODEL reported in this paper is the original one developed by the author in 1974-1975. Revisions and improvements to the language and system are under way in various directions both at the Univ. of Michigan and Univ. of Pennsylvania. These are mentioned at the end of the paper under "Further Research".

3. The Program Specification Language-- MODEL

This section discusses the MODEL language in greater detail. After a general description of the language characteristics, an illustrative example is presented. That is followed by a discussion of the class of problems currently describable in MODEL and the limitations of the current language and system. This section then ends with a brief comparison with other related languages and systems.

MODEL language features

The "Module Description Language" (MODEL) is intended to be used by a system designer to state the requirements of a desired application program non-procedurally. As stated above, it inherently differs from programming languages in several respects. It is a non-procedural language, which has been loosely defined as a language that expresses "what" rather than "how" [e.g., 19]. MODEL is characterized as non-procedural for the following additional, explicit reasons. First, the statements are all descriptive or declarative, not imperative. Each statement given by the user stands independently and is used to describe a unit of data, a relationship, a desired subset of records, a computation, or a decision rule. More important is the lack of necessity on the part of the user to compose statements in any particular order. The order or sequence in which statements are given by the user is immaterial, since all sequencing of tasks or

events to be performed are deduced by the Processor. Therefore, there are no control structures in the language. This ability by the user to submit statements in any sequence reduces the expertise required on the part of a user. It enables him or her to concentrate on and submit one unit of information or computational requirement at a time. Unlike conventional language compilers, the MODEL system allows the system designer to add statements to the specification without regard to their location in the specification. It allows statements to be added in incremental stages and provides feedback on the completeness and consistency of the specification between iterations. (Examples of such feedback are given later.) In short, there is no computer programming knowledge necessary in order to compose MODEL statements in several respects: there is no procedural or sequential thinking required, there is no reference to computer programming terminology, and there is an absence of concern with processing details such as data access, program "housekeeping" tasks, control flow, etc.

MODEL language statements and an example

A description and examples of the statements in MODEL follow. One limitation to the use of MODEL in its present form is the fact that it is a formal language and has a somewhat rigid syntax. Although it is a specification language and not a programming language, it nevertheless

would require some training on the part of a systems analyst.

A specification in the MODEL language consists of several parts: a header which has a list of source and target data files of the module, a data description section, record selection rules, and assertions that state computational or decision requirements.

In order to describe and exemplify the nature of the language, Figure 2 presents a system flowchart of a sales problem. There are three source files: the transaction from the point-of-sale terminal, an inventory stock status file, and a customer file of charge account information. The output files are the sales slip to be produced at the terminal, and the updated inventory and customer records. The program to be specified is to process the transactions producing the sales slip, and to update the inventory and customer records to reflect the sale. For tutorial purposes, Figure 3 shows the specification of a small subset of this problem in the MODEL language. In the example given, only the transaction, inventory, and sales slip files are shown and the files have only a small subset of the fields that would actually be carried in such files. A subset was chosen here for the purposes of a sufficient example, and it will be used to describe the Processor methodology in Section 4. Furthermore, the specification depicted shows only two tasks to be carried out: to compute the charge and to update the

