

University of Pennsylvania
Department of Computer and Information Science
Moore School of Electrical Engineering
Philadelphia, Pennsylvania 19104

Technical Report

GENERATING DATA FLOW PROGRAMS
FROM NONPROCEDURAL SPECIFICATIONS

March 1983

by

Maya Balkrishna Gokhale

Submitted to
Information System Program
Office of Naval Research
Under Contract N00014-76-C-0416

Moore School Report

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Generating Data Flow Programs from Nonprocedural Specifications		5. TYPE OF REPORT & PERIOD COVERED Technical Report
7. AUTHOR(s) Maya Balkrishna Gokhale		6. PERFORMING ORG. REPORT NUMBER Moore School Report
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Pennsylvania, Moore School of Electrical Engineering, Department of Computer Science, Philadelphia, Pennsylvania 19104		8. CONTRACT OR GRANT NUMBER(s) ND0014-76-C-0416
11. CONTROLLING OFFICE NAME AND ADDRESS Office of Naval Research Information Systems Program, Code 437 Arlington, Virginia 22217		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		12. REPORT DATE March, 1983
		13. NUMBER OF PAGES 208 pages
		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) General Distribution		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Dataflow, Automatic Program Generation, Nonprocedural Languages, Programming Dataflow, MODEL, MaD, Program Efficiency		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Data flow is a mode of parallel computation in which parallelism in a program can be exploited at the fine grained as well as macro level. A data flow computer executes a data dependency graph rather than the program counter controlled sequence of instructions executed by conventional machines. Nonprocedural languages appear to be especially appropriate high level languages for data flow computers. Nonprocedural languages have only two		

statement forms: data description and assertion. The assertions enumerate the relationships among the data. A data dependency graph is also a suitable representation for a nonprocedural language program (or specification).

This research is concerned with translating the dependency graph form of a specification to a program graph for a data flow machine. Specifications in the MODEL language are translated into an intermediate form, the data flow template. The template is a language-independent representation of the specification. The template is then translated into a data flow language (Manchester Dataflow) for the Manchester University machine.

The translation consists of creating an array graph to represent the specification; generating the data flow program template from the array graph; and translating the template into MaD.

**GENERATING DATA FLOW PROGRAMS
FROM NONPROCEDURAL SPECIFICATIONS**

by

Maya Balkrishna Gokhale

A DISSERTATION

in

Computer and Information Science

**Presented to the Graduate Faculties of the University of Pennsylvania
in Partial Fulfillment of the Requirements for the Degree of Doctor of
Philosophy.**

1983



Supervisor of Dissertation



Graduate Group Chairman

CHAPTER 1	INTRODUCTION	
1.1	MOTIVATION	1
I.2	CONTRIBUTIONS	7
I.3	ORGANIZATION OF THE DISSERTATION	9
CHAPTER II	DATA FLOW	
II.1	THE DATA FLOW COMPUTATION MODEL	11
II.1.1	Properties Of The Data Flow Model	14
II.1.1.1	Parallelism	14
II.1.1.2	Lack Of Address	15
II.1.1.3	Referential Transparency	15
II.2	ORGANIZATION OF DATA FLOW MACHINES	17
II.2.1	The Data Flow Instruction	17
II.2.2	Scheduling Of Instructions	20
II.2.3	Processor Network Topology	25
II.3	PROBLEM AREAS IN DATA FLOW DESIGN	29
II.3.1	Reentrancy In Graph	31
II.3.2	Partitioning The Program Among Processors	33
II.3.3	Data Structures	34
II.3.4	Structure Processing On Some Data Flow Machines	35
II.3.5	Implications Of The Structured Data Problem	39
II.4	THE MANCHESTER DATA FLOW MACHINE	40
II.4.1	Machine Layout	41
II.4.2	Data And Instruction Formats	43
II.5	CONCLUSION	44
CHAPTER III	LANGUAGES FOR DATA FLOW	
III.1	CONVENTIONAL PROGRAMMING LANGUAGES	45
III.2	DATA FLOW LANGUAGES	47
III.2.1	Common Properties	47
III.2.2	Iteration	48
III.2.3	Parallel Constructs	49
III.2.4	Streams	50
III.3	THE MAD PROGRAMMING LANGUAGE	50
III.3.1	The Program Heading	51
III.3.2	Structured Types In MaD	52
III.3.3	The Expression	54
III.3.4	The Basic Expression	57
III.4	NONPROCEDURAL LANGUAGES	58
III.4.1	Common Properties	58
III.4.2	LUCID	60
III.4.3	MODEL	60

III.5	THE MODEL SPECIFICATION LANGUAGE	62
III.5.1	Data Declaration	62
III.5.2	Array Data	64
III.5.3	Subscript Data Type	64
III.5.4	Range Arrays	65
III.5.5	Assertion	68
III.6	THE EXAMPLE SPECIFICATION	69
III.7	CONCLUSION	70
CHAPTER IV	THE MODEL SYSTEM	
IV.1	THE UNDERLYING GRAPH	73
IV.2	DATA STRUCTURES DEFINING THE ARRAY GRAPH	74
IV.3	NODE DIMENSIONALITY	80
IV.4	PRECEDENCE RELATIONSHIPS	81
IV.4.1	The EDGE Data Structure	83
IV.4.2	The EDGE Subscript Expression List	84
IV.5	RANGE SETS	85
IV.6	PHYSICAL AND VIRTUAL DIMENSIONS	87
IV.7	CONCLUSION	89
CHAPTER V	SCHEDULING THE ARRAY GRAPH FOR A DATA FLOW MACHINE	
V.1	INTRODUCTION	90
V.2	DATA STRUCTURES USED BY THE SCHEDULER	91
V.2.1	The Component Graph	91
V.2.2	The Data Flow Template	93
V.3	A SIMPLE SCHEDULING ALGORITHM	97
V.3.1	Initialization	98
V.3.2	Procedure SCHEDULE_GRAPH	99
V.3.3	Procedure SCHEDULE_COMPONENT	100
V.4	CONCLUSION	111
CHAPTER VI	SCHEDULING II: EFFICIENCY CONSIDERATIONS	
VI.1	INTRODUCTION	112
VI.2	BLOCK ENLARGEMENT	113
VI.3	DATA STRUCTURE SIMPLIFICATION	119
VI.3.1	Virtual Dimensions For Local Data Nodes	121
VI.3.2	Virtual Dimensions In Iterative Blocks	122
VI.4	EXPERIENCE WITH THE MANCHESTER MACHINE	123
VI.5	THE MODIFIED SCHEDULING ALGORITHM	124
VI.5.1	Criteria For Merging Components	125
VI.5.2	Adding Edges To The Component Graph	132

VI.5.3	The Revised SCHEDULE_GRAPH	136
VI.5.4	Locating Virtual Dimensions	141
VI.6	CONCLUSION	142

CHAPTER VII CODE GENERATION FOR THE MANCHESTER MACHINE

VII.1	INTRODUCTION	143
VII.2	RESTRICTIONS OF THE MAD LANGUAGE	147
VII.3	ORGANIZATION OF THE CODE GENERATION PHASE	148
VII.4	GENERATING DATA DECLARATIONS	149
VII.4.1	The Program Header	150
VII.4.2	Data Declarations For Global Variables	152
VII.4.2.1	Interim Variables	153
VII.4.2.2	Variables To Hold The Program Result	156
VII.5	GENERATING THE ASSIGNMENT STATEMENTS	156
VII.5.1	Procedure GenAssr	157
VII.5.2	Procedure GenBlk	158
VII.5.3	Procedure LocalVar	160
VII.5.4	Procedure ForEach	161
VII.5.5	Procedure Iter	163
VII.6	GENERATING THE RETURN STATEMENT	163
VII.7	CODE GENERATION FOR EXAMPLE	164
VII.8	CONCLUSION	166

CHAPTER VIII CONCLUSION

VIII.1	SUMMARY OF CONTRIBUTIONS	167
VIII.1.1	Desirability Of Nonprocedural Languages For Data Flow	167
VIII.1.2	Scheduling The Array Graph For Data Flow	168
VIII.1.3	Generating Data Flow Programs	168
VIII.2	FUTURE RESEARCH	169

APPENDIX A MAD BNF

APPENDIX B MODEL BNF

APPENDIX C EXAMPLES

C.1	MATRIX MULTIPLY	180
C.2	TRAPEZOIDAL INTEGRATION	185
C.3	FAST FOURIER TRANSFORM	191

BIBLIOGRAPHY

List of Figures

Figure 1.1 Matrix Multiply in Model	5
Figure 1.2 Matrix Multiply in Id	6
Figure 1.3 From Problem Specification to Solution	8
Figure 2.1 An Arbitrary Computation	13
Figure 2.2 A Linear Sequence of Instructions	13
Figure 2.3 Data Flow Graph of the Computation	13
Figure 2.4 TI DDP Instruction	19
Figure 2.5 Processing Unit of the TI DDP	21
Figure 2.6 Manchester Data Flow Machine	23
Figure 2.7 MIT (Arvind) Data Flow Machine	24
Figure 2.8 TI Network Topology	26
Figure 2.9 MIT (Arvind) Network Topology	26
Figure 2.10 DDM Tree	28
Figure 2.11 Multiple Input Sets to an Instruction	30
Figure 2.12 Manchester Token Label	30
Figure 3.1 A Two-Dimensional Array and its Size Array	67
Figure 3.2 A Two-Dimensional Array and its END Array	67
Figure 3.3 The EXAMPLE Specification	70
Figure 4.1 The NODE Data Structure	76
Figure 4.2 A Node Subscript	77
Figure 4.3 The EDGE Data Structure	78
Figure 4.4 Subscript Expression	78

Figure 4.5 Nodes and Edges for the EXAMPLE array graph	79
Figure 5.1 The Component Graph	92
Figure 5.2 The Data Flow Template	96
Figure 5.3a The Initial Component Graph	100
Figure 5.3b The New Component Graph	100
Figure 5.4 The Template For EXAMPLE	105
Figure 5.5 A Cycle in the Graph	107
Figure 6.1a The Original Component Graph	117
Figure 6.1b The New Component Graph	117
Figure 6.2 The New Template For EXAMPLE	118
Figure 6.3a Array Graph For Assertion 3	126
Figure 6.3b Underlying Graph For Assertion 3	126
Figure 6.4a Edges from MSCC to Single Node Component	129
Figure 6.4b Edges to MSCC from Single Node Component	130
Figure 6.4c Edges between MSCC and Single Node Component	131
Figure 7.1 The Generated Program Structure	145
Figure 7.2 Structure of the EXAMPLE Program	146
Figure 7.3a, 7.3b Transformation of a Generalized Tree to a Set of Restricted Trees	155
Figure 7.4 Block Description Entries for EXAMPLE	164
Figure C.1 Input Bit Reversal	192

CHAPTER 1

INTRODUCTION

I.1 MOTIVATION

The demand for increasing computational power continually outruns the increasing capabilities of sequential computers. This fact has long been recognized, and models for parallel computation have been developed to allow a problem to be decomposed and solved concurrently by more than one processing unit. However, decomposition of a problem into independent subunits communicating through well-defined protocols has been difficult to achieve in the normal sequential memory-processor paradigm of computation. Processes must communicate by changing portions of a common state. Synchronization, mutual exclusion, and variable sharing require significant effort on the part of the programmer. Parallel programs using such directives as FORK, JOIN, and mutual exclusion variables are more complex than sequential programs, and more difficult to verify correct.

An alternate model of computation has been proposed in which parallelism is expressed automatically at the expression as well as block and task level. In this model, the data flow model, a computation has the form of a directed graph rather than the traditional linear sequence of instructions. A node in the graph represents a unit of computation; an arc represents the flow of data into and out from the units of computation. A node may "fire," or be executed, whenever data is available on the input arcs, and as a result, produce data on the output arcs. This firing process is determinate. It depends only on data availability. In the data flow model, there is no concept of a global, shared state whose update must be synchronized through complex directives.

A class of programming languages has been defined for the data flow computation model. These value (rather than address) oriented data flow languages are characterized by properties of single assignment and referential transparency. Single assignment means that a variable may be assigned a value only once, that is, that a variable may "appear on the left hand side of an assignment only once within the area of the program in which it is active" [Acke82]. This is because a variable in a data flow language corresponds to an arc in the data flow graph rather than to a memory location. An arc can have one and only one name. Referential transparency means that a computation has no context

or environment which influences the result. A given set of inputs to the computation always result in the same output. These characteristics give data flow languages a functional semantics.

Nonprocedural languages, although not defined specifically for data flow, seem particularly appropriate as data flow languages [Acke79, Ashc79]. First, they satisfy the properties of single assignment and referential transparency which data flow languages have. In addition, they support a very high level, concise form of problem description. In Model, for example, both recurrence relations and familiar array operations can be expressed using subscripted arrays. In contrast, these tasks require loop construction in conventional high level languages and in data flow languages. Thus, with a nonprocedural language, more of the problem can be expressed in the problem domain rather than in the programming domain. The specification is more self-documenting than a conventional program; can be written more quickly than a conventional program; and can be more easily changed than a conventional program- all these benefits because the problem rather than the implementation is described. Another advantage to a nonprocedural language such as Model is in the area of data memory usage. The user can describe data structures in a form most convenient to the problem. The translator can then generate from that description a data structure most efficient to the implementation.

To illustrate the desirability of using a nonprocedural language rather than a lower level language to specify a problem , consider the nonprocedural specification of Figure 1.1, a matrix multiply program. The longest part of the program is the data declaration. There are only two assertions. The first assertion computes a three dimensional array X, obtained by multiplying the appropriate rows and columns of the input matrices A and B. The second assertion uses a reduction function SUM to add together elements of the innermost dimension of X and produce the matrix product C.

```
MODULE: MM;

SOURCE: INFILE1, INFILE2; /* input files */
TARGET: OUTFILE; /* output file */

INFILE1 IS FILE (INREC1); /* Matrix A */
INREC1 IS RECORD (IN1(10));
IN1 IS GROUP (A(10));
A IS FIELD (NUMERIC);

INFILE2 IS FILE (INREC2); /* Matrix B */
INREC2 IS RECORD (IN2(10));
IN2 IS GROUP (B(10));
B IS FIELD (NUMERIC);

OUTFILE IS FILE (OUTREC); /* Matrix C */
OUTREC IS RECORD (OUT1(10));
OUT1 IS GROUP (C(10));
C IS FIELD (NUMERIC);

X IS FIELD (NUMERIC); /* temporary */

I IS SUBSCRIPT (10);
J IS SUBSCRIPT (10);
K IS SUBSCRIPT (10);

X(I,J,K) = A(I,K) * B(K,J); /* Assertion 1 */
C(I,J) = SUM(X(I,J,K),K); /* Assertion 2 */

END MM;
```

Figure 1.1 Matrix Multiply in Model

In contrast, the corresponding program in the lower level Id language contains two procedures and a three level nested loop.

The call

```
mmt(a, transpose(b, m, n), l, m, n)
```

returns the product of the l by m matrix a and the m by n matrix

b. The example, taken from [Cost80], is shown in Figure 1.2.

```
procedure transpose(b,m,n)
  (initial trans<-LAMBDA
   for i from 1 to n do
     new trans<-append(trans,i,(
       initial row<-LAMBDA
       for j from 1 to m do
         new row<-append(row,j,b[j,i])
       return row))
   return trans);

procedure mmt(a, bt, l, m, n)
  (initial c<-LAMBDA
   for i from 1 to l do
     rowa<-a[i];
     new c<-append(c,i,(
       initial rowc<-LAMBDA
       for j from 1 to n do
         colb<-bt[j];
         new rowc<-append(rowc,j,(
           initial innerprod<-0
           for k from 1 to m do
             new innerprod<-innerprod+rowa[k]*colb[k]
           return innerprod))
       return rowc))
   return c)
  mmt(a, transpose(b,m,n),l,m,n);
```

Figure 1.2 Matrix Multiply in Id

The Model specification is concerned with the problem domain rather than the implementation domain. In contrast, one must understand clearly the semantics of the data flow machine to be able to program in the lower level language.

I.2 CONTRIBUTIONS

The major contribution of this research is the development of a system which translates a program specification in a very high level nonprocedural language to a lower level data flow language.

In the process of specification analysis and translation, a problem description goes through the following transformations:

1. The problem is defined in the Model language to form a specification.
2. The specification is analyzed by the Model Processor, and a form of data dependency graph is created, the array graph.
3. The array graph is processed to yield a data flow program template.
4. The template is translated into the MaD data flow language.
5. The MaD program is compiled to produce machine code.
6. The code is run on the data flow machine to produce the problem solution.

These transformations are outlined in Figure 1.3.

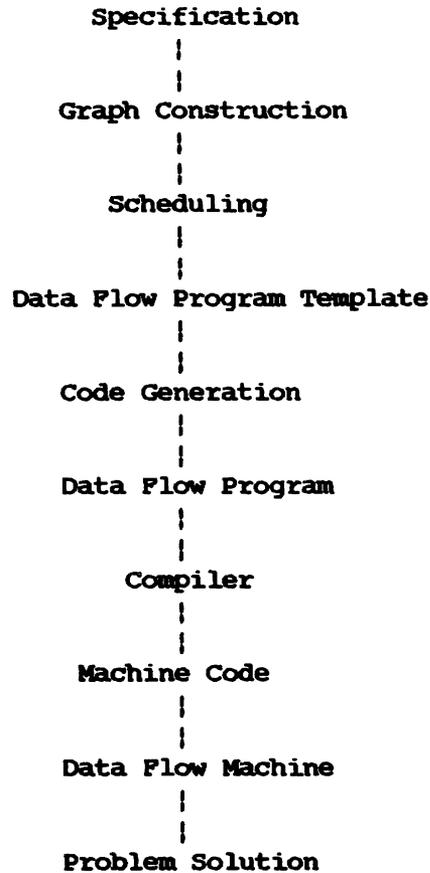


Figure 1.3 From Problem Specification to Solution

The nonprocedural language Model has been used as the source language for translation. Parts of the existing Model System, which generates a PL/I program from the specification, have been used. These programs create the data dependency graph and perform semantic checking. The graph is input to the data flow subsystem, which partitions the graph to produce data flow program units. The program units are called blocks. Two types of blocks are distinguished— iterative and parallel. Blocks are generated as a

result of analyzing the Maximally Strongly Connected Components (MSCC's) of the graph. A block may be generated either from a single MSCC or from a larger component created by merging MSCC's. The process of generating iterative and parallel blocks from a nonprocedural specification is called scheduling. The result of scheduling is a language-independent intermediate form of program, the data flow template. Using practical experience with the Manchester machine, as well as a comparative analysis of several other data flow machines, criteria by which to merge components to produce an efficient template have been developed.

This research studies in detail one data flow implementation, the Manchester machine, and a data flow language developed for the machine, MaD. Program templates generated from a Model specification are translated to the MaD Programming Language.

I.3 ORGANIZATION OF THE DISSERTATION

Chapter 2 reviews data flow, from the conceptual model to various architectural implementations, and explores problem areas in data flow. In addition, it describes the architecture of the Manchester University data flow machine. Chapter 3 discusses languages for data flow computers. It focuses in particular on the MaD (Manchester Data flow) language and on the Model Specification Language. In Chapter 4, Model internals are discussed: the array graph, subscript processing, and the notion

of range. Chapters 5 and 6 describe the scheduling and storage allocation algorithms. Translation of the data flow template to MaD is done in the code generation phase of processing, the topic of Chapter 7. Chapter 8 summarizes and suggests further areas of research in nonprocedural languages for data flow. Appendix A contains a description of the MaD Programming Language in BNF form, and Appendix B, a description of Model. Appendix C contains several examples in which Model specifications are translated to MaD.

CHAPTER II

DATA FLOW

In this chapter, the conceptual framework and some machine realizations of data flow architecture are presented. In the first section, the data flow model is contrasted with the conventional sequential model. Next, several architectural implementations of the model are reviewed. The next section discusses problem areas for data flow implementations. Finally, the Manchester data flow machine is described.

