DESIGN REPORT

ON

OPERATIONS ON ARRAYS AND DATA STRUCTURES

by

A. Pnueli* and N.S. Prywes

Prepared With Support From
The National Science Foundation
Under Grant MCS-79-00298

December 1980
Joint Computer Science-Economics Project
Department of Computer and Information Science
Moore School
University of Pennsylvania, Philadelphia, Pa. 1910

* Prof. A. Pnueli is affiliated with the Weizmann Institut
Rehovot, Israel. The reported research was conducted wh
serving as a consultant to the University of Pennsylvani

ABSTRACT

The ability to specify computations of aggregate data is very important for mathematically oriented applications and for manipulating data bases. This report describes the concepts, syntax, semantics and implementation of the capability to specify computations through use of matrix and vector algebra equations, and through equations with functions that operate on multi-level list and tree structu

These new capabilities are discussed in the context of the MODEL language. The MODEL language is a non-procedural language for specifying computations. The two components o a specification in MODEL are a set of equations using gener and boolean algebra, for defining the relations between the variable, and the description of the structure and organiza of these variables including bases and reports. The MODEL processor conducts extensive checks of the mathematical completeness, non-ambiguity and consistency of a submitted specification and constructs a program for the computation the specified variables. The program is optimized to achie efficiency in use of the memory and computer time. The fin product is an efficient and reliable program in PL/1. All aspects have been extended to the new capabilities for spec operations on aggregate data.

The research reported here is a component of the joint Computer Science-Economics project. The objective of the p

is to investigate and devise a system for the specificat: simulation and estimation of very large complex econome: models which typically represent a cooperative developme: by independent, geographically dispersed, groups and ins* tutions. The new capabilities would greatly ease specif cations of sets of equations, of complex statistical pro cedures and of manipulation of data bases•

TABLE OF CONTENTS

TABLE OF CONTENTS/Cont's.

1. INTRODUCTION

1.1 Objectives and Background

The MODEL language is a very high level nonprocedural specification language for use in computer programming. Tl MODEL compiler accepts a specification of a computational task, written in the MODEL language, and translates it intc a computer program in a conventional programming language: PL/1* The purpose of this report is to describe the desigi of extentions to the MODEL language and processor which wi: enhance the expressive power of the system. The new opera- tions will offer the use of vector and matrix algebra as w< as other operations that will be applied to data structure: including merging and selection of data. These new operat are needed to obtain ease and economy in expressing comple; computational tasks*

The research reported here constitutes one of the tas! in a joint Computer Science-Economics project which'invest, gates new approaches to performing large and complex compu- tational tasks, in particular computations in econometri studies. The awareness of the inadequacy of present metho and systems is based on the experience with the LINK Inter national Trade Model at the Economics Department of the University of Pennsylvania. The LINK project is a coopera tive effort by 21 institutions which develop each respect: regional or country econometric models. These models are

integrated at the University of Pennsylvania into a worl wide econometric model. The LINK model is very large, co sisting at the present of extensive data files on the eco omies of the respective regions and countries and approxi ately 6,000 equations. To fully realize the results of this cooperative activity it is necessary to expand the L model progressively to a level of 20,000 equations. The of LINK and the independence in developing respective sub models are the main reasons why existing econometric mode ling systems have not been used in LINK. The computation tasks involved have required of the economists much skill in mathematics and programming. Large amounts of effort and costs have been involved. A most severe problem has been the difficulties in responding quickly to recent rap economic changes. This has been a serious deterrent to e experimentation and investigations. These factors indica need to find more powerful techniques.

The objectives of the joint Computer Science-Economi project at the University of Pennsylvania are as follows:

1) Economy and ease in specifying representation of data and equations.

2) Deeper analysis of the representation of data and equations to discover as early as possible any mathematical ambiguities, incompletenesses or inconsistencies, and the application of auto- matic correction to such problems. This is par-

ticularly important in order to reduce the labor in debugging and obtain reliable computations.

3) Efficiency in execution, especially in estimating the coefficients and evaluating solutions of very large equational systems.

4) Modularity and integrability of large scale computation and allowing distributed computation, whereby the respective country or region models could be computed in the local computers of the cooperating institutions, and the computers would communicate in a network to exchange the informat needed to perform integrated world-wide computati

These objectives of ease of expression, reliability, efficiency, modularity and distributed processing constitut key research areas in Computer Science. Hence the cooperat between Computer Science and Economics. Respective activi the project have been directed to these four objectives. This report is concerned with the first objective of provid economy and ease in specifying data structures and equation

Several systems have been developed to date for aiding economists in econometric modelling. The general approach in these systems has been: 1) to require use of a standard data structures for the data files and 2) to perform the com putation by interpreting equations provided by the user. Our approach has been to investigate the compilation of a program in PL/1 or other high level languages, based on

the user's arbitrary data structures and equations. This approach leads to a greater freedom and economy in specifying computations. It allows much deeper analysis of the specification and a much greater efficiency in execution of the computation. To achieve the above objectives we envisage a new type of a modelling system, based on the compilation approach. The research is based on the MODEL language and processor that have been developed at the University of Pennsylvania. In composing a specification in MODEL, the user describes the data structures of the variables and provides equations which define the desired output variables. In the present MODEL language, the variables in equations must be elements of arrays or fields (i.e., leaf nodes) in more general data structures.

The objective of the research reported here is to augment the system with operations on aggregates of the more elemental variables. By use of matrix algebra an entire set of equations may be represented as a single equation. Similarly, complex estimation methods may be expressed by a single expression using matrix algebra. Economy in representation of other data manipulation tasks is achieved by referring directly to structures that represent entire or major portions of data bases. Such operations consist of selecting a subset of the data bases, or of merging data bases or their subsets. All these operations

may be characterized as concerning higher level data struc
The objective of this report is then to describe the synta
semantics and an algebra for higher level data structures
their implementation.

1.2 The MODEL System

The MODEL language and processor were selected as the b for attaining the above objectives. This section briefly re views the advantages of the MODEL system. Section 2 of this port presents the MODEL language.

MODEL is a very high level nonporcedural language for specifying computation tasks. It consists mainly of declara of data strucutres and of equations that define variables. The MODEL language is thus distinguished in being purely descriptive. The user of MODEL needs to concentrate on the mathematical correctness of the specification and may ig nore to a large extent the efficency of representation or al gorithms. The MODEL processor accepts a specification in th MODEL language and translates it into a conventional procedu programming language: PL/1. In this translation it orders the arbitrarily presented equations into a procedural sequence of executable statements, and provides missing instructions, such as reading and writing of external data, loop controls of repetitive calculations, etc.

The MODEL processor performs in depth analysis of the specification to check mathematical consistency and complet eness and to achieve efficiency in execution of the compu- tational task. Much emphasis is also placed on the effici- ent utilization of memory space.

The MODEL language is domain independent and a general language. The language and processor have been demonstrated

as innovative and useful program construction tools in several application areas. There is a pronounced enhancement in the ease and naturalness of expressing a task i MODEL compared to conventional programming languages, and a distinctly higher reliability of the checked program. The l ratio of resulting object (PL/1) lines to the number of sou (MODEL) lines provided by the user is indicative of the amou of detail which the system spares the MODEL user. The syste was thoroughly excercised in business data processing and proved to be a great asset for managing a family of related tasks. It stands superior not only to conventional programming languages but also to other experimental very hi level languages under investigation.

The dissertation research of J. Gana (1978) examined th applicability of the MODEL system to econometric studies. This area is much more mathematically and computationally oriented than business data processing. Econometric, and other models, are essentially represented by sets of equatio which are also the basic elements in a MODEL specification. will be shown in this report, the MODEL approach is suitable for further augmenting the ease and economy in specifying co plex computation tasks found in complex modelling. By demon strating the advantages and promise of the MODEL system in those two widely differing fields - business data processing

which is data oriented, and modeling which is more computation oriented,— we plan to provide a sound argument for the universal applicability of the MODEL concept.

The purpose of this report is to describe the augmentations of the MODEL language by additional powerful constructs which will enhance the expressive power of the language* Some of the new operations will allow easier expression of typical data processing tasks such as selection of data merging of files, while the others will afford a compact vector and matrix notation for the more mathematically oriented applications. Yet, all these augmentations can be described under one heading as being operations" on high level structures,

The MODEL language in its current form is' 'element oriented Even though arbitrarily complex data structures can be describe in the language, the equations which express the relations between the variables must all be expressed in terms of variables which are data elements at the lowest level of the data structures • Thus to express the fact that an output record is a cop of an input record, we have to state this repeatedly for each element of the record. Instead it is proposed to extend the language to allow expressing directly the equality of entire data structures. Another example is the merging of two files. The need of explicitly writing an expression which will tell us where the i^{f}th element in the merged file comes from is by no means a trivial task. Thus, we will regard a

file merge as a global operation, which given two input f
produces, as though in one transformation, the merged fil
as a result. Similarly mathematical notation for vector
matrix operations and expressions will be used to denote
global operations to be applied to a vector or a matrix a
whole.

1.3 The Extensions to the MODEL System

The extensions are basically in two categories: matrix and vector algebra, and aggregate data structure algebra. In addition, to facilitate the use of these extensions it is proposed to expand somewhat, first, the present data description syntax and semantics, and second, assure efficient implementation through expanding the analysis of variable subscripting. These four areas are briefly described below.

1. Matrix and Vector Notations and Operation

This extension enables the user to write equations and expressions involving vectors and matrices, using the conventional mathematical notation of matrix algebra. The matrix (and vector) operations provided are: matrix (and vectors) addition, subtraction, multiplication, inversion and transposition. In addition all the component by component operations such as multiplication by a scalar are included. The special UNIT matrix is standardly provided. Unique naming conventions enables the selection of a vector or a matrix out of structures of higher dimensions. Thus any row or column of a matrix can be selected to be treated as (row or column) vector.

2. Aggregate Data Structures Notations and Operation

These operations consist of structure definition and a number of functions.

Structure Definition

The simple defining equation

A = B

can be used in the current version of the system only to denote the definition of the element (field) A by the value B. In the extended language such definition may be applied to any high level structures A and B provided they have similar structures. This definition implies that each field in A is defined to have the value of the corresponding field in B. Structures in A inherit any attributes (length, shape and scale) of the corresponding structures in B.

Functions:

Several structure functions are proposed. The most important, SELECTION and MERGE are briefly reviewed here. The SELECTION function selects a subset of components of an array, a file or other structures, and forms a new structure of the selected components. The selection can be done according to any specified condition. Current operations in MODEL produce structures with dimensions of the same size as the arguments of the operations. In the extended system the SELECTION function and MERGE functions enable the definition of arrays and lists of different size than the argument. The selected components can be elements or structures of higher level.

The MERGE function merges two linear structures, such as files or lists. The criterion by which substructures are selected from each of the structures may be generally specifiec by the user. This enables the concurrent use of two (or more) source files as a single file.

3. Variable Structures

In the current system the composition of a data structure is fixed, i.e. it must always consist of the same substructures or elements. The proposed extension enables the definition of data structures to describe several possible alternatives or variants. Each variant may consist of different substructures or elements. A special attribute variable, named DISCR, is used to define for each instance of the data the chosen structure variant. The variants must be explicitly defined by the user for input structures and are automatically defined by the system for output structures.

4. More General Subscript Expressions

Currently MODEL analyzes only subscript expressions of restricted forms. The analysis enables the system to decide whether the specification is well defined and to check the consistencies in shape of data structures. The analysis also leads to the construction of more efficient object code. Under the extension discussed here, subscript expressions of a more general type are analyzed as well.

This will enable the user to define size modifying tra
formations, in addition to the standard ones to be pro
vided by the select and merge functions. It will also
enable the system to generate efficient subprograms fo
the thus specified computations.

1.4 Outline of the Report

The reading of this report requires background knowledge concerning the present version of the MODEL language and the operation of the processor. Some of the background information is provided in this report. In particular Section 2 describes briefly the present version of the MODEL language which forms the base for the extensions in this report. An example of an econometric model specification is used as an illustration.

Section 3 presents the syntax and semantics of the proposed extensions with illustrative examples.

Basically the implementation is based on translating the higher level structure operations and equations into the more basic MODEL equations. Section 4 describes the source to source translation, i.e. translation of higher level assertions into elemental MODEL statements.

The efficiency in the object PL/1 program depends on the extension of the system to handle indirect subscripting. The implementation of the indirect subscripting related analysis is the subject of Section 5. Some background information on the analysis and scheduling performed in the MODEL system are provided in this section. For additional background the reader is referred to a paper by N. Prywes and N. Pnueli "Compilation of A Nonprocedural Specification Into A Computer Program", October 1980.

Finally the appendices discuss changes in the implementation of the syntax analysis, indirect subscripting variant structures, and other miscellaneous changes. Readin of the appendices requires prior familiarity with the documentation on "MODEL Program Generator" by A. Pnueli, K. Lu and N. Prywes March 1980.

A bibliography of relevant reports and paper on the MODEL system is provided at the end of the report.

2. THE MODEL SPECIFICATION LANGUAGE - EXCLUDING THE PROPOSED CHANGES

This section describes briefly the current version of the MODEL language and processor, which are used as a basis for the modifications proposed in subsequent sections of this report. MODEL is a general purpose language for specifying computation tasks. A specification in the MODEL language consists of an unordered set of statements. The statements in the language are primarily of two types: data description statements and equations which we call assertions. The data description statements describe the structure and attributes of the variables participating in the specification. The assertions define the values of some variables in terms of other variables. The variables appearing in a specification are designated in a header statements as source variables or target variables. The header statements are used to name the computational task and the aggregates of source and target data. The values of the source variables are considered to be available on external input files. Target variables are to be produced on external output or update files. Target variables may alternately be designated as interim, to indicate that they need not be retained as output. The two subsections below describe the syntax and semantics of data and assertion statements respectively.

We use in this section a reduced econometric model of Spain to illustrate the current syntax and capabilities of the MODEL system. The example is shown schematically in Figure 2.1 as consisting of three input files and one output file. The input or source files are:

1) SIMDEF, which consists of the parameters of the simulation (BEG_YR = beginning year, PD_SIM - number of simulation periods, NUM_CO - number of coefficients, NUM_VAR - number of variables, DELTA - the relative starting period for the solution, and LAG - the maximum lag periods referenced in equations).

2) COEFF, which consists of coefficients for the equations.

3) TIM_SER, which consists of a time series for eleven variables of which seven will be considered exogenous and four endogenous.

There is one output, or target file named SOLUTION. This file is actually a report of the simulated variables for the periods of the simulation.

The equation box in Figure 2.1 specifies the dependencies of the target variables on other variables in the specification.

The schematic diagram of Figure 2.1 is represented in the header to the MODEL specification shown in Figure 2.2. There are 3 statements in the header shown in lines 1-5.

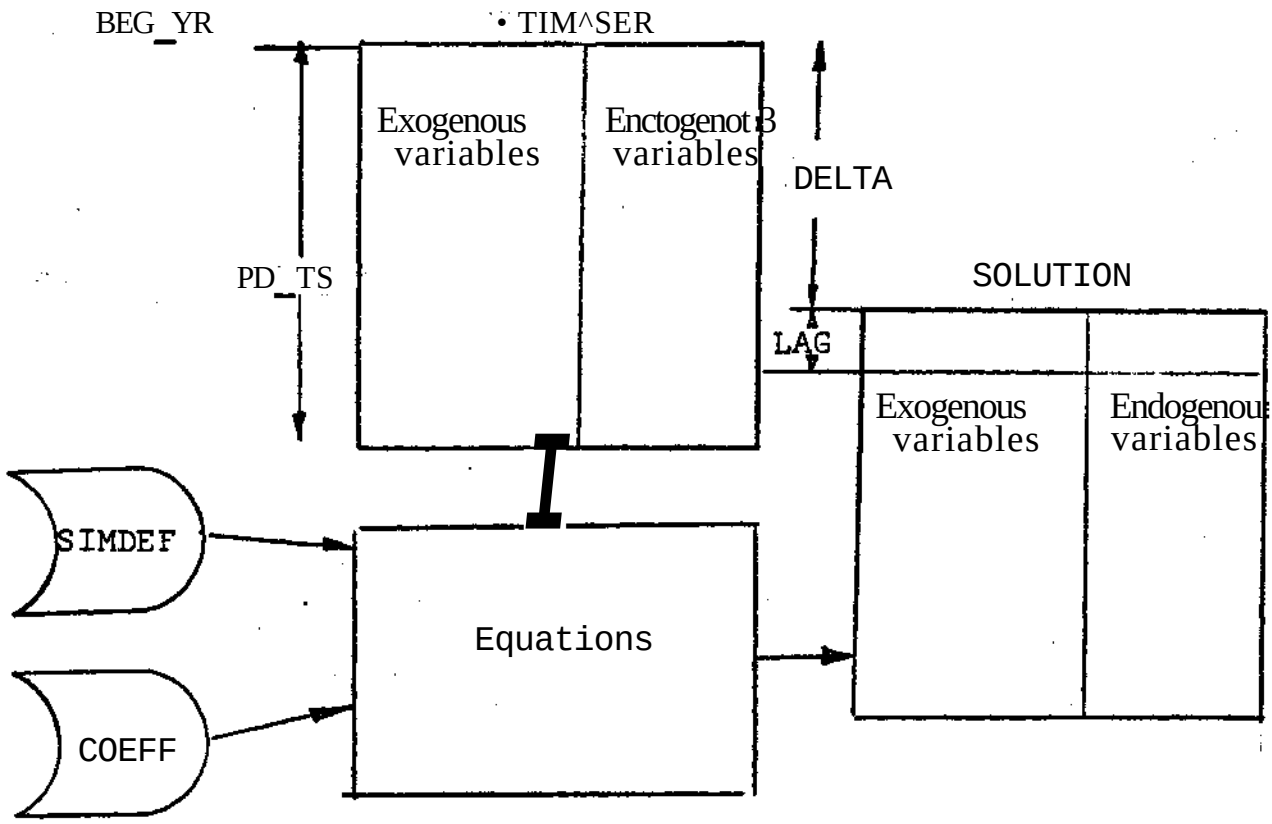


Fig 2.1. Schematic Diagram of Example of a Small Econometric Model,

```
/*****************************************************************/
/*                    SPAIN MODULE SPECIFICATION                 */
/*****************************************************************/
 1    MODULE: SPAIN;          /*          SPANISH MODEL          */
 2    SOURCE FILES: SIMBEF,   /* SIMULATION PARAMETER DEFINITION */
 3                  COEFF,    /*    MODEL EQUATION COEFFICIENTS  */
 4                  TIM.SER?  /* TIME-SERIES (HISTORICAL) DATA   */
 5    TARGET FILE: SOLUTION;  /* SIMULATION RESULTS              */
```

Fig. 2.2. The Header for the Example Specification in the MODEL Language.