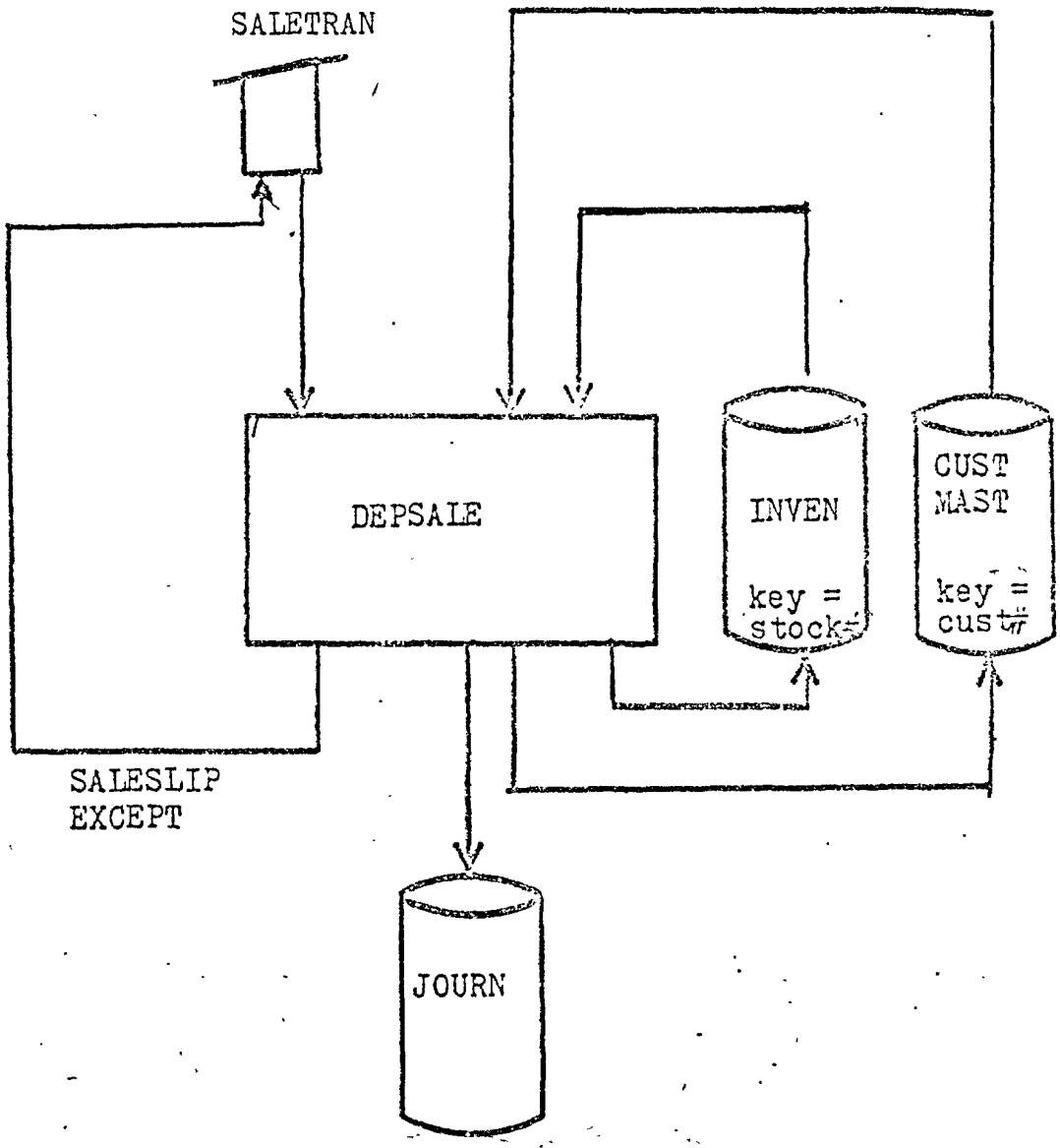

Figure 2 : Illustration of Department Store Sale

```
/*************************************************************/
/*                                                           */
/*          "MINSALE" PROGRAM SPECIFICATION                  */
/*                                                           */
/*************************************************************/

 1   MODULE: MINSALE;
 2   SOURCE FILES: SALETRAN, INVEN;
 3   TARGET FILES: SALESLIP, INVEN;

/*************************************************************/
/*                                                           */
/*            FILE DESCRIPTIONS                              */
/*                                                           */
/*************************************************************/

 4   SALETRAN IS FILE(RECORD IS SALEREC);
 5   SALEREC IS RECORD(CUST#,STOCK#,QUANTITY);
 6      CUST# IS FIELD(CHAR(5));
 7      STOCK# IS FIELD(CHAR(7));
 8      QUANTITY IS FIELD(NUMERIC(3));

 9   INVEN IS FILE
      (RECORD IS INVREC, STORAGE IS INVDISK, KEY IS STOCK#);
10   INVREC IS RECORD(STOCK#,SALEPRICE,QOH);
11      STOCK# IS FIELD(CHAR(7));
12      SALEPRICE IS FIELD(NUMERIC(5));
13      QOH IS FIELD(NUMERIC(5));
14   INVDISK IS DISK(ORGANIZATION=ISAM,UNIT=3330);

15   SALESLIP IS REPORT(REPORT_ENTRY IS SLIPREC);
16   SLIPREC IS REPORT_ENTRY(CUST#,STOCK#,CHARGE);
17      CUST# IS FIELD(CHAR(5));
18      STOCK# IS FIELD(CHAR(7));
19      CHARGE IS FIELD(NUMERIC(8));

/*************************************************************/
/*                                                           */
/*            INTER-RECORD RELATIONSHIPS                     */
/*                                                           */
/*************************************************************/

20   SALETRAN.STOCK# IDENTIFIES INVREC;

/*************************************************************/
/*                                                           */
/*                  ASSERTIONS                               */
/*                                                           */
/*************************************************************/

21   CALCCHRG: CHARGE=QUANTITY*INVEN.SALEPRICE;
22   UPDQUANT: NEW.INVEN.QOH=OLD.INVEN.QOH - QUANTITY;
```

Figure 3: Sample Set of MODEL Statements

quantity on hand in the inventory record. In this sample, only one item purchased is assumed, but the specification could be expanded to handle multiple items.