II.1 THE DATA FLOW COMPUTATION MODEL

Data flow is a model of computation which represents an algorithm as a directed graph showing data dependencies. In the graph, each node represents an instruction. Tokens (the term used for data) travel on the directed arcs from the node which produced them to nodes

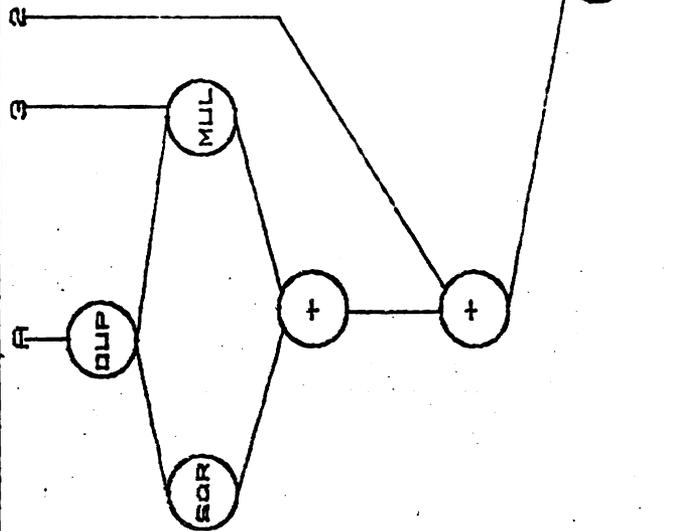
which consume them. In contrast, an algorithm expressed in the conventional model is represented as a sequential list of instructions. To illustrate the difference between data flow and conventional models, Figure 2.1 shows an arbitrary computation. In a conventional sequential machine, the linear sequence of instructions shown in Figure 2.2 compute the expressions. A data flow machine however, would execute the graph of Figure 2.3.

$$X = A^2 + 3A + 2$$

$$Y = B + 2$$

$$Z = X - Y$$

FIGURE 2.1 AN ARBITRARY COMPUTATION



LOAD B
 ADD 2
 STORE Y
 LOAD A
 MUL A
 STORE TEMP1
 LOAD A
 MUL 3
 ADD TEMP1
 ADD 2
 STORE X
 SUB Y
 STORE Z

FIGURE 2.2 A LINEAR SEQUENCE OF INSTRUCTIONS

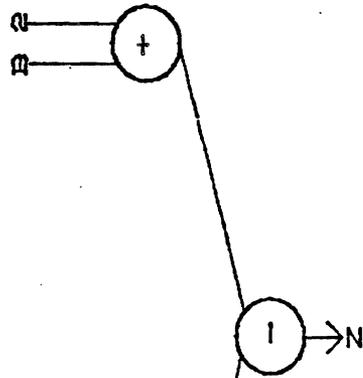


FIGURE 2.3 DATA FLOW GRAPH OF THE COMPUTATION

II.1.1 Properties Of The Data Flow Model

The example illustrates three desirable properties of the data flow model. First, the data flow graph shows parallelism of the computation at all levels, from machine op level up. Second, under the data flow model, there is no concept of memory address. And third, the graph displays referential transparency.

II.1.1.1 Parallelism -

Parallelism of an algorithm is exposed because the graph is a partial order rather than the total order imposed by a sequential machine. Sequential machine instructions are executed in a fixed order according to a single program counter. Possible parallelism may be inferred to a limited extent from evaluation of the algorithm. However since memory cells can hold different values at different times, the potential for detection of parallelism is reduced. If the programmer reuses a variable for more than one purpose, computation involving the second usage, even if it is independent of the first, must occur after the first.

With the data flow model, asynchrony and parallelism are implicit. Each computation is constrained only by the availability of its input data. The programmer does not have to specify concurrent operation of procedures. "Co-begin", "fork", or "activate task" constructs are not necessary. The data dependency graph representation of the program exposes all possible parallelism.

II.1.1.2 Lack Of Address -

In the data flow model, data is described in terms of values rather than addresses. Inherent in the conventional model is the idea of an address [Arvi78]. The address of a memory cell is invariant, while the value stored in the cell changes with time. In the data flow model, there is no address. (We will see later that implementations of the model do use address rather than data to transmit structured values). Values, or tokens, produced by processors travel on the arcs to other processors. Data values input to a node are applied to the computation represented by the node. Output values are produced. These values are, in turn, input to other nodes. For convenience, the tokens are usually named. Since it would be ambiguous for one name to refer to more than one value, data flow programs usually follow the "single assignment rule". One name can denote only one value. Multiple assignment such as in the statement

$$I := I + 1$$

is not allowed in the same fashion as in conventional programs.

II.1.1.3 Referential Transparency -

The third property of data flow is referential transparency. This property implies that there is no environment or context or side effect to influence the result of a computation. In the conventional model, the change in value of a memory cell might depend on some complex interrelationship of the current values of other memory cells.

Values of cells change in time, so that when the interrelationship is tested is as important as the interrelationship itself.

Under data flow, however, any parameter which is needed by the node must be input to the node. The input values uniquely determine the results produced. A given set of values input to a processor produce the same results regardless of when the computation occurs, so that the node can be characterized as a mathematical function. The graph, therefore, displays a locality of effect in which "the mathematical equations for a data flow program can be derived simply by conjoining the equations for the various parts of the program in an 'additive' manner" [Kosi79].

These properties of data flow graphs make the semantics of data flow programs tractable to formal description. Being able to describe precisely the "meaning" of language statements is helpful in many respects. It is then possible to have a standard against which to test compilers for compliance to a language specification. Having a mathematically precise set of axioms defining a language makes it possible to attempt to prove theorems about the behavior of programs. The relative ease with which denotational semantics has been developed for data flow languages has encouraged research into suitable architectural implementations for data flow.

II.2 ORGANIZATION OF DATA FLOW MACHINES

Data flow machines have been built or proposed by various groups. Texas Instruments, University of Utah, the CERT Laboratory in Toulouse, and University of Manchester have all built prototype systems. Dennis and Arvind at MIT were responsible for much of the ground work in data flow. Each is building a prototype machine. In this section we will develop concepts common to many of the implementations.

II.2.1 The Data Flow Instruction

The node of the data flow graph is realized in the machine as an instruction. A data flow instruction differs somewhat from a conventional machine instruction. In addition to the op code, the data flow instruction must carry information about the arcs- that is, the inputs and outputs to the instruction. An instruction might look as follows:

Opcode

Number of Destinations

for each Destination:

node number of the Destination

input point into the Destination (which input arc?)

Number of Inputs

for each input:

input type

There can, of course, be many variations to this format. Space may be reserved with the instruction for input parameters to be collected. The Texas Instruments Distributed Data Processor (TI DDP) instruction, illustrated in Figure 2.4, follows this convention. There may be a limitation as to the number of input parameters and/or destination addresses. The Manchester data flow machine allows a maximum of two inputs and two destinations.

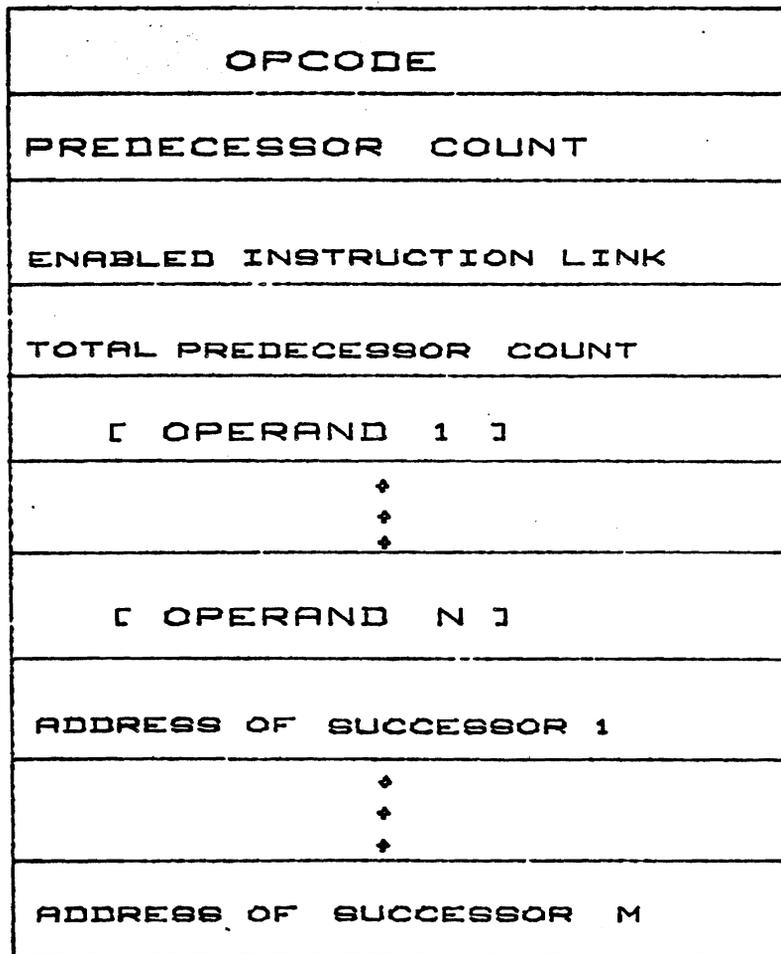


FIGURE 2.4 TI DDP INSTRUCTION

II.2.2 Scheduling Of Instructions

The data flow machine must have some way to deliver tokens to instructions and to recognize when an instruction is ready to "fire" (or execute). In the TI DDP, executable instructions are removed from a Pending Instruction List, a queue of instructions in memory which are ready to fire, whenever a processing element is available. The "enabled instruction link" in each instruction is used chain together the entries in the queue. Output from the arithmetic unit goes to an update controller, which delivers the result to all instructions which need that result as input. Depending on the destination address, the result may go either to the local memory or onto the bus to another processing unit. When all results have been delivered to an instruction, the instruction is put on the Pending Instruction List. The configuration of one processing unit is shown in Figure 2.5.

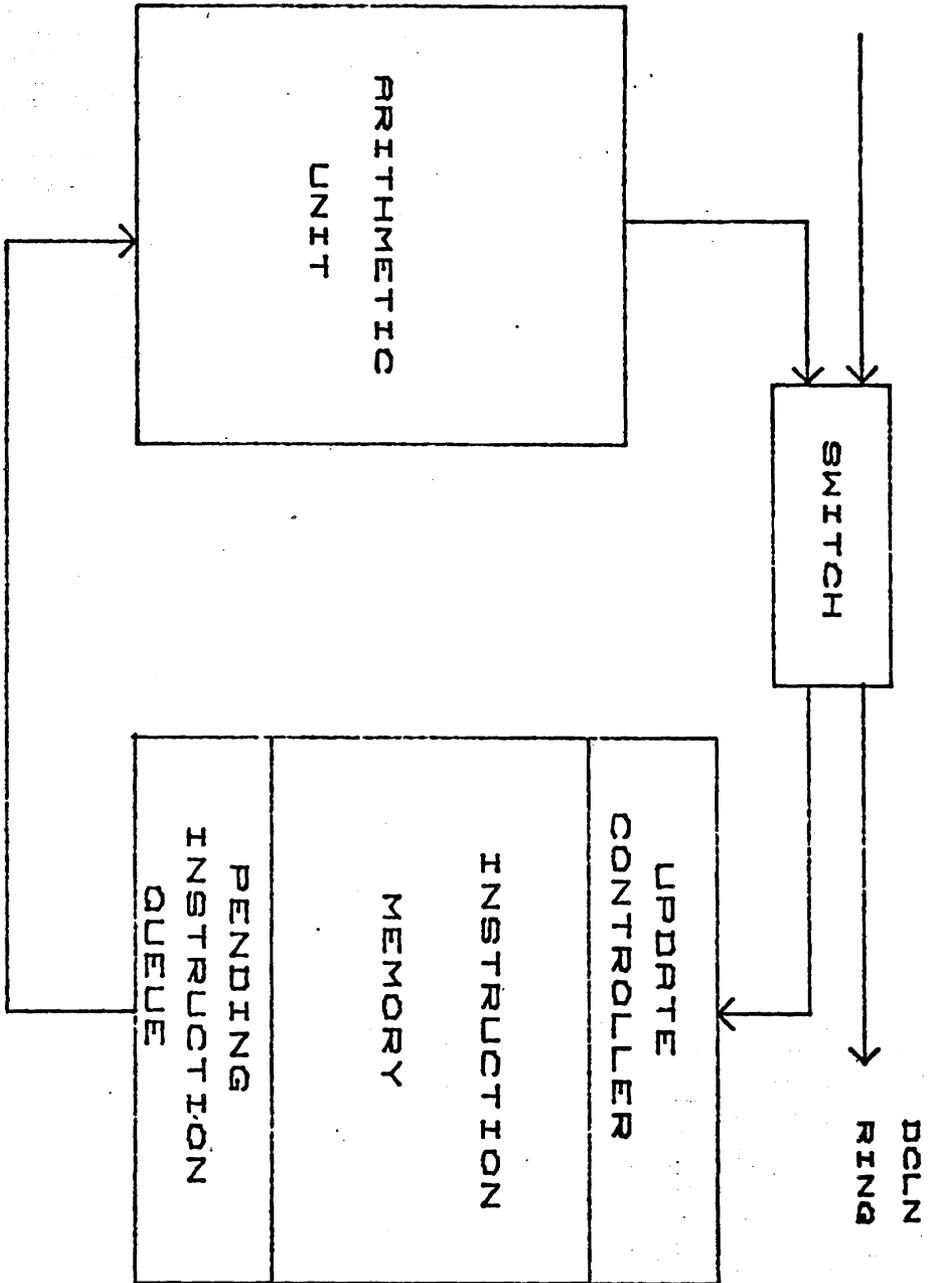


FIGURE 2.5 PROCESSING UNIT OF THE TI DDP

In the LAU System, which was constructed at the CERT Laboratory in Toulouse, instruction memory has control bits associated with each instruction. A processor continually scans the instruction memory for enabled instructions.

In the Manchester machine and in Arvind's machine, there is a Matching Unit. This unit assembles the parameters to an instruction. When all parameters have arrived, a "group packet" consisting of opcode, input parameters, and destination address is sent to a processing element in the Processing Unit. A block diagram of the Manchester Data Flow Machine is shown in Figure 2.6. A diagram of Arvind's machine is shown in Figure 2.7. The Token Queue in the Manchester machine corresponds to the Input Section of Arvind's machine. The Matching Store corresponds to the Waiting-Matching Section. The Node Store corresponds to the Instruction Fetch Section and Program Memory. The Processing Unit corresponds to the Service Section. The Switch corresponds to the Output Section. In the Manchester machine the function of Arvind's Data Structure Memory is performed in the Matching Store.

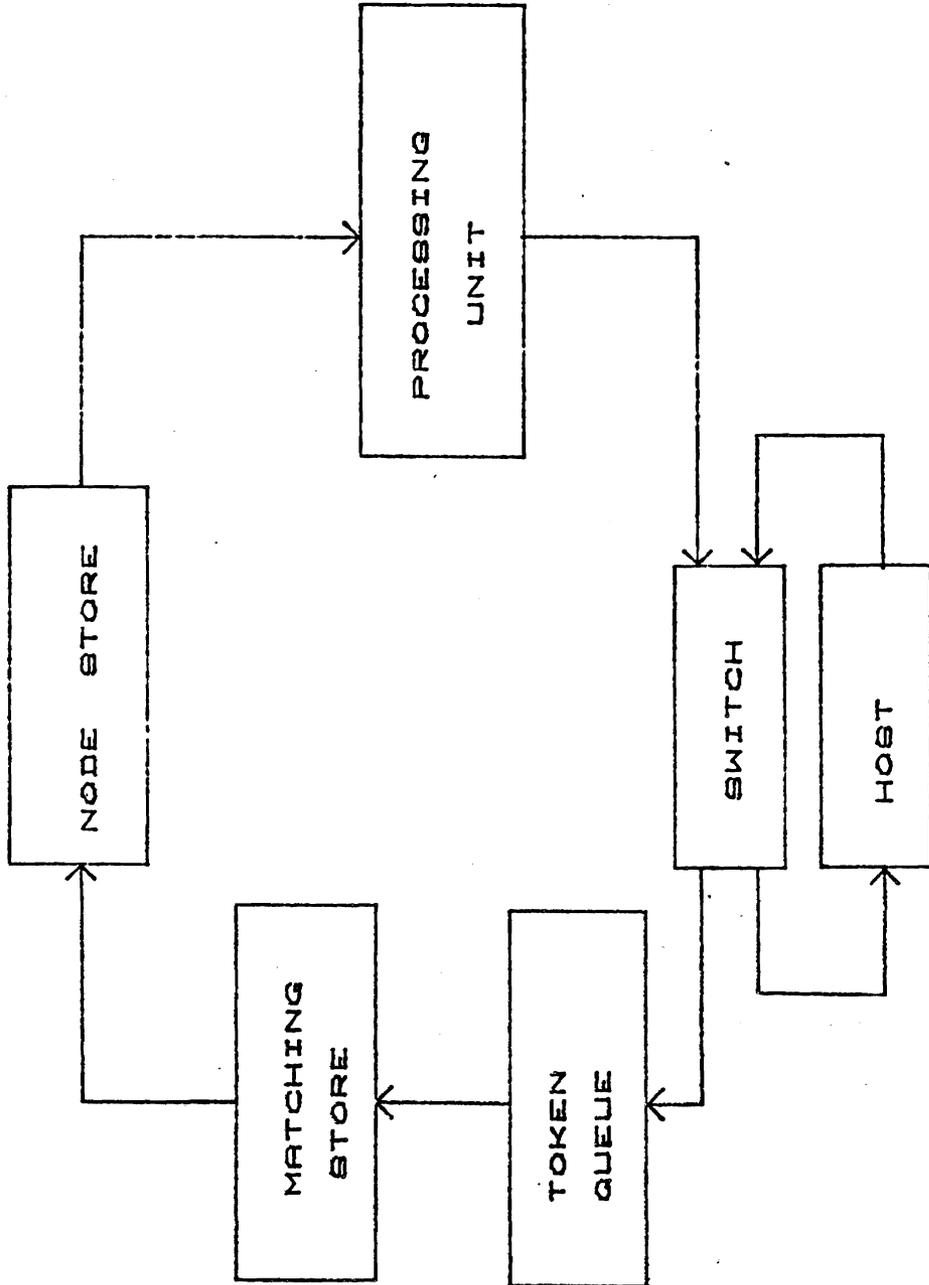


FIGURE 2.8 MANCHESTER DATA FLOW MACHINE

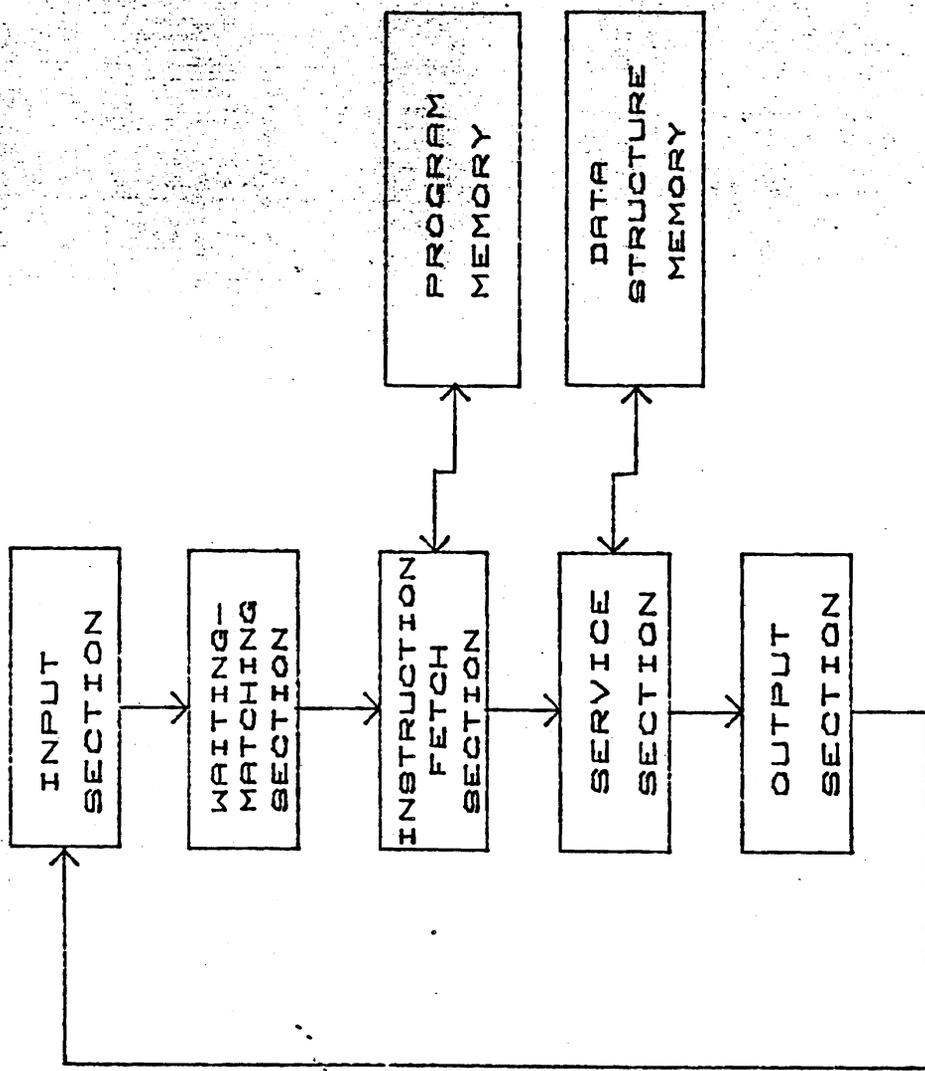


FIGURE 2.7 MIT CARVINDS DATAFLOW MACHINE

II.2.3 Processor Network Topology

It is necessary to provide a path of communication between processors of a data flow machine. Many architectures specify a two-level hierarchy. An individual processor usually consists of a local instruction memory, a local token memory, and one or more processing elements. The higher level connects the processors. The TI DDP may have up to four processors on a common bus, called the DCLN ring. Data produced at one processor and used by another processor is routed through a switch at the sending processor (see Figure 2.5). The data then travels along the DCLN ring to the destination processor, and is routed through the switch at the destination processor into its local memory. Arvind is designing a machine having 64 processors in the network. Data produced by one processor and used by another must travel through the network. Data produced by a processor and used within the same processor may short-circuit the network. In these network topologies, each processing unit can address any other one directly. The TI and Arvind network topologies are illustrated in Figure 2.8 and 2.9 respectively.