The task of specifying the econometric model of Spain in the MODEL language, consists of specifying the five boxes in Figure 2.1. We will start with the description of the four file boxes in Section 2.1. This will be followed with the description of the equations in Section 2.

2.1 Data Statements

Data in a MODEL specification may be highly structured. The description of the data structure is tree-oriented, similar to PL/1 or Cobol. The node at the root of the data structure tree typically represents a file. A file may be composed of substructures, each of which may be further composed of substructures, and so on. A structure is referred to as the parent of its component substructures. The latter are referred to as descendents. A data structure is visualized as a tree where structures form nodes with branches leading to lower level components. The syntactic definition of data statements is shown in Figure 2.3. The data name is the name of a node in the tree. The node type indicates a level in the tree. A FILE node type may only appear at the root of the tree. A terminal tree node is denoted as FIELD node type. An intermediate node in the tree which is also the unit of transfer of data between input/output and memory is of RECORD node type, as in PL/1 or Cobol. A GROUP node type is any other intermediate node in a tree.

< data statement > : = < data name > IS < node type > (<arguments>)

< node type > : = FILE | GR[OU]P | REC[ORD] | F[IE]LD+

< arguments > : = < file arguments > | < group/record arguments > | < field arguments >

< group/record arguments > : = < immediate descendent name > [(< number of repetitions >)] [, < immediate descendent name > [(< number of repetitions >)]]*

The square brackets ([X]) denote optionality; when followed by an asterisk ([X]*) they mean zero or more repetitions.

+ The node type may be preceded by the key word INT[ERIM] when the respective data structure is target data but is not needed on an output medium.

Figure 2.3 Major Syntactic Components of Data Statement

The optional < file arguments > describe the media of the data. They are unimportant to the discussion below and will be omitted in the following.

The number of repetitions of a descendant structure is included as an argument in the statement describing the parent. If the descendant occurs only once, then the < number of repetitions > is omitted. If the < number of repetitions > varies, then the minimum and maximum bounds may be specified. Also, unknown number of repetitions may be specified by an asterisk (*) in place of a repetition count. The definition of a variable number of repetitions is further discussed below.

The field arguments are: data type, size and scale, with the same meanings these attibutes have in PL/1.

The MODEL specification of the four files of Figure 2.1 is shown in Figure 2.4. The specification is explained in the comments on the right-hand side of Figure 2.4. The discussion below supplements these comments.

The SIMDEF file is described first. The first statement (line 6) "SIMDEF IS FILE (RECORD IS INPUTREC)" means that the file has the assigned name SIMDEF and it consists of the record structure INPUTREC, of which there is only one instance (namely, it does not repeat). The statement in line 9 shows that INPUTREC is a record and it lists in parenthesis the names of its constituent fields. Finally, lines 11 - 18, consists of statements describing each one

```
/*****************************************************************/
/*                    FILE DESCRIPTIONS:                         */
/*****************************************************************/

/*****************************************************************/
/*              DESCRIPTION OF SIMDEF    FILE                    */
/*****************************************************************/

 6    SIMDEF IS FILE RECORD IS INPUTREC;  /* DEFINITIONS OF PARAMETERS WHICH */
 7                                        /* CONTROL THE SIMULATION          */
 8
        INPUTREC  IS RECORD(PD-TS,DELTA,BEG-YR,PD.SIM,NUM_VARS,NUM_CO,LAG);
10
11        PD.TS    IS FIELD (PIC'999');   /* NUMBER OF PERIODS OF TIME-SERIES DATA */
12        DELTA    IS FIELD (PIC'999');   /* NUMBER OF PERIODS FROM START OF TIME- */
13                                        /*  SERIES DATA TO START OF SIMULATION   */
14        8E6.YR   IS FIELD (PIC9999')!   /* YEAR OF START OF TSJDATA              */
15        PD.SIM   IS FiaD (PIC'999');    /* NUMBER OF PERIODS IN OUR SIMULATION   */
16        NUOARS IS FIELD (PIC9999');     /* NUMBER OF VARIABLES IN SIMULATION     */
17        NUH_CO   IS FIELD (PIC9999');   /* NUMBER OF CONSTANT COEFFS. IN MODEL   */
18        LAG      IS FIELD (PIC'999');   /* MAXIMUM NUMBER OF LAG PERIODS IN MODEL*/

/*****************************************************************/
/*              DESCRIPTION OF COEFF     FILE                    */
/*****************************************************************/

19    COEFF IS FILE RECORD IS C0-REC(l:99);  /* COEFFICIENTS OF MODEL EQUATIONS */
20      CO.REC IS RECORD(A);              :
21        A ISFiaD.(DECFL0AT(10>)5
22      SIZE.CO.REC = NUM.CO;                 /* fWIBER OF MOECL COEFFICIENTS    */
```

Fig. 2.4. Data Description Statements for Example.

```
/*********************************************************************/

/*                  DESCRIPTION OF TIM_SER    FILE                   */

/*********************************************************************/

 23     TIM_SER IS FILE RECORD IS TS_REC(1:99);  /* FILE CONTAINING TIME SERIES */
 24                                              /* DATA FOR SPAIN              */
 25       TS_REC IS RECORD(VNAME,VNUM,NUM_PDS,TS_DATA(1:99));
 26
 27         VNAME    IS FIELD (CHAR(4));     /* NAME OF MODEL VARIABLE         */
 28         VNUM     IS FIELD (PIC'9999');   /* NUMERIC IDENTIFIER OF VARIABLE */
 29         NUM_PDS  IS FIELD (PIC'9999');   /* NUMBER OF PERIODS OF TIME      */
 30                                          /* SERIES DATA FOR THIS VARIABLE  */
 31        TS_DATA IS FIELD (PIC'S99.V999');  /* TIME SERIES DATA VALUE        */
 32       SIZE.TS_REC  = NUM_VARS;   /* ONE TIME SERIES ARRAY PER MODEL VARIABLE */
 33       SIZE.TS_DATA = NUM_PDS;    /* ONE DATA VALUE PER PERIOD PER VARIABLE   */

/*********************************************************************/

/*                  DESCRIPTION OF SOLUTION   FILE                   */

/*********************************************************************/

 34     SOLUTION IS FILE GROUP IS SOL_GRP(*);  /* SIMULATION SOLUTION FILE     */
 35       SOL_GRP IS GROUP(HDR_REC,SOL_REC);   /* EACH SOLUTION GROUP CONTAINS */
 36                                            /*  A HEADER AND A BODY         */
 37       HDR_REC IS RECORD(SM_PD_ID,SM_YR_ID);
 38
 39         SM_PD_ID    IS FIELD(PIC'9999');      /* SOLUTION PERIOD NUMBER  */
 40         SM_YR_ID  IS FIELD(PIC'BB9999');      /* SOLUTION YEAR           */
 41
 42       SOL_REC IS RECORD(II,EX,GOV,IMSER,IM01,IM24,IM3,CONS,INV,IM,GDP);
 43
 44         CONS  IS FIELD (PIC'BB99.V(6)9');  /* THESE ARE ALL VARIABLES IN  */
 45         INV   IS FIELD (PIC'BB99.V(6)9');  /*  OUR ECONOMETRIC MODEL FOR  */
 46         II    IS FIELD (PIC'BB99.V(6)9');  /*  WHICH WE WILL DETERMINE    */
 47         EX    IS FIELD (PIC'BB99.V(6)9');  /*  VALUES, ONE VALUE PER PD.  */
 48         IM    IS FIELD (PIC'BB99.V(6)9');  /*  OF OUR SIMULATION          */
 49         GOV   IS FIELD (PIC'BB99.V(6)9');
 50         GDP   IS FIELD (PIC'BB99.V(6)9');
 51         IMSER IS FIELD (PIC'BB99.V(6)9');
 52         IM01  IS FIELD (PIC'BB99.V(6)9');
 53         IM24  IS FIELD (PIC'BB99.V(6)9');
 54         IM3   IS FIELD (PIC'BB99.V(6)9');
 55
 56       SIZE.SOL_GRP = PD_SIM;       /* ONE SOLUTION RECORD FOR EACH PD. OF SIM. */
 57       T IS SUBSCRIPT;              /* T (TIME) IS A COUNTER OF SIMULATION PDS. */
```

of the fields in INPUTREC. These fields provide the parameters of the simulation. They have been explained above. Each field name is followed by the respective data type. As shown, we use in this example the PICTURE data types of PL/1.

Next is a description of the COEFF file (line 19) which consists of the coefficients used in the equations. There is a record CO_REC for each coefficient A (line 20). The number of corresponding A coefficients may vary between 1 and 99. The actual size (i.e. number of repetitions) of CO_REC (and of A) is provided in the SIMDEF file in the NUM_CO (number of coefficients) field. This relationship is given in the assertion SIZE.CO_REC=NUM_CO in line 22. The specification of size using an assertion is discussed further in Section 2.2.

The TIM_SER file description starts on line 23. It consists of a variable number (1 to 99) of TS_REC records. Each record consists of the respective variable name, variable identification number, the number of periods in the time series for this variable, and the values of the variable for all the periods. Notice that the time series values, named TS_DATA constitute a two-dimensional array with the dimensions corresponding to repetitions of TS_REC and TS_DATA. The numbers of repetitions or the size of the two dimensions are also specified in the SIMDEF file by NUM_VAR and PD_TS, respectively.

The corresponding assertions are shown in lines 32 and 33. They are discussed further in Section 2.2.

Finally, the SOLUTION file is in fact a report that consists of two parts, a header part (HDR^REC) which specifies the titles in the report (the period year and period number), and data part (SOL_REC) with the values of eleven exogenous and endogenous variables for the corresponding period. A printed report line must correspond to a record. Therefore, for each period of simulation one report line provides the titles (HDR_REC) and the next line provides the data (SOLJREC). The SOLJREC record consists of the eleven variables for which values are computed for each period of the simulation. Each one of the variables constitutes a vector with its elements corresponding to the periods of simulation. The size of the dimension, i.e. the number of simulation periods, is given in the SIMDEF file by PD_SIM. This is expressed by the assertion in line 56.

The last line (57) in Figure 2.4 specifies that T represents a subscript which is used in the equations to denote a respective simulation period.

As shown, the description of data is simple and straightforward, listing each structure with its constituent parts, until all the data has been described.

Although data are pictured in MODEL (as in PL/1) as tree structures, it will be more convenient for the dis-

cussion here to refer to data as arrays. There is a direct correspondence between the tree and array views of a data structure. For instance, specifying a <number of repetitions> means that the data structure repeats, constituting a vector. Generally, a structure, may be viewed as a multidimensional array, where <number of repetitions > specifications of own or predecessor nodes in the data tree give the ranges of respective dimensions. Thus for instance, TS_DATA in TIM_SER file is viewed as a two dimensional array. The first, more significant dimension corresponds to repetitions of TS_REC and the second dimension corresponds to repetitions of TS_DATA. Therefore, we refer in the following to the < number of repetitions > of a node as a size or range specification, and also as the size or range of the dimension. Viewing the data as arrays allows referring to a specific instance of the data as an element of an array which can be identified by the appropriate indices for each dimension. For instance TS_DATA (n1,n2) denotes the n2 th TS_DATA of the n1 th TS_REC. Element indices are denoted by free subscript variables that may assume integer values in the range of the respective dimension. T, the declared subscript in Figure 2.4 is such a free subscript variable.

The range of a dimension may depend on the values of higher order subscripts. Therefore the range of a dimension of an array need not have the same value for all higher

order dimension indices. Such an array is not rectangular and is referred to as a jagged edge array. For example, TSJDATA has two dimensions with variable ranges associated with the repetitions of TSJREC and TSJDATA. The number of TS_DATA instances is specified by NUM_PDS, which may vary from one instance of the parent TS_REC to another. TSJDATA may be viewed as a two dimensional jagged edge array, with a row corresponding to each instance of TS_REC and the TS_DATA instances corresponding to elements of the respective rows. Since the number of TSJDATA instances is specified by NUMJPDS which may vary from row to row (i.e. from one TSJREC to another), the resulting matrix is not rectangular, but jagged edge.

The referencing of an element through subscripting, and the definition of a variable range by use of an assertion are further discussed below in connection with the use of assertions.

2.2 Assertion Statement

While the data statements describe the existence and structure of data to be operated upon, the description of the transformations applied to the data is given by the assertions. Rather than give detailed procedural instructions on a step-by-step execution, the user of MODEL identifies relationships between the variables from

which the processor deduces the actual execution sequences. These relationships are called assertions in MODEL. The building blocks for assertions include conventional arithmetic and boolean expressions and more structured operations such as IF-THEN-ELSE. This subsection describes the syntax and semantics of assertions with the aid of the example in Figures 2.1.

The syntax used for assertions in this paper is similar to that of computation statements in conventional programming language. The language allows explicit defining relations of the form:

<variable > = <expression >

The variable on the left hand side, the dependent variable of the assertion, is defined by the expression on the right hand side. The independent variables for this assertion are the variables participating in the defining expression on the right hand side. An expression is built out of variables and constants to which basic operators and functions are applied. PL/1 conventions for constants, variables and boolean and arithmetic operators are used in composing expressions. These include the IF-THEN-ELSE operator whose syntax is:

<variable >= IF < condition > THEN < expression_1>
ELSE < expression_2>

meaning that if <condition >evaluates to TRUE, then < expression_1 >defines the value of the variable, other-

wise < expression_2> is used*

An assertion statement, though similar in syntax to an assignment statement in conventional programming languages, should be regarded by the user quite differently* The assertion meaning is identical to the mathematical notion of equivalence between the two sides of the equal sign. Namely it is an equation. This aspect is basic to the difference between procedural and nonprocedural languages* Because of the nonprocedural nature of MODEL, each variable name may denote only one value. Also the "historical" values of data, namely those that would not be needed further in a computation must be explicitly represented by symbolic names. In contrast, procedural programming languages allow assigning differing values to the same variable and "historical" values may be overwritten if not further needed. For instance, an assignment statement within a loop: X=X+1 would make no sense as an equation. In MODEL it would be necessary to consider each value of X as a separate variable. Assume that these values constitute a vector, with N elements. An element is denoted by subscripting: X(T). T is the subscript variable which can take the value of an integer in the range of the variable X. The MODEL equivalent of the above assignment statement is the assertion: X(T)=X(T-1)+1.

Both the dependent and the independent variables should be subscripted by a list of subscript expressions corresponding to the dimensions of the variables as specified in the data description. Any integer valued expression can be used as a subscript expression for the variables. The general syntax for subscripted variables is:

< element of array > ::= < field name >
(<subscript expression> [,subscript expression>

The subscript expressions must be ordered according to the dimensions. Free subscript variables, as well as other variables and constants, with arithmetic operators, may be used in composing subscript expressions.

A free subscript variable may be global to an entire specification or local to an assertion. The same global subscript name in a number of assertions refers to free subscript variables of the same range. Global subscript names use the syntax form of FOR_EACH. <data name>. They may then have any integer value in the range of the <number of repetitions> associated with the <data name>. Use of the same local subscript name in different assertions does not imply referring to free subscript variables of the same range. Local subscript names use the syntax form of S[UB]<n>. The use of local subscripts is easier in many cases as the user need not consider the ranges of dimensions of different data structures. The syntax

of a global subscript name is somewhat awkward and a shorter global subscript name, such as commonly used symbols for subscripts, I,J,K etc*, may also be declared. The syntax for declaring a global subscript name is:

< subscript names>{*|E>SUBSCRIPT (<number of repetitions>)

Figure 2.4- shows such a subscript declaration for T. The use of global and local subscripts is illustrated below in the discussion of the assertions for the example of Figure 2.1.

Subscript expressions are classified into four types according to use of the following syntactic forms:

1) <free subscript variable>
2) <free subscript variable>-l
3) <free subscript variable>-K, K is integer> 1
4) Any form of arithmetic expression except types 1, 2 and 3 above.