Looking now at the sample specification, the header is a formalization of the system flowchart, showing the name of the desired program and the input and output files. In the data description portion of the specification, a network-like model is used. Each record type is described with its component groups and fields. Such data descriptions are not unlike a COBOL or PL/1 hierarchical structure. The individual fields are given their specific attributes, such as field type and length. Furthermore, various statements describing inter-record relationships are provided to handle a network structure. In this example, the STOCK# field in the transaction file corresponds to an inventory record in the inventory file. All the above such data descriptions could be stored in a library for use by other program specifications using those files.

In order to describe the specific tasks of the program, selection rules can be provided. These are logical expressions for subsets of data to be processed. In this example, none are stated and therefore all records are processed as a default. Examples of statements for record selection rules are given after discussion of this sample problem.

The computational requirement rules, called "assertions" in MODEL, are specified to provide the information relationships, data computations, and decision rules. It is important to note that these could appear in any order and do not in themselves denote anything about sequence of events. An assertion of the form "CHARGE=QUANTITY*PRICE" is to be regarded as an algebraic equation, and not as an assignment statement as in a programming language. In fact, the entire set of statements in the specification could be shuffled with no effect on the meaning.* Statements in conventional programming languages such as "total=total+amount" would of course not be in the MODEL language for they are procedural and could have no non-procedural interpretation. Such computations as totals would be performed in this language by high-level functions. Therefore, in order to augment the basic arithmetic and logical operators, a library of common high-level functions, such as for totaling or counting, has been built. For example, a statement such as the following could be specified:

---

* The one exception to the principle that statements can appear in any order is the following: Since the same field name may appear in more than one file, an assumption is made by the Processor that a field described in a FIELD statement is associated with the file most recently described.

TOTCHRG=TOTAL(CHARGE);

where TOTCHRG could be in an output report or a field that could be used in other statements.

In the context of an expanded version of this example, the analyst could specify rules for decisions, accounting, and other rules that would indicate dispositions in certain eventualities, such as when a stock item is not available for inventory or when a purchaser exceeds the allowed credit limit. The accounting rules could specify the determination of charges for purchases and the method of determination of the customer's balance.

In the sample of Figure 3, two computational statements are given. The first is a straightforward computation for the charge of an item. In the second statement, the words OLD and NEW before a name are used to distinguish between corresponding field names in a file that is both input and output. Here the quantity-on-hand field in the inventory record is updated. The other fields in the record are assumed to be unchanged.

After submitting a set of statements such as those in Figure 3 to the Processor, the user of MODEL gets feedback as to the correctness, consistency, and completeness of a set of statements beyond the feedback of a conventional compiler. The user is informed in the form of listings and reports as to inadequate descriptions, contradictory

statements, and assumptions made by the Processor. Examples of logical inconsistencies and incompleteness detectable by the Processor are circular computational definitions, undefined output fields, two contradictory computations for the same field, etc. Section 4 of this paper outlines in more detail how the system processes a set of MODEL statements, including the feedback the user gets. A complete and consistent MODEL specification would be achieved through several interactions with the Processor.

There are other features of the MODEL language not exemplified in the subset of Figure 3 such as the capability to define subsets of files, the capability to define conditional assertions, and the capability to define operations on repeating groups and fields. Some brief examples of such statements follow.

In order to provide a way of specifying logical expressions to determine which subset of a file is to be considered for a desired application program, a subset selection statement is provided. For example, to restrict the set of transactions to be processed to only those records that have a quantity-ordered of more than 100, the user could have the following statement:

SUBSET.SALETRAN: QUANTITY > 100;

In other words, any record in the SALETRAN file not meeting this criterion is excluded.

MODEL lets the user state computational rules that hold only under certain conditions, in two parts. In the first part, the user can designate a name for the condition, and give the corresponding Boolean expression. For example, to designate a condition called SALE as occuring when the customer's old balance (in the CUSTomer file) + total charge is not greater than his credit limit, the user could write:

CONDITION: SALE: OLD.CUST.BALANCE + TOTCHRG <= OLD.CUST.CREDLIM;

Then anywhere else in the specification the user can define one or more computations that are contingent upon a condition defined elsewhere. For example, to update a customer balance only when the SALE condition has been met, the user can write

IF SALE THEN NEW.CUST.BALANCE = OLD.CUST.BALANCE + TOTCHRG;

Keep in mind that these could be submitted in any order. The reason for splitting up conditional relationships in this way is that other computations could reference the same condition and could be added later in any location. Thus, the user can insert computations that are to take place under certain eventualities as they occur to him or her. It

will be the Processor's task to group the actions for a condition together at the appropriate place. On the other hand, if a user finds the traditional IF...THEN...ELSE statement (such as in PL/1) more convenient, the Processor would accept it as well.

A more detailed discussion of these language features is actually beyond the intended scope of this paper. The reader is referred to [1] for a more complete description and examples of the language.

Class of problems and some current limitations

From the previous sections, it can be inferred that the MODEL language can be used to describe desired application programs in many areas of data processing. MODEL is particularly suited to the automatic generation of traditional transaction processing systems where transactions are processed, master records are updated, added, or deleted, and reports are produced.