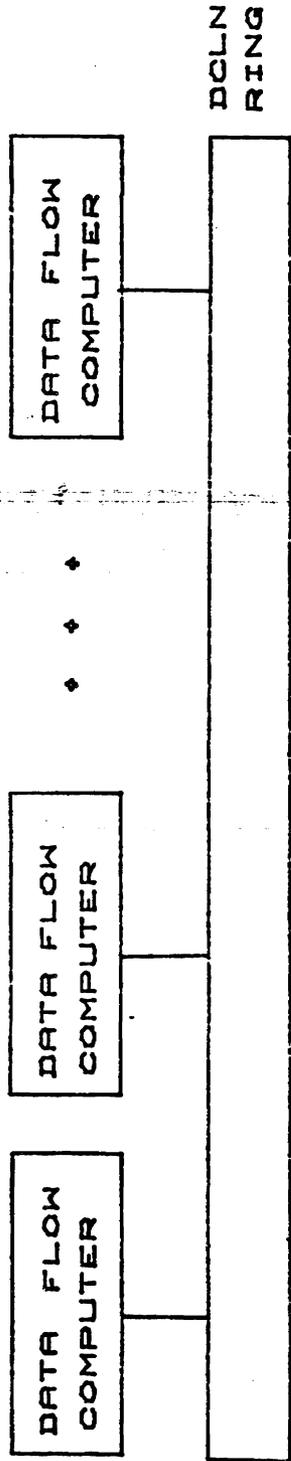


FIGURE 2.8 TI NETWORK TOPOLOGY

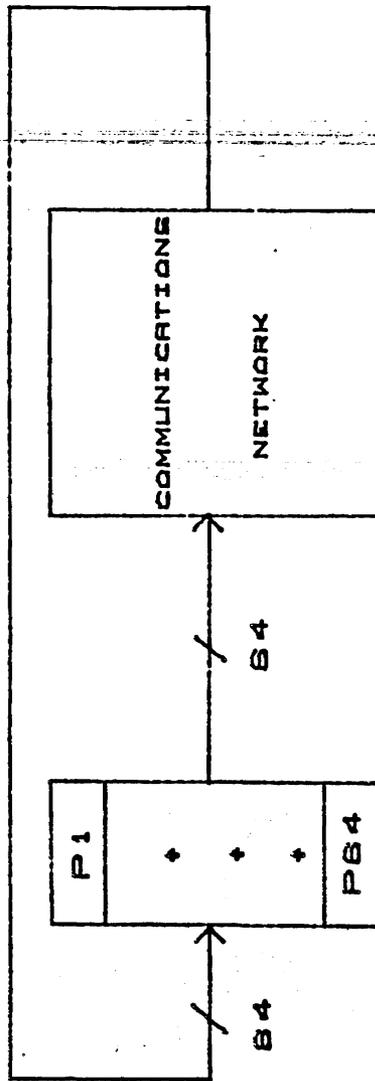


FIGURE 2.8 MIT (ARVIND) NETWORK TOPOLOGY

In contrast, DDML, the University of Utah data flow machine, is configured as a tree. Each processing unit is either atomic or is the root of a subtree of up to eight others. Thus a DDM consists of a rooted tree of breadth up to eight at each level but arbitrary depth [Davi79]. A processor may communicate directly only with its parent and children. All other communication must go through one or more intermediate link. The DDM tree is shown in Figure 2.10.

II.3 PROBLEM AREAS IN DATA FLOW DESIGN

There are several problem areas in designing a data flow machine. We will discuss three topics of special relevance to the task of scheduling a nonprocedural specification for data flow.

Some questions of concern are:

First, can more than one token be on an arc at one time? Figure 2.11 shows an instruction X with two inputs. On the left hand input, there is one token waiting. On the right hand input, there are two tokens. How does the machine recognize that input A1 belongs with input B1 to instruction X rather than with input B2?

Second, how is a program graph to be partitioned among processors in a data flow machine? What properties of the graph should be considered in generating such a partition?

Third, how is structured data handled on the data flow machine? What data type can a token be— can an array be considered one token, or is a token constrained to be of elementary type such as integer word? How does the machine handle structured data such as an array?

We will examine these questions and find how various data flow machine designs handle the problems.

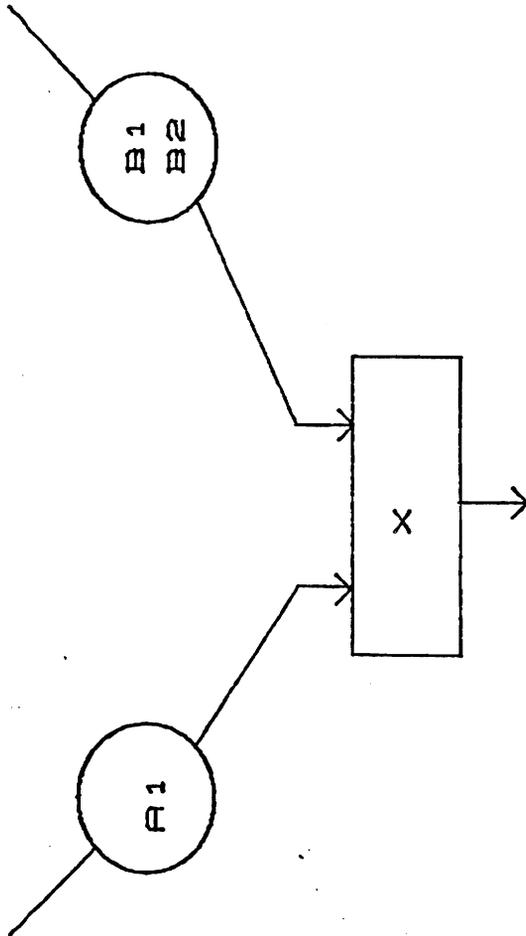
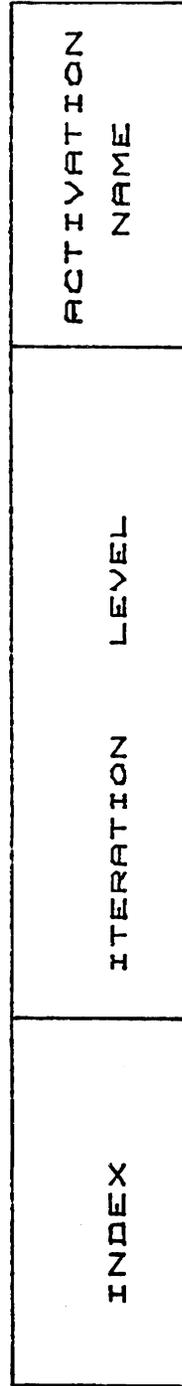


FIGURE 2.11 MULTIPLE INPUT SETS TO AN INSTRUCTION



0-20 BITS 0-32 BITS 0-20 BITS

FIGURE 2.12 MANCHESTER TOKEN LABEL

II.3.1 Reentrancy In Graph

The first question concerns reentrancy of graph. If an instruction in the graph can have more than one set of input existing at one time, the graph is reentrant. When such a condition exists (several sets of tokens input to an instruction), there must be some way of keeping the token sets distinct. If the implementation does not support reentrancy, then there may be only one set of tokens active at one time. Only when that set has been processed may another set be input to the node. The TI DDP and LAU follow this strategy. In the TI DDP, input values to a node are stored with the node. When all required inputs have arrived, the instruction (node) is linked into an enabled-instruction list to be executed as processor availability permits. Because of this implementation, the graph is not inherently reentrant. If a portion of the program needs to be reentrant, the subgraph corresponding to that program fragment must be duplicated so that distinct input sets may be allocated unique storage areas.

On the LAU system, the compiler recognizes an EXPAND directive which duplicates a subgraph a programmer-specified number of times. This is required in order to allow parallel computation of array elements, for example. If the number of duplicate copies needed is not known at compile time, the computation must proceed iteratively rather than in parallel.

The Manchester machine and Arvind's machine, which support reentrancy, use token labeling to keep sets of tokens distinct. Tokens which belong to the same input set to a node have the same label.

Figure 2.12 shows the label format for the Manchester machine. A label consists of an index and a color. The index is used to distinguish elements of an array when multiple array elements might be on the same input arc to a node. The color portion of the label permits reuse of the graph. It is divided into an activation name, used to distinguish concurrent calls to a function, and an iteration level, used to distinguish concurrent iteration instances. There is flexibility in the size of each field. The index may use 0-20 bits, activation name 0-32 bits, and iteration label 0-20 bits.

The Manchester machine design incorporates opcodes to manipulate labels. The Yield opcodes accept an input of any type and return the label or individual label fields. The Extract opcodes accept a token of type label as input and extract fields from the label. The Set opcodes accept label or label field tokens as input and produce new label tokens.

The data flow template produced by the Model Processor has been applied to the Manchester System, which supports reentrancy. However, in order to be useful for non-reentrant data flow machines, sufficient information is stored in the data flow template to allow construction of an EXPAND-like directive.

II.3.2 Partitioning The Program Among Processors

As we have seen from Section II.2.3, a data flow machine consists of one or more processors communicating through an interconnect. A processor has only local storage for data flow programs and data tokens. There is no common memory for instructions or for data. If data produced by a processor is needed by an instruction in another processor, that data must be transmitted along the communication path to the other processor. In the TI DDP, the communication path consists of a common bus. In Arvind's data flow machine, there is an interconnection network to join the processors in the machine. If there is no direct path from one processor to the second, the data has to be routed through one or more intermediate links. The Utah DDM interconnect has this property.

To minimize the cost of data transmission between processors in the machine, it is desirable to follow the principle of locality of reference. A partition of the graph allocated to one processor should contain a related set of instructions. The instructions are related in the sense that data produced by one instruction in the program subgraph is used by other instructions resident in the same processor. Such a partition minimizes the number of tokens which must be transmitted over the interconnect to other processors in the machine. The scheduler, therefore, attempts to partition the data flow program into blocks of related instructions in which each block is an independent unit of allocation.

II.3.3 Data Structures

The third question to be considered is that of handling data structures. What mechanism does the machine use to access structured data such as arrays and records? Arvind's proposed machine requires a "structure controller" to access structured data. The Manchester machine has special opcodes added on to the basic machine to handle structures. Since structured data and the array data type is very important to the Model Specification Language, understanding how structures are implemented on data flow machines is important in translating Model structures to data flow.

The use of structured data poses difficulties to a "pure" data flow machine. Accessing structured data on a conventional sequential machine is, by contrast, much simpler. The base sequential machine has a linearly addressed memory (or a hierarchy thereof). Structured data defined through high level programming languages is mapped to the linear memory. A compiler usually generates code to calculate the array offsets. If an offset is known at compile time, the composite program-relative address can be used directly. In either case, after the initial address calculation, data in a structure is accessed and stored just as if it were simple data.

On a data flow machine, accessing structured data is more complicated. By way of example, consider the reference to an array element $A(5)$ in a computation. In the data flow graph, $A(5)$ is input to the computation. However, does this mean that only one element is

input to the node, or does it mean that the entire array is input? If the former is inferred, then where is the rest of the array A? There is no memory in the conventional sense in a "pure" data flow machine. If the entire array is input to the node, serious efficiency considerations appear. A token, whether structure or simple field, must be duplicated and sent to every node which needs it. Each time an array element or one field of a record is changed, according to the strict semantics of data flow, the entire structure must be duplicated, and a new structure, with a new name, created. Even when this is done conceptually rather than literally, an implementation needs additional functional units and/or machine opcodes to support structured data. The following section describes some ways of handling structures in a data flow environment.

II.3.4 Structure Processing On Some Data Flow Machines

One approach which has been taken in some machine designs (Arvind, DDM) is to attach another processor to the data flow unit, a structure controller, to handle accessing and storing structured data. The Utah DDM uses an intelligent memory at each Processing Unit to handle all data, structured or simple. The Atomic Storage Unit (ASU) provides a "location independent method for dealing with an arbitrary structure of variable length fields" [Davi79]. The elementary item of storage is the field. A field is either a variable number of characters delimited by two reserved characters (left and right

parentheses) or else a sequence of any number of fields enclosed in parentheses. A parenthesized field, therefore, represents a data structure. It is equivalent to a generalized tree. A field which has subfields is called a file. The first subfield of the file serves as a descriptor to the rest of the file. The non-descriptor fields of the file may be ordered or unordered. Unordered fields must be accessed by name. Ordered fields may be accessed by name or by position. The ASU can execute such commands as inserting or deleting fields and positioning to a certain field. Since fields can be modified at this primitive level, the higher levels must ensure that the modified structures are referred to by new names in the programs using the fields. The ASU also manages all memory allocation and reclamation.

Dennis has proposed a Structure Controller to handle structured data for his machine. Like the Utah ASU, the Structure Controller accepts read and write requests from the data flow processor, and manages the dynamic memory allocation and reclamation. The Dennis Controller realizes structures as acyclic binary trees. A node is either an elementary value or a pointer to other nodes. A node is addressed by its selector, which is its position in the tree. The two basic operations on a structure are SELECT and APPEND. SELECT, when given a structure name and selector, returns the value at that selector. APPEND, given a structure, selector, and value, returns a new structure identical to the original one except that the new value

has been inserted at the indicated selector. In order to increase efficiency, structures can be shared. Each node has a reference count, which is the total number of pointers to that node. A SELECT operation to a node causes the node's reference count to be increased. When an APPEND occurs at a selector, then the node which is being replaced must have its reference count decremented. When an APPEND occurs and the reference count is greater than one, new pointers must be created to the unchanged portion of the structure. When the reference count reaches zero, a cell may be reclaimed [Acke78].

A variation to this form of structure controller is Arvind's I-structure controller. An I-structure is an array-like data structure. The storage for the structure is allocated before each element of the structure is defined. In order to support the unordered nature of structure construction, each element has a presence bit associated with it. If an undefined element is referenced, the read is deferred. Once the element is defined, all deferred reads must be honored [Arvi80].

The benefit of having a separate structure controller is in eliminating the overhead of creating and propagating large, complex, and unwieldy structures in the same manner as simple tokens. The drawbacks are in the introduction of sequential access and update (for links and reference count) and the introduction of a memory access bottleneck just as in sequential machines [Gajs82].

In contrast to the separate structure controller, the Manchester machine has special opcodes executed by the data flow machine itself to support structured data. These opcodes manage a data type called stream. This data type will be discussed in further detail in the next chapter. In this section, a stream can be considered equivalent to a file or array.

In the Manchester machine, the iteration level field of the token label is used to differentiate elements of a stream. Examples of stream ops include opcodes to generate and check the iteration level field of the token label; to separate a stream into FIRST and REST; to check for end-of-stream; and to add an element to the head of a stream. The opcodes which set and update the token's iteration level perform much the same function as instructions in a conventional machine to compute the address of an array element. The difference is in what happens to this label or address. On a sequential machine, the address is sent to the memory to access the data. On a data flow machine, the label is attached to the token. The token then is matched with its partner with the same label, and the two are delivered to the appropriate instruction.

In addition, the Manchester data flow machine can store an entire stream in a Storage Node. Elements of the stored stream are accessed in a demand-driven fashion. Operations on stored streams include maintenance of reference count, getting a pointer to the stream, fetching stream elements, and garbage collection. Besides these extra

operations, there are matching store functions to support stored streams. These functions include Increment, to increment a token of type ordinal, which is used in a token label; Decrement; Preserve, to make a copy of a token but leave it in the matching store; and Defer, to defer storing a token in the matching store in case of collision of labels [Kirk82]. The advantage of using the stored stream rather than the simple stream is that a parameter to function or iteration which is of type array may be passed through a single token (the address of the array). The disadvantage is that the system may fill up with a large number of unconsumed tokens [Gurd81]. This might occur because the Defer matching function is used to keep stored stream tokens circulating on the ring.

II.3.5 Implications Of The Structured Data Problem

Regardless of the implementation strategy, creating and accessing structured data can be a major source of bottleneck in a data flow machine. Using a Structure Controller to store and retrieve tokens forces sequential operation. It is necessary to read and update pointers and reference counts sequentially in order to guarantee their validity. With the I-structures, Gajski points out, it is difficult to know ahead of time the optimal memory allocation scheme to partition large arrays. Memory contention problems may occur for frequently accessed elements stored in the same memory module. Gajski observes that these are the same problems which affect vector machines

and other multiprocessors [Gajs82]. Incorporating stream handling into the data flow processor can result in a large number of tokens circulating through the machine.

In generating a data flow template, therefore, it is beneficial to simplify the structure of data used in the specification. One way to do this is to recognize cases in which data of type array can be reduced in dimension in the generated program. When a two-dimensional array can be reduced to a one-dimensional array or to a scalar, overhead in creating and accessing the array is reduced. The Model processor attempts to partition the program graph so that array dimensions can be minimized whenever this is consistent with maintaining parallelism inherent in the problem. Doing so reduces the complexity of the data structure which the data flow machine must handle.

II.4 THE MANCHESTER DATA FLOW MACHINE

This section describes the Manchester University data flow machine. A prototype of the machine has been operational since 1981, and a second unit is under construction. An emulator for the Manchester machine has been used to run data flow programs generated by the Model system.