The user is advised to give preference to use of subscript expressions of types 1, 2 and 3, as the version of the MODEL system reported here analyses the correctness of the specification and endeavors to obtain efficiency of the resulting program more thoroughly when these types of subscript expressions are used. In our example, we will refer to lagged variables using "subscript expressions of types 2 and 3, i.e. T-l, T-2, etc.

The subscripting of variables is a complex task which is difficult for many users. Subscripts may be implicit in

cases which do not lead to ambiguity. Allowing omission of such subscripts eases the composition of assertions. Following are the rules when subscripts must be specified:

1) Subscripts used in subscript expressions of types 2, 3 and 4 (see above) must be specified.

2) Subscripts of dimensions that are reduced or added in an assertion (i.e., where independent variables have more or less dimensions that the dependent variable) must be specified.

3) Once a subscript is specified for one variable in an assertion it must be consistently specified with the other variables in the assertion where the subscript applies.

4) Subscripts on the right of any specified subscripts must be specified.

5) Missing local subscripts are assumed inserted in all variables of an assertion montonically (i.e., S1,S2...) from right to left. Subscripts must be specified in cases where this assumption is not valid.

Subject to these rules, the MODEL system performs analysis to insert missing subscripts.

Qualified names may be used in assertions, using a period (.) to connect individual names (similar to PL/1). The most common use of a qualified name is to eliminate ambiguity through prefixing a name of a higher level

Another common use of qualified names in MODEL is to eliminate ambiguity in data that are updated. The keywords OLD and NEW are then used.

There are parameters of the data structures which depend on values of source or target variables. We refer to these as data parameter variables. Characteristically, these parameters provide specifications for sizes of arrays, lengths of character strings, keys for access to files, etc. They introduce to MODEL the flexibility of variable size or dynamic structures. The syntax of qualified names is used for data parameter variables:

< data parameter variable> ::= <reserved keywords>. <variable>

Data parameter variables may be explicitly defined by assertions. They may denote entire arrays and be used with subscript expressions in the same way as other variables. These keywords used as prefix in data parameter variables are listed below and further discussed in the seque

END.<data name>	denotes whether the named data element is the last one in the range of dimension.
ENDFILE.<file name>	denotes an end-of-file marker of the named file.
FOUND.<record name>	denotes existence of the record in an index sequential file that is accessed through a POINTER variable (see POINTER below).
INITIAL.<variable>	denotes a starting value for an iterative solution of the variab

LENGTH.<field name>	denotes length of the named field
NEXT.<field name>	denotes a named variable in the next adjacent record on the medium source data.
POINTER.<record name>	denotes value of a key used to reference a keyed record in an index sequential file. (the key name is identified in the FILE statement.)
SIZE.<data name>	denotes the range of the lowest order dimension of the repeating data structure named in the suffi

These variables are INT [ERIM], i.e., they are not output, but are otherwise considered

description statements for these variables may be provided optionally. If not provided, each of these variables will be automatically assigned the appropriate dimensionality. These variables are further explained below.

When the range of a dimension is variable, the range is viewed as denoted by an auxiliary array variable which may be defined by an assertion. A variable range data structure X may have its range denoted by a structure named SIZE.X, of one dimension less than that of X (the rightmost) and identical ranges for the other dimensions. Thus if X is m dimensional the elements of SIZE.X have the values of the ranges of the lowest order dimension of X for each of the higher order dimensions indices. Thus I_m, the subscript for the m-th dimension of $X(I_1 \ldots I_{m-1}, I_m)$ must be in the range $1 \leq I_m \leq$ SIZE.$X(I_1 \ldots I_{m-1})$. Consequently if the values of the elements of SIZE.X are not equal, then X is not a rec-

tangular array but -a jagged edge array* The range must be ≥. 0. Figure 2.4 shows assertions that specify SIZE. CO_REC, SIZE.TS_REC, SIZE.TSJ5ATA and SIZE.PRTJ3RP.

Another option for defining the size of structure X is by an auxiliary boolean array named END.X that has the same dimensions and ranges as X* A 0 value of an element of X denotes that it is not the last element within the range of the rightmost dimension, and a 1 denotes that it is the last element. When END.X is used for range specifications then the range must be 1 1.

POINTER.<record name >, defines an access key to an index sequential or random access file. Consider the assertion: POINTER,X(I)=EXPR(I). The array of records X is considered as indexed in the order of the elements of the retrieval keys POINTER.X. Namely, the record retrived by using EXPR(I) as a key is considered to be the I-th element in the array X.

Finally, function references may be used within right hand side expressions of assertions. The built-in functions of PL/1 may be used with the MODEL program generator that produces PL/1 pbject programs. Additional functions may be coded in the object language and placed in the system function library.

A variable is said to be recursively defined if it is an element of an array which depends, directly, or through a chain of assertions, on other elements of the same array.

If an element of this variable depends on elements in the same array with index values that are smaller than the value of the subscript for the dependent element, then the variable elements can be evaluated progressively as the value of the subscript is incremented from 1 to the end of the range in steps of 1. This condition is checked, and if it is not satisfied then a warning message is issued and a Gauss-Seidel iterative procedure is used in order to evaluate the recursive dependent array variable elements.

Figure 2.5 shows the specification of the equations for the reduced econometric model of Spain, previously introduced in Figure 2.1. The assertions in this part of the specification illustrate subscripting and use of qualified names in MODEL. The assertions in Figure 2.5 are shown as belonging to four groups as follows:

The first group consists of equations that define the exogenous variables in the solution file. As shown there are seven such variables. These equations specify that for the lag periods (T≤LAG) the values of the variables are copied from the respective TS_DATA time series, and for the other periods (T>LAG) they will be constant, namely equal to the value in the previous period (T-1). Note that in these equations the subscript T is used to denote the appropriate period element of the respective variable. The sub-

```
/*********************************************************************/

/*                EQUATIONS FOR EXOGENOUS VARIABLES                  */

/*********************************************************************/

 58    II   (T) = IF (T>LAG) THEN II   (T-1) ELSE TS_DATA( 3,T+DELTA);
 59    EX   (T) = IF (T>LAG) THEN EX   (T-1) ELSE TS_DATA( 4,T+DELTA);
 60    GOV  (T) = IF (T>LAG) THEN GOV  (T-1) ELSE TS_DATA( 6,T+DELTA);
 61    IMSER(T) = IF (T>LAG) THEN IMSER(T-1) ELSE TS_DATA( 8,T+DELTA);
 62    IM01 (T) = IF (T>LAG) THEN IM01 (T-1) ELSE TS_DATA( 9,T+DELTA);
 63    IM24 (T) = IF (T>LAG) THEN IM24 (T-1) ELSE TS_DATA(10,T+DELTA);
 64    IM3  (T) = IF (T>LAG) THEN IM3  (T-1) ELSE TS_DATA(11,T+DELTA);

/*********************************************************************/

/*                EQUATIONS FOR ENDOGENOUS VARIABLES                 */

/*********************************************************************/

 65    CONS(T)  = IF (T>LAG) THEN A(1) + A(2)*GDP(T) + A(3)*CONS(T-1)
 66               ELSE           TS_DATA(1,T+DELTA);
 67    INV(T)   = IF (T>LAG) THEN A(4) + A(5)*GDP(T) + A(6)*GDP(T-1) + II(T)
 68               ELSE           TS_DATA(2,T+DELTA);
 69    IM(T)    = IF (T>LAG) THEN IM01(T)+IM24(T)+IM3(T)+A(61)+A(62)*GDP(T)+IMSER(T)
 70               ELSE           TS_DATA(5,T+DELTA);
 71    GDP(T)   = IF (T>LAG) THEN CONS(T) + INV(T) + EX(T) + GOV(T) - IM(T)
 72               ELSE           TS_DATA(7,T+DELTA);

/*********************************************************************/

/*       EQUATIONS FOR INITIAL VALUES FOR ENDOGENOUS VARIABLES       */

/*********************************************************************/

 73    INITIAL.CONS(T) =  IF (T>LAG) THEN CONS(T-1) ELSE TS_DATA(1,T+DELTA);
 74    INITIAL.INV (T) =  IF (T>LAG) THEN INV (T-1) ELSE TS_DATA(2,T+DELTA);
 75    INITIAL.IM  (T) =  IF (T>LAG) THEN IM  (T-1) ELSE TS_DATA(5,T+DELTA);
 76    INITIAL.GDP (T) =  IF (T>LAG) THEN GDP (T-1) ELSE TS_DATA(7,T+DELTA);

/*********************************************************************/

/*                EQUATIONS FOR REPORT HEADER                        */

/*********************************************************************/

 77    SM_PD_ID(T) = T;                    /* NUMBER OF SOLUTION PERIOD  */
 78    SM_YR_ID(T) = BEG_YR + T;           /* YEAR OF SIMULATION RESULTS */
```

Fig. 2.5. Assertions for the Example.

and for the second dimension the expression T+DELTA. 'Both subscripts illustrate type M- subscripts. These assertions are all recursive as the defined variable elements depends on other elements of the same variable•

The second block of assertions defines the endogenous variables. Again, for the lag periods (T≤LAG) these variable; are defined to be the same as the data series for TS_DATA for the respective periods. For the other periods (T>LAG) these variables constitute a set of simultaneous econometric structural equations. The system will automatically recognize that these four equations are simultaneous and will employ an iterative solution method to define these variables If the method is not specified, the Gauss-Seidel method will be employed.

The third block of equations defines qualified name data-parameter variables with the prefix INITIAL and the endogenous variables as suffix. If these equations were not provided the system would by default take the initial values in the iterative solution for the simultaneous equations to be zero. This block of equations provides the system with a better zero order estimate for the solution. As shown the initial values of the four endogenous variables involved in the simultaneous equations are defined to be equal to the corresponding variable element values for the' previous period (T-l). For the lag periods, the values are to be copied from the respective TS_DATA time series data.

The last block of equations defines the report titles for SM_PD_ID - the simulation period, and SM_YR_ID-the simulation year.

This concludes the specification of the reduced model of Spain. As shown the specification consists of a total of 60 statements. The program generated based on this specification amounts to approximately 500 PL/1 statements. In addition, the MODEL processor produces documentation consisting of a cross-reference and attribute report, analysis of shapes of variables and finally a flow chart showing in schematic form the order of execution of statements in the program that is produced.

In the following sections additional examples will be given of MODEL statements in order to illustrate the proposed enhancements to the MODEL language. These will provide additional illustration of the use of the MODEL language.

Note that, unlike most econometrics software systems available to date, we require an explicit indication of how to compute the exogenous variables and the initial values for endogenous variables. The reason for this requirement is that we allow arbitrary source and target data structures while the econometric software available to date requires standard data structures for both the time series and the solution. A second reason is that the impli-

3. Extensions to the MODEL Language and Processor

This section describes in detail the proposed extensions to the MODEL language, including the syntactic forms, the meanings of the new constructs, and the advantages they offer to the user. The extensions are illustrated by examples. The extentions are presented below in respective subsections according to four categories: matrix and vector notation and operations, structure variables notation and operations, variable data structures and indirect subscripting. The last two categories support the constructs in the first two categories.

3.1 Matrix and Vector Notation and Operations

The purpose of this notation is to enable the user to write equations dealing with vectors and matrices in a form which is very similar to the mathematical notation for these equations. As indicated above, this capability is of' great importance in econometric modelling for specifying correlation and estimation methods, and for generally expressi ordinary and partial differential equations in mathematically oriented applications. At the present, MODEL allows the user to represent matrix and vector operations in terms of their elements. Also the language presently allows omitting the subscripts associated with these variables. In the following, the proposed extensions are defined in terms

Some matrix operations are already available under existing MODEL language facilities. Consider for

example the data description statements for three structures F, H and G:

```
F IS GROUP (M(*))
 M IS GROUP (X(*))
  X IS FIELD

H IS GROUP (P(*))
 P IS GROUP (Z(*))
  Z IS FIELD

G IS GROUP (N(*))
 N IS GROUP (Y(*))
  Z IF FIELD
```

X,Y,Z are field components of these structures. They represent two dimensional arrays or matrices. The statement

```
Z = X+Y
```

is interpreted in MODEL as the subscriptless version of

```
Z (I,J) = X(I,J) + Y(I,J)
```

Thus, even without the proposed extensions, the present MODEL system possesses the capabilities for addition, subtraction, and for element by element multiplication and division. The extensions concern more complicated matrix operations, namely: matrix multiplicaion (inner product), matrix inversion and transposition. To attain these operations we add new operations, symbols and naming conventions.

3.1.1 Matrix Operations

The following symbol combinations denote the new matrix operations:

"* Matrix multiplication.

"/ Unary or Binary matrix inversion.

"∿ Matrix transposition.

In addition the reserved word UNIT denotes a unit matrix variable.

Matrix Multiplication

The meaning of the matrix multiplication:

```
Z s X"*Y
```

is equivalent to the subscripted MODEL statement:

```
Z (I,J) = SUM (X(I,K) * Y(K,J), K)
```

The SUM function sums the values of the first parameter along the dimension of the subscript of the second parameter

Consider the following structure K in addition to the above F, H_3 G structures:

```
T IS GROUP (U<*), V(*))
 U IS 'FIELD
  V IS FIELD
```

U and V are one dimensional arrays which are interpreted here as row vectors. Thus:

```
V = U^ff*X
```

is interpreted as

```
V(J) = SUM (U(K) * X(K,J), K)
```

Let A be a scalar. A matrix multiplication notation may be used to compute the scalar A:

```
A = U^fl*ff*V
```

having the interpretation:

```
A = SUMCUCK)* V(K), K)
```

Note that V is transposed before multiplication.

Matrix Transposition:

Z = X

is interpreted as

Z (I,J) = X(I,J),

Z = "∿X

is interpreted as

Z(I,J) = X(J,I).

When applied to vectors it transforms a row vector ir a column vector and vice versa.

Matrix Inversion:

This operation can be used as a unary or as a binary operation. As a unary operation it produces the inverse o the matrix to which it applies:

Z = "/X

is interpreted as

Z(I,J) = M(I,J)

where $M = X^{-1}$ (the inverse matrix).

As a binary operation it's right argument must always be a (non singular) matrix. In general:

Y"/X = Y"*"/X

that is, the left argument is matrix multiplied into the inverse of its right argument. The use of "/ as a binary operation is illustrated in representing a linear set of equations:

SUM(X(L)*A(L,J),J) = B(J)

which can be expressed mathematically by

$\underline{X}"*\underline{A}=\underline{B}$

The solution of this system is given by:

$\underline{X} = \underline{B}$ "/ $\underline{A}$

Where $\underline{X}$ is the solution (row) vector, $\underline{B}$ is the vector of right hand side values and $\underline{A}$ the matrix of coefficients.

Another appropriate example here is the estimation of the coefficients of a single linear equation based on a number of observations of its variables. The equation may be represented as

Y(K) = SUM(B(I)*X(I,K),I) + U(K)

K and I are observation and variable indices respectively. There are also more observations than variables. U is the residual. This can be stated as

$\underline{Y} = \underline{B}"*\underline{X}+\underline{U}$

The least square method determines B such that ΣU^2 is minimi

X"*"∿X"/(X"*"∿X)

corresponding to the conventional matrix algebra expression:

$B = Y*X^T(X\cdot X^T)^{-1}$

The UNIT Matrix:

A standard unit matrix is provided using the reserved array name UNIT. Its definition is given by

UNIT (I,J) = IF I=J THEN 1 ELSE 0

The shape of the UNIT matrix is implied from the sizes of the dimensions of the matrices in the equation where it is referred to. As an example consider the expression

Z = "/ (UNIT - LAMBDA*X)

equivalent to the mathematical expression:

$Z = (I- \lambda X)^{-1}$

Z and Y are matrices and LAMBDA is a scalar. The shape of UNIT is that of X or Z.

3.1.3 Naming Conventions for Arrays

There are three ways to designate variables as vectors or matrices (to which the matrix operations can be applied):

The subscriptless form - using an element symbolic name with subscripts omitted.

Use of asterisk subscripts - indicating the matrix or vector dimensions by asterisks (*) as subscripts.

Using higher level structure names.

These three ways are explained below.

The subscriptless form. Any variable which appears without any subscripts in an expression with matrix operations is considered to be a vector or a matrix, depending on its dimensions as described in the data statements.. If it is described as a scalar (i.e. no repetitions of its structure and above'it) it is treated as a scalar. If it is a one dimensional array it is considered to be a row vector. If it has more than one dimension then it is considered to be a $matrix_5$ where the matrix operations apply to the two lowest order dimensions. All dimensions of higher order are treated as indicating higher order repetitions of the matrix. . . . .

As an example consider X and Y defined as above and the additional three dimensional arrays W and T:

```
L IS GROUP (B(*))
 B IS GROUP (Q(*))
  Q IS GROUP (W(*))
   W IS FIELD

P IS GROUP (C(*)>
 C IS GROUP (R(*))
  R IS GROUP (T(*))
   T IS FIELD
```

The expression:

```
W = X%T * Y(l,l)
```

is interpreted as follows:

```
W(I,J,K)=SUM(X(J,L)*T(I,L,K),L) * Y(l,l)
```

Y(l,l) is a scalar. The matrix multiplication of X and T is performed on the two lowest order subscripts. The highest order subscript I indexes W and T which are three dimensional•

Use of asterisk subscript list: In some complicated cases the user may want to specify the exact dimensions to which the matrix operations apply - unlike the above form where the lowest order dimensions were implied. This can be achieved by specifying a subscript list with some (at most two) of its elements being an asterisk: tf&".