The features of MODEL as outlined above enable it to describe almost any aspect of application programs. However, the current version of MODEL has several limitations. For one thing, as mentioned, the somewhat rigid syntax of MODEL in its current form limits the range of users enabled to use it, but a more user-friendly front-end interface to it could alleviate this problem.

The current version does not provide for report formatting facilities. Programs generated by the Processor treat output reports as output files. This is not considered a conceptual drawback, since report generation facilities are known technology and could be incorporated in future versions.

Although not a problem in practice, another limitation is in the use of computational statements. The lack of capability to specify sequence provides a great degree of ease of use and does not in itself pose any limitations. However, computational statements are limited to single equations with single unknowns on the left-hand side; that is, statements of the form Y=arithmetic-expression or Y=F(X1,...,Xn). Thus it is not possible to generate new algorithms with the system, nor is that its intent. It is expected that the vast majority of data processing programs could be handled with a dozen or so high-level functions (such as totalling, counting, maximum, minimum, etc.) in a library.

Another assumption made by the current version is that the physical organization of the files used by the target application program is pre-designed by a system designer before MODEI is used as a tool for automatic program generation. The current version uses the operating system sequential and indexed sequential access methods in its implementation of input and output statements for the

generated application program. However, one of the extensions of this language under way is directed towards generation of application programs that use generalized data base management systems.

Finally, the Processor currently generates one program per specification. It is envisioned that in the future a further optimization phase would modularize the generated program based on efficiency and other considerations.

Comparison with other languages and systems

Comparing MODEL to PSI [9], PSL does not have sufficient facilities in it in order to generate an executable program, nor has that been its focus. By the same token, MODEL is a much smaller and simpler language because it has only those facilities necessary in order to specify and generate a program, and is not concerned with the target environment as a whole as is PSL.

Other "automatic programming" projects have had similar goals as MODEL, but preliminary indications (e.g., [21]) are that they require more procedural knowledge and are not as user oriented.

There are many other classes of non-procedural languages but they are for use in areas other than generation of application programs (such as data translation).

Some query languages show some similarity to MODEL, but they are generally concerned with one-time ad-hoc retrieval rather than with generation of transaction processing application systems for repetitive use.

Comparisons could also be made between MODEL and other program generators, but they deal with other problems. For example, [22] deals with automatic generation of DBTG information retrieval programs from specifications of "relations" and not with generation of transaction processing systems.

4. The Automatic Program Generation Methodology

The Processor has the task of accepting such specifications and generating a PL/1 program complete with all the necessary declarations, control structures, sequenced statements, input/output statements, housekeeping tasks, etc.

The system goes through various phases, labelled PHASES I-V in Figure 4, in order to generate the program from the specification. Although the detailed algorithms are beyond the scope of this paper, the general methodology and the five phases are outlined below, and the reader interested in greater detail may refer to [1].

Phase (I): Syntax analysis of the MODEL module specification

Syntax analysis technology is well known and follows that of a compiler. In this phase, the provided MODEL specification is analyzed to find syntactic and some semantic errors. This phase of the Processor is itself generated automatically by a meta-processor called a Syntax Analysis Program Generator (SAPG), whose input consists of syntax rules provided through a formal description of the MODEL language [1]. Thus, changes to the syntax of MODEL during development and in the future can be made easily.

A further task of this phase is to store the statements in a simulated associative memory for ease in later search, analysis, and processing [1]. Some needed corrections and warnings of possible errors are also produced in a report for the user.