II.4.1 Machine Layout

The Manchester machine consists of five functional units connected in a pipelined ring. The machine architecture is illustrated in Figure 2.6. Input to and output from the ring are controlled by an LSI-11, the host. Tokens travel on the ring in the following manner:

1. Tokens enter the system from the LSI-11 through the Switch. The Switch is the machine interface to the LSI-11. The Switch can receive tokens either from the Processing Unit or the LSI-11. It can route tokens either to the Token Queue or the LSI-11.
2. After entering the system through the Switch, tokens are stored in the Token Queue. The Queue can provide temporary storage for up to 16K 96-bit tokens.
3. Next on the ring is the Matching Unit. This Unit gathers pairs of tokens with the same destination node address and label. The Unit operates associatively. When a token arrives at the Unit, there is an associative search for a partner. One control field of a token is the matching store function. This function specifies what the Matching Unit should do both in the case that a partner can be found and in the case that a partner cannot be found. The most common matching store functions are:

- Bypass (BY). This means that the token does not have a partner. The instruction only needs one token, or else the other token is a literal carried with the instruction. The token bypasses the Matching Unit.

- Extract-Wait (EW). This is the normal action. If the partner is found, the partner is extracted from the store and joined with the incoming token. The two form a complete token set. If the partner is not found, the token is stored in the Matching Unit to await the partner.

Variations of these common actions have been added as matching store functions so that the machine has primitives with which to implement resource managers and to handle stored arrays. The latter functions have been described above.

4. The token or token pair leaving the Matching Unit next addresses the Node Store. This is the unit in which the data flow instructions are stored. Each instruction consists of an opcode and destination address(es). This information is added to the token(s) to produce a group packet.

5. The group packet goes to the processing unit for execution.

6. The result goes to the Switch. Depending on the address, it may

either exit the machine or recirculate to the Token Queue.

II.4.2 Data And Instruction Formats

A token on the Manchester machine consists of a 96-bit word. This word is divided into a 32-bit data field, a 36-bit label, and 18-bit destination address, and 10 bits for type information and control. The destination is subdivided into a node address, the input point (left or right), and the matching function. The label subfields are discussed in Section II.3.1. Every token and literal has a data type. The standard types are Integer, positive integer, real, character, and boolean. The ordinal, activation name, and label data types are used in labels. Other types include types to create dynamic arcs (for function return, for example), stream numbers, end-of-stream, trigger, the set data type, error, and lambda-i, which is a number in the range 0 to $2^{24}-1$ indicating one of 2^{24} different user-defined types.

An instruction consists of an opcode, and a maximum of two destinations. One of the destination fields may be used for a literal input to the instruction. A maximum of 71 bits is used for an instruction. There is a 12-bit opcode, 18 bits are available for one destination, 32 bits are available for the second destination or for a literal, and 9 bits are used for control.

The instruction set is divided into various sets of operators. The ordinary arithmetic operators constitute the largest set. The flow control operators include the branch operators. These double result operators send the left input to the left or right output depending on a condition involving the right input. There are operators to manipulate label and destination fields, operators to control dynamic arcs, data structure handling operators, and input/output operators. The label-changing and data structure operators have been discussed above. Dynamic arc control operators are used for function call/return sequences. Input/output operators provide a communication path to the host.

II.5 CONCLUSION

This concludes the survey of data flow machines. From the discussion above, it is clear that many different approaches have been taken to implementing a machine to interpret a data flow graph. The implementations must deal with issues far more complex than simple expression evaluation. Reentrancy of graph, program allocation to processors, and data structures are major problems for machine implementations. Both the proposed and prototyped implementations have trouble handling data structures.

The next chapter discusses languages for data flow machines, both existing languages and languages designed specifically for data flow machines.

CHAPTER III

LANGUAGES FOR DATA FLOW

The development of suitable languages to program data flow machines has been an area of active research. Languages suggested as appropriate range from conventional programming languages [TITE80] to new languages designed especially for data flow [Denn74] [Arvi78] [Acke79] [McGr80]. In this chapter, we will examine some of these programming languages. In addition to the high level procedural programming languages, we will consider nonprocedural languages as data flow languages. We conclude the chapter with a description of the Model Specification Language.

III.1 CONVENTIONAL PROGRAMMING LANGUAGES

Existing conventional programming languages have been used to program data flow machines. There are several advantages to using conventional languages. One advantage is that there is a considerable

body of programs already written in such languages as FORTRAN. Compiling FORTRAN to a data flow machine language makes those programs available immediately to the new machines. Another advantage is that there is a considerable body of programmers conversant in such conventional languages as FORTRAN. Proponents of the use of conventional languages for data flow argue that programmers will find it difficult to program with the unfamiliar syntax and functional semantics of the new data flow languages. In addition they claim that a sufficiently smart optimizing compiler can recover much of the parallelism from a sequential program [Gajs82].

The programming language for the TI DDP is FORTRAN. FORTRAN was chosen primarily for pragmatic reasons: TI has a large investment in supporting FORTRAN on the TI Advanced Scientific Computer (ASC) and a library of scientific application oriented benchmark programs developed for the ASC could be used on the DDP. The ASC FORTRAN compiler was modified to produce program graphs for the DDP [TITE80].

Although existing software and familiarity with it are powerful arguments for using conventional languages for data flow, there are also disadvantages. A conventional programming language such as FORTRAN mirrors almost exactly the processor-memory paradigm of sequential machines. It does not follow the functional semantics of the data flow machines. Computation in FORTRAN is not accomplished strictly by function application. The tasks of assigning values to variables, fetching stored values of variables, and updating values of

variables are essential parts of the language. Even the most sophisticated compiler cannot recover all the parallelism in the algorithm which has been disguised by the sequential nature of the programming language. In addition, the programmer is forced to (over)specify the algorithm in the sequential paradigm even if this is not the most natural way to describe the problem. The programmer must supply a linear sequence of instructions. Parallelism which might have existed in one form of problem description might be completely absent when the problem is translated to FORTRAN.

III.2 DATA FLOW LANGUAGES

III.2.1 Common Properties

Even though conventional languages will undoubtedly be used on data flow machines (for back compatibility, among other reasons), functional languages also have been designed for data flow. In this type of programming language, a computation is accomplished by function application, that is, by supplying parameters to a function and evaluating the function. The results produced by the function depend solely on the parameters supplied. The context or environment in which the function was invoked has no bearing on the results produced. There are two immediate benefits when programming is accomplished as function application. First, already mentioned, is the simplified semantics of the programming language. Another, is the

ability to provide in the language powerful combining forms for synthesizing functions into a program. Instead of rewriting the same basic algorithm in differing contexts, it becomes possible to isolate functional components of the algorithm, and then put those components together in different ways as the particular situation demands.

Languages such as Id, Val, the LAU programming language, and MaD (Manchester Data flow) have been defined for various data flow machines. These languages follow the functional semantics of data flow machines. The basic language form is the expression. All of the languages are single-assignment. In addition, most of them support two new constructs: the parallel loop FORALL as distinguished from the iterative loop ITER, and the STREAM data type.

III.2.2 Iteration

Just as WHILE and REPEAT loops in conventional languages allow controlled branching, the ITER construct in data flow languages allows controlled reassignment to variables [Acke82]. The ITER block is used to express recurrence relations, such as in computing factorial. The ITER block consists of three parts. First there is an initialize part to give iteration variables their initial values. Then there is the body of the iteration. Only in this section may a variable be reassigned a value. The iteration variables appear here as target of assignment. The new values of all the iteration variables are available only at the end of the block, so that within the block, the

single assignment rule is still enforced. The third part of the ITER is test for termination. An example of an ITER block is the factorial program [Acke82]. In Id, the program is written as follows:

```
(initial J := N; FACT := 1; /* initialization */
while J <> 0 do           /* termination condition */
  new J := J - 1;        /* controlled */
  new FACT := FACT * J;  /* reassignment */
return FACT)            /* value returned */
```

The program in MaD is almost identical to the Id version.

III.2.3 Parallel Constructs

The FORALL construct allows the programmer to construct a block which may have multiple independent incarnations. The scope of repetition may be a set of indices, as in VAL, or a stream of values, as in MaD. The ITER and FORALL represent two classes of blocks, iterative and parallel. There is a direct correspondance in the translation of the data flow template to a data flow language between a parallel or iterative block in the template and a parallel or iterative program block. The two kinds of blocks are the fundamental units which the Model data flow scheduler generates from the nonprocedural specification.

III.2.4 Streams

The stream was introduced earlier by analogy to a file or array. Here we define the data type more formally. A stream is defined to be a possibly infinite sequence of elements. The elements of a stream have a total linear ordering and are not required to exist simultaneously [Arvi78]. The stream data type is useful in handling I/O in data flow languages. A sequential file can be thought of as a stream of elements. The stream data type can also be used to implement arrays. A two-dimensional array of integers, for example, can be implemented as a stream of a stream of integers. If a stream is a parameter to a function, the function operates on the input stream as it is read from an input device and produces an output stream. Operations on streams include extracting the first element (CAR), the rest of the stream (CDR), and constructing a stream out of simple tokens.

III.3 THE MAD PROGRAMMING LANGUAGE

MaD is a high level data flow language designed at Manchester University [Bowe82]. It is considered an interim effort toward a data flow language for the Manchester machine. MaD is single assignment and function-oriented. The program unit is similar in form to a Pascal program. We will describe here a subset of the language which has been used in translating the Model data flow templates into MaD.

The complete BNF is included in Appendix A.

In the following BNF description, optional language elements are bracketted by "[" and "]". A "*" following the right bracket denotes zero or more repetitions, and a "+" following the right bracket denotes one or more repetitions. A "|" is used to separate options in the expansion of the left hand side. Any punctuation mark used in the language itself is quoted (for example, ';'). Syntactic elements are put in <...>. Syntactic elements ending in "id" usually denote an identifier.

III.3.1 The Program Heading

```
<programme> ::= PROGRAM <program-id>
               [ <parameterlist> ] <result> ';'
               [ <typedeclaration> ]
               [ <funcdefs> ]
               [ <expression> ]
               END
               [ <assembly-code> ]
```

The program heading has an optional parameter list. However, the <result> is required. There may follow type declarations and nested function declarations. The <expression> is the result returned by the program. Assembly language code may follow the END statement.

```
<parameterlist> ::= '(' <parmlist>
                  [ ';' <parmlist> ]* ')'
<parmlist> ::= <parmid> [ ',' <parmid> ]* ':'
              [ [STORED] STREAM [STREAM]* ]
```

```
      <typeid>
<result>      ::= '[' <result> [ ',' <result> ]* ']'
               | [ [STORED] STREAM] <typeid>
```

Example: PROGRAM FACTORIAL(N: INTEGER):STREAM INTEGER;

The <parameterlist> is a standard Pascal format parameter list. The <result> shows the data types of values returned by the program. The header may either be a composite structure or may refer to a predefined type. The stream and stored stream data types are described in Section II.3.4.

III.3.2 Structured Types In MaD

Structured types as defined in MaD are slightly different from those defined by Pascal. The stream data type is used instead of array or file. Records cannot be declared as in Pascal. Structures can be defined; however, fields cannot be named individually. An expression consisting of a composite structure can be created as an "informal" record. For example, the expression [5.2, I*5] is a composite structure consisting of a real and an integer.

When the value of a structured variable is defined, the entire structure must be defined by one expression. Individual structure elements may not be defined separately. For example, the statement A[5] := 10 is invalid in MaD because a specific element of A is being defined. In addition, structured variables must be defined strictly

hierarchically. For example, to define the value of a matrix of integers, M, the computation of the least significant dimension must be nested within the block in which computation of the most significant dimension occurs. This restriction is caused by MaD's data structure implementation. The lower order dimensions of a multidimensional structure are always implemented as stored streams. A two-dimensional array is implemented in MaD as a stream of context tokens, or pointers. Each context token points to a storage node. A storage node is an area of the Matching Store. The storage node for a row of the matrix contains the integer values for that row of the matrix. The matrix values must be defined row-wise, that is, the row M(1) must be defined before any element of the next row M(2) can be defined. There is no way to express in the language column-wise definition of matrix values. The following statement shows a two level nested block which computes the value of a two-dimensional array X1.

```
DECLARE X1: STREAM STREAM INTEGER;  
        I: INTEGER;  
[* Top Level Loop- all the rows are defined *]  
  FOR EACH I IN IXSET DO  
    X1 := DECLARE  
          J: INTEGER;  
          X2: STREAM INTEGER;  
[* Inner Loop- a single row is defined *]  
    FOR EACH J IN JXSET DO  
      X2 :=  
        DECLARE K: INTEGER;  
        X: INTEGER;  
        FOR EACH K IN KXSET DO  
[* The value of an element in the array is  
  computed by a function FUNC.          *]
```

```
      X := FUNC(K);  
      RETURN ALL X;  
      RETURN ALL X2;
```

In this example IXSET and JXSET are assumed to be a stream of integer indices from 1 to the range of the first dimension of X1 for I, and from 1 to the range of the second dimension of X1 for J. Each instance of the inner loop defines one row of the matrix X1. Individual elements of a structured variable such as X1 cannot be defined separately. When a structured variable is referenced in an expression, however, individual components may be selected. Other examples of definition and usage of arrays are given below.

III.3.3 The Expression

```
<expression> ::= DECLARE <block> |  
                IF <condexp> |  
                <basicexp> |  
                CASE <casexp>
```

The <expression> can be one of four constructs: a DECLARE block, an IF expression, a basic expression, or a CASE expression. The first two are described here. The basic expression is defined in the next section. The case expression is not used in translating templates to MaD. It is described in the complete MaD BNF in Appendix A.

```
<block> ::= <id> [ ',' <id> ]* :  
           <typedefn> ';' <legalblock>
```

```
<legalblock> ::= <id> [ ',' <id> ]* :  
                <typedefn> ';' |  
                <let> |  
                <initforwhile>
```

Local variables may be declared in a block. Scope rules for variable declaration are the same as for conventional block structured languages. Following the local variable declarations, the block may contain either a LET statement containing one or more assignments or a looping construct.

```
<let> ::= LET [ <lhs> := <expression> ';' ]+  
        RETURN <expression>  
  
<lhs> ::= <id> | '[' <lhs> [ ',' <lhs> ]* '']
```

In a LET statement, a previously declared variable may be assigned a value. Each variable may be defined only once in the program. Since the block must return a value, the LET must be terminated by an <expression> to be returned. The value of the block is the value of this expression.

Example:

```
LET XSTREAM :=  
    DECLARE  
    XI, AI, BI: INTEGER;  
    FOR EACH AI IN A; EACH BI IN B DO  
    XI := AI + BI;  
    RETURN ALL XI;
```

In this example, XSTREAM is assumed to be of type STREAM INTEGER. The

value of XSTREAM is computed by the above DECLARE block. This DECLARE block contains a structured loop.

```
<initforwhile> ::= [ INIT [ <lhs> ':=' <expression> ';' ]+ ]
                FOR EACH <id> IN <streamid>
                [ ';' EACH <id> IN <streamid> ]*
                DO
                [ WHILE <expression> DO ]
                [ [ NEW ] <lhs> ':=' <expression> ';' ]+
                RETURN <expression>
```

A structured loop consists of three parts. The optional initialize part assigns initial values to variables declared locally in the block. The FOR EACH identifies streams whose elements are selected within the loop. The optional WHILE part sets up termination conditions for the loop. If the WHILE is not used, end-of-stream signals loop termination. Next in the loop come the assignment statements. In this section, the loop variables may be reassigned. The NEW qualifier indicates reassignment. The final part of the loop is the RETURN statement. The value of the block is the value of the RETURN expression.

Example:

```
S :=
  DECLARE I: INTEGER; S: TS;
          [* TS is a user-defined type *]
  INIT S := ((FA + F[N+1]) / 2);
        I := 1;
        WHILE I <= SIZES - 1 DO
          NEW S := S + F[I+1];
          NEW I := I + 1;
        RETURN S;
```

III.3.4 The Basic Expression

```
<basicexp> ::= <all-remainder> |  
            <simpleexp> [ <relop> <simpleexp> ]
```

A basic expression in MaD is either an expression using the stream constructs ALL or REMAINDER or a more conventional arithmetic expression.

```
<all-remainder> ::= ALL <basicexp> [ BUT <basicexp> ] |  
                REMAINDER [ <id> ]
```

ALL constructs a stream by concatenating together each instance of <basicexp> in the loop. The BUT qualifier allows some of the stream members to be filtered out. The REMAINDER <id> must be a stream identifier referenced in a FOR EACH. Examples of these expressions occur in the FOR EACH loop above.

A simple expression is a standard arithmetic expression augmented with a couple of unusual features. The operators '*', '+', AND, OR, MAX, and MIN can be used as reduction operators. Similar reduction operators are used in Model. An "informal" record may be constructed in MaD with the left and right bracket notation described above. Operations may be performed with portions of a stream by specifying high and low bounds for the stream subset.

In addition to user-defined functions, the language provides standard mathematical functions such as SIN, COS, LN, SQRT, and exponentiation. Stream functions include the CONS to construct a stream, FIRST, REST, GET (for a stored stream), EMPTY, and SIZE.

Example:

```
IF EMPTY(XSTREAM) THEN 1 ELSE 1 + FIRST(XSTREAM)/((1 +  
FUNC(REST(XSTREAM)))
```

In this basic expression, XSTREAM is assumed to be a numeric stream. FUNC is an arbitrary function whose call parameter is a numeric stream. The standard stream functions EMPTY and FIRST are used in the expression.

III.4 NONPROCEDURAL LANGUAGES

III.4.1 Common Properties

Nonprocedural languages have been suggested as being particularly well suited to data flow. The Model nonprocedural language has only two statement forms, data description and assertion. The assertions describe output data in terms of input data. The term "assignment", whether single or multiple, is not applicable to nonprocedural languages. Data cannot be assigned. Instead, using the assertions, one can describe properties of the data items. The independent data item, on the left hand side of the assertion, is described as a

function of one or more dependent data items. This defining function is on the right hand side of the assertion. The use of these two simple statement forms allows a very high level form of problem description.

Environment or context is also missing in a nonprocedural language. The validity of the assertions is not time dependent. It is not tied to a state of the machine. In fact, a nonprocedural language has no sequencing or control constructs. Data dependencies alone control the sequence of execution. There might be several linear execution sequences of a specification which are correct. From the discussion of the data flow model in Section II.1, it is clear that specification languages follow exactly the semantics of the model. The availability of data drives program execution. Since there is no programmer-controlled sequencing, a data flow graph can be constructed directly from the specification.

Nonprocedural languages have the same functional semantics as the applicative data flow languages described above. However, a language such as Model is at a higher level than the data flow languages. A programmer using a data flow language must recognize iterative and parallel aspects to the algorithm and must explicitly construct iterative and parallel blocks in the program. In contrast, a programmer using Model need only specify operations on data structures and arrays. The Model Processor detects iteration and parallel invocation from the subscript expressions used in array reference. It

then generates the appropriate kind of block in the data flow template.

III.4.2 LUCID

LUCID is a nonprocedural language invented by Ashcroft and Wadge [Ashc77]. The design of LUCID was motivated by the desire to combine program writing and program proving into a single language. The assertions in LUCID are the axioms from which properties of the program are derived. LUCID has sequencing control operators such as FIRST X and NEXT X to specify values occurring during an iterative computation. The AS SOON AS operator can be used to extract values from a loop. A data item in LUCID can be thought of as an infinite sequence. The elements of the sequence form the history (or "world line") of the data item in an iterative computation. This is equivalent to the STREAM type in data flow languages.

III.4.3 MODEL

The MODEL nonprocedural language has many concepts in common with LUCID. The "world line" analogue in MODEL is one dimension of an array. The elements of the array represent successive values of a data item during an iterative computation. This notation is more powerful than that of LUCID because data element values other than the current, next, and first can be referenced by using the appropriate

subscript expression. The notation facilitates ease of reference to multi-dimensional structures. The same array notation can be used. In LUCID, multi-dimensional structures must be constructed with nested loops.