Thus the expression:

```
W(I,*^) = X":, T < * ^ D * Y(I,*)
```

is interpreted as:

```
W(I,J,K) = SUM(X(J,L) * T(L,K,D) * Y(I,K)
```

• Note that this notation enables us to treat rows arid columns of a matrix as vectors. Thus to form a matrix of all inner products of the rows in a matrix we could write:

of course the same computation can be written more compactly as

Z = X"*"∘X

The rules for interpreting an asterisk subscript list is that the matrix or vector active subscripts replace the asterisks from right to left. If there are two asterisks the operation variable is interpreted as a matrix, while if there is only one the operation variable is interpreted as a row vector.

Use of higher level structure names. Another option for designation of the dimensions for matrix operations is the use of an ancestor data name of the elements of the matrix or vector. Each intermediate level in a structure, and hence each dimension, is associated with a data name which is in general a group or a record. We call these higher level variables - structure variables. They name a structure rather than the individual fields or elements. By using the name of a structure variable in a matrix expression we mean that the name denotes the complete structure Thus in the declaration of X above, F is the name of the matrix X(I,J). Alternately the vector M(I) refers to the matrix X(I,J) as M(I) is the name of the vector X(J). Consequently we may write:

Z(I,J) = M(I)"*"∘M(J)

for the matrix of all cross products of rows in X. In contrast with the previous options where the dimensions of the

array are determined by the repetitions above a variable, here the dimensions are determined by the dimension below the structure variable.

3.2 Structured Variables Notation and Operations

The purpose of this notation is to enable the user to write equations where the variables are entire data structures. It is particularly important for manipulating operations select tures. Section 3.2.1 discusses structure variables and their use in equations. The remaining subsections discuss ture variables functions: SELECT, MERGE, SORT, COLLECT and FU

3.2.1 Simple Equations for Structure Variables

A structure variable is denoted by the name of the respective structure. The data statements define a tree like structure in which the nodes are designated as fields or elements and the root (file type) and intermediate nodes (group or record types) are the structure variables. Thus in the example below:

```
F IS FILE (G1, G2)
 G1 IS GROUP (A(*),B)
  A IS GROUP (C,D(1:5))
   C IS FIELD
   D IS FIELD
  B IS FIELD
 G2 IS GROUP (X(2))
  X IS FIELD
```

F, G1, G2, and A are structure variables. B, C, D and X are field variables. Qualified names may be used to fully name a variable instance, such as:

```
F. G1. A(I)
```

Note that since A is repeating a subscript is needed in order to designate the exact instance. A structure variable is the root of the structure below it. A structure operation operate on the structure as a whole. The simplest structure operation is that of defining one structure to have a value identical to that of another structure. This implies a definition of each component of the defined structure. Consider for example the structure:

```
L IS FILE (K,H(*))
 K IS GROUP (X,Y(1:5))
 H IS GROUP (Z,U(1:100))
  (X,Y,Z,U) ARE FIELDS
```

The equation

```
K = A(2)
```

defines the structure K as having a field values equal to that of the structure of A(2). In order for a structure definition to be valid it is required that the defined structure (K) and the defining structure (A(2)) be compatible. This means correspondence of roots and subtrees in the defined and defining structures, including the same data types of respective fields.

The user can express correspondence of nodes in the respective structures in a number of ways.

1) The correspondence is based on node positions, i.e. level of the tree and the node position (from left to right) in the level, in the respective data trees.
2) If subtrees in the defining structure are to be omitted in the defined structure (then the position

does not define the corresponding of subtrees in the two structures) then the corresponding subtrees which are not omitted must use the same names.

3) Correspondence may be expressed by using the same field names in the defined structure as the corresponding fields in the defining structure.

In all these cases, to be compatible the corresponding nodes must have same dimensionality in respect to the respective structure roots, and if fields, same data type. Having such correspondence, the meaning of a structure definition can be taken as the definition of each component in the defined structure by the value of its corresponding component in the defining structure. Thus the statement above K=A(2)) is equivalent to the following set of field defining equations:

```
X = A.C(2)
Y(I) = A.D(2,I)
SIZE.Y = SIZE. A.D(2) or
END.Y(I) = END.A.D.(2,I).
```

Note that the defined structure inherits not only the values of all components in the source structure, but also any variable attributes of the structure such as SIZE, END, LEN etc.

The form of an equation which defines a structure is

vector of numeric elements, and 'expr' is a vector or matrix expression (see subsection 3.1)

2) A =[IF cond 1 THEN] A_1[ELSE [IF $cond_k$ THEN] A_k]*

where A is a compatible structure with A_1,[A_2....

3) A = structure function(----)

The last form will be further discussed in more detail. The above forms may be compounded by conditional statements or conditional expressions used in MODEL.

3.2.2 The SELECT function

The SELECT function is list oriented. Namely, it treats a one dimensional set of data structures as an ordered list. It forms an output sublist of structures which satisfy a given condition, preserving the original order between the selected structures.

Consider the following F source and E target files:

```
F IS FILE (G(X))
 G IS GROUP (K,X)
  K IS FIELD (CHAR)
  X IS FIELD (NUM)

E IS FILE (H(*))
 H IS GROUP (KEY,Y)
  KEY IS FIELD (CHAR)
  Y IS FIELD (NUM)
```

Where G and H are compatible structures. There are two forms to the SELECT function, differing in the structure level of the target variable.

E = SELECT (G(I), cond (I) [,I])

H(L) = SELECT (G(I), cond (I,L) [,I])

In the first format the target is the parent E of the repeating structure H. E therefore refers to a list of H

G, Cond(I) is tested in order to determine whether G(I) should be selected. The optional parameter I denotes the dimension used in the selection.

An example of use of this function is the case where only G groups in which X is positive are to be selected. This is defined by the assertion:

E = SELECT <G(I), XCD >0)

In general, the target list of the SELECTION FUNCTION is shorter than the source list. SIZE or" END attributes for the target list are automatically defined by the SELECT function. The optional parameter I denotes the subscript of the source substructures. If this parameter is absent the right subscript (i.e. of G(D) is selected.

In the second format the target variable is a structure on the same level as the source repeating structure. It appears subscripted by L. The subscript L must be distinct from any subscript appearing in the source structure G(I). L may appear also as an argument in the condition fcond(I,L)[1] This adds power to our selection capability. Consider the above E and F structures. Assume also that F is sorted on the field K. In the following example we select only the first of every consecutive structures G which have a common value of K. The desired selection can be expressed as:

H(L) = SELECT (G(I), KEY(L-l)n=K(I)) .

Note that L-l refers to the element preceding the element which is referenced.

The MERGE function is also list oriented. Its multiple source lists are merged into a target list by interleaving elements (structures) of the source lists. Each of the lists is separately indexed. A test of a condition is applie to element candidates from each list to determine which structure is selected as the next member of the target list. When one of the source lists is exhausted, elements are taken only from the remaining lists.

The following example illustrates the use of the MERGE function. Let F be the target group and G and H the source groups.

```
F IS GROUP (AC*))
G IS GROUP CBC*))
H IS GROUP CCC*))
```

Assume also that A is compatible with both B and C. L, I and J are used below as subscripts of A, B. and C, respectively. Again, the MERGE function has two formats where the target structure is represented by an entire list or typical element of the list. These formats are:

```
F = MERGE(B(I),C(J), Cond(I,J) C,I,J1)

A(L)=MERGE (B(I),C(J), CondCI,J,L [,I,J,])
```

In the first form F is the parent of the merged list. $^{f}CondCl_{3}J)^{f}$ depends in general on B(I) and C(J). The condition determines which structure is next selected to be in the merged output list.

In the second format, the target structure is a variable ACL), an element, of F, and the condition may also depend on L

function is merging two sorted lists to form a new sorted list. Assume A, B, C ARE FIELDS(NUM). Then either of the assertions:

F=MERGE(C(I),B(J),C(I) ≤sB(J))

ACL) = MERGE(CCl),B(J),C(I) sB(J))

merges the strings of C and B elements* The next (Lth) element of F is the one with the lower value. This produces a sorted list provided G and H are sorted.

The MERGE function automatically defines the attributes of the target structure•

3.2.4 The' SORT function

This function rearranges the order of the structures in a source list to form a target list of compatible structures which are ordered by increasing or decreasing values of certain fields. Assume the source and target structures.

```
F IS GROUP (AC*));
 A IS GROUP (X,Y,Z);
G IS GROUP (B(*) ;
 B IS GROUP (U,V,W);
```

The format of SORT function is then:

$B(I_1,..,.I_K)$ = SORT(A(I_1,____$^{I}K>^{J}l»^{\#\#\#J}m$)> INC DEC> (X>"

INC or DEC are used for sorting by increasing or decreasing values of the fields in A indicated in the next argument. The sorting by a multiple key may be indicated by <
* list of fields of A, i.e. X, Y etc, with the order in the lisi indicating the priority, i.e. order first on X, and then with: X on Y, etc. As shown above, let: '...•"

dim(A) = K+m and dimCB) = K, dim(A) ≥dim(B),

i. e. the sorting is performed for each permutation of ^1### ^K—1 m+"L dimensions. The selection of sort algorithms is automatic and is discussed further in connection with the implementation of this function.

3,2.5 The COLLECT function

The COLLECT function is used in order to convert a list (or a file) which is a one dimensional array of structures into a two dimensional jagged edge array, or list of lists.

Consider the structures:

```
F IS FILE (G(*))
G IS GROUP (X(*))
H IS FILE (Y(*>)
```

where X and Y are assumed to be compatible structures. X may be defined as:

X(I,L) = COLLECT(YCJ)$_v$ Cond(I,J,L))

Its meaning is that the list Y(J) is subdivided into a list of sublists with I the index of the sublist and L the index of an element in a sublist. The first element Y(l) forms X(l,l). Subsequently an element Y(J) forms an element X(I,L) if ^ondCljJjL)^T holds, otherwise it forms the element X(.I+1,1), i.e. the first member of the next sublist.

This function is useful in the case that we wish to view a file (a list) which is a homogenous stream of records as consisting of groups of records. The elements in each group may share some common characteristic such as having a common range of keys. The source file (Y) is described

dimensional representation according to an arbitrary grouping criterion.

In the present MODEL language if the user desires to refer to an operand in an assertion as being multidimensional then the corresponding file must be described as being structured in this manner. This violates the principle of data independence, namely the independence of external data description on how the data is actually referenced in a desired transformation. The COLLECT function allows specifying internal structures with the desired dimensionality, without forcing this structure on the external file.

The function COLLECT is redundant since a very similar effect may be obtained by using the SELECT function:

X(I,L) = SELECT(Y(J), cond(I,J))

The difference between the uses of SELECT and COLLECT is that of efficiency. The specification of SELECT implies a scan of the complete source string of Y in order to select the appropriate records that correspond to the index I. The COLLECT function explicitly states that only one scan of the source string Y is required. This aspect is further discussed later in connection with the implementation of the COLLECT and SELECT functions.

3.2.6 The FUSE function

The FUSE function is the inverse of the COLLECT function It takes as input a two dimensional jagged edge array which

can be considered a list of sublists and forms a single fused list. For example consider the source structure:

```
F IS FILE (G(*))
 G IS GROUP (R(*))
```

and the target structure:

```
H IS FILE (R(*))
```

Then:

```
H = FUSE(F.R(I,J) [, I,J ])
```

defines the list H, which consist of the following sequence of structures:

```
R(1,1),..R(1,SIZE.R(1)), R(2,1)...R(SIZE.G,SIZE.R(SIZE.G)
```

3.3 Variant Structure Variables

At the present a MODEL data statements must specify for each structure variable (FILE, RECORD or GROUP) a unique set of components. It is possible to indicate an alternate possibility of components, based on some condition, through the artificial description of the alternate components as optional components.

Thus for example:

```
F IS FILE (G(*))
 G IS GROUP (A(0:1), B(0:1))
```

means that G consists of a sequence of alternate A and B components which are optionally repeating either 0 or 1 times. To choose only one of these it is necessary to explicitly defind SIZE.A and SIZE.B so that either A or B exists but never both. This is a roundabout way of saying that G consists of a sequence each of whose elements is either of the structure A or alternately of the structure B.

The following modification is proposed in order to make the selection between variant components much easier. Presently the syntax of description of a structure variable is:

<structure variable>::=<name> is {GROUP|RECORD} (<component list >)

The proposed extension consists of a separator '/' denoting alternative components in the following way:

<structure variable>::=<names> IS {GROUP|RECORD }

(<component list> [/<component list>]*)

Thus, the definition above for declaring G to be either A or B can be simply represented as:

G IS GROUP (A / B).

A more complicated example is:

G IS GROUP (A, B(*), C(2) / D, E(1:2))

where G either consists of the components A, B and C or of D and E.

To determine the choice of the alternative, it is necessary also to define a qualified name variable with the prefix keyword DISCR and the suffix name of the parent, i.e. DISCR.G. The value of this "discriminator" variable determines the choice of numbered alternatives, i.e. the firs alternative is chosen if DISCR.G is 1, and the second alternative of DISCR.G is 2, etc. The dimensions and ranges of th DISCR prefixed variable are the same as the variable named ir the suffix.

This is illustrated below through an example of the specifications of personnel records in which if a person is divorced the record includes the date of divorce while if married the name of the spouse is included. A description of such a record might be given by

```
PERSON IS RECORD (NAME, MARITAL_STATUS, ADD_INFO)
ADD.INFO_IS GROUP (DIV_DATE / SPOUSE_NAME)
```

For source variables the DISCR data-parameter has to be explicitly defined:

```
DISCR.ADD_INFO = IF(MARITAL_STATUS = 'DIVORCED') THEN 1
                                                 ELSE
                 IF(MARITAL_STATUS = 'MARRIED')  THEN 2
```

High level data structure equations (i.e. definition of a high level structure by a single equation) can be used only when the operand data structures do not consist of components which have variant structures. The case where there are struc variants require the user to explicitly match source or targe variants with other variables. This can be done by writing directly the elemental equations as shown in the discussion of source to source translation of high level data structure equations in Section 4.2.

To avoid circular definitions the DISCR data-parameter for source structures cannot depend on information which is contained in the variant structures themselves.

The variant structure concept resembles the variant recor concept of PASCAL. It is somewhat more general in being able to specify an arbitrary discriminating condition.

3.4 Indirect Subscripting

The implementation of the list functions described in Section 3.2 is based on expressing these functions by simpler statements in the current MODEL language. Additionally, the efficient implementation of these functions requires extending the subscript expression analysis performed by the MODEL processor. As noted in Section 2, presently the MODEL processor analyses three types of subscript expressions of the forms of I, I-1, I-K (K>1), where I is a subscript and K a positive integer constant. All other forms of subscript expressions are denoted as being of type 4 and are not further analyzed. The proposed extension consists of adding the analysis of subscript expressions of the form A(I) where A is a variable which is used to subscript another variable B, i.e. B(A(I)). This form of subscript expression is referred to as indirect subscripting, where a secondary array (A) defin the subscript value used to subscript the main array (B).

Indirect subscripting may be used in the left hand side dependent variable:

$$B(I_1,----I_{m-1},A(I_1,----I_{m-1},I_m))=C(I_1,---I_m,---)$$

or in a right hand side independent variable

$$B(I_1,----I_m)=C(I_1,----I_{m-1},A(I_1,----I_{m-1},I_m),------).$$

These two forms are referred to as left and right indirect subscripts, respectively.

Generally the produced program is made more efficient both in computing and memory use if the scope of loops

is enlarged. In the present MODEL system, subscript expressions of types 1, 2 and 3 are analyzed to determine such sets of statements where in each set appropriate respective subscript expressions have compatible ranges so that these statements can be computed in a single program loop. Also the variable arrays in these statements require only a single range specification for the respective subscripted dimension. Type 4 subscript expressions indicate that the statement must be computed in a separate loop for the respecti subscript.and never in the same loop as the one that computes the variable in which the type subscript appears. This requires also that the array which is subscripted by a type 4 expression must be placed in memory. The proposed analysis of indirect subscripts will allow the inclusion of more statements in the same loop.

As will be shown below, this extension of the analysis to indirect subscripts will also allow the user to compose list oriented specifications which will be checked more thoroughly and implemented more efficiently.

The secondary array used in indirect subscripting must be integer valued with positive entries. The system will analyze indirect subscripts only if the secondary array A(I) is sublinear, namely if it is:

a) Monotonic: i.e. for I>J=>(A(..I)≥A(..J)

b) Grows slower than I, i.e. A(..I) ≤ I

The system will test the secondary array automatically to determine if it is sublinear by applying the following simple criteria. In the assertion that defines the second-

with clauses of conditionals) must be either 0 or 1 for I=1 and must be equal to A(..I-1) or A(..I-1)+1 for I>1.