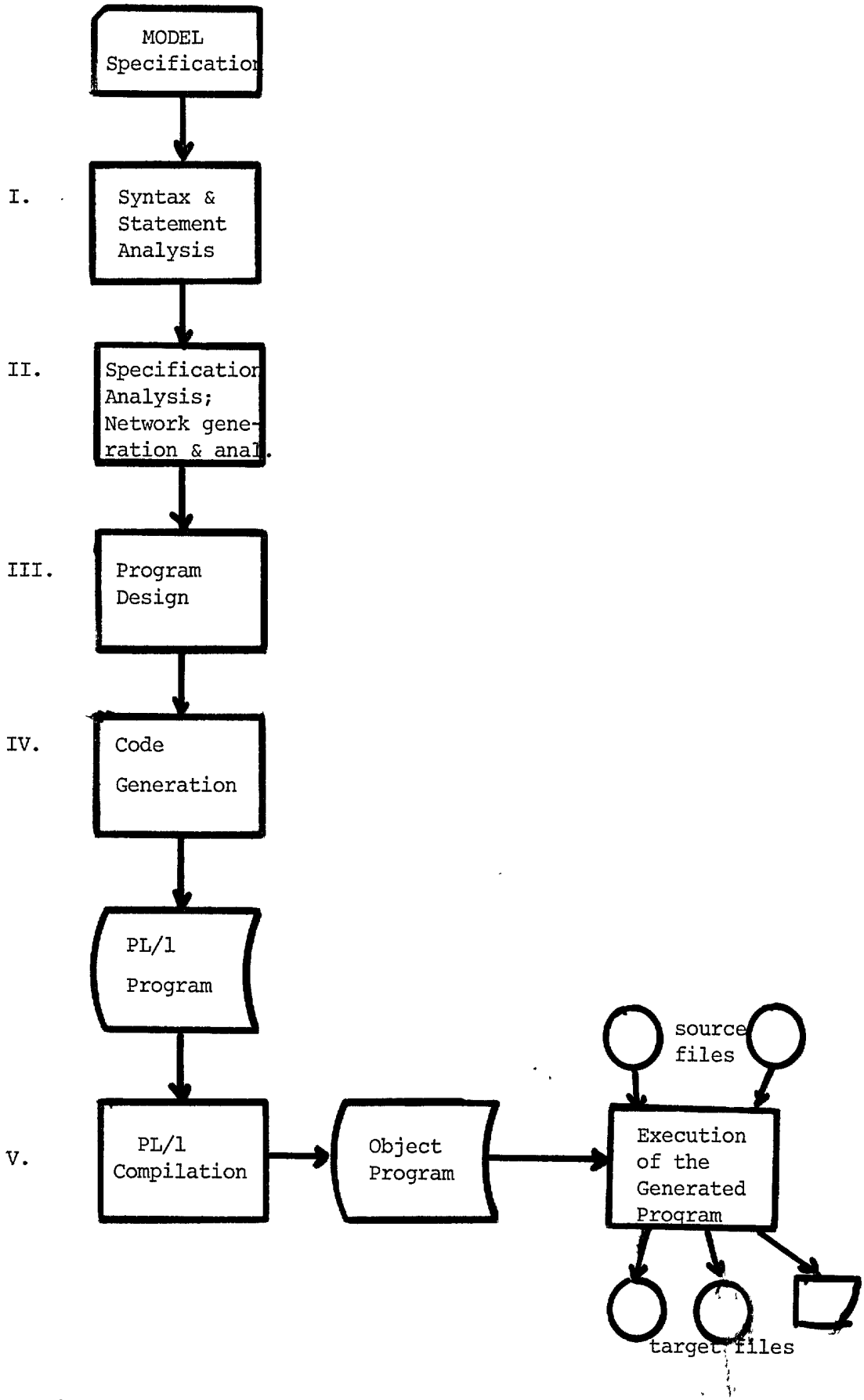

Figure 4: Phases of the MODEL Processor

Phase (II): Analysis of MODEL specification

In this phase, precedence relationships are determined from analysis of the MODEL data descriptions and assertions, and the specification is analyzed to determine the consistency and completeness of the statements. Each MODEL statement may be considered to be an independent, stand-alone statement. The order of the user's statements is of no consequence. However, in analysis of the statements, precedence relationships are determined on the basis of description components. Precedence rules, based on programming knowledge, have been built into the Processor. Two frequently used precedence rules are hierarchical precedence rules (e.g., having to read a record before its component fields can be used) and value dependency precedence rules (e.g., having to compute A=B+C before D=A*C). These relationships are used to form a precedence graph on which the completeness, consistency, and ambiguity of the specification can be checked. Reports are produced for the user indicating the data, assertions, or decisions that have been inadequately described, assumptions that have been made by the Processor, or contradictions that have been found, and reports are provided to cross-check the descriptions. For example, if an output field has not been given a value by a user rule, the Processor has a series of rules it uses in order to determine its value, such as using the value of same-named field in an input file. If a value

cannct be assumed by the Processor, an incompleteness message is printed tc the user and he is prompted to complete the missing infcrmatien. Examples of inconsistencies that are detected by the system are circular definitiens (e.g., A is a functien cf B, B is a function of C, and C is a function cf A) cr contradictory computational rules for the same field under conditiens which are not mutually exclusive. Thus, it is envisioned that a complete and consistent specification of a program would be achieved through several iteratiens and through interaction with the Processcr.

An example cf a directed graph that represents the prcblem expressed in MCDEL in the previous section is given in Figure 5. Note that the ncdes represent the records, fields, corputations, etc., as described by the user. They could have been entered incrementally and in any sequence. The system "draws the arcs" based on its internal precedence rules. Many representatiens cf such a directed graph appear in the literature. Figure 6 shows an adjacency matrix which MODEL uses for the prcblem described abcve. The various entries in the matrix show the relationships existing between the various objects. This matrix representatien enables easy checking cf ccnsistency and completeness criteria described above. For example, a cycle enumeration algcrithm on the matrix can be used to detect circular corputational definitiens.