Iterative loop termination can be expressed in MODEL in several ways: the array can be given a constant upper bound; the SIZE attribute of the array can be defined in an assertion; or the END.array condition can be defined in an assertion. The use of these attributes is described in Section III.5.4.

The objectives of MODEL are different from those of LUCID. MODEL has been designed as a software aid tool to automate program development. The system does extensive consistency and completeness analysis of a specification. The sequential Model Processor generates a PL/1 program from the specification [LuKa82]. The aim of the MODEL program generation system is to provide a language for very high level specification of a problem by non-programmers. The system has been designed to resolve ambiguities or errors in the specification and report corrections to the user [Shas78]. Since the work reported here has been done with the Model System, we describe a subset of the Model Specification Language in greater detail in the following section.

III.5 THE MODEL SPECIFICATION LANGUAGE

A specification in Model consists of a program header followed by data declarations and assertions. The header contains the specification name and the names of the input parameters to and output results from the specification. The input and output parameters must be of type file. The following is an example of a program header:

```
MODULE: MATRIXMUL;  
SOURCE: INFILE;  
TARGET: OUTFILE;
```

This header indicates that the specification name is MATRIXMUL and that there is one input file, INFILE, and one output file, OUTFILE. The files are assumed to be sequential. There must be a target file declaration: the specification must have output. The data structures of the files are given in the data declaration statements.

III.5.1 Data Declaration

Model supports structured data in the syntax style of PL/I. The data declaration statement defines the structures of the input and output files and of any interim data used. Levels of structure include FILE, RECORD, GROUP, and FIELD. The RECORD is the unit of input/output. The FIELD is the smallest unit of data. The field must be of elementary data type. Some representative data declaration

statements follow. Note that each statement is independent of any other. The order is not significant to the Processor. The order of statements and indentation used merely enhances readability.

```
INFILE IS FILE (INREC);
  INREC IS RECORD (IN1, IN2);
    IN1 IS GROUP (IN12(10));
      IN12 IS GROUP (A(10));
        A IS FIELD (NUMERIC);

    IN2 IS GROUP (IN21(10));
      IN21 IS GROUP (B(10));
        B IS FIELD (NUMERIC);

OUTFILE IS FILE (OUTREC);
  OUTREC IS RECORD (OUT1);
    OUT1 IS GROUP (OUT2(10));
      OUT2 IS GROUP (C(10));
        C IS FIELD (NUMERIC);

XO IS GROUP (X1(10));
  X1 IS GROUP (X2(10));
    X2 IS GROUP (X(10));
      X IS FIELD (NUMERIC);
```

These declarations indicate that the input file consists of one instance of one record, INREC. INREC contains two elements, IN1 and IN2, which are also aggregates. A and B are the leaf nodes of this tree structure. They are fields. INREC therefore consists of two matrices. The output file is similarly defined. OUTREC is also a matrix. XO illustrates the declaration of an interim structure.

III.5.2 Array Data

Array data has two major uses in Model. In addition to the conventional usage of array in the mathematical sense of vector or matrix, the array is also used as an indication of repetition. Iterative computation can be expressed in Model using the same array notation as for matrix manipulation. Examples of the use of arrays in iteration are given in the next section. Here, we describe several special constructs which support use of arrays.

III.5.3 Subscript Data Type

The SUBSCRIPT data type is a unique feature of Model which helps describe the index of an element in an array. A variable declared as type SUBSCRIPT may assume all values in a range from 1 to some upper bound. For example, the declaration

```
I IS SUBSCRIPT (100);
```

defines I to have all values in a range from 1 to 100. When I is used to qualify an array variable, as in the reference A(I), the entity A(I) denotes all 100 elements of A. In contrast, if X is defined by the statement

```
X IS FIELD NUMERIC;
```

then the reference A(X) denotes only one element of A- the element whose index corresponds to the value of X.

Model distinguishes between global and local subscripts. The explicit subscript declarations define global subscripts. Local subscripts are of the form SUB n , where n is a positive integer. They are predefined to the Model Processor. The subscript is local to an assertion. The name may be reused in other assertions to denote different ranges.

III.5.4 Range Arrays

The number of elements in one dimension of an array is called the range of that dimension of the array. An array dimension may have a constant range definition or either of two system-defined auxiliary arrays to define the range. They are called range arrays.

If A is an n -dimensional array, then $SIZE.A$ is a k -dimensional integer array, where $0 \leq k < n$. $SIZE.A$ defines the range of A 's rightmost (least significant) dimension. The range of each dimension of $SIZE.A$ is equal to the range of the corresponding dimension of A . However, $SIZE.A$ must have fewer dimensions than A . For instance, if A is a vector, then $SIZE.A$ is a scalar. If the value of $SIZE.A$ is 15, this implies that A has fifteen elements. The $SIZE$ array may be defined in an assertion to establish the range of the last dimension of the corresponding array. The value of the $SIZE.A$ array cannot depend on a subscript j of A where $k < j \leq n$. Figure 3.1 shows the correspondance between a two-dimensional array and its $SIZE$ array.

The other array qualifier is END. END can also be used to define the range of the least dimension of an array. If A is an n-dimensional array, END.A is an n-dimensional boolean array whose value is defined as follows:

If $\text{END.A}(I_1, I_2, \dots, I_{n-1}, q)$ is true and $\text{END.A}(I_1, I_2, \dots, I_{n-1}, p)$ is false for $1 \leq p < q$, then $A(I_1, I_2, \dots, I_{n-1})$ has q elements [LuKa82]. For example, if A is a vector, and $\text{END.A}(I)$ is false for $1 \leq I < 5$, and $\text{END.A}(5)$ is true, then A has 5 elements. The values of END.A may depend on data in A. Thus the range may be defined by data generated at run time. Figure 3.2 shows the correspondance between a two-dimensional array and the corresponding END array.

Although there are many special prefixes in addition to SIZE, and END, only these two are used in the data flow version of Model.

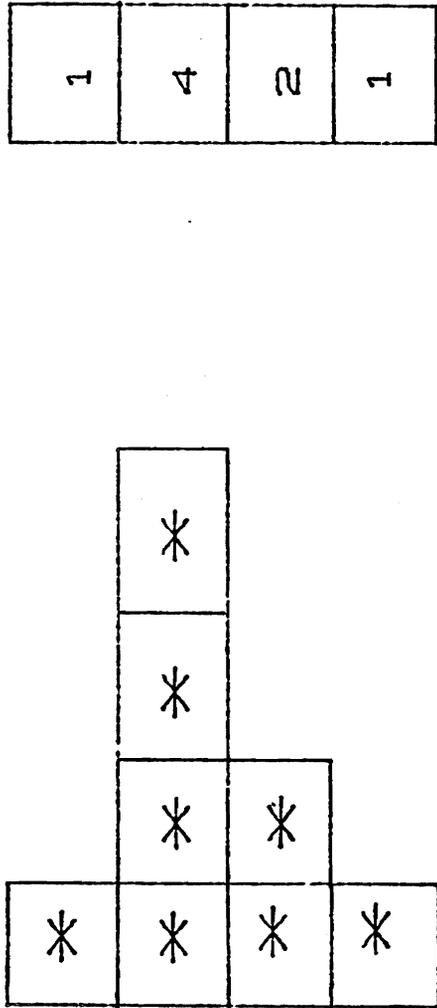


FIGURE 3.1 A 2-DIMENSIONAL ARRAY AND ITS SIZE ARRAY

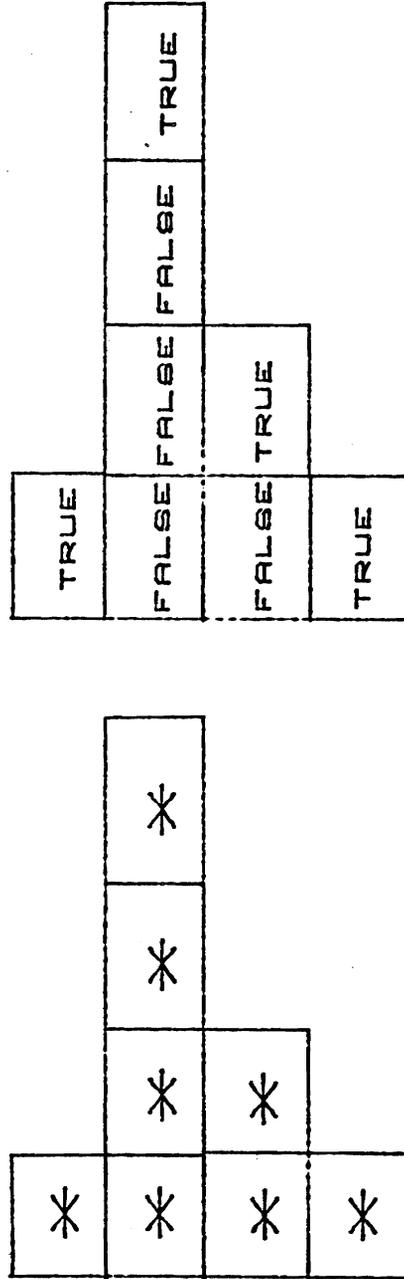


FIGURE 3.2 A 2-DIMENSIONAL ARRAY AND ITS END ARRAY

III.5.5 Assertion

The assertion is used to define the values of variables declared in the specification. An assertion in Model is of the form

`<target variable> = <expression>`

The target variable, or dependent variable, may be a scalar or may be subscripted. If the variable is subscripted, the form of each subscript expression may not be an arbitrary expression. It must be of the form `<global subscript name>` or `<local subscript name>`. This restriction exists because the target language MaD does not support the structure operations SELECT and APPEND. Instead, a structure must be completely defined by one statement. Use of an arbitrary subscript expression implies selective definition of one array element rather than definition of each element. The expression on the right hand side contains the source, or independent, variables. The expression may be an arithmetic or conditional expression. Subscripts on the right hand side may contain arbitrary subscript expressions.

A conditional expression is of the form

`IF <boolean expression> THEN <expression>`

`ELSE <expression>`

The following are examples of Model assertions:

`X(I,J,K) = A(I,K) * B(K,J);`

`S(I) = IF I = 1 THEN (FA + F(N))/2;
ELSE S(I-1) + F(I-1);`

```
END.J(SUBI,SUBJ)= IF J(SUBI,SUBJ)>= IFILE.N THEN TRUE;  
ELSE FALSE;
```

In the first assertion we assume that I, J, K are of data type SUBSCRIPT. With this assumption in mind, we can see how the assertion illustrates the power of the specification language. In a conventional high level programming language, the statement would have to be enclosed in a three level loop. In Model, use of the subscript data type permits the programmer to specify operations on a three-dimensional array with a single assertion. The second assertion illustrates the use of recurrence relations. In this example, the THEN clause establishes the initial condition because it defines the value of S for the first iteration. The ELSE clause defines the recurrence relation. The termination condition for the recurrence could be established in any of the ways discussed above: with range arrays or through the range of the subscript I. The third assertion illustrates use of the range array END. Note that the value of END.J depends on data available only during program execution.

III.6 THE EXAMPLE SPECIFICATION

The specification shown in Figure 3.3 below demonstrates some of the constructs in the Model Language. This specification is also used as an example in the following chapters.

```
MODULE: EXAMPLE;

SOURCE: INFILE1, INFILE2;
TARGET: OUTFILE;

INFILE1 IS FILE (INREC1(100));
  INREC1 IS RECORD (A);
  A IS FIELD (NUMERIC);
INFILE2 IS FILE (INREC2(100));
  INREC2 IS RECORD (B);
  B IS FIELD (NUMERIC);

OUTFILE IS FILE (OUTREC(100));
  OUTREC IS RECORD (C);
  C IS FIELD (NUMERIC);

XO IS GROUP (X(100));
  X IS FIELD (NUMERIC);

I IS SUBSCRIPT (100);

/* Assertion 1: AASS220 */
X(I) = A(I) + B(I);
/* Assertion 2: AASS230 */
C(I) = X(I) * X(I);

END EXAMPLE;
```

Figure 3.3 The EXAMPLE Specification

A BNF description of Model is included in Appendix B.

III.7 CONCLUSION

This concludes the discussion of languages for data flow. We have described novel features of data flow languages. Data flow languages distinguish between iterative and parallel loops and have a new data type, the stream, which is used for I/O and for

multi-dimensional structured data. We have presented in greater detail the MaD data flow language and the Model Specification Language. The next chapter describes how a specification is stored and processed by the Model System.

CHAPTER IV
THE MODEL SYSTEM

This chapter describes the array graph and other data structures of the Model System which are used in scheduling and code generation [LuKa82] [Pryw82a]. The first step in generating a high level language program from a specification is syntax analysis. The Model System reads the specification and checks the syntax. If any errors are found, they are reported, and the system halts. If the specification is verified to be syntactically correct, it is then checked for semantic correctness, completeness, and consistency. In many cases, incompleteness and ambiguity in the specification are corrected, and warnings reported to the user [Lock82]. If it is not possible for the system to correct the specification, error messages are issued, and further analysis is curtailed. If the specification passes these stages, the system makes an internal representation of each entity in the specification, and builds the array graph. The array graph is the major input to the scheduler.

The Model System implementation is described in [Luka81]. In addition, in [Shas78] may be found description of a more powerful general-purpose programming variant of Model. [Gree81] describes a Model variant which can generate a PL/I program to solve a set of simultaneous equations. [Pryw82b] is a report on using Model for cooperative computation in a distributed computing environment.

The next section describes the underlying graph of a specification. Next, we describe the array graph, node dimensionality, and the precedence relations which determine the edges of the graph. The concepts of range and range set are introduced next. The final section defines the storage allocation attributes of physical and virtual storage.

IV.1 THE UNDERLYING GRAPH

A natural way to represent the specification for analysis is in the form of a graph. There is a node in the graph for each entity in the specification. Edges are inserted into the graph to indicate precedence relationships between the nodes. To be an accurate representation of the specification, the graph should contain a node for each array element and a node for each instance of an assertion. For example, if variables A and B are assumed to have 10 elements each, then the assertion

$$A(I) = B(I)*5$$

has 10 instances, one for each value of I from 1 to 10. This graph is called the underlying graph of the specification. If such a graph were constructed, it would certainly be very large. In addition, in many cases the graph could not be constructed because the number of elements in an array might not be determined until run time. Thus the system would not know at compilation time how many nodes to create. For these reasons, a compact form of the underlying graph is constructed instead of the underlying graph itself. This graph is called the array graph. It is so named because each node and each edge may be multi-dimensional. A node represents an entire array (of zero or more dimensions) and an edge represents the corresponding array of relationships between nodes.

There is a node in the array graph for each data item declared in the specification and each assertion. In addition, nodes are created for range arrays if these are used in assertions. Such basic information as the node type (file, record, group, field, assertion, etc.), node dimensionality, and node predecessors and successors is stored with the node.

IV.2 DATA STRUCTURES DEFINING THE ARRAY GRAPH

The following description of the array graph is adapted from [Pryw82a]. The array graph is represented by three data structures:

- DICT. A dictionary of nodes. There is a NODE entry in the

dictionary for each assertion and each data item.

- **NODE.** All the attributes of interest for an assertion or data item are stored in the NODE data structure. Figure 4.1 lists the NODE attributes. Figure 4.2 shows the data structure of a local subscript list entry of a node. The local subscript list is a description of the node dimensions.

- **EDGE.** Information about the relationships between nodes is stored in the EDGE data structure. In the EDGE are shown the source and target of the edge, and the subscript expressions used in the edge. Figure 4.3 shows the data structure of the EDGE, and Figure 4.4, the data structure of a subscript expression. Figure 4.5 shows the nodes and edges in the array graph for the specification EXAMPLE (Figure 3.3).

- NODE_ID. Node number and Name
- NODE_TYPE. Node type (data or assertion)

If the NODE_TYPE is data, the NODE contains the following attributes:

- PARENT: the array graph node number of the parent of this node.
- #SONS: the number of immediate descendants of this node.
- SON1: the node number of the leftmost descendant.
- BROTHER1: the node number of the sibling to the immediate right of this node.
- REPEATING: whether this node is repeating or scalar.
- SUBSLST. The local subscript list, a list of node subscripts associated with the node.
- PRED_LST. The Predecessor Edge list.
- SUCC_LST. The Successor Edge list.

Figure 4.1 The NODE Data Structure

- REDUCED. Whether the dimension is reduced in that subscript. Only applicable to assertion nodes.
- STOTYP. Whether the dimension is physical or virtual. Used only for data nodes.
- RANGE. The range set number of the range of this dimension.
- RALP. The real arguments to a range array.
- IDWITH. The nesting level. Determined in scheduling and used in code generation.

Figure 4.2 A Node Subscript

- EDGE_TYPE. Data Dependency, Hierarchical, Parameter Dependency.
- SOURCE. The node number of the predecessor node.
- TARGET. The node number of the successor node.
- DIMDIF. The difference in dimensionality between the source and target.
- SUBX. The subscript expressions used in the edge, ordered by position in the predecessor node.

Figure 4.3 The EDGE Data Structure

- LOCAL_SUB#. Local subscript position in the successor node.
- APR_MODE. Subscript expression type ("I", "I-k" (where $k \geq 1$), other).
- I_K. The "k", if the APR_MODE is "I-k".

Figure 4.4 Subscript Expression

Nodes:

1	: OUTFILE	8	: INFILE2.B
2	: OUTFILE.OUTREC	9	: INFILE2.INREC2
3	: OUTFILE.C	10	: INFILE2
4	: AASS230 (Assertion 2)	11	: INFILE1.A
5	: INTERIM.XO	12	: INFILE1.INREC1
6	: INTERIM.X	13	: INFILE1
7	: AASS220 (Assertion 1)	14	: EXAMPLE

Edges:

Predecessor	Successor	Edge Type
2	1	2
3	2	2
3	2	10
4	3	7
6	5	2
6	4	3
7	6	7
8	7	3
9	8	1
10	9	1
11	7	3
12	11	1
13	12	1
14	13	21
14	10	21
14	1	21

Figure 4.5 Nodes and Edges for the EXAMPLE array graph

IV.3 NODE DIMENSIONALITY

Each node in the array graph has zero or more dimensions. For example, X in the declaration

```
X IS FIELD (NUMERIC);
```

has a dimensionality of zero, and V in the declaration

```
VO IS GROUP V(10);
```

has a dimensionality of one. The dimensionality of an assertion node is set to the union of the dimensionality of all the data nodes in the assertion. After a data node's dimensionality has been determined from the data declaration statement, the data usage must be checked in the assertions to ensure consistency of dimension. In one dialect of Model, variables may be used without subscript or with an incomplete subscript list in assertions. The actual number of dimensions is then constructed from the variable's data declaration, and the node dimension fields of the variable and the assertions which use it are augmented appropriately [LuKa82]. In Figure 4.1, the node dimensionality determines the number of node subscripts in the local subscript list. Each node subscript contains a further description of a dimension.

The REDUCED field in a node subscript is used only in an assertion node. The assertion's node subscript list is the union of the subscript lists of source variables and of the target variable. If a subscript appears of the right hand side of the assertion but not on the left hand side, then we say that the subscript has been

reduced, and the REDUCED field for that subscript for the assertion node is set to 1.

The IDWITH field is determined in scheduling. Each node dimension is identified with a block level when the node dimension is scheduled. The block level number is stored in IDWITH.

RANGE and RALP hold information about the range of the subscript. Definitions of range and of a range set are given below. The field STOTYP is used only for data nodes. It records how the node dimension is represented in the generated program. During scheduling, each node dimension is determined to be either physical or virtual. If a node dimension is physical, the STOTYP field is negative. If the dimension is virtual, the STOTYP field is positive. The absolute value of the STOTYP indicates the amount of storage required by the data node in the generated program. The terms physical and virtual are defined below.

IV.4 PRECEDENCE RELATIONSHIPS

The predecessor and successor lists of a node in the dictionary define the precedence relationships between a node and other nodes in the specification. If a node N_i is on the predecessor list of a node N_0 , we say that there is an edge from N_i to N_0 . Similarly, if N_j is on the successor list of N_0 , then there is an edge from N_0 to N_j .