Thus the system will examine the assertion to check if A(I) is in the form:

A(I) = IF I=1 THEN (1 | 0) ELSE(A(...I-1) | A(....I-1)+1))

This extension not only makes possible the checking and implementation of the list oriented functions, but also the user composition of list oriented applications. Consider the following example:

Let IN be a source file as follows

```
IN IS FILE (R(*))
 R IS RECORD (ACCT#, NAME, ACTIVITY)
  (ACCT#, ACTIVITY) ARE FIELDS (NUM)
  NAME IS FIELD (CHAR)
```

It is sorted by ACCT#. It is desired to edit this file to obtain a report with the following requirements:

a) Every new ACCT# (the file is sorted on this field) should start on a new page.

b) There would be a line in the report for each record R.

c) An account which extends beyond 60 Records should have a new page every 60 lines.

d) Each new page should have a header which lists the ACCT#, a global page number (accumulative), and a local page number for the current account.

A separate HEADER source file contains just header records as follows:

```
HEADER IS FILE (HD(*))
 HD IS RECORD (CC,ACCT#,LOCAL_PAGE,GLOBAL_PAGE)
  CC IS FIELD (CHAR(D)
  (ACCT#, LOCAL_PAGE,GLOBAL_PAGE) ARE FIELDS (NUM)
```

"The output file will be a merge of the IN and HEADER files.

```
REPORT IS FILE (G(*))
 G IS GROUP (P|HD)
```

The index of records (P or HD) in the REPORT file is denoted by the subscript I. NEW_ACCT(I) and NEW_PAGE(I) are boolean variables which denote whether the Ith record of REPORT represents a new account number, and start of a new page, respectively. CNT60(I) denotes the sequence number of c record R in its page.

A(I) is a secondary array which denotes the index of a record R in IN which corresponds to the Ith line(PjHD) in REPORT. Note that since the output file uses both header records and input records, AU> increases slower than I. Thus

```
AC I) = IF 1=1              THEN 1
                            ELSE IF NEW_PAGE(I) THEN A(I-l)
                            ELSE ACI-D+1      or i
NEW_ACCT(I) = ((I=1)|(A(I-1)>1S  IN.ACCT#(A(I-1))-l=IN.ACCT#(
              8CCNT60CI-1)>0)
NEW_PAGECI) = NEW_ACCMT(I)|(CNT60(I-l)=60)
CNT60(I) = IF NEW_PAGE(I) THEN 0 ELSE CNT60(I-l)+1
HD.ACCT#(I) =  IN.ACCT#(A(D)
HD.LOCAL_PAGE(I) = IF NEW__ACCNT(I) THEN 1
                   ELSE IF NEW_PAGE(D THEN HD.LOCAL_PAGE
                   ELSE HD.LOCAL_PAGE(I-l)
```

```
HD.GLOBALJPAGE(I). = IF I•« 1 THEN 1
                     ELSE IF NEW_PAGE(I) THEN HD. GLOBA
                     ELSE HD..GLOBALPAGE(I-1)
REPORT.G(I) = IF NEW_PAGE(I) THEN HD (I)
              ELSE IN.R(ACI) -1)
```

Where the declarations of the secondary variables are given by

```
AUX IS INTERIM FILE (GR(*))
GR IS GROUP (NEW__ACCOUNT, NEW__PAGE, CNT60,A)
(NEWJVCCOUNT, NEW_PAGE)ARE FIELD (BOOLEAN)
(A, CNT60) ARE FIELD (NUM)
```

Also CC is the indicator of a new page: CC = 'l'.

4. Source to Source Transformations

As already stated in the previous section, the principal method for implementing the extension of MODEL with high level data structure assertions is to transform such assertions into assertions containing only operations on data elements. Namely the source high level data structure assertions will be replaced by equivalent elemental data assertions. The latter type of assertions are already implemented in the current version of the MODEL processor. The transformation of high level structure assertions into the elemental data assertions is divided below as follows: Section 4.1 deals with matrix and vector algebra assertions, Section 4.2 deals with the data structure algebra, and finally Section 4.3 deals with data structure equations.

The MODEL processor architecture contains a number of phases, as described in the MODEL documentation (A. Pnueli, L. Lu, and N. Prywes "MODEL Program Generator" March 1980). The processor consists of the following phases:

Phase 1: Syntax-Analysis of the MODEL Specification

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 is a table of syntax rules provid

a formal description of the MODEL language in an extended BNF language. In this manner, changes to the syntax of MODEL during development can be made more 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. Some needed corrections and warnings of possible errors are also produced in a report for the user. Also, a cross-reference report is produced.

Phase 2: Analysis of MODEL Specification

In this phase, dependancy relationships between statements are determined from analysis of the MODEL data and assertion statements. The specification is analyzed to determine the consistency and completeness of the statements. Each MODEL statement is first considered independently and checked for syntactic correctness. The or of the user's statements is of no consequence. The statements are represented by nodes, and the dependency relationships are represented by directed edges in an array graph on which completeness, consistency, ambiguity and feasibility of constructing a program can be checked. Various ommissions or errors are corrected automatically, especially in connection with use of subscripts. 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. In addition, a report showing the range of each subscript is generated.

Phase 3: Automatic Program Design and Generation of Sequencing Control.

This phase of the Processor determines the sequence of execution of all the events and the iterations implied by the specification. Subsequently it determines the sequence and control logic of the desired program. The result of this phase is a flow of events, sequenced in the order of execution. Thus, the output of this pahse is similar to a program flowchart of the desired program. It is subsequently used to produce a flowchart-like report. At the end of this phase it is also possible to produce a formatted report of the specification.

Phase 4: 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. In particular, read and write input/output commands are generated whenever the flowchart indicates the need for moving records. The assertions are developed into PL/1 assignment statements. Wherever program iterations and other control

them is generated. Declarations for object program data structures and variables are generated. Code is also generated for recovery from program failures when bad data is encountered during program execution. 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 is produced.

The current version of the MODEL processor already utilizes the techniques of generating source MODEL statements where a user specification is incomplete and the missing data and assertions statements are generated to correct the specification. However the analysis in the presently considered transformations is far more complex It is proposed therefore to add procedures in phase 2 of the MODEL processor to handle the transformations. This in fact creates a new subphase within phase 2.

The new subphase has to come in phase 2 after the attribu of all data structures have been computed, because it needs to know for its execution the dimensions of variables as well as their structure. On the other hande it has to precede the present analysis of the assertions as this analysis should be applied also to the statements generated in the subphase. Consequently the ideal location would be between the procedures ENHRREL which analyzes the data statements and constructs the data nodes and edges and ENEXDP which analyzes assertions.

The statement generator subphase must be organized in several further subphases which have to be followed in a certain order, because the earlier subphases produce assertions which may be further transformed by later subphases. Note also that some of the statements generated are data statements.

The output of the statement generator subphase is a set of additional MODEL statements replacing some of the original assertions in the specification. .These new MODEL statements have to be reanalyzed and properly represented as though they were part of the original specification input and the original statements which were, replaced must be deleted from the associative memory. The simplest but somewhat inefficient method is to maintain the specification on an external file, and to edit this external file. This facilitates deleting old assertions and inserting new ones. After this phase we may restart the whole translation from the beginning, presenting SAP with the edited version of the specification. A more efficient method is to perform the transformation at the assertion analysis level/.including the reapplication of the previous phases, such as dictionary updating, etc. as appropriate.

U.I The Transformation of Matrix and Vector Algebra Assertions.

This subphase scans the assertions and interprets the matrix and'vector assertions into more elemental state-

ments. If this transformation is successful, the original assertion is replaced by one or more assertions produced by the transformation. If the assertion does not contain any matrix or vector operations then the assertion is unchanged.

The heart of the transformation and analysis algorithms is the recursive algorithm MATRIFY described below. This algorithm has two inputs. It accepts an expression, from an assertion that was identified as being a matrix or vector assertion, and a list of two subscripts. It attempts to translate the inputs into elemental assertions. It analyzes the input expressions in the assertion tree in the associative memory, expression by expression. If successful, it returns the transformed expression with the information of the dimension of the resulting expression. It may also return an indication of failure.

PROCEDURE MATRIFY:

Input Parameters:

EXP - The expression to be translated.

SUB1, SUB2 - The names of the subscripts to be used for translation.

Output Parameters

TEXP - The translated expression.

SHAPE = Can assume one of the values:

'MAT' - for a matrix

'ROW' - for a row vector

'COL' - for a column vector

'SCL' - for a scalar.

The SHAPE parameter reports the shape of the translate expression.

Algorithm

The algorithm analyzes each expression to determine one of five cases:

1. The expression is a variable - then this is case M
2. The expression is a unary operation followed by an expression (OP EXP)-then this is case M2
3. The expression is a binary operation (EXP1 OP EXP2)-then this is case M3
4. The expression is a function with arguments (F(ARG1, ---))-then this is case M4
5. The expression is equality: (the top level of an assertion) EXP1=EXP2)-then this is case M5

Ml. Consider first the case that EXP is a variable.

M.I.I Let EXP = A where A is a field. Let d be its dimensio:

If d=0 return TEXP = A, SHAPE = 'SCL'

IF d=l return TEXP = ACSUB2), SHAPE = 'ROW

IF d^2 return TEXP = A(SUB1,SUB2).SHAPE = 'MAT'.

Ml.2 Let EXP = AC^,.. J_m), m>0 where A is a field.

Return TEXP = EXP , SHAPE = 'SCL'.

Ml.3 Let EXP = A(J^,.. J^^) , msO where A is a field.

Return TEXP = ACJj,..^, SUB2), SHAPE = 'ROW.

Ml.4 Let EXP = A(Ji,..$J_{m,ft}$,J_{m+}l) , m^O where A is a field.

Return TEXP = A(*!•,_,.. J_m, SUB.l, J_{m+1}), SHAPE = 'COL'.

Ml.5 Let EXP = AC,.. * »• •$^J{}_{m+}$i»$^J{}_{in+}$2) where A is a field.

Return TEXP=A(...SUB2,...J_{m+1},J_{m+2}>, SHAPE ='ROW».

"Ml.6 Let EXP = AC..*,..ft,..) where A is a field. Then

Return TEXP = A(,..SUB1,..SUB2,..) , SHAPE ='MAT4.

Ml.7 Let EXP = AC^,..^), m>.0 where A is a REC[GROUP and it has only one descendant field X. Let d = dim(X)-dim(A). Then

If d=0 return TEXP =. *X(*J_{l9}..J_m),* SHAPE = 'SCL'

If d=l return TEXP = X(J_{1}}..J_m, SUB2), SHAPE 'ROW

If d=2 return TEXP=X (J^..J_m. SUB1,SUB2), SHAPE = 'MAT

Ml.8 If EXP = UNIT, the special name for the unit matrix , return TEXP = .IF(SUB1=SUB2)THEN 1 ELSE 0, SHAPE = 'MAT'.

In all other cases where EXP is a variable, report an error condition.

M2. Consider now the case that EXP is an expression formed out of a unary operation and a subexpression, i.e.

EXP = OP EXP1

M2.1 If OP is an elemental operation, i.e. a non matrix operation then

Call MATRIFY (EXP1,SUB1,SUB2,TEXP1,SHAPE1)

Return TEXP=OP TEXP1,SHAPE = SHAPE1

M2.2 If OP = '~, a transposition, then

Call MATRIFY (EXP1,SUB2,SUB1,TEXP1,SHAPE1)

Return TEXP = TEXP1 and

SHAPE = IF SHAPE1='SCL' | 'MAT' THEN SHAPE1

ELSE IF SHAPE1='COL' THEN 'ROW'

ELSE IF SHAPE1='ROW' THEN 'COL'

M2.3 If OP = '/, inversion, then

Call MATRIFY (EXP1,SUB1,SUB2,TEXP1,SHAPE1)

If SHAPE1 ≠ 'MAT' report an error.

Describe new auxiliary arrays AUX and INV by issuing the statements:

G IS GROUP (R(*))

R IS RECORD (AUX(*), INV(*))

(AUX,INV) ARE FIELD (NUM)

The names G,R,AUX,INV, should be unique to each specification.

Also generate the assertions:

AUX (SUB1,SUB2) = TEXP1

INV = MATINV (AUX,SIZE.AUX) and return with

TEXP = INV(SUB1,SUB2), SHAPE = 'MAT'

MATINV will be a run-time provided procedure for inverting a matrix.

M.3 Consider next the case that EXP is a binary expression:

EXP = EXP1 OP EXP2

M3.1 If OP is an elemental operation, i.e. a non matrix or vector operation then:

Call MATRIFY (EXP1,SUB1,SUB2,TEXP1,SHAPE1)

Call MATRIFY (EXP2,SUB1,SUB2,TEXP2,SHAPE2) and return

TEXP = TEXP1 OP TEXP2 and

SHAPE = IF(SHAPE1 = SHAPE 2)THEN SHAPE1 ELSE
IF (SHAPE2 = 'SCL') THEN SHAPE1 ELSE
IF (SHAPE1 = 'SCL') THEN SHAPE2 ELSE
'MAT'.

M3.2 If OP = "* - i.e. matrix multiplication. Let K be a new system generated subscript.

Call MATRIFY (EXP1,SUB1,K,TEXP1,SHAPE1)

Call MATRIFY (EXP2,K,SUB2,TEXP2,SHAPE2)

If SHAPE1 = 'COL' and SHAPE2 = 'MAT' or
SHAPE1 = 'MAT' and SHAPE2 = 'ROW' or
SHAPE1 = SHAPE2 and SHAPE1 ≠ 'MAT', report an error.
otherwise return

TEXP = SUM(TEXP1*TEXP2,K), and

SHAPE = IF SHAPE2 = 'MAT' THEN SHAPE1 ELSE
IF SHAPE1 = 'MAT' THEN SHAPE2 ELSE

IF(SHAPE1 = 'ROW')THEN 'SCL' ELSE

'MAT'

M3.3 If OP = "/, matrix division,

Call MATRIFY (EXP1,SUB1,SUB2,TEXP1,SHAPE1)

Call MATRIFY (EXP2,SUB1,SUB2,TEXP2,SHAPE2)

If SHAPE1 = 'SCL' or SHAPE1 = 'COL' or

SHAPE2 ≠ 'MAT' report an error.

Declare a two dimensional array AUX(*,*)

and issue the assertion:

AUX(SUB1,SUB2) = TEXP2

Now distinguish between the following two subcases:

M3.3.1. If SHAPE1 = 'ROW' then declare two one dimensional

arrays LEFT(*), SOL(*) and form the assertions

LEFT(SUB2) = TEXP1

SOL = SOVEQ(LEFT,AUX)

Return TEXP = SOL(SUB2)

SHAPE = 'ROW'

SOLVEQ (b,A) is a run time provided procedure which

for a row vector a and b matrix A finds the row vector

solution x of the linear system

b = xA

M3.3.2. If SHAPE1 = 'MAT' then declare two two dimensional

arrays LEFT(*,*), SOL(*,*) and form the assertions

LEFT(SUB1,SUB2) = TEXP1

SOL = SOLVMAT(LEFT,AUX)

Return TEXP = SOL(SUB1,SUB2)

SHAPE = 'MAT'

SOLVMAT (B,A) is a run time provided procedure which for matrices B and A finds the matrix solution X to the linear system:

$B = XA$

M4. Consider next the case that EXP is a function

EXP : $F(ARG_1,..ARG_R)$

For each ARG_i, i=1..R

Call MATRIFY(ARG_i,SUB1,SUB2,$TEXP_i$,$SHAPE_i$)

Return

$TEXP = F(TEXP_1,..TEXP_R)$ and

$SHAPE = SHAPE_1$ if all $SHAPE_i$ are equal to one another

If all the non scalar $SHAPE_i$ are equal to one another, then let SHAPE be that non scalar value.

Otherwise let SHAPE = 'MAT'.

M5. Finally consider an equality:

EXP : $^{1}TARGET = SEXP^{1}$

Where SEXP is the source expression of an assertion. Then:

Call MATRIFY (TARGET,SUB1, SUB2,TEXP1,TSHAPE)

Call MATRIFY (SEXP,SUB1,SUB2,SOURCE,SSHAPE)

M5.1 If TEXP1 = 'ERROR' or SOURCE = 'ERROR' and SEXP contains matrix operations this is an error condition.

M5.2 If TEXP1 = 'ERROR' or SOURCE = 'ERROR' and SEXP does not contain matrix operations the assertion just cannot be interpreted as a matrix assertion and return

TEXP = EXP, the original assertion.

M5.3 If TSHAPE = SSHAPE or SSHAPE = 'SCL'
or TSHAPE = 'MAT' return

TEXP = TEXP1 = SOURCE

M5.4 Otherwise check if SEXP contains matrix operations. If it does it is an error situation, else return TEXP = EXP, again with no translation

End of Algorithm

4.2 Transformation Of Data Structure Functions

Another check performed during the scanning of the assertions is for the presence of high level data structure functions: SELECT, MERGE, SORT, COLLECT and FUSE. By requirement these functions can only appear at the top level of the right hand side expression of an assertion tree. Thus detection in this case is relatively easy. Each assertion which utilizes one of these functions is then translated in this subphase into a set of elemental assertions.

These functions operate on source lists of data structures to define a target list of structures. One form of the assertion using the functions is:

T(...L) = f(S1(,..I), S2(,..J),..Cond(I,J,...,L))

where the structure T must be compatible with the structures S1, S2, etc. Compatibility has been defined in Section 3.2.1. T,S1,S2 etc. may not include components which are structure variants. Characteristic to the [illegible] the presence of the free subscript L which

There exists an alternate format in which the left hand side variable is the parent of T and the subscript L does not appear• For example if:

P is GROUPCT(I))

the alternate format is:

PC.) = f(Sl(,..I),S2(..J),..cond(I,J,...))