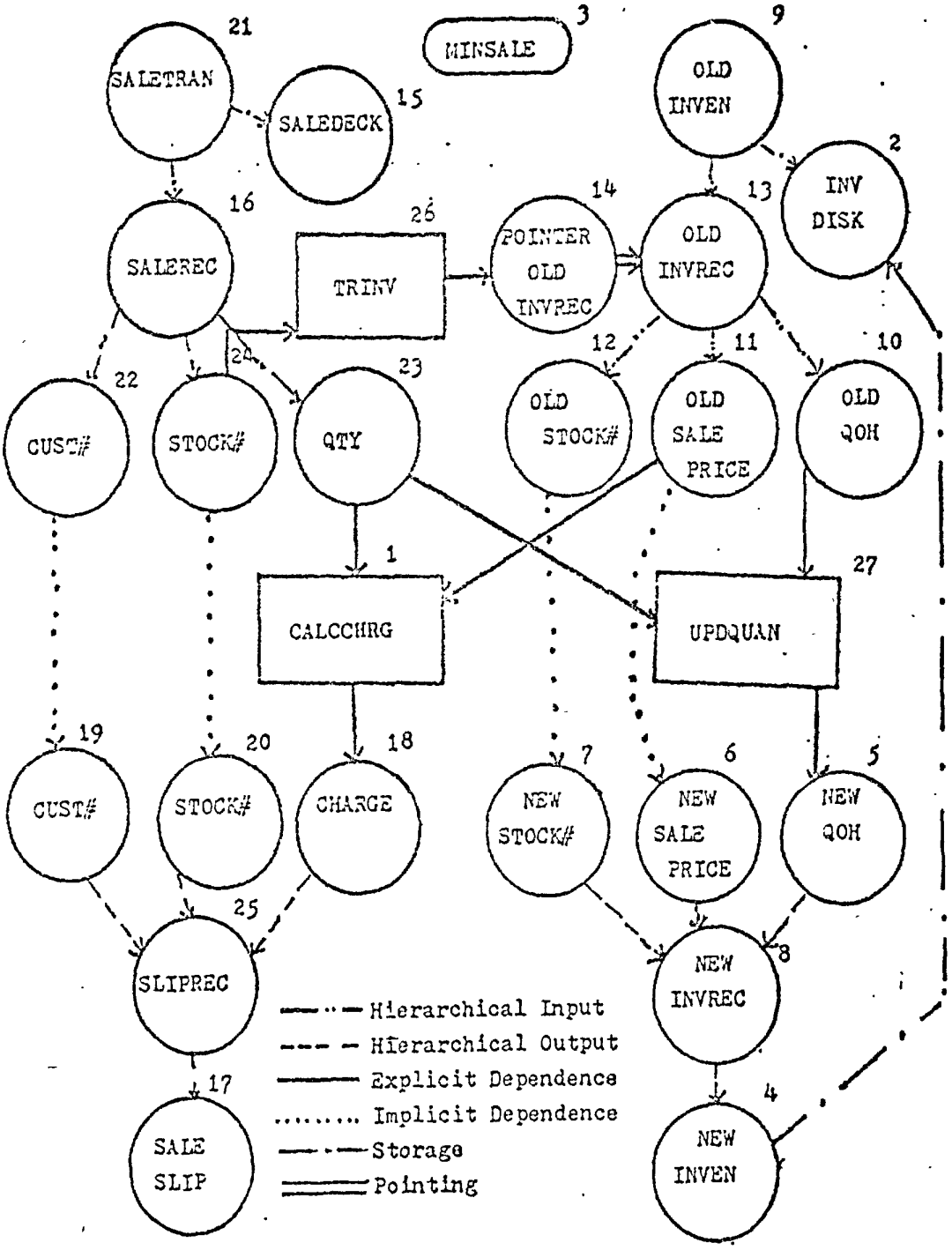

Figure 5

Digraph for MODEL Specification of Figure 3

ADJACENCY MATRIX OF NAME RELATIONSHIPS

		5	10	15	20	25
1	CALCCHRG	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 1
2	INVDISK	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
3	MINSALE	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
4	NEW.INVEN	0 6 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
5	NEW.INVEN.QOH	0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6	NEW.INVEN.SALPRICE	0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
7	NEW.INVEN.STOCK#	0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8	NEW.INVREC	0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
9	OLD.INVEN	0 6 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
10	OLD.INVEN.QOH	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
11	OLD.INVEN.SALPRICE	3 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
12	OLD.INVEN.STOCK#	0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
13	OLD.INVREC	0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
14	POINTER.OLD.INVREC	0 0 0 0 0 0 0 0 0 0 0 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
15	SALEDCK	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
16	SALEREC	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0
17	SALESLIP	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
18	SALESLIP.CHARGE	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
19	SALESLIP.CUST#	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
20	SALESLIP.STOCK#	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0
21	SALETRAN	0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 1 0 0 0 0 0 0 0 0 0 0 0
22	SALETRAN.CUST#	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0
23	SALETRAN.QUANTITY	3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3
24	SALETRAN.STOCK#	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 3 0 0
25	SLIPREC	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0
26	TRINV	0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0
27	UPDQUAN	0 0 0 0 3 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