An edge from a node N_0 to a node N_j has the following form:

$$N_j(U_1, \dots, U_k) \xleftarrow{t} N_0(J_1, \dots, J_m)$$

where t is the edge type, k is the dimensionality of N_j and m is the dimensionality of N_0 . The subscript expressions U_1, U_2, \dots, U_k are stored with the dictionary entry for N_j . J_1, J_2, \dots, J_m are the subscript expressions associated with dimensions 1, 2, \dots , m of N_0 . For example in the assertion,

Assertion: $A(I) = B(I-2) + B(I-1);$

there is a subscript expression "I-2" associated with the first dimension of an edge from B to Assertion and a subscript expression "I-1" associated with the first dimension of a second edge from B to Assertion.

There are three broad categories of precedence relationships, data dependency, hierarchical, and data parameter. Data dependency means that data must be generated before it can be used. Therefore, for each assertion, source variables appear on the predecessor list of the assertion, and the target variable appears on the successor list. A hierarchical relationship exists between a higher level data structure and its components. For example, there is a hierarchical relationship between a file and the records it contains, or between a group and its component fields. Therefore, the records of a file

appear on the successor list of the file's dictionary entry, and the file appears of the predecessor lists of each of the records' dictionary entry. A data parameter relationship exists between a range array and the nodes whose range it defines. A data parameter edge is drawn from a range array to the nodes whose ranges it defines.

IV.4.1 The EDGE Data Structure

Each element in a node predecessor or successor list points to an EDGE data structure (Figure 4.3). The DIMDIF field in this structure is the dimensionality of the target node minus the dimensionality of the source node. The EDGE_TYPE indicates the type of precedence relationship between source and target. There are several edge types with which we will be concerned. A Type 3 edge is drawn from an independent variable of an assertion (on the right hand side) to the assertion. A Type 7 edge is drawn from an assertion to the variable it defines (the variable on the left hand side). Edge Types 1 and 2 are hierarchical edges drawn from a node in an input file to its immediate descendants for the Type 1 and from a node in an output file to its immediate ancestor for the Type 2 edge. Edge Types 13 and 14 are used for the range arrays SIZE and END respectively. They are drawn from the range array node to the node whose range is being defined. The Type 8 edge is drawn between siblings in an input or output file if the nodes concerned are below the record level or if they belong to sequential files. A Type 21 edge is drawn from the

module to each file. It indicates the precedence of the module over the file nodes.

In the sequential version of the Model Processor, Type 8 edges are also drawn from the last field of a repeating structure of an input sequential file back to the node representing the structure, and from a repeating structure in an output sequential file to the first field in the structure. For an input structure, this edge is inserted because the last field of the previous instance of the structure must be processed before the next instance of the structure may be accessed. For the output structure, the edge means that an instance of a repeating structure may not be written until the previous instance has been completely written.

However, as we have seen in Chapter 2, data structures are not accessed and stored on data flow machines in the same way that structures in sequential files are accessed and stored in conventional machines. Several instances of a repeating structure may be accessed concurrently on a data flow machine. Therefore, these two cases of Type 8 edges are not used in the Model Data Flow Processor.

IV.4.2 The EDGE Subscript Expression List

The final field of the edge data structure, SUBX, is the list of subscript expressions used in the source. If the subscript expression is of the form Uq or $Uq-c$ for some constant c , then LOCAL_SUB# is q , that is, the ordinal number of the subscript in the target. APR_MODE

is the subscript expression type. The system distinguishes seven types of subscript expressions. A Type 1 subscript expression is a simple subscript reference, as in $A(I)$. A Type 2 subscript expression is of the form $I-1$, as in $A(I-1)$. A Type 3 subscript expression is of the form $I-k$ for some positive integer k greater than 1, as in $A(I-5)$. A Type 4 subscript expression, or generalized subscript expression, is anything other than the first three, for example, $A(N*(R+T))$. In this example, N , R , and T are assumed to be variables declared elsewhere in the specification. The expression " $N*(R+T)$ " is the Type 4 subscript expression. Types 5-7 are used for indirect indexing vectors. An indirect indexing vector can be used to hold index values for another array. The Model Processor detects the use of indirect indexing. The sequential version of the Scheduler optimizes the generated program when this feature is used [Luka82]. If the subscript expression is Type 3, the I_K field holds the constant offset. For example, for the reference $A(I-5)$, the value of I_K is 5.

IV.5 RANGE SETS

The size of an array dimension is called the range of the dimension. The system must determine the range of each node dimension, either as a constant, or in terms of the range arrays described earlier. If the node is placed by the scheduler into an iterative block, then the range of a dimension determines the number of iterations of the node for that dimension. If the node is placed

in a parallel block, the range determines the number of incarnations of the node which may be active concurrently.

Prior to scheduling, the system determines the range for each node subscript. It places each node subscript into a range set. All the node subscripts in a range set have the same range. However, no two dimensions of a node can be in the same range set. This is because the range set is the basis for block of iteration or duplication, and no two dimensions of the same node can be at the same block level. The range set number is stored in the RANGE field of the node subscript.

The range of a node dimension can be established in one of several ways. A constant bound may be used in declaring the array. If the array is part of an input file, the end-of-file condition defines the range. If the array is referenced with a subscript, the range of the subscript is used to define the array dimension range. The SIZE or END qualifiers may be used to define the range.

If a range array is used to define a node dimension, the system stores this information with the dimension. In particular, it records whether any more significant (to the left) dimensions are used in computing the range array. These are called the real arguments to the dimension. When real arguments exist for the node dimension, this indicates that the ranges corresponding to the real arguments precede the range of the dimension. Existence of real arguments to a range array imposes a partial order on the range sets. In generating nested

blocks, the scheduler must observe the partial order. This requires in the synthesized program that the blocks corresponding to the real arguments enclose the block corresponding to the range in question. The real argument list is stored in the RALP field of the node subscript.

IV.6 PHYSICAL AND VIRTUAL DIMENSIONS

A node in the array graph is an aggregate composed of zero or more dimensions. At each node dimension, the number of instances of that dimension is determined by the range of the dimension. For example, let A be a one-dimensional array. Let the range of that dimension be 5:

A IS FIELD (5) FIXED BINARY;

The aggregate character of the data node A is interpreted in the generated program as representing a variable whose data type is stream. A two-dimensional data node corresponds in the generated program to a stream of a stream, that is, a two level stream. The scheduler determines the number of levels of stream required for the variable in the generated program from 1) the data node dimensionality and 2) the ways in which each dimension of the data node is referenced. The scheduler establishes a mapping from each data node dimension in the array graph to a level of stream for the variable in

the data flow template. The mapping differentiates two possible cases:

1. A data node dimension is mapped to a stream, and the number of stream elements is equal to the range of the dimension. In this case, the node dimension is called physical. The field STOTYP in the node subscript description is set by the scheduler to -(range of the dimension). For the above example, the STOTYP for dimension 1 of A, where A is physical, is set to -5.

2. A data node dimension is mapped to one or more representative elements. The number of elements required in the generated program is fewer than the range of the dimension. The dimension is called virtual. The number of elements representing the dimension is called the window into the dimension. In this case, the STOTYP for the dimension is set to the width of the window. If dimension 1 of A is determined by the scheduler to be virtual with a window width of 1 element, then the STOTYP for that dimension is set to 1.

If a variable in the generated program can be represented by fewer representative elements than dictated by the data node dimensionality, the generated program is more efficient. Chapter 6 discusses efficiency gained through detection of virtual dimensions.

IV.7 CONCLUSION

This completes the introduction to the Model System. We have introduced the array graph, a compact representation of the underlying graph of a specification. We have identified the nodes of the array graph, and have shown how edges (indicating various types of precedence relationships) are inserted into the graph. In addition, the concepts of node dimensionality, range, and physical versus virtual storage allocation have been defined.

The next two chapters describe the process of scheduling: translating the array graph to a data flow template. Chapter 5 introduces the process of scheduling and presents a simple algorithm to generate a data flow template. Chapter 6 discusses methods to increase efficiency in the generated program.

CHAPTER V

SCHEDULING THE ARRAY GRAPH FOR A DATA FLOW MACHINE

V.1 INTRODUCTION

Translation from the array graph to a data flow program is performed in two steps. The translation occurs in two phases to separate the problem of determining the structure of the data flow program from the problem of implementing that structure on a specific machine. The array graph is translated first to a data flow program template. The template is an intermediate form of the data flow program. It is a machine- and language-independent representation for a data flow program. The template serves as input for the second step of translation: to a program for a specific data flow language and machine on a chosen level of a high level language, assembly language, or a machine language. Since the format of the data flow template is independent of the object language and machine, the same template can be translated to programs for different data flow languages and

machines. In our case, the program template is translated to the MaD Programming Language for the Manchester University data flow machine. The first step of the translation, from an array graph to a program template, is called scheduling.

During scheduling, nodes of the array graph are merged into components. A data flow computation block is generated from each component. In the following section, the source and target representations of the scheduling process are described. Then, a simple algorithm is defined for translating the array graph to a data flow program template. With this algorithm, the components from which blocks are generated are the smallest consistent with the precedence relationships defined by the array graph.

V.2 DATA STRUCTURES USED BY THE SCHEDULER

V.2.1 The Component Graph

The input to the data flow scheduler is the array graph. The data structure representation of the array graph is described in Section IV.1. The scheduler first builds from the array graph a component graph. The component graph is a more compact representation of the array graph. Each component corresponds to a Maximally Strongly Connected Component (MSCC) of the array graph. (An MSCC is a subgraph of the array graph in which there is a path from any node to every other node.) The component graph is represented as a vector

NODELST of pointers, as illustrated in Figure 5.1.

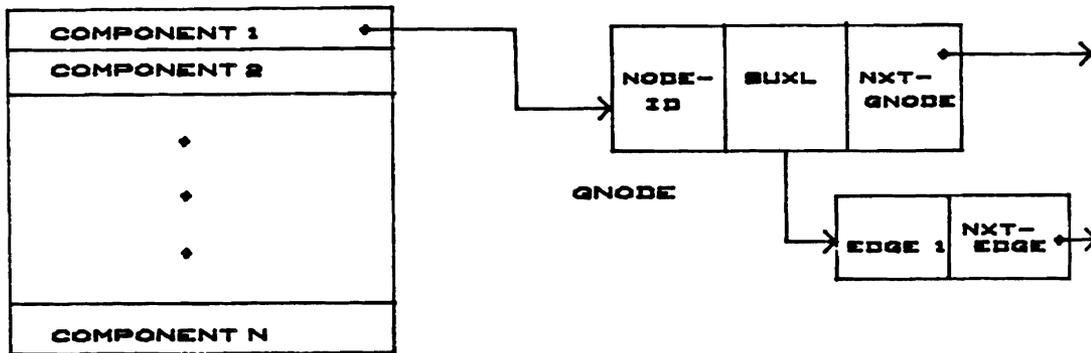


FIGURE 5.1 THE COMPONENT GRAPH

NODELST(I) points to the list of nodes contained in the I'th component. Each node is described by a GNODE entry. A GNODE has the following fields:

- NODE_ID. The NODE_ID is the index of the node in the array graph.
- SUXL. The SUXL is the intra-component list of successors to the node. Each entry in the successor list of a node describes

an edge in the array graph from the node to another node in the same component.

- `NXT_GNODE`. This field is used to link together all the nodes in a component. A `NXT_GNODE` entry points to the the next node in the component.

V.2.2 The Data Flow Template

The template consists of two parts, data description and block description. The data description entries define the characteristics of data to be used in the final program. There are three types of data: input, output, and interim. The input and output data description entries are formed from the source and target file descriptions of the specification. Any data not declared as part of a source or target file is part of the interim data entry. The data description entries contain the file nodes of the specification, and a node for the interim "file." These nodes are represented by their array graph node numbers. Each of these file nodes is the root of a generalized tree. Intermediate nodes of the tree represent `GROUP` or `RECORD` nodes. Leaf nodes of the tree correspond to data fields of the specification. Thus all the descendants of the file can be accessed from the description of the file node. In the second step of translation, code generation to a lower level data flow language, data declarations appropriate to the target language are generated from the data description entries in the template.

The second part of the data flow template is the block description section. A block description entry contains information needed to construct the computation blocks of the final program. A program element indicating repetition or duplication is generated from each block description entry. The block description entry also specifies the dimension and node number of each data node produced as a result of block evaluation. The final portion of the block description entry is a list of block members. The members may be assertions or other blocks nested within the given block. A block contains the following fields:

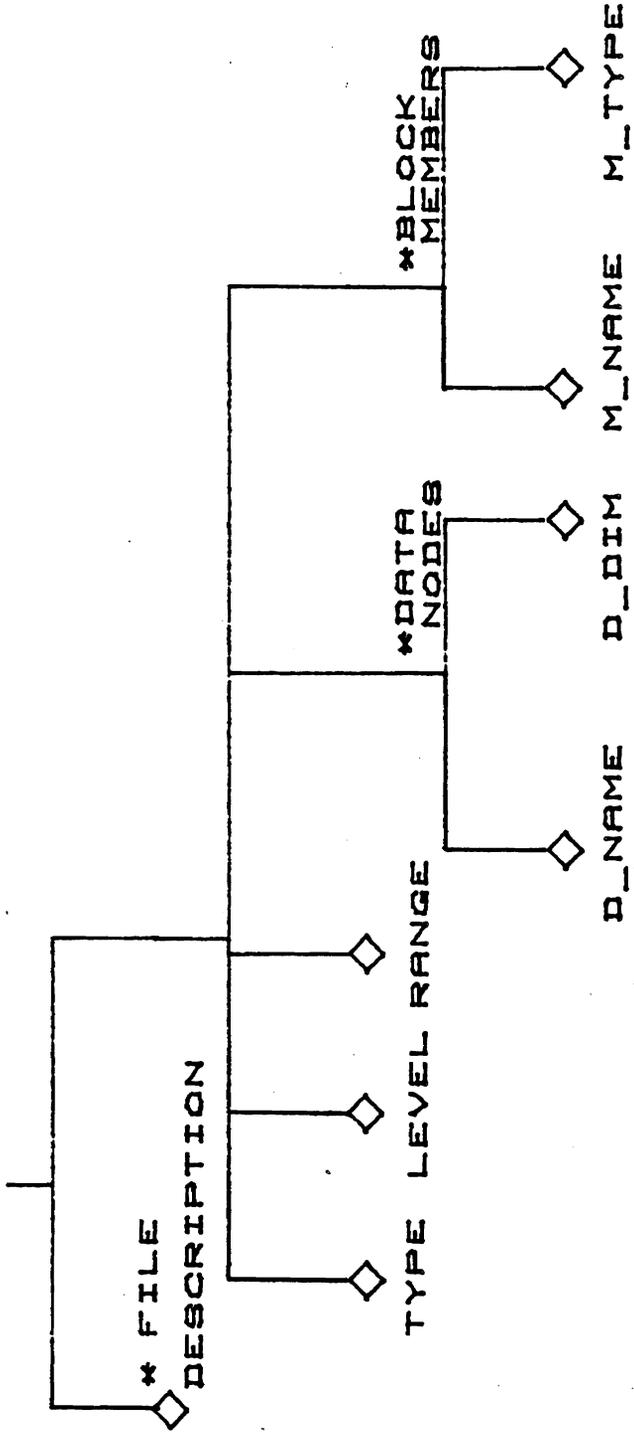
- Number. An integer index assigned to the block.
- Type. A block may be of type simple, iterative, or parallel. A simple block is not to be duplicated or expanded at run-time. It consists of exactly one incarnation. An iterative block is expanded sequentially at runtime. A parallel block is expanded in parallel at runtime. There may be many incarnations of a parallel block active at the same time during execution of the data flow program.
- Level. The nesting level of the block. The outermost block is at nesting level 0.
- Range. This field defines the number of repetitions or incarnations of a block which will be active as the block is executed on a data flow machine. For iterative or parallel

blocks, the range field holds the range set number associated with the block. For simple blocks, the field is 0.

- Data nodes. There is one entry for each data node of type field which is defined as a result of block evaluation. The entry consists of three parts. The first part is the node number of the data node being defined in the block. The second part is the ordinal position of the dimension of the data node which is defined. For a scalar data node this field is zero, since a scalar has zero dimension. The dimensions are numbered (right to left) from least significant to most significant. The notation (D,i) is used to refer to a data node. The D is the identifier associated with the data node. The i is the dimension being defined. The next section describes how it is determined that a data node is defined in a block.

- Block members. The member entry consists of two parts. The first part is the member type. A block member may be an assertion or another block. The second part is the member number. If the member is an assertion, the number is the array graph node number of the assertion. If the member is a block, the number is the block number.

The data flow template is illustrated in Figure 5.2.



* INDICATES MULTIPLE INSTANCES

FIGURE 5.2 THE DATA FLOW TEMPLATE

V.3 A SIMPLE SCHEDULING ALGORITHM

Because Model follows the data-driven semantics of the data flow computation model (see Section III.4), a program template can be synthesized easily from the component graph. The data description section is created by placing each node of a component which is of type file into this section of the template. The block description section of the template is produced by the procedure SCHEDULE. SCHEDULE calls on a mutually recursive pair of procedures to perform the actual scheduling. The first procedure, SCHEDULE_GRAPH, is given a component graph as input and produces the block description entries for the all components. SCHEDULE_GRAPH calls on SCHEDULE_COMPONENT to generate a block description entry for each component of the component graph. If the component C contains more than one array graph node, SCHEDULE_COMPONENT may delete certain intra-component edges, and recursively call SCHEDULE_GRAPH with the component C as parameter. If intra-component edges were deleted, C may no longer be an MSCC. Therefore, SCHEDULE_GRAPH can perform the same actions on this subgraph as it did on the original graph. The simple scheduling algorithm is described in greater detail below. The process is illustrated with the EXAMPLE specification (Figure 3.3) and array graph (Figure 4.5).

The procedure SCHEDULE first performs initialization of data structures needed by SCHEDULE_GRAPH and SCHEDULE_COMPONENT. SCHEDULE then calls SCHEDULE_GRAPH to synthesize a block description entry for

the entire specification. This entry corresponds to the outermost block of the generated MaD program.

V.3.1 Initialization

A component graph is constructed from the array graph. The component graph is viewed initially as consisting of only one component. Each node of the array graph is a member of the single component. The SUXL field of a node in the component is built from the successor list of the corresponding node in the array graph. SCHEDULE allocates space for a block description entry for the initial component. During initialization, some of the fields of this entry are defined. The block is of type simple since there is only one incarnation of the entire program. The nesting level is 0. The data node portion of the block description entry is constructed. A data node in the component is added to the data node section of the block description entry for the initial component if the data node is a field, has zero dimension (a scalar), and is the target of some assertion. The block description entry for EXAMPLE at this point is as follows:

- Type = Simple
- Level = 0
- Range = 0
- Data nodes = null (there are no scalar data nodes)

SCHEDULE then calls SCHEDULE_GRAPH. Call parameters to SCHEDULE_GRAPH

are the initial component graph, the block description entry, and the nesting level (1+ the current nesting level = 1).

V.3.2 Procedure SCHEDULE_GRAPH

SCHEDULE_GRAPH receives as input a component graph and a block description entry. It produces as output block members for the block description entry. SCHEDULE_GRAPH performs the following actions:

1. Find the MSCC's of the component graph. An MSCC of the component graph is a set of array graph nodes which are maximally strongly connected. The component graph is modified to reflect the MSCC's. Figure 5.3a illustrates the initial component graph for module EXAMPLE. After the MSCC's of this component graph have been found, the new component graph contains two components. The new graph is illustrated in Figure 5.3b.