Thus in scanning assertions containing data structure functions it is necessary to determine first which of the two formats is used. The determination is based on checking compatibility between the target structure (lhs) and the source structures (rhs). For example in the above cases P and T cannot both be compatible with SI, S2, etc.

The first format is assumed as more general, and assertions using the second format will first be converted into the first format.

4.2.1 Translation of the SELECT Function

Consider the assertion:

$A(I_J,*.,I_m,L)$ = SELECT($B(J_l,...,J_K,1)$, cond ($K_1,..K_m$,I

First a check must be made if the structure of A is compatible with B.

•• Two versions of the translation are presented below for the*cases where the selection condition does and"does-not depend on L*. Assume first that a function does not depend on L, then assertions must be generated to define a secondary array X, it!s*last element condition END.X, the dependent variable A and its range SIZE.A, as follows:

(a) $X(I_1,..I_m,I)$ =

IF I = 1 THEN IF cond $(K_1,..K_m,1)$ THEN 1

ELSE 0

ELSE IF cond $(K_1,..K_m,I)$ THEN

$X(I_1,..I_m, I-1)+1$

ELSE $X(I_1,..I_m,I-1)$

This defines a sublinear secondary vector X through which the dependence of A upon B will be expressed.

Next the assertion defining A is:

(b) $A(I_1,..I_m,X(I_1,..I_m,I))$ = IF cond $(K_1,..K_n,I)$ THEN $B(J_1,..J_k,I)$

The next assertion defines the size of the target array A.

(c) $SIZE.A(I_1,..I_m)$ = IF $END.B(J_1,..J_k,I)$ THEN $X(I_1,..I_m,I)$

Also

(d) $END.X(I_1,..I_m,I) = END.B(J_1,..J_k,I)$

(c) can not be replaced by an assertion (c') fo

END.A

(c') $END.A(I_1,..I_m,I) = END.B(J_1,..J_k,I)$

as A is shorter than B and this assertion may cause multiple definitions in the END.A array.

The other case of where the condition depends explicitly on L is discussed next. In this case (a) and (b) are replaced by(a') and(b'):

IF 1=1 THEN IF cond (K^. .K_m,l,l) THEN 1

ELSE 0

ELSE IF condCK-L,..!^,!, XC.,I-1)+1)

THEN X(I,...I_m,I-l)+l

ELSE X(I,...I_m,I-l)

(b«): A(I_1,..I_m,X(I_1,..I_m,I)) = IF (1=1 S cond

(K_1,..K_m,1,1))|(I>1 6 cond(K_1,..K_m,I,X(I_1,..I_m,I-

THEN BC^,..^,!)

4.2.2 Translation of the MERGE Function

Let the MERGE assertion be:

A(..,L) = MERGE (B(,..I), C(. . , J), Cond(..1,J,L))

In this case two secondary arrays X and Y are needed, both being sublinear such that whenever B is elected it is Subscripted by X and whenever C is selected it is subscriptec by Y. In addition we use the following arrays:

B_DONE denotes that the B list is exhausted.

C_DONE denotes that the C list is exhausted.

SEL denotes that B is selected.

The main assertion is

A(___,L) = IF SEL(L) THEN B(..,X(L)) ELSE C(..,Y(L) •]

The additional variables are:

B_D0NE(..,L) = L>1S(B_DONE(..,L-1) |

(END.BC. ,X(L-1)) S SEL(..,L-1)))

C__DONE(. . ,L) = L>1S(C__DONE(. . ,L-1) |

(END.CC..,Y(L-1)) g "* SELC...L-1)))

XC.L) = IF L=l THEN 1 ELSE

IF SEL(..,L-1) & ¬ B_DONE(...,L) THEN X(..,L-1)+1

ELSE X(...,L-1)

Y(..L) = IF L=1 THEN 1 ELSE

IF (SEL(..L-1)| C_DONE(..,L)) THEN Y(..,L-1)

ELSE Y(..,L-1)+1

SEL(..,L) = C_DONE(..,L)|(¬B_DONE(..,L) &

Cond(..,X(..,L),Y(..,L),L))

END.A(..L) = (B_DONE(..,L) & END.C(..,Y(L)) & ¬ SEL(..

(C_DONE(..,L) & END.B(..,X(L)) & SEL(..,L))

4.2.3 Implementation of the SORT Function

Unlike the other functions, there is little advantage in efficiency in translation of the SORT function into more elemental assertions. The reason is that the other functions can be performed in most cases in a simple pass over the source data. Also in such cases it suffices to locate one record at a time and not keep the entire source data in main memory. The program loop that implements a pass over the records or groups in the source data may also include other computations, thereby further improving efficiency. This is not the case in sorting where multi-passes over the data are necessary and major portions of source data must be located in main memory. For these reasons SORT may be implemented by substituting for it appropriate sub-programs during the Code Generation phase. If the data which is the argument of the function is input then it is advantageous to presort the data before other computations

are performed. Similiarly, if the target of the assertion is output data then it is advantageous to post-sort the data after the appropriate computations. Note that sorting may also be achieved by repeated use of the SELECT function, to select progressively the lowest/highest value substructures. The format of an assertion with a SORT function is

$$B(L_1,\ldots,1_k)=SORT(A(l_n..,1_k,J.,J_m),INC|DEC,(X,...))$$

The sort function reduces the lowest order dimensions (J ....J of A, which are in excess of the dimensions of B.

If A is source data on an external medium then pre-sorting is indicated and B will be a source data on an external medium. If A is target data on an external medium then post-sorting is indicated. Any computation that depends on B must be scheduled after the sorting and an analysis must be conducted of the feasibility of such computation. If A or B are interim data, the implication is that they must be entirely located in memory.

*4.2.4 Translation of the COLLECT function

The COLLECT function creates a two dimensional structure A from a one dimensional structure B. The condition argument defines the first element in a' next "row" structures of *A*

The form of an assertion-with a COLLECT function is:

A(..I,L) = COLLECT(B(..J), cond(I,J,L))

Two secondary arrays X(J) and Y(J) are needed to subscript A in the generated substitute assertion:

A C , X(J), Y(J)) = B(.,.J)

X(J) denotes I and Y(J) denotes the values of L for each value of I. Y(J) by itself is not monotonic, only within I, and (X(J),Y(J)) is lexicographically monotonic. The successor of A(..I,L) may be either A(..I,L+1) if Cond(I,J,I is ..false or A C . 1+1,1) if it is true.

then the generated assertions that define X and Y are.

X(J) = IF J=l THEN 1 ELSE

IF Cond(X(J-l>,J,Y(J-l)) THENX(J-1)+1

ELSE X(J-l)

Thus X is stepped only if the condition

Y(J) = IF J=l THEN 1 ELSE

IF Cond(X(J-l),J,Y(J-l).) THEN 1

ELSE Y(J-1)+1

Also let the row name (parent) of A be AA then

END.AA(X(J)) = IF cond(X(J),J,Y(J)) THEN END.B(J)

END.A(X(J),Y(J)) = cond (X(J),J,Y(J))

fr.2.5 Translation of the FUSE Function

The FUSE function assertion has the form:

A(..L) = FUSE(B(..I,J))

Here the two auxiliary arrays

X(L) and Y(L) denote the indices of B:

A(..L) = B(X(L),Y(L))

This last assertion is generated in addition to the definitions of X(L) and Y(L):

```
X(L) = IF L=1 THEN 1 ELSE
       If END.B(X(L-1),Y(L-1) ) THEN X(L-1)+1
                                ELSE X(L-1)
Y(L) = IF L=1 THEN 1 ELSE
       If END.B(X(L-1),Y(L-1) ) THEN 1
                                ELSE Y(L-1)+1
END.A(L) = END.BB(X(L) ) where BB is parent of B
```

U.3 Translation of Data Structure Equations

This section concerns assertions of the general form:

A = IF Condi THEN B ELSE

IF Cond2 THEN C ELSE

•«••

Where A,B,C... are of record or group node type (not fields) A,B,C... may be multidimensional with n_a, n_o, n_c... dimensions respectively. The objective is to substitute for the above assertion one or more assertions that define the components of A in terms of the components of B,C... .

A must be compatible with B,C... • Compatibility here means either of the following two casesCin order of priority):

1. For each field in A there is a unique matching field in B,C... with the same name, same dimensionality in respect to the root of A,B,C.., respectively, and same data type.

2. For each node in A there is a unique matching node in B,C... with the same data tree level in respect to the root of A,B,C.., respectively, either the same name or otherwise of the same sibling positi . (in order of sibling node's from, left to right) , same node type (i.e. either record/group or field), and.if the node is a field, also the same data type.

As noted A,B,C... may be-multidimensional. The respective subscripts expressions of these variables will be

retained as entered by the user. Subject to the restrictions listed in Section 2, the subscripts may be omitted by the user, whereupon they will be automatically inserted by the system. These subscripts will be omitted below, to simplify the presentation here. To further simplify the presentation we will consider the form

A = B

with the understanding that when 'Cond THEN...ELSE' clauses exist then the rules concerning A and B must also apply to A and each of the other variables following the ELSE(i.e.C, etc.). Note that the operands in the conditional clauses must be fields.

In previous scanning of the assertions, those assertions that have the above form and the operands are of record or group node types are passed as arguments to the translation of data structure equations. Basically this process is viewed as consisting of the following two steps:

1. Matching the nodes in A with corresponding nodes in B (and C,...if there are corresponding conditionals). We will refer to this stage as either MATCH_BY_NAME (assertion) for case 1 above, or MATCH_BY_STRUCTURE (assertion) for case 2 above.
2. Translation of the equation into an equation with lower level data structure operands (the lower level nodes may be record/group or field node types) This stage will be referred to as TR(assertion,

list of matched components).

In many cases SIZE and END attributes are propagated automatically from other operands in the same assertion for which the user has defined these attributes. In these cases it is not necessary to define attributes.

Applicability of compatibility by name (case 1) is checked first. If the check is negative then compatibility by structure (case 2) is checked. The algorithm for case 1 is as follows:

1) MATCH_BY_NAME(A=B) consists of checking of each field in A for a unique field in B with the same name, dimensionality (in respect to roots of A and B) and data type.

2) If the check is successful and the fields in A are $X_1 \ldots X_k$ then the translation stage TR(A=B, $A.X_i$, $B.X_i$ for i=1 to k) is called to generate assertions of the form

$$A.X_i(I_1 \ldots I_{na}) = B.X(I_1, \ldots I_{nb}) \text{ for } i=1 \text{ to } k$$

(for the more general case the assertions will be: $A.X_i$ = IF Cond1 THEN $B.X_i$ ELSE IF Cond2 THEN $C.X_i$ ELSE etc.)

Later in the MODEL analysis phase these assertions will be analyzed for consistency of ranges of variable arrays. Normally, the specified ranges of respective dimensions (from right to left) for $B.X_i$ will be passed to $A.X_i$ or

vice versa. If there is an inconsistency the user is requested (in an error message) to specify the missing range definition.

In case 2 the matching and translation are performed recursively one level at a time. Namely the assertion A=B is translated into assertions for the corresponding immediate descendents of A and B, respectively. The process will be repeated if the latter are records or groups, until the respective corresponding descendents are fields.

1) The MATCH_BY_STRUCTURE(assertion) process is as follows: The immediate descendents of $A:A_1 \ldots A_k$ are examined first if they each have the same names as a immediate descendents of $B:B_1 \ldots B_m$. The same named components A_i and B_j are then candidates for further matching. If there is no name correspondence then A_i and B_i, for i=1 to k, are candidates for further matching.

   The candidate pairs are then checked for the same dimensionality (within A and B respectively) and if they are fields, for the same data type, to determine if they all match.

2) In case of a match the translation process is called:

   TR(A=B, A_i,B_j for all k components of A)

   to generate the assertions.

A_i = Bj for all the matching sets•

If A_i^* and B_j are repeating, and the user inserted subscripts with A^and &$_j$ then a subscript on the right must be added by generating instead the assertions:

A_i(..,SUB1) = B_jC..,SUB1)

If A_i or B_j are not fields, then the matching and translation processes are called recursively.

The insertion of appropriate subscripts and determination of ranges of dimensions are performed in latter parts of the analysis phase called 'dimension propagation and range prop-agation respectively, which are described in the MODEL Program Generator documentation. These processes are also briefly reviewed in the next sections.

5. Analysis and Implementation of Indirect Subscripting

The significance of indirect subscripting is in improving the efficiency of the produced program* The understanding of how efficiency is improved and the associated analysis requires knowledge of two subprocesses of the MODEL system: range propagation and scheduling. These processes are briefly described below. For further explanation the reader is referred to the documentation on "MODEL Program Generator". A. Pnueli, K. Lu and N. Prywes, March 1980, or to a more general paper "Compilation of Nonprocedural Specifications Into Computer Programs" by N.Prywes. and A. Pnueli, October 1980.

Much of the information needed for generating a program is implicit in the MODEL specification. It is therefore necessary to perform the analysis to make such information explicit. As a first step a MODEL specification is represented in a convenient form, based on which implicit information may be derived and entered, checks be conducted and finally a schedule of program execution be derived. The usual approach to such analysis has been to use a form of a directed graph to represent dependencies between the different elements of the specification. "We have developed a generalize type of directed graph wich we termed array graph, where a single node represents an aggregate of elements and a single' edge an aggregate of edges or dependences between the belonging respectively.to the two nodes.

The nodes in an array graph represent potential process! steps associated with accessing and evaluating array variable

Each data structure and each equation are represented by a node. Each node is potentially compound, namely it represents the instances of the data structure or equation for all the array elements from 1 to N. Information on dimen sionality and ranges must therefore be associated with the nodes in the array graph, A node that corresponds to a data structure has associated with it subscripts corres- ponding to its dimensions. A node that represents an assertion (i.e. equation) has associated with it subscripts corresponding to the union of subscripts of the variables appearing in the assertion. Thus a compound m dimensional node A represents the elements from A(l,l,...l) to A(N-., $N_0 \ldots N_m$) where $N_1 \ldots N_m$ are the ranges of dimensions 1 to m, respectively.

Similarly a directed edge may be compound in that it represents all the instances of dependencies among the array elements of the nodes at the ends of the edge. These depen- dencies imply precedence relationships in the execution of the respective implied actions. There are several types of dependencies or precedences. For example, a hierarchial precedence refers to the need to access a source structure before its components can be accessed or, vice versa, the

need to evaluate the components before a structure is stored away. Data dependency precedence refers to the need to evaluate the independent variables of an equation before the dependent variable can be evaluated. Similarily, data parameters of a structure (range, length, etc.) must be evaluated before evaluating the respectiye structures. These edges are determined based on the analysis of the information in statements associated with the respective end nodes. Since each edge may be compound it is necessary to associate with it information on dimensionality and ranges.

An array graph AG is then a pair (N,E) where N is a set of compound nodes and E is a set of compound edges. The array graph AG=(N,E) represents an underlying graph $UG=(N_u,E_u)$ which is a conventional directed graph obtained by considering each of the elements of array nodes as individual nodes:

$$N_u=\{A(I_1,I_2\ldots I_m) \mid 1 \le I_1 \le N_1,$$
$$1 \le I_2 \le N_2,\ldots \text{where } A(I_1,I_2,\ldots.I_m)\&N\}$$

Similarily

$$E_u = \{A(I_1,I_2,\ldots.)\leftarrow B(E_1,E_2\ldots)$$
$$\mid 1 \le I_1 \le N_1,\ 1 \le I_2 \le N_2 \ldots \text{where } A(I_1,I_2\ldots)\leftarrow B(E_1,E_2$$

The analysis of a MODEL specification is based on the respective array graph. Consistency can be checked in a three step process. The first step, dimension propagation, traces the array graph in order to determine consistent

dimensionality of the nodes. Next comes insertion of subscripts in assertions where they have been omitted. The last step, range propagation, identifies the ranges of dimensions of arrays, for which the user has not provided specifications, with corresponding user specified ranges of dimensions of other arrays. This process also detects and reports conflicting, redundant, or missing range specifications.

It would be cumbersome for the user to define the range of each dimension of each node. Therefore, in the absence of a range specification for a dimension of a variable, the assertions where the variable is used are analyzed for implications of the range. For example, the assertion

$$X(I_m \ldots I_1) = Y(I_m \ldots I_1)$$

may imply that the ranges of the dimensions in X and Y referred to by the same subscript name are the same. This is referred to as range propagation. The range in this case is defined through propagation of the range from another node for which the range is known. The function of the range propagation process is also to determine the range sets, namely the sets of nodes and respective dimensions that have a common range definition. Consider an edge e: s←p. The correspondence of respective dimensions in nodes p and s is given in the subscript entries associated with the edge e.

For subscript expression of types 1,2 and 3 (I, 1-1 or I-K) and in the absence of contradictory range specifications, the indicated corresponding subscripts in p and s are assumed to have the same range and be members of a corresponding range set. By repeated propagations, a range set is determined, consisting of node-number and position-number pairs which have only one common range specification. Note that in the current version of MODEL the range is not propagated where a subscript expression is of type 4 (i.e. constant or any other form differing from types 1,2 and 3). One necessary extension to the MODEL system is to allow propagation of a range where indirect subscripts are used. This is further explained below.