A NON-ZERO ENTRY IN ROW I & COLUMN J INDICATES THAT ITEM I PRECEDES ITEM J. THE CODE REPRESENTS THE FOLI

1 = HIERARCHICAL(SOURCE); 2 = HIERARCHICAL(TARGET); 3 = EXPLICIT DEPENDENCY; 4 = IMPLICIT DEPENDENCY;
5 = POINTING RELATIONSHIP; 6 = STORAGE RELATIONSHIP; 7 = CONDITIONAL DEPENDENCY

Figure 6. Weighted Adjacency Matrix for Sample MODEL Specification

Phase (III): Automatic program design and generation of sequence and control logic

This phase of the Processor determines the sequence of execution of all events implied by the specification, using precedence graph theory algorithms, and thereby determines the sequence and control logic of the desired module. The result of a topological sorting of the graph presented above is a table such as the one shown in Figure 7. This table forms the skeleton of a flowchart for the target program, as it shows the order of the nodes in their executable sequence. It also shows the ranks of these events, which as a by-product denotes some of the possible parallelism. Design of the object program proceeds with scope and iteration analysis and flow optimization. Iterations are based on the existence of repeating groups or fields in the data description, or are designed for processing a set of records sequentially. The result of this phase is a set of data structures representing the desired sequence of processes and flow of events, sequenced, ranked, and optimized in their order of execution.

SEQUENCE OF PROCESSING

ORDER VECT. INDEX	ORDER VECTOR	NAME	RANK
1	3	MINSALE	0
2	9	OLD.INVEN	0
3	21	SALETRAN	0
4	15	SALEDECK	1
5	16	SALEREC	1
6	22	SALETRAN.CUST#	2
7	23	SALETRAN.QUANTITY	2
8	24	SALETRAN.STOCK#	2
9	19	SALESLIP.CUST#	3
10	20	SALESLIP.STOCK#	3
11	26	TRINV	3
12	14	POINTER.OLD.INVREC	4
13	13	OLD.INVREC	5
14	10	OLD.INVEN.QOH	6
15	11	OLD.INVEN.SALPRICE	6
16	12	OLD.INVEN.STOCK#	6
17	1	CALCCHRG	7
18	6	NEW.INVEN.SALPRICE	7
19	7	NEW.INVEN.STOCK#	7
20	27	UPDQUAN	7
21	5	NEW.INVEN.QOH	8
22	18	SALESLIP.CHARGE	8
23	8	NEW.INVREC	9
24	25	SLIPREC	9
25	4	NEW.INVEN	10
26	17	SALESLIP	10
27	2	INVDISK	11

Figure 7

Digraph Nodes Sequenced in Precedence Order

Phase (IV): Code generation

At this point in the process it is necessary to generate, tailor, and insert the code into the entries of the flowchart to produce the program. Code is produced in two steps for purposes of modularity and independence of the target language. The first step produces a language-independent version of the flowchart-entries, as noted above, while this second step produces code in the PL/1 programming language. In particular, input/output statements are generated whenever the flowchart indicates the need for records. Calls to procedures embodying the assertions are generated in the appropriate places in the flowchart. Wherever program iterations and other control structures are necessary, program code for them is generated, such as for repeating groups or fields. Declarations for object program data structures and variables are generated. The product of this phase is a complete program in a high level language, PL/1, ready for compilation and execution. A listing of the generated program as well as the flowchart-like report is produced. This documentation is not expected to be of significance to the casual user, but it would be available for a computer programmer in the event that it may be needed for deeper understanding of the procedure.

Phase (V): Program compilation and execution

This phase starts with the PL/1 program module that has been automatically produced. With the generated PL/1 program being submitted to the PL/1 optimizing compiler, program compilation and optimization of code on the machine-language level is effected. The automatically generated program is then available for use in execution.

5. Concluding Remarks and Further Research

This paper has outlined and exemplified the basic features of a non-procedural specification language for describing modules of an information system and a processor for generating programs automatically from specifications expressed in that language. The system described here has demonstrated the feasibility of such an approach. It is expected that such a language and system could be an important step towards the automation of the software development process if given strong industrial level support, reliability, and funding. Some conclusions are already evident. Such a system reduces the amount of expertise and time needed to generate today's typical data processing program. Since MODEL is non-procedural, it enables the user to concentrate on describing data and their inter-relationships by providing a set of statements that can appear in arbitrary order. The Processor, unlike conventional compilers, is able to deduce the sequence of

events and checks much of the user's logic and provides effective feedback. In addition, it produces the desired program complete with sequence and control logic, declarations, input/output, relieving the user of such procedural thinking.