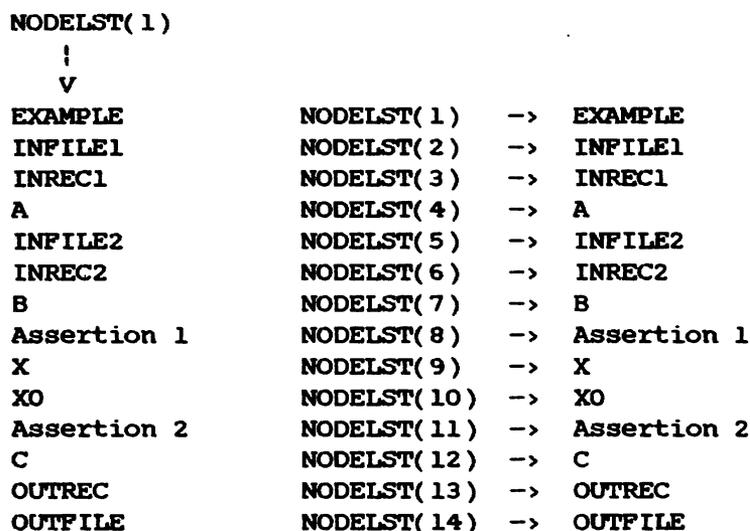


Figure 5.3a

Figure 5.3b

Initial Component
Graph

New Component Graph

2. Call the procedure `SCHEDULE_COMPONENT` once for each component of the component graph. `SCHEDULE_COMPONENT` adds one block member and one or more data nodes to the block description entry for the component.
3. Return a completed block description entry.

V.3.3 Procedure `SCHEDULE_COMPONENT`

Input to `SCHEDULE_COMPONENT` are a single component, C_i , from the component graph; a block description entry, BD ; and the nesting level, NL . The output is a modified block description entry: information about a new block member is inserted into BD . This

process is discussed below.

SCHEDULE_COMPONENT searches the nodes of C_i for node dimensions which have not yet been processed. A dimension of a node which has not been processed is called an unscheduled dimension. The procedure first finds the the minimum number of unscheduled dimensions of any node in the component. It then performs the following analysis:

Let C_i be the component being analyzed, and $|C_i|$ denote the number of nodes in C_i . Let MINFREE be the minimum number of unscheduled dimensions.

Case 1. If $|C_i| = 1$ and MINFREE = 0:

If the node type of the single node in the component is "assertion", then return the node as the block member of the block description entry. The member type is "assertion." The member number is the array graph node number of the single node in the component. If the node is not an assertion node, then a null entry is returned.

Case 2. If MINFREE > 0: This indicates that there is at least one unscheduled dimension in each node in the component. SCHEDULE_COMPONENT attempts to find a range common to an unscheduled dimension of each node. It verifies that the dimension corresponding to the range chosen, the distinguished dimension, is in a consistent position.

Example

Assertion: $A(I,J) = A(I,J-1) + A(J,I)$

C_i contains two nodes, A and Assertion. Each node has two dimension. Assume that SCHEDULE_COMPONENT chooses to schedule the least significant dimension (J). However, there is a conflict in the choice of dimension for A. The least significant dimension of A in one case corresponds to the range set of I and in the other case corresponds to the range set of J. The same problem occurs if SCHEDULE_COMPONENT chooses to schedule the most significant dimension (I). In either case, there is inconsistency in the position of the chosen dimension. This anomaly is discussed below. For now, we assume that a common range is located, and that the distinguished dimension is in a consistent position in each node. If a common range can be found and the component contains at least one assertion, SCHEDULE_COMPONENT can create a new block from the component. After the block is constructed, the block description entry for this new block, BD₁, is returned to SCHEDULE_GRAPH as a member of BD. The member type is "block," and the member number is the index assigned to BD₁ in the data flow template. This new block is nested in the block which SCHEDULE_GRAPH is building. The data structure of a block is described above. The values of the fields in a block are defined as follows:

- a) Block number. The block number is the index of the next available template entry.

b) Block type. If there are any edges with subscript expressions of Types 2 or 3 in the position of the distinguished dimension, the block is of type iterative. Otherwise the block is of type parallel.

c) Block Level. The block level is set to the current nesting level, a call parameter.

d) Block Range. This the range set number of the range found common to an unscheduled dimension of each node.

e) Data nodes defined in the block. For each assertion in the component, the array graph node number of the target of the assertion is added to the data node portion of the block description entry. The ordinal position of the distinguished dimension is added as the dimension of the data node being defined.

f) Block members. SCHEDULE_COMPONENT deletes all edges with subscript expression of Types 2 or 3 in the distinguished dimension. It then calls SCHEDULE_GRAPH recursively with the resultant subgraph, BD1, and (1 + the current nesting level) as parameters. SCHEDULE_GRAPH returns all the members of the block. Each member may be an assertion or may itself be a block.

Cases 1 and 2 can be illustrated with the component graph of the module EXAMPLE. Let SCHEDULE_COMPONENT be called with component $C_i = \text{Assertion } 1$. C_i has one unscheduled dimension, the

dimension corresponding to the range of I. This is the distinguished dimension for Ci. SCHEDULE_COMPONENT creates a new block description entry BD1 for Ci. Since there are no Type 2 or 3 edges, the block type is parallel. The block level is the nesting level, NL. The range is the range set number of the range set containing I. There is one assertion Ci. The target of the assertion is X. Therefore, (X,1) is the data node defined in this new block. The '1' refers to the ordinal position of the dimension of X being defined, in this case, the first. This dimension of X is marked as processed, and SCHEDULE_GRAPH is called with parameters Ci, BD1, and NL=2. Since Ci is an MSCC, SCHEDULE_GRAPH calls SCHEDULE_COMPONENT with parameters Ci, BD1, and NL=2. Now Case 1 applies since $|Ci| = 1$ and there are no unscheduled dimensions. The node in Ci is of type "assertion," so the node is inserted into BD1 as a member of BD1 of type "assertion."

SCHEDULE_GRAPH also calls SCHEDULE_COMPONENT later on with Ci=Assertion 2. The sequence outlined above is repeated. Another member BD2 is added to the block description entry BD. BD2 is defined similarly to BD1. The block type is parallel, the level is 1, and the range is the range of I. The data node (C,1) is defined in BD2. Figure 5.4 shows the completed data flow template for EXAMPLE.

File Description: {INFILE1, INFILE2, OUTFILE, INTERIM}

Block Description:

```
Block 1: Type=Simple Level=0 Range=0
  Data Nodes: None
  Block Members: {(m_name=Block 2 m_type=block)
                  (m_name=Block 3 m_type=block)}

Block 2: Type=Parallel Level=1 Range=1
  Data Nodes: {(d_name=X d_dim=1)}
  Block Members: {(m_name=Assertion 1 m_type=assertion)}

Block 3: Type=Parallel Level=1 Range=1
  Data Nodes: {(d_name=C d_dim=1)}
  Block Members: {(m_name=Assertion 2 m_type=assertion)}
```

Figure 5.4 The Template for EXAMPLE

Case 3. If $|C| > 1$ and $MINFREE = 0$: This case represents a cycle in the array graph which the scheduler is unable to eliminate. The cycle may indicate a true cyclic dependency which would cause the data flow program to hang in execution. An example of such a dependency is the following pair of assertions.

```
ASS 1: A(I) = B(I) + C(I);
ASS 2: B(I) = A(I) - C(I);
```

In this example, the first assertion can only be evaluated when the value of array B has been defined. However, the value of the B depends on A. Figure 5.5 shows the graph for this example. Since the value of B depends on the value of A and the value of A depends on the value of B, neither assertion can be evaluated.

The data flow program generated from this pair of assertions would hang. "Hanging" in the sense of a data flow program means that there is no instruction ready to be executed and therefore no data being generated to enable other instructions.

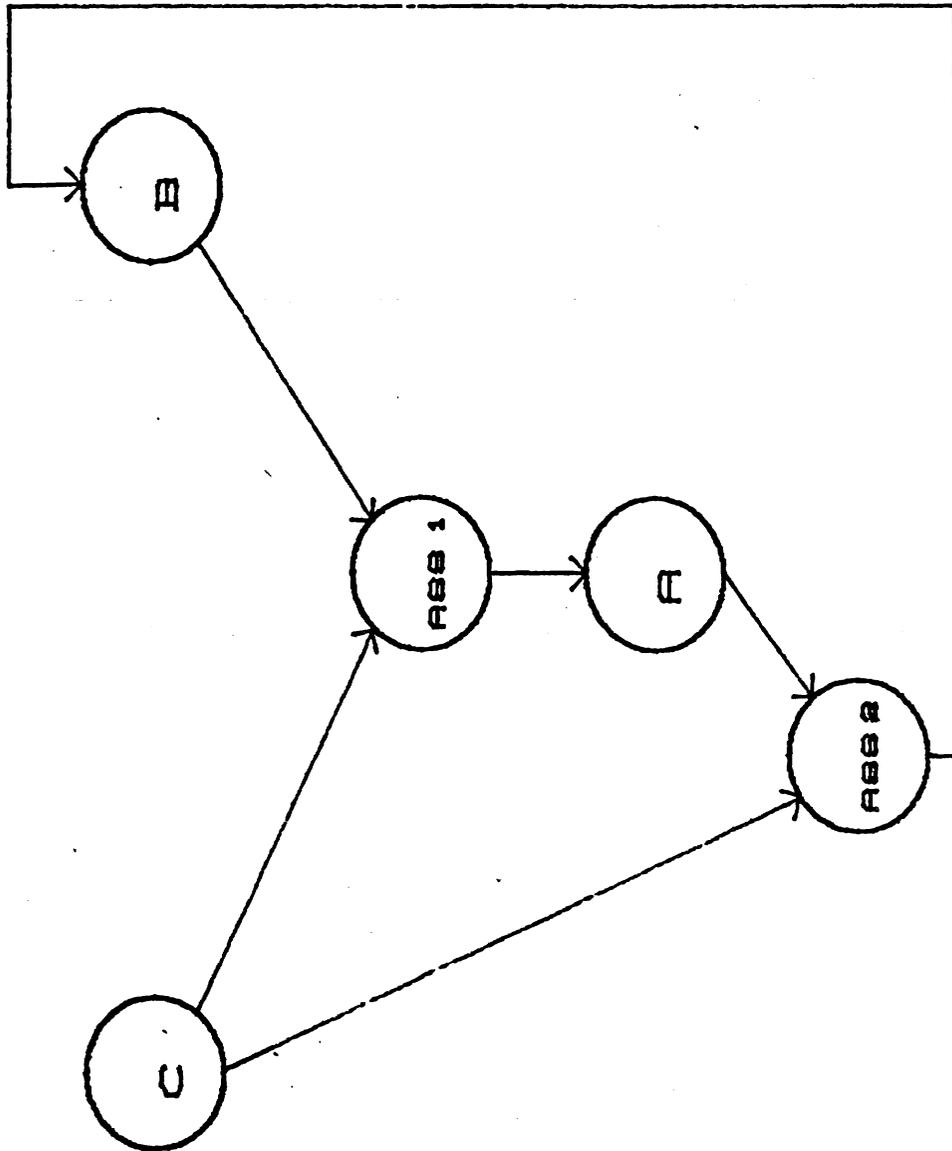


FIGURE B.5 A CYCLE IN THE GRAPH

However, in other cases, a cycle in the graph does not necessarily indicate that the corresponding data flow program would hang. For example, the following assertions can be evaluated:

```
A(I) = IF I=D THEN B(I) + C(I) ELSE C(I);  
B(I) = IF I ^= E THEN A(I) - C(I) ELSE C(I);
```

Here, the only element of A which depends on B is A(D). The value of D is available, however, only at run time. Let the runtime value of D=5, and the run time value of E be some index other than 5. Then, in this case, B does not depend on A(D). The data flow program executes as follows: Each element of A except A(5) can be evaluated as C becomes available. Once these values of A are defined, B can be evaluated. B(5) receives a value as soon as C(5) is available. After B(5) is defined, A(5) can be evaluated. Therefore, each element of A and each element of B can be evaluated. The program does not hang. The cycle in the graph cannot be eliminated because it is not always determinable at compile time that individual array elements will be defined before they are used.

The cases described above cause SCHEDULE_COMPONENT to detect that $|C| > 1$ and MINFREE = 0. The sequence of events which cause this are as follows:

a) SCHEDULE_GRAPH calls SCHEDULE_COMPONENT with the graph of

Figure 5.5.

b) SCHEDULE_COMPONENT finds an unscheduled dimension common to all the nodes in the component, the dimension corresponding to the range of I. In this case, $|C| > 1$ and MINFREE > 0 . Therefore, SCHEDULE_COMPONENT marks the distinguished dimension of each node in the component as scheduled, and attempts to delete any edges with Type 2 or Type 3 subscript expressions. In this case, there are none. It then calls SCHEDULE_GRAPH with this component as a new subgraph. SCHEDULE_GRAPH attempts to find the MSCC's of this subgraph. Since no edges were deleted, there is still only one MSCC. SCHEDULE_GRAPH calls SCHEDULE_COMPONENT with this component as parameter. Now SCHEDULE_COMPONENT discovers that $|C| > 1$ and, since the one dimension of each node is already being scheduled, MINFREE = 0.

Since the cycle is potentially resolvable when the program is run on a data flow machine, SCHEDULE_COMPONENT continues with scheduling. It reports a warning to the user of a possible cycle in the graph. It then deletes an arbitrary edge from the component, and calls SCHEDULE_GRAPH with the component. Deletion of the edge may result in an acyclic subgraph which SCHEDULE_GRAPH can then handle in the normal manner. For example, in Figure 5.5, if the edge from the second assertion to B is deleted, the graph is acyclic. If deletion of the first

edge chosen does not produce an acyclic graph, the process will be repeated recursively until an acyclic graph is produced.

Case 4. `MINFREE > 0` but no common range can be found: This case was introduced earlier. There is no range for which a consistent dimension can be located in each node. This case is similar to Case 3. However, with the present implementation of structured data on the Manchester machine, the program generated from this graph cannot run successfully. A structure must be completely defined in one block before it is available to be accessed by instructions in other blocks. The graph, however, indicates cyclic dependency. Generating an element in one array depends on an arbitrary value in another array, and vice versa for the other array. Therefore, an element from each array is being selected before the entire array is defined. In addition, there is no consistent range, so the assertions defining the two arrays cannot be included in the same block.

To warn the user of this problem, `SCHEDULE_COMPONENT` issues an error message reporting a cycle in the graph, and makes no change to the block description entry. The graph cannot be scheduled.

V.4 CONCLUSION

This concludes the discussion of the simple scheduling algorithm. The next chapter describes modifications to the algorithm to increase efficiency in execution time and in storage requirements on the data flow machine.

CHAPTER VI

SCHEDULING II: EFFICIENCY CONSIDERATIONS

VI.1 INTRODUCTION

The algorithm described in the previous chapter produces a data flow template from an array graph. However, it is possible to further analyze the array graph and to produce a template from which a more efficient program can be generated. The algorithm described below merges components so that the generated block contains more elements than in the simple algorithm.

The dimensions along which to measure efficiency of a data flow program are still being formulated. Most designs of data flow computers are on paper or in very early prototype stage. One study of the performance of a proposed machine has been done using simulation [Gost80]. Prototypes which have been built have not been studied extensively in terms of efficiency in programming them.

We can demonstrate in a data flow scheduler, therefore, some optimizations which seem appropriate 1) from results obtained from running programs on a specific machine and 2) from a comparative study of several different machines (see Chapter 2). Two general areas of optimization are pursued here. These areas are block enlargement and data structure simplification.

VI.2 BLOCK ENLARGEMENT

The first area of optimization is to enlarge the size of a block generated by the scheduler. The size of a block is the number of block members. Enlarging the size of a block generates a more efficient data flow program in two ways. The first reason that having one large block may be more efficient than having several small blocks has to do with reducing the number of allocations of program units to processors in the data flow machine.

A data flow machine is composed of one or more processors communicating through an interconnect. An individual processor has only local storage for programs and data tokens. There is no memory shared by all the processors in the machine. If data produced in one processor is needed by an instruction in another processor, that data must be transmitted along the communication path to the other processor. The transmission may require routing through one or more intermediate link. To minimize the cost of data transmission between processors in the machine, the scheduler follows the principle of

locality of reference. The scheduler may enlarge the scope of units of allocation so that a program unit contains a related set of instructions: that is, data produced by an instruction in the program unit is used by other instructions in the same unit.

The other way in which generating one large block produces a more efficient program than generating several small blocks has to do with cost of transmitting streams from one block to another. This cost occurs whether or not the blocks are located in the same processor. On the Manchester machine, the cost is manifested in the work associated with generating and updating token labels.

Each token generated or referenced in a MaD language block has a label which identifies it as being in a unique block instance. When a token enters a block, the old Activation Name (AN) and Iteration Level (IL) fields of the token are replaced with new AN and IL fields. The new fields indicate that the token has a new context- the context of the associated block. Variables local to a block, that is, generated and used completely within the scope of the block, are also labeled with the new AN and IL. A token exiting the block must have the old AN and IL inserted into the label fields, indicating that the token is no longer associated with the block. When a stream of tokens is generated within a block, the label of each token exiting the block must be changed. This is accomplished by generating two label streams, each with the same number of elements as the data streams. There is one label stream for the AN and one for the IL. Pairs of

elements (Data and AN) are input to a Set Activation Name instruction to update the AN. Pairs of elements (Data and IL) are input to a Set Iteration Level instruction to update the IL. The cost of having a data stream exit the block is that two label streams must be generated, and the two fields of each data token must be updated. Therefore it is advantageous to detect data which is local to a block. If a variable is local to a MaD language block, it need not exit the block. Therefore the label fields need not be restored to another context.

Enlarging the scope of a block (or program unit) is accomplished by merging adjacent components in the component graph. Component merging to enlarge a block can be illustrated with the EXAMPLE specification. Figure 6.1a shows a portion of the component graph for the specification (the file nodes have been omitted). Each component contains a single node. Application of the simple scheduling algorithm to this graph results in a data flow template with two block description entries in addition to the outer block (the outer block is the block representing the entire program). In one block, Assertion 1 is the sole member, and X is the data node being defined. In the other block, Assertion 2 is the sole member, and C is the data node being defined. If, however, adjacent components are merged to create a single component containing multiple nodes, the data flow template contains only one block. Figure 6.1b illustrates the component graph with merged components. If this component is given as parameter to

SCHEDULE_COMPONENT, the procedure returns two block members in the data flow template entry, one corresponding to Assertion 1, and the other corresponding to Assertion 2. This block is a larger unit of allocation than the two blocks produced from the component graph without merged components. Figure 6.2 shows the block description entries produced from the component graph of Figure 6.1b.