The other extension concerns scheduling of program events. There are two special interdependent problems that must be coped with in scheduling execution of a node in the array graph. First, the array graph may contain cycles which prevent ordering the nodes in accordance with the edges. A maximally strongly connected component (MSCC) results from cycles in the array graph. Secondly, each node represents an array of data or equations and it is necessary to assure that all the elements are individually accessed and evaluated. Consider the simple example of a sin;gle node consisting of assertion a:

```
a:  AC I.  . . I )=f(B(I ...I, ,J,...J ))
       1     n     : a   b* 1   m
```

The I and J subscripts are distinct. $I_a....I_b$ is a subset of $I_1...I_n$. Assume that $CondI_1...CondI_m$ recognize the last elements in the ranges of $I_1...I_m$. To evaluate all the elements of assertion a it may be bracketed by iteration statements for all it's subscripts. The elements will then be evaluated while progressively varying the indices in each dimension from 1 to the last element defined by cond I.

The general approach to scheduling consists of creating a component graph which consists of all the MSCCs in the array graph and the edges connecting the MSCCs. The component graph is therefore an acyclic directed graph. It is then topologically sorted, resulting in a linear arrangement of the components which can be regarded as a gross level representation of the flowchart. The subscripts for each component are determined and appropriate iterations for these subscripts bracket the respective components.

An attempt is made to decompose a MSCC by deleting appropriate edges. Consider the simple example of a two node MSCC consisting of a one dimensional array X and the assertion

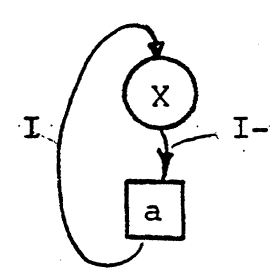
a: X(I) = IF I=1 THEN 1 ELSE X(I-1)+1

N is the range of I. Therefore the schedule would be:

do I from 1 to N.

MSCC consisting of nodes a and X

end I



The edge a←X has associated with it a subscript I of type 2, (I-1). It indicates that evaluation of the I-1 th element of X must precede the evaluation of the I th element. But this is already assured by the order of iterations for I from 1 to N. Therefore this edge may be deleted, which may cause decomposition of the MSCC and allow for its scheduling. More generally, to decompose a multi-node MSCC it is necessary to:

1) Find a dimension position in each node of the MSCC. These positions must all have a common range that can be given a corresponding common subscript name to use in an iteration statement. The iteration statement will than bracket the entire block of nodes that constitues

2) Find edges that represent dependencies on lower index elements of the selected subscript; these edges are deleted and may cause decomposition of the component.

For complex MSCCs the decomposition and scheduling may be performed recursively until all the cycles are opened.

The second extension concerns deleting edges in an MSCC in some cases where indirect subscripts are used.

The last extension has an important effect on the use of memory. If a variable is only referenced inside the same program loop in which it is computed, and the subscri

expressions for the variable are of the forms I, I-1,I-K ($K \geq 0$) then only K+1 elements of the dimension associated with I need be co-located in memory at any time during program execution. We want to extend the MODEL system to allow similar savings in memory when indirect subscripts are used as well. If all the elements along a dimension denoted by I of a variable X(..,I,....) must be co-located in memory, we refer to the dimension I of X as physical. Otherwise, if only K+1 elements need to be co-located at any time in the main memory, then we refer to such a dimension I of X as virtual.

General implications of using indirect subscripting are discussed further in Section 5.1. Section 5.2 discusses scheduling of assertions with indirect subscripts when the respective nodes are in an MSCC. Additional information on the design of the extensions associated with indirect subscripting analysis and program design is given in Appendix II.

Indirect subscripts are subscript expressions of the form A(.,X(I),.), where the secondary array X(I) is sublinear, i.e.

$X(I) \leq I$ and $X(I) \leq X(I+1)$.

It is possible to check whether a secondary array is sublinear by checking its definition. A sufficient condition for sublinearity is that the definition is of the form:

X(I) = IF I = 1 THEN (1|0) ELSE IF cond THEN X(I-1)
ELSE X(I-1)+1

5.1 Effects of Indirect Subscripting

Four possibilities of instances of indirect subscripting within a maximally strongly connected component (MSCC) are considered below.

a) A variable A is defined by an assertion

a: A(..1) = •. .

and all its rhs usages are of th ; form

= ..A(.•,1 -c, •.)

This situation is handled in the rresent system. The edge A(•. ,I-c,.)-*a may be de"; eted decomposing the MSCC and A may be virtual*

b) A variable A is def. led by an assertion

a i J\\'t j ī j •\' = ••

and some of its rhs usages are indirect, i.e. of the form

It is still possible to delete the € e A(. . ,X(I-c)-d,. .)-*a since by the assumption of sublineai ity I>X(. ,I-c,.)- d and an element is not accessed before it is evaluated. However, under these circumstances A, must be physical since X(..,I,.#) may lag arbitrarily far behind I requiring the storage of 'all the elements

A(.MX(I),.), A(..,X(I)+1,..).. A(..,I,).

These elements have been evaluated but not yet.used.

In scheduling a program loop on I which consists of the nodes of a MSCC we may remove edges of the following types:

B(..,X(.,!,•),.)«-A(.,X(•.,I[-c],..)-d>..) c,d>o

B(.,I,.)«-A(..X(.,H>c]j.)-d,..) c,d>o

Also it is proposed trust the user and remove edges of the types:

B(..,X(•,1,.),.)*A(»,X(••,I-c..),..) c>o

B(••,1,•..)«-A(•.$_5$ •$_5$I-c,..),..) c>o

These latter edge. re not guaranteed to refer to previous instances of A, since there may be cases in which X(.jl,..)= X(..,I-c,.). however, we will rely on the user in such cases to refer to X(.,I,«.) instead of X(.,I-c,*,)*

c) A variable A is defined by an assertion

[illegible]

and some of its usages are direct, i.e. of the form

=..A(..,1C-c],*.)

these assertions can not be scheduled in a loop on I, since it is possible that X(Il<IC-c] and an edge may be deleted only if A(I) depends on lower index elements of A.

d) A variable A is defined by an assertion

[illegible]

and all its usages are indirectly subscripted, i.e. of the fo:

...3...AC... X(..,I[-cl)C-d],..)

As stated in (b) above, the user is trusted to assure that:

XCI) > XClC-cDC-d]

Also:

max CXCI) - X(I[-cp C-d3)=c+d

The index of A on the lhs is greater than on the rhs. The rhs value of the element of A is evaluated prior to the evaluation of the assertion defining the lhs element of A. The edge

A(.., X(I[-c])[-d],...)→a

may be deleted thus decomposing the MSCC. Furthermore the maximum number of elements of A needed to be co-located in memory is c+d+1. Therefore the dimension I of A may ve virtual, and we may schedule the generation and usages of the variable in the same loop.

To summarize:

Case a: a: A(...,I,...) = ...A(..,I-c,...)

The edge A(...,I-c,...)→a may be deleted, the MSCC decomposed (if possible) and if all references to A can be placed in a program loop then A may be virtual in the dimension of I.

Case b: a: A(..,I,...) = ...A(..,X(I[-c])[-d],...)

The edge A(..,X(I[-c])[-d],...)→a may be deleted and the MSCC decomposed. However, A must be physical in the dimension of I.

Case c: a: A(...,X(I),...) = ...A(...,I[-c],...)

The edge A(...,I[-c],..)→a may not be deleted.

Case d: a: A(•. ,X(.I) ,..) = .-. .A(.. ,X(IE-c]) C-d] ,. .)

The edge A(...,X(I^c])C-d],...)->a may be deleted and A may be virtual in the dimension of I#

5,2 Scheduling a Maximally Strongly Connected Component (MSCC

The process of scheduling a MSCC calls for propagation of a range to at least one dimension of each assertion and each variable in the component. The resulting range set must consd of only one dimension of every node in the MSCC, barring the case where a range can be propagated to more than one dimension of a node. If such a range set is found, then the nodes in the MSCC may be scheduled in a program loop for the respective subscript.

Note that any instance of the above case(b) implies that a certain dimension of an array must be physical. This should be used in computing the storage penalty associated with the loop candidate subscript $I_{\#}$ If several subscripts are legal loop candidates we pick the one with the lowest penalty•

Consider the assertion:

[illegible]

If I appears in a position of B which was selected for the range set, it must also appear in the position of A which was selected for the range set and it cannot be in the form of a type 4- subscript expression.

These conditions can be extended to cases of the assertions of the forms:

B(..,X(.,I,.),..) =

or

.. = A(.,X(.,I,.),..)

provided I appears in the range set selected position of X, and X(.,I,.) is in the range set selected positions of A and B, respectively.

Consider again case (c) above, where an assertion defines A in the form:

A(.,X(I),.) = ..

and A is used on the lhs of the same or another assertion a of the form:

a: B(...,I,..) = .. A(..,I,..)

then these assertions cannot be scheduled in a loop, without some changes. In the case that such a situation is detected conceivably I may be replaced in a by X(I) to obtain:

a': B(..,X(I),..) = ..A(..,X(I),..)

This replacement should of course be carried out for all instances of I in a'.

As a result of this replacement the new variable B is defined by indirect indexing. Consequently it may be necessary to replace I by X(I) also in other assertions which use B(.,I,..) on the rhs.

This sequence of replacements may:

1. Terminate, in which case we are assured that no

more instances of case(c) exist. We may then proceed with the scheduling.

2. There is continuous replacement in a loop. This becomes evident when we apply the replacement twice to the same assertion. Consider for example the assertion:

   A(X(I)) = f(A(I-1))

   Applying a replacement to it yields:

   A(X(X(I)))=f(A(X(I)-1))

   which is again case(c) and doubly indirect. The appearance of doubly indirect subscripting is a sign of a cycle of replacements which may continue forever. Consequently on the detection of a newly created double indirect subscript expression we conclude that the subscript candidate cannot be selecte for a loop for the MSCC.

There is also another problem associated with the appeara of an indirect subscript on the lhs of an assertion. Consider the assertion:

a': A(.,X(I),..) = ...

Such a statement in a loop on I implies inefficiency since it will be executed for each I regardless of whether X(I)=X(I-1), in which case it is a duplication of a computation performed before. The problem is not only that of efficiency, but if instead of an assertion we have a record node R(.,X(I),..).

We do not want to have it read for each I but only when X(I) increases. Consequently we may add a condition to such assertions in the form:

a": A(..,X(I),..) = IF(I=1&X(1)=1) |(I>1 & X(I)>X(I-1))

THEN ....

Typically several statements will all have such a common condition and can be grouped into one block. The condition will then be tested only once for the complete block.

5.3 Range Propagation Through Indirect Subscripting

Consider an assertion

a: A(X(I)) = ..B(I)..

In the absence of conflicting range specifications we infer that the ranges of B and of A are associated. This is different than saying that A and B have identical ranges. We can say that the range of A is associated through X with that of B. Thus in the event of an assertion:

C(J) = A(J)+B(J)

unless an independent range specification is given for C, it is not known whether J should range up to N_B, the range of B, or up to $X(N_B)$ which is the range of A.

In the case of two other assertions:

C(X(I))=..B(I)..

E(J)=C(J)+D(J)

the range of E is associated through X with that of B, i.e. if the size of B is N_B then the size of E is $X(N_B)$.

In the present version of MODEL a range specification

(i.e. a termination condition for the respective loop) can be direct -- covering the cases:

1. A fixed value specified in the data statement.
2. END.A
3. SIZE.A
4. END.A implicitly defined by end of file for a record or a group above the record level in an input file.

Consider the assertion

B(X(I))=A(I)

The termination condition may be propagated indirectly from A to B. The pointer RANGEP points from node (A) to node (B implying RANGE(A) = RANGE(B). This indirect range specification must be exteded to an assertion of the form B(X(I))= An indirect pointer INDR is added to each node data. This indirect field, if not empty, points to a secondary array X which relates the range of B to the range of A, i.e.

RANGE(B) = X(RANGE(A))

RANGEP(B)=A

INDR(B)=X

If the assertion is of the form:

B(I)=A(X(I))

Then (through backwards propagation)

RANGEP(A)=B, INDR(A)=X, RANGE(A)=X(RANGE(B))

representing the fact that

X(RANGE(B)) ≤ RANGE(B)

The concepts of a range set should also be extended to the situation where two nodes may be put into the same loop, even though their ranges are not identical but are associated through a secondary array.

APPENDIX I

Syntax and Syntax Analysis Checks

In this Appendix we discuss the modifications which are necessary in the syntax definition of the language. We will also consider the needed revisions in the processing following immediately after the syntax analysis.

Consider first the syntactic changes needed in order to implement the matrix operations:

I.1 Syntax Modifications for Matrix Operations

The first step is to cause the syntax analyzer to recognize the symbol combinations for the new operations which are "* for matrix multiplication, "∿ for matrix transposition and "/ for matrix division and inversion. The simplest modification would be to change the syntactic definition of <term> and <factor> into

<term>::= <mfactor> [<mops> <mfactor>]*

where

<mops> :: = *|/|"*|"/

<mfactor>::=<unop> <mfactor> | <factor>

where

<unop>::=" ∿ |"/

This extends the class of binary multiplication operato to include also "* and "/ and introduces a new class of unar operators "∿ and "/. The definition is such that it will allow multiple unary operators to be applied to a single argument, such as:

"/"∿A or "∿"/B

It will also correctly interpret a sequence of a binary operator followed by a unary operator:

A"*"~B.

In fact the actual EBNF/WSC will be of the following form:

<term>::= /WRTERM1//SVTERM/<mfactor>/SVCMP1/
[<mops>/SVNXOP/<mfactor>/SVNXCMP/]*/STALL/

<mops>::= /MOPREC/

<mfactor>::= /WRMFAC//SVMFAC/<unop>/SVOP1/
<mfactor>/SVCMP1//STALL/I<factor>/STALL/

<unop>::= /UNOPREC/

In the above:

MOPREC is the routine for recognizing multiplication operators and it' should be extended to recognize tf* and "/ as well. In the internal representation of delimiters we should allocate some numbers to these operators. It is suggested to have (see page 59 in the Documentation Manual)

25 - "*

26 - "/

27 - "\

The internal type assigned to <mfactor> should be identical to that of <factor> namely 11.

WRMFAC-Should be a new error stacking routine. It should stack the error message —

"EXPECTING A FACTOR"

SVMFAC - Is a node saving (and presetting) routine for initiating a node to be of type <mfactor>•

Since the internal type is 11 the same as <factor>. we may replace WRMFAC and SVMFAC by WRFAC and SVFAC respectively. In this case no new error and saving routines are needed.

UNOP - This new routine is necessary. It is similar in operations to MOPREC and should recognize and derive the internal representation of the unary matrix operations " / and "∿.

The system should recognize the reserved name UNIT denoting the unit matrix.

I.2 Extensions for Variant Structures

I.2.1 Syntax and Syntax Analysis

The syntactical definition of the list of substructures into which a record or a group breaks is currently given by the variable <item_list>. This variable appears in the definition of <group_start> and of <record_start> . Consequently it is sufficient to modify the following definitions:

<record_start>::=<record>/MEMINIT/[(<alt_item_list>)]
/STREC/

<group_start>::=<group>/MEMINIT/[(<alt_item_list>)]
[,TABULATED/SVTAB/]/STGRP/

where

<alt_item_list>::=/ITEM02/<item_list>
[//SVALT/<item_list>]*

Thus we add an additional level to the description of the substructure of a record or a group. Consequently the storage entry structure associated with the record or group described will have to be changed. A relatively easy and compatible modification will be to add a tag field to the auxiliary description structure associated with the storage entry (RECGRP Structure). This tage will contain the value 'SEQ' if the members listed in RECGRP are the substructures into which the current structure breaks. It will contain the value 'ALT' if the listed members are the alternatives or variant terms which the current structure may assume. This restricts the representation to describe either a sequential breakdown or a list of variants but not both. Consequently a statement such as

G IS GROUP (A,B /C)

will have to be represented by the system as though it were presented with two statements:

G IS GROUP ($ST /C)

$ST IS GROUP (A,B)

The newly introduced name $ST is system generated name and should be entered into the associative memory as well as represented by an appropriate storage entry.

In order to manage the acquisition of an <alt_item_list> we maintain a double storage area. The first area stores names of variants while the second area stores a sequence

of substructures. We will refer to them as the variant and sequence area respectively. Any item list encountered will be entered first into the sequence area. On encountering a $^{f}/^{f}$.we cheak whether the sequence area contains one or more items. If it contains a single item this item is moved into the variant area. Otherwise let the sequence area contain the sequence A,B,.. . Then we create a new name $ST and simulate the storage of the data declaration

$ST IS GROUP (A,B,..)

In this case we move $ST into the variant area. In both cases we clear the sequence area. The task of managing this acquisition is shared between the following routines:

MEMINIT - Should initialize the double area.

SVALT - A new routine. It is called on encountering an $^{f}/^{f}$« It performs the movement of an item list from the sequence area into the variant area, as described above.

STREC, STGRP - They finalize the storing of entries for record and group respectively. If they find that the variant area is still empty, a regular storage entry is created by moving the sequence area into RECGRP and setting the tag field to ^{f}SEQ^{f}. If the variant area is not empty we call on SVALT one more time to move the last sequence area into the variant area. We then define RECGRP by moving the variant area into it and

setting the tag field to 'ALT'.