From preliminary indications, the execution time efficiency of a program generated by the Processor is good. Unlike "generalized packages," the Processor generates an ad hoc program for a particular problem. Therefore, the code generated is peculiar to a given use and contains no unnecessary instructions. Unlike some "pre-compilers," the system requires no procedural knowledge to use nor is it limited to particular application areas.

Further research is taking several directions at various institutions including the University of Michigan and the University of Pennsylvania. Refinements are being made in the language and in the degree of logical analysis that the system can perform on a specification. Some work is being conducted on automatic data aggregation and modularization of generated programs based on efficiency and other considerations. At the University of Michigan, there is also an effort to apply some of this technology to the automatic design, generation, and restructuring of programs that use network data base management systems. The data manipulation language (DML) of many current network model

data base management systems is highly procedural and technical, and requires "data navigation" and other procedural knowledge by an experienced programmer. Another language based on the principles outlined in this paper is now being developed for use in defining non-prodcedurally application programs for network data base management systems. An interface between PSL [3,9] and MODEL is also under investigation, so that PSL users could use the code generation capabilities of MODEL.

Bibliography

[1] Rin, N. Adam "Autcmatic Generation of Business Data Processing Programs from a Non-Procedural Language," Ph.D. Dissertation, Computer and Information Science Department, University of Pennsylvania, 1976.

[2] Prywes, N. S. "Autcmatic Generation of Software Systems -- a Survey," Data Base, Vol. 6, No. 2, Fall 1974.

[3] Teichrcew, D. and Sayani, H. "Automation of Systems Building," Datamation, Aug. 1971.

[4] Teichrcew, D. And Merten, A. "The Impact of Problem Statement Languages on Evaluating and Improving Software Develcpment Perfcrmance," Fall Joint Computer Conference, 1972.

[5] Hax, A. C. and Martin, W. A. "Automatic Generation of Custcmized Mcdel Based Infcrmation Systems for Cperaticns Management," Proceedings on the Wharton Ccnference on Research on Computers in Crganizaticns, Philadelphia, Oct. 1975, edited by H. L. Mcrgan, pp. 117-121.

[6] Balzer, R. M., "A Global View of Autcmatic Programming," also Memcrandum cn "Autcmatic Programming," USC Information Sciences Institute, Marina del Rey, Califcrnia, Sept. 1972.

[7] Teichrcew, D. "A Survey of Languages for Stating Requirements fcr Ccmputer- Based Information Systems," Fall Jcint Computer Conference, 1972.

[8] Ycung, J. W and Kent, H. K "Abstract Formulation of Data Processing Prcblems," National Cash Register Co., also given at 13th ACM National Meeting, June 1958.

[9] Hershey, Rataj, Teichrcew, and Berg PSL Manual, ISDOS Working Paper #68, University of Michigan, Cct. 1973.

[10] Severance, D. G. "Some Generalized Modeling Structures for Use in Design cf File Organizations," Ph.D. Dissertation, University of Michigan, 1972.

[11] Cardenas, A.F., "Evaluation and Selection of File Organization -- a Model and System," Communicatcns cf the ACM, Sept. 1973, pp. 540-548.

[12] Nunamaker, J. F. Jr. "On the Design and Optimization of Infcrmaticn Processing Systems," Ph.D Dissertation, Case Western Reserve University, June 1969.

[13] ---------- "A Methcdology for the Design and Optimizatcn of Infcrmatien Processing Systems," AFIPS Proceedings, 1971, SJCC, pp. 283-294.

[14] ---------- "Processing Systems Optimization Through Autcmatic Design and Reorganization cf Program Modules," CSD TR77, Purdue University, Dec. 1972.

[15] ---------- SCDA, ISDCS Working Paper #36, University of Michigan, Feb. 1971.

[16] Graham, Clang, and DeVeney "A Software Design and Evaluation System," Communications of the ACM, Feb. 1973.

[17] ADS, National Cash Register Cc., 1968.

[18] TAG Sales and Systems Guide, IBM, GY20-0358-1, 1968.

[19] Leavenworth, Eurt M. and Sammet, Jean E. "An Overview cf Non-prccedural Languages," SIGPLAN Notices, Proceedings of a Sympcsium on Very High Level Languages, Santa Mcnica, California, March 28, 1974.

[20] Beck, Ieland "An Approach to the Creation of Structured Data Processing Systems," SIGMOD Proceedings, 1976.

[21] Ruth, G. R. "Status of Protosystem I," Automatic Programming Grcup, Massachusetts Institute of Technolcgy, 1976.

[22] Gerritsen, R. "Understanding Data Structures," PhD. Dissertaticn, Computer Science Department, Carnegie-Mellcn University, 1975.