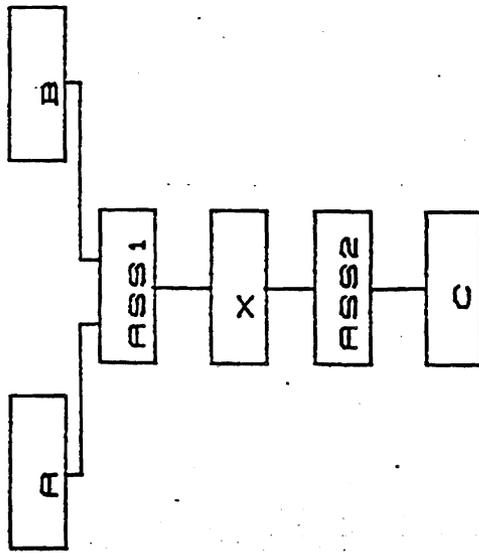


FIGURE 8.1A ORIGINAL COMPONENT GRAPH FOR EXAMPLE

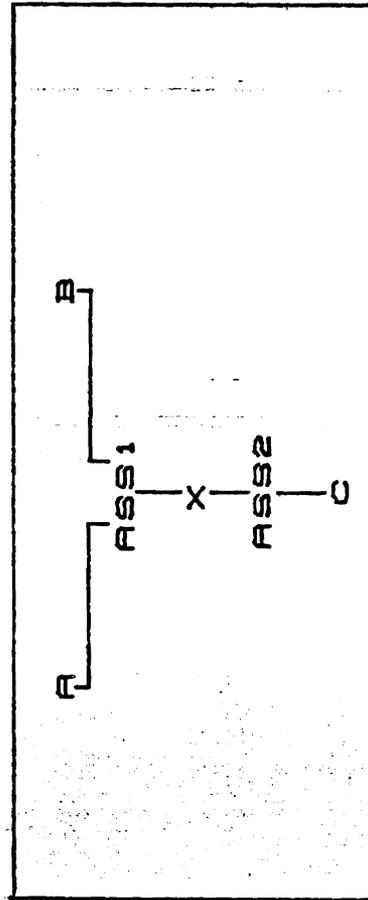


FIGURE 8.1B MERGED COMPONENT GRAPH FOR EXAMPLE

File Description: {INFILE1, INFILE2, OUTFILE, INTERIM}

Block Description:

Block 1: Type=Simple Level=0 Range=0

Data Nodes: None

Block Members: {(m_name=Block 2 m_type=block)}

Block 2: Type=Parallel Level=1 Range=1

Data Nodes: {(d_name=X d_dim=1) (d_name=C d_dim=1)}

Block Members: {(m_name=Assertion 1 m_type=assertion)
(m_name=Assertion 2 m_type=assertion)}

Figure 6.2 The New Template for EXAMPLE

There is a trade-off involved in merging components of whether to merge a component containing cycles with a cycle-free component. In many cases, an iterative block is generated from a cycle-containing component. A parallel block is generated from a cycle-free component. If the two types of components are merged, the block generated from the resultant component is iterative. Merging the two types of components enlarges the scope of the generated block, which is a desirable optimization. However, the block so created is an iterative block. Potential parallel computation from the cycle-free component could not be exploited on some data flow machines. Merging the two could, therefore, result in a decrease of parallelism in the data flow template, and in the generated program. The scheduler's objective is to provide as much information as is available at compile time in the data flow template. Combining an iterative and parallel block into one iterative block causes information about parallelism in the

program to be lost. If the two are kept distinct, the information is still available. The run time processor allocation program can still allocate an iterative block and a parallel block to the same processor if a run time evaluation of locality issues so dictates. The scheduler therefore does not merge a cycle-containing component (which might produce an iterative block) with a cycle-free component (potentially a parallel block).

VI.3 DATA STRUCTURE SIMPLIFICATION

The second area of optimization is data structure simplification. Data flow machines, which typically support the transmission of data values of elementary type, need special hardware to handle data structures. The support hardware may take the form of an auxiliary structure controller or of additional machine instructions to manipulate structured data. (See Section II.3). The data flow scheduler, therefore attempts to simplify the data structure of variables declared in the specification. The scheduler simplifies a variable's data structure by reducing the number of elements required in the generated program to represent a node dimension.

A data node dimension is defined to be physical if the dimension is mapped to a stream, and the number of elements in the stream is equal to the range of the dimension. A data node dimension is defined to be virtual if the dimension is mapped to a "window" of elements, and the width of the window is smaller than the range of that

dimension. If a dimension can be recognized to be virtual, there may be considerable savings in the generated data flow program. There is substantial cost (demonstrated on the Manchester machine) in creating and accessing a stream.

The MaD language defines two varieties of streams: token streams and stored streams. A token stream can be thought of as a number of tokens traveling down the same arc [Bowe81]. The tokens are distinguished by the index field of the token label. A stored stream follows more along the lines of a conventional array. The stream is stored in Matching Store and represented by a single "context" token, or pointer. Recognizing that a dimension of either type of stream is virtual may produce a more efficient data flow program.

The cost of processing a token stream is as follows: When the stream is created, the index field of the label of each element must be initialized. To select an element from the stream, a new stream must be created to hold the index of the element selected. This new stream, which has as many elements as the token stream, is matched against the token stream. When the value of the index stream matches the index of the data stream, the element has been located. Therefore, it is advantageous to have as few elements as possible in the token stream. If a data node dimension can be identified as virtual in the least order dimension, then it may be possible to represent the node in the data flow program as one or more scalars rather than as a stream.

It is also advantageous to find virtual dimensions in a node which is generated into a stored stream. A stored stream occupies space in the Matching Store. If a data node dimension can be found to be virtual, fewer storage locations are needed in the Matching Store. This is beneficial, since overflow of the Matching Store is not recoverable.

VI.3.1 Virtual Dimensions For Local Data Nodes

One example of a variable with a virtual dimension is the data node X in Figure 6.2. X is local to Block BD1. A data node is local to a block if the data node is produced by an assertion in the block, and the node is used only by assertions within the block. Since X is produced by Assertion 1, which a member of BD1, and X is used by Assertion 2, also a member of BD1, X is local to BD1. There are as many instances of Block BD1 as the range of I. Only the instance of X corresponding to the block instance is needed in the block instance. Only X(7) is needed in the seventh instance of BD1. Therefore the dimension of X associated with the range set of I can be marked virtual. In the data flow program which is generated from the template, X can be declared as a scalar local to BD1.

VI.3.2 Virtual Dimensions In Iterative Blocks

Another example in which a data node dimension is virtual is within an iterative block. If a data node is defined by a recurrence relation, that is, each element of the array is defined in terms of elements of lower index, and only the last element of the array is used in other assertions, then the dimension corresponding to the recurrence may be marked virtual. The Factorial example illustrates this situation:

```
Assertion 3: FACTORIAL(I) = IF I = 1 THEN 1;  
              ELSE I * FACTORIAL(I-1);
```

```
Assertion 4: OUTT = FACTORIAL(SIZE.FACTORIAL);
```

Only the final value of the factorial iteration is needed to define the value of OUTT. Therefore, the dimension may be marked virtual. In this case, the subscript expression in Assertion 3 is Type 2, "I-1". This indicates that only two successive elements of the array are required at any one time. Therefore, a "window" of 2 array elements is required. In general, a window of k+1 elements is required for "I-k" subscript expressions.

VI.4 EXPERIENCE WITH THE MANCHESTER MACHINE

Experiments done as a part of this work on the Manchester data flow machine confirm that block enlargement and data structure simplification result in more efficient data flow programs. One series of experiments were performed with specification EXAMPLE. The data flow template based on Figure 5.4 was translated to MaD and compiled. The execution run resulted in the following run time statistics:

```
TOTAL NUMBER OF INSTRUCTIONS EXECUTED : S1 = 624
ASSUMING : 1.UNLIMITED NO.OF PROCESSORS
          2.ALL INSTRUCTION EXECUTION TIMES = 1 STEP
TOTAL NUMBER OF PROCESSING STEPS : SINP = 148
AVERAGE PARALLELISM OF THE PROGRAM : S1/SINP = 4
          9 RESULT TOKENS WRITTEN
          865 TOKENS PASSING THROUGH THE RESULT QUEUE
```

Then the template from Figure 6.2 was translated to MaD, compiled, and run on the emulator. In this template, the block was enlarged to include both assertions; X was recognized to be a local data node; and dimension 1 of X was marked virtual. The following results were obtained from running the second version of the program:

```
TOTAL NUMBER OF INSTRUCTIONS EXECUTED : S1 = 446
ASSUMING : 1.UNLIMITED NO.OF PROCESSORS
          2.ALL INSTRUCTION EXECUTION TIMES = 1 STEP
TOTAL NUMBER OF PROCESSING STEPS : SINP = 133
AVERAGE PARALLELISM OF THE PROGRAM : S1/SINP = 3
          9 RESULT TOKENS WRITTEN
          616 TOKENS PASSING THROUGH THE RESULT QUEUE
```

The second program executed 178 fewer instructions (a 28% improvement) and completed in 15 fewer processing steps (a 10% improvement). 249 fewer tokens passed through the token queue (a 28% improvement). Enlarging the block and simplifying the data structure of X resulted in a computation which completed faster (fewer processing steps), required that fewer instructions be executed, and generated fewer tokens to flow through the ring. Similar results were obtained with the Factorial program.

In the following section, the algorithms used to achieve the two optimizations discussed here, block enlargement and data structure simplification, are described.

VI.5 THE MODIFIED SCHEDULING ALGORITHM

The SCHEDULE procedure described in the simple scheduling algorithm is also used in the new scheduling algorithm. Initialization is carried out as in the simple algorithm. SCHEDULE calls SCHEDULE_GRAPH with

- 1) the component graph representing the entire specification,
- 2) a block description entry for the outermost block, and
- 3) a nesting level of 1

as call parameters. A new version of the procedure SCHEDULE_GRAPH incorporates the algorithms to do block enlargement and data structure simplification.

VI.5.1 Criteria For Merging Components

SCHEDULE_GRAPH attempts to enlarge the scope of blocks in the template by merging adjacent components of the component graph. Components C and C' are said to be adjacent if there is an edge $C \rightarrow C'$ or an edge $C' \rightarrow C$. Let G be the component graph. Each member C of G is an MSCC. If $|C| > 1$, then the component graph contains a cycle. There is a path from any node in C to every other node in C . For example, the array graph for Assertion 3 above is shown in Figure 6.3a. The graph contains a cycle because Factorial is both source to the assertion and a target of the assertion. Since a node represents an entire array instead of an array element, there appears to be a cyclic dependency. Further analysis of the subscript expressions reveals that the cyclic dependency does not exist in the Underlying Graph, as illustrated in Figure 6.3b. The program generated from this specification should compute the array iteratively from index 1 to the range of I . In many cases, a cycle in the array graph indicates the possibility of iterative computation, a computation in which the value of an array element depends on the values of elements of the array of lower indices. Since a cycle-containing component is not to be merged with other components, only components C with $|C| = 1$ are candidates for merger.

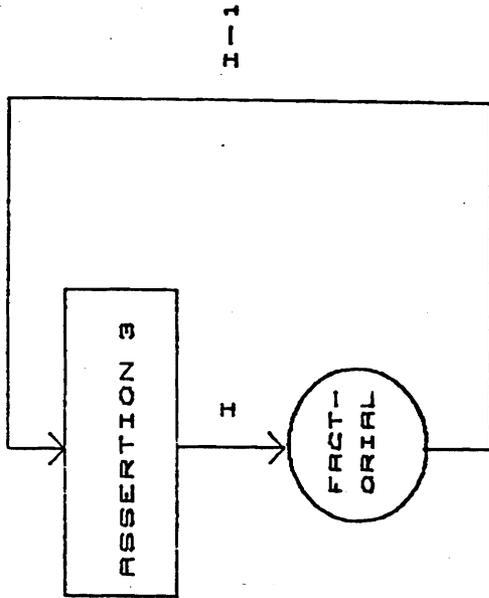


FIGURE 8.3A ARRAY GRAPH FOR ASSERTION 3

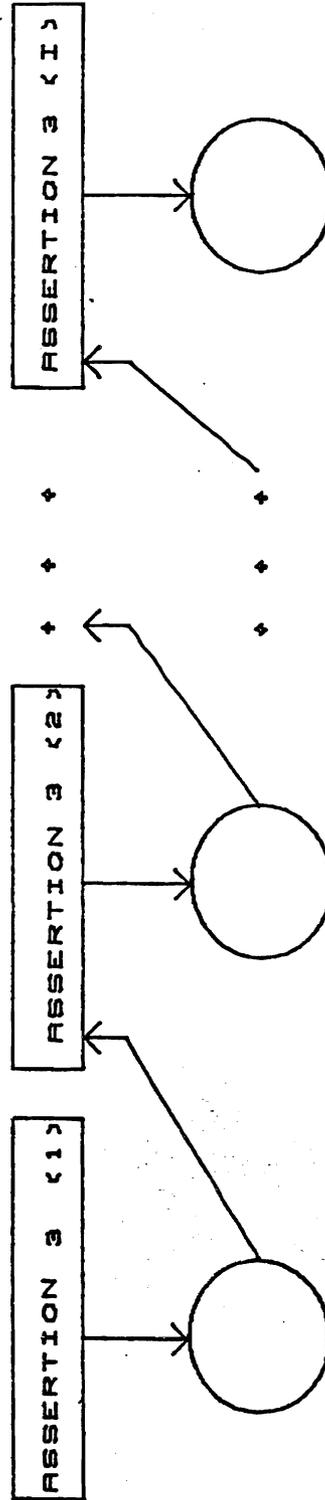


FIGURE 8.3B UNDERLYING GRAPH FOR ASSERTION 3

Components are merged by adding new edges to the component graph.

If C1 and C2 are two components such that

1. $|C1| = |C2| = 1$
2. There is an edge from N2 in C2 to N1 in C1 (N2 \rightarrow N1)
3. Adding the edge does not cause an iterative computation to be merged with a parallel computation

then a new edge is added from N1 to N2 (N1 \rightarrow N2). This will cause N1 and N2 to be in the same MSCC. There is an edge N2 \rightarrow N1 in the original graph. When a new edge N1 \rightarrow N2 is added, a cycle is formed. As successive pairs of single node components are considered, the cycle may be enlarged, enlarging the number of elements in the component and, therefore, the number of members in the parallel block generated from the component.

It should be noted that a distinction is made between MSCC's in the original array graph and MSCC's caused by new edges being added to the graph to merge single node components. MSCC's in the original graph reflect data dependencies of the specification. These MSCC's usually indicate iterative computation of successive array elements. The MSCC's caused by new edges added to the graph are components from which parallel blocks are generated.

Condition 3 ensures that each time an edge is added to the graph, addition of the edge does not cause the new MSCC being constructed to be merged with an MSCC of the original graph. Addition of the edge under such circumstances would cause an iterative block and a parallel

block to be merged into one iterative block. As discussed above, iterative blocks are not merged with parallel blocks, even though doing so increases the number of elements in the block. It is considered more important to retain the distinction between iterative and parallel blocks in the generated program than to increase the number of elements in an iterative block.

Let N_1 and N_2 be the single nodes in C_1 and C_2 respectively ($|C_1| = |C_2| = 1$) such that there is a path from N_1 to N_2 in the component graph ($N_1 \rightarrow^+ N_2$) which does not go through a node in a multi-node MSCC from the original graph. Let M be any MSCC in the original graph. Consider the following three relationships which could hold between $\{N_1, N_2\}$ and M .

1. There is an edge from M to N_1 and an edge from M to N_2 .
2. There are edges from both N_1 and N_2 to M .
3. There is an edge from N_1 to M and from M to N_2 .

Figures 6.4a, 6.4b, and 6.4c illustrate these relationships. The fourth relationship, an edge from N_2 to M , and an edge from M to N_1 could not occur in the component graph. If it did, there would be a path from N_2 back to itself and from N_1 back to itself, and M would contain these nodes in the first place.

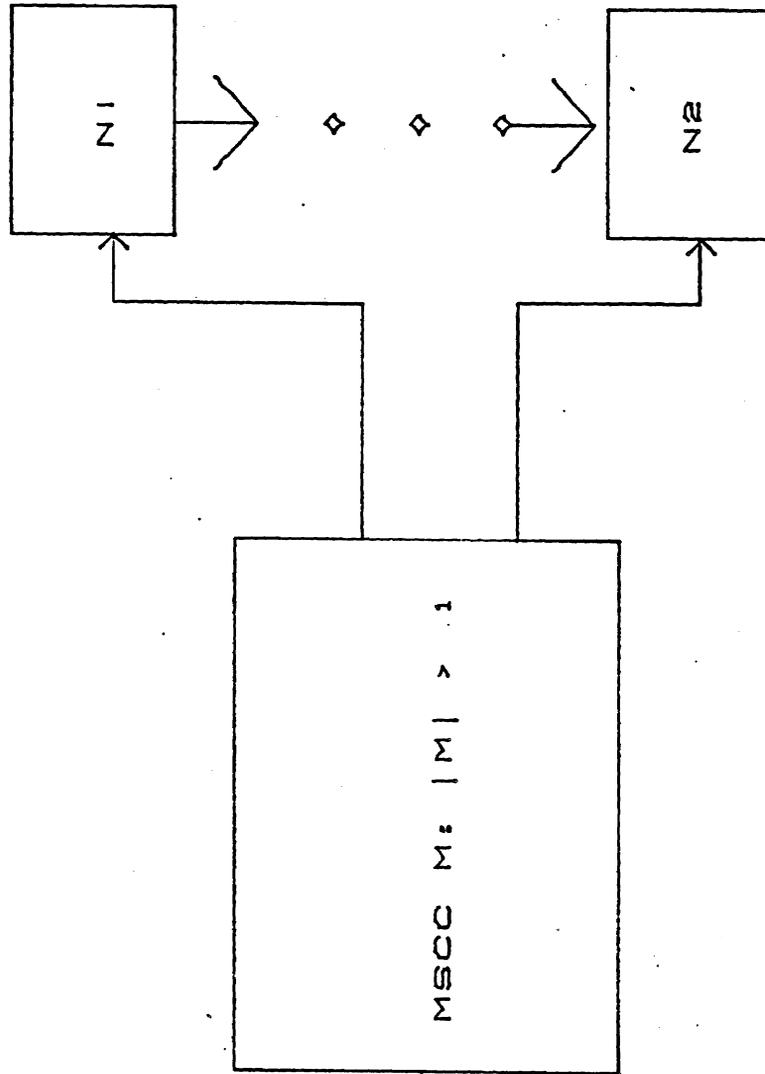


FIGURE B.4A EDGES FROM MSCC TO SINGLE NODE COMPONENTS

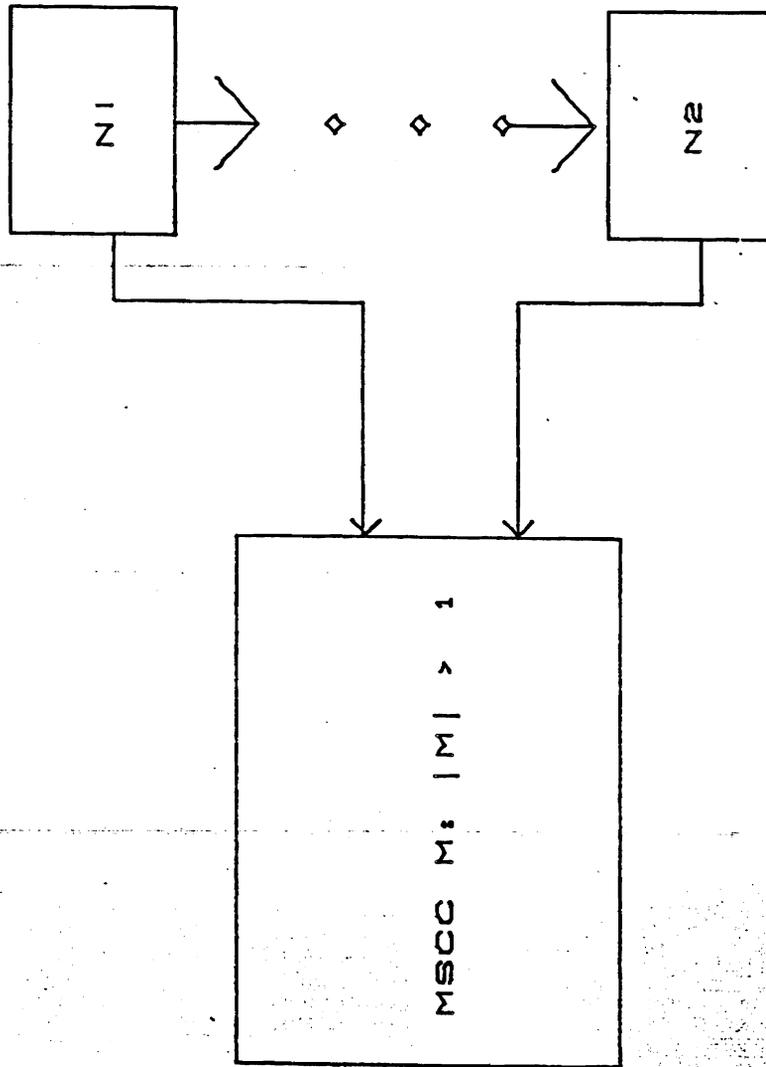


FIGURE 6.4B EDGES TO MSCC FROM SINGLE NODE COMPONENTS