The system should recognize the special prefix DISCR applied to groups which are declared to be described by variants.

I.2.2 Computation of Attributes For Variant Structures

The routine ENHRREL computes certain attributes of declared data structures. Some modification in the computation is necessary in order to allow for variant structures. Following is a list of the attributes which are affected:

LEN-DAT - The length (size) of the structure. If the structure is a variant, its size in bytes is computed by taking the maximum of the sizes of its variant substructures.

VARS - This attribute marks structures which have varying substructures. It is used in deciding whether a structure can be moved in one statement or has to be broken into a sequence of statements moving each of its substructures. If a structure is defined by a variant, it must be marked (as well as all its ancestors) as varying, i.e. VARS='1'B.

BROTHER - This attribute points at a node which is the right brother of a given node, both nodes sharing the same father. Even though variants do not follow one another sequentially. The BROTHER link should be established for siebling variants.

DISCRB - This new attribute is set to '1'B for all groups or records A such that the attribute variable DISCR.A is the target of some assertion.

I.2.3 Siebling Edges for Variant Structures

In general when several structures form the sequential decomposition of a higher structure, dependency edges of type 8 are drawn between them is the array graph. The purpose of these edges is to delay the processing of a later node until its predecessor has been processed. If the structures are variants, i.e. form a parallel decomposition of a higher structure these edges should not be drawn since only one variant will be available at any time. Thus, type 8 edges should not be drawn between nodes whose father's tag is 'ALT'. This modification should also take place in ENHRREL.

I.2.4 The DISCR Edge, type 22

Associated with each variant structure A there is an data-parameter DISCR.A. This integer valued variable has the values 1,2,.. according to whether the first, second, etc. variant is to be chosen for A. There is an obvious precedence order which says that DISCR.A has to be computed before any of the fields in the variants of A can be accessed. This dependency relation is represented by a type 22 edge. The natural form of this edge would have been:

$$A(U_k,..U_1) \overset{22}{\leftarrow} DISCR.A(U_k,..U_1)$$

Unfortunately, in some cases the DISCR attribute may depend on values of the variant fields themselves. Consider the example:

A IS GROUP (B/C)

B IS RECORD (X,Y)

C IS RECORD (X,Z)

(X,Y) ARE FIELD (NUM)

Z IS FIELD (CHAR(10))

a : DISCR.A = X

Here B and C are variant records. Both begin with common numeric field X which has the value 1 for B and 2 for C. The continuation of B and C differs after the similar X field. It is clearly feasible to read the record A, retriev the field X without knowing if it belongs to B or C and then using the assertion for DISCR.A in order to decide if B or C a implied. Drawing the 22 edge between DISCR.A and A would lead in the above case to a cycle involving:

$X \overset{1}{\leftarrow} A \overset{22}{\leftarrow} DISCR.A \overset{7}{\leftarrow} a \overset{3}{\leftarrow} X$

Hence we must have a more complicated rule:

let A : $(B_1/B_2/,../B_m)$

And let each B_i be describable as

$B_i: X_1, X_2, .. \; X_k, Y^i_{k+1}, ..$

i.e. all the B_i structures have identical initial segment $X_1, .. X_k$ and then different continuations $Y^i_{k+1}, ..$ dependent on i. Then we may allow DISCR.A to depend on the common prefix $X_1, .. X_k$. Correspondingly we will draw the edges:

$Y^{i}_{k+1} \overset{22}{\leftarrow}$ DISCR.A for i=1,..m.

In the example above there would be two edges:

Y $\overset{22}{\leftarrow}$ DISCR.A and

Z $\overset{22}{\leftarrow}$ DISCR.A

In order to draw the 22 edges we have to perform concurrent analysis of B_1,.. B_m to find the smallest k such that at least two of them disagree on Y^{i}_{k+1}.

In this and later applications we have to form an exact definition of when two structures agree and to construct a recursive algorithm for checking for agreement.

Def. Two structures A and B agree (have similar structures) if:

A and B are identically named (assuming no multiple definitions), or

A and B are both fields with the same type and same fixed attributes (repetition, length, precision, etc.), or

A: $(X_1,./X_k)$, B: $(Y_1,./Y_k)$, A and B have the same type and same fixed repetition attribute, and X_i agrees with Y_i for i=1,..k.

I.3 Syntax Analysis For List Functions

No syntax modifications are required for the list functions. The table of recognizable standard functions should

be extended to include the new functions:

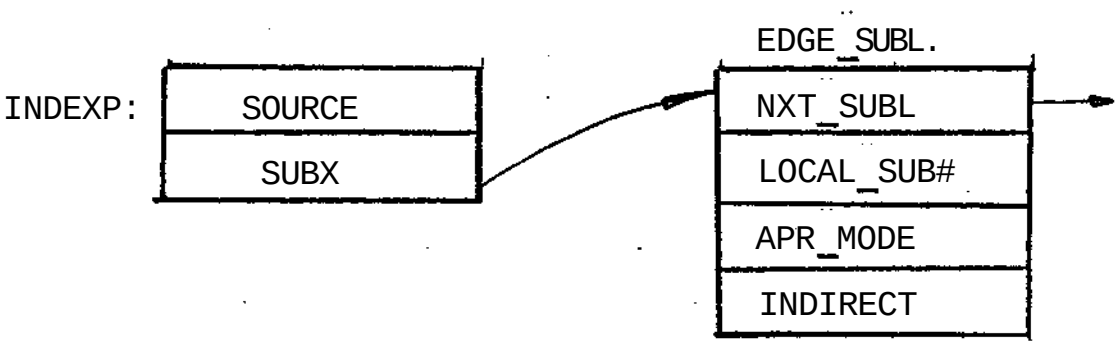
SELECT, MERGE, SORT, COLLECT, FUSE.

APPENDIX II

Implementation Details -Edge Types and Representations

The representation of assertions with indirect indexing calls for the extension of the structures representing the edges, as well as for some new types of edges. The representation of the edges for a given node in the current system includes a list of subscripts for the left hand side which is assumed common to all edges. This list is built out of a linked list of elements called L0CAL_SUB. The righ hand side of each edge consists of a linked list of structur called EDGE_SUBL. Both structures have to be extended by th addition of a pointer field called INDIRECT. This field if empty (NULL) means that subscripting is direct, otherwise it points to an additional structure called INDEXP which descri the subscript list of the indirect expression.

The structure INDEXP consists of the following:



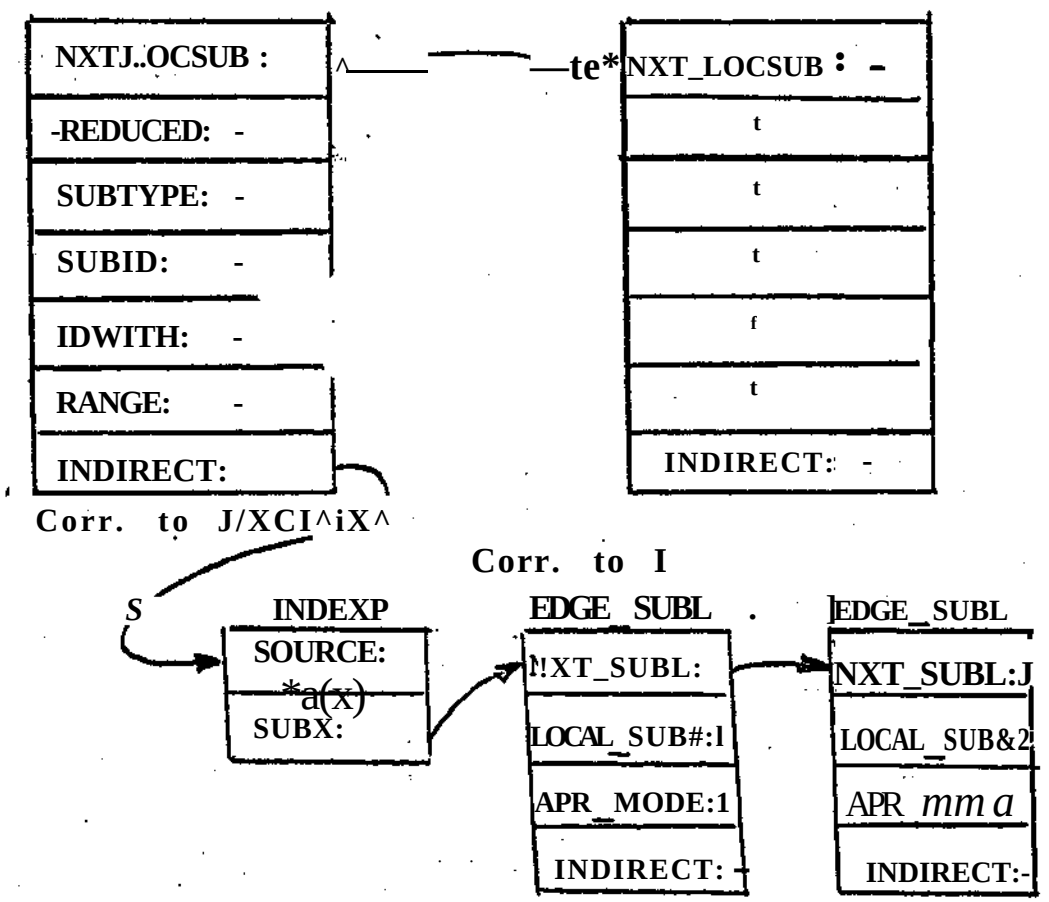
The SOURCE field of INDEXP is the node number of the secondary array and the list of EDGE_SUBL describes its subscript expressions (from right to left). Similarly to the extension of EDGEJ3UBL, the structure L0CAL__SUB is extended by adding the field INDIRECT,

Consider the representation of the edge:

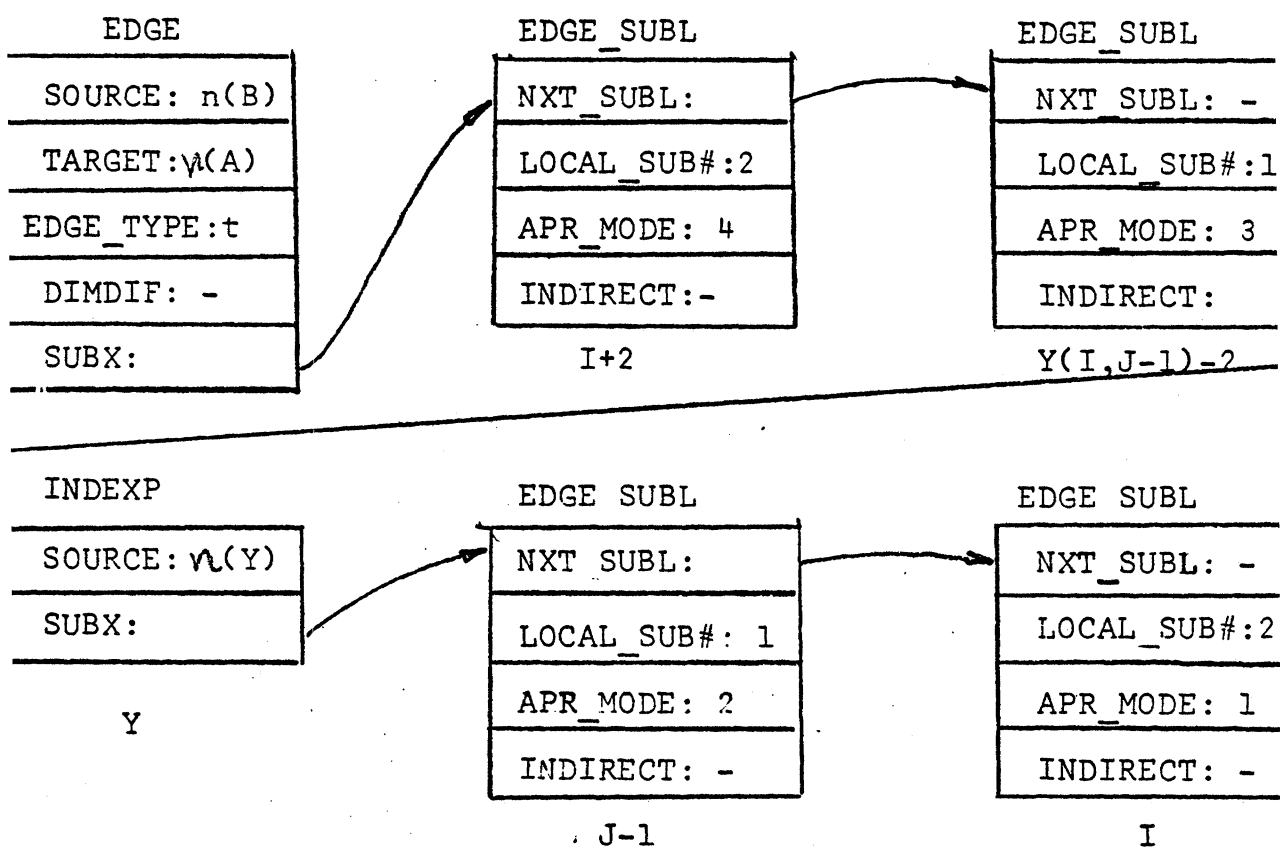
A(I,X(I,J)) ← B(Y(I,J-1)-2,I+2)

In this representation we will refer to n(Z) as the node number of Z.

We start by the representation of the left hand local subscript list:



Note that the first LOCAL_SUB actually represents both X(I,J) and J. Its representation of X(I,J) is given by its INDIRECT field pointing to the INDEXP structure. On the other hand it is associated with J in that everywhere else the first local subscript is J (including in the descript of INDEXP structure). Also the identification of the subscript range for J will be done in the IDWITH and RANNGE fields of the first LOCAL_SUB structure. Consider next the representation of the edge itself:



Note that Y(I,J-l)-2 is of node i'pe 3 on the outer level (K-2), and node type 2 (K-l) on the inner level.

We have to introduce a new type of edge for expressing the dependence of an assertion a on an index array. Let

at ....A(}*.X(J_5.?J }1)}..).«
1 m

where A may appear on the left or right hand side of a. Then we draw an. edge of new type 2^

24: a(..) %7 $X(J_n,..J_m,1)$

The subscript list for a on the left hand side will be the local subscript list of a.

Consider next the drawing of edges type 3 and 7 and their reverses in the virtual case:.

Let a be an assertion defining the variable A which may assume any of the forms

a: $A(I_k,\ldots,I_l) = \ldots$ or

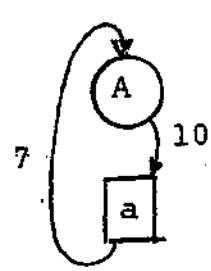
a* J\\ • $ j\\ y •} -in • / $ • . / " - .

Regardless of the form we will always have the following edge:

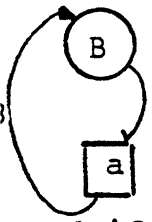
$A(U_k,..U_l) \overset{7}{\leftarrow} a(U_k,..U_l).$

and the reverse, virtual edge

$a(U_{JC},..U_m,..UL_x) \overset{10}{\rightarrow} A(U_{V5x}..U_{m-1},..U_l)$



Here we have to distinguish between three cases:



a: $A(I_k,..I_1) =.. B(J_n,..J_1)$

We draw

$a(I_k,..I_1) \overset{3}{\leftarrow} B(J_n,..J_1)$

and if I_m is a virtual subscript, identified with J_1 we draw the reverse edge

$$B(J_n,..I_m,..I_1) \overset{18}{\leftarrow} a(I_k,..I_m-1,..I_1)$$

If

$$a:\ A(I_k,..I_m,..I_1) = ..B(..X(.,I_m)..)$$

we draw

$$a(I_k,..I_m,..I_1) \overset{3}{\leftarrow} B(..X(..I_m),..)$$

and its reverse

$$B(..X(..I_m)..) \overset{18}{\leftarrow} a(I_k,..I_m-1,..I_1)$$

If

$$a:\ A(I_k,..Y(.,I_m),..I_1) = ..B(.,X(.,I_m),..)$$

we draw

$$a(I_k,..Y(..I_m)..I_1) \overset{3}{\leftarrow} B(..X(..I_m)..)$$

and its inverse

$$B(..X(..I_m..)..) \overset{18}{\leftarrow} a(I_k,.,Y(..I_m-1..),..I_1)$$

similarly for

$$a(A(I_k,..X(..I_m..)..) = ..B(...I_m,..)$$

R E F E R E N C E S

1. Gana, J.E., "An Automatic Program Generator For Model Building In Social and Engineering Science," Technical Report, Submitted to ONR, Ph.D. Dissertation In Computer Science, University of Pennsylvania, October 1978.

2. "MODEL II - Automatic Program Generator, User Manual," Revisionf of Version III, July 1978.

3. Pnueli, A., Lu, K., and Prywes, N., "Model Program Generator: System and Programming Documentation", Fall 1980 Version.

4. Prywes, N., Pnueli, A., and Shastry, S., "Use of a Non-Procedural Specification Language and Associated Program Generator In Software Development," TOPLAS, October 1979.

5. Sangal, R., "Modularity In Non-Procedural Languages Through Abstract Data Types", The Moore School of Electrical Engineering, University of Pennsylvania, August 31, 1980.

6. "Verification and Correction Of Non-Procedural Specification In Automatic Generation of Programs", Technical Report, Ph.D. Dissertation In Computer Science , University of Pennsylvania, 1978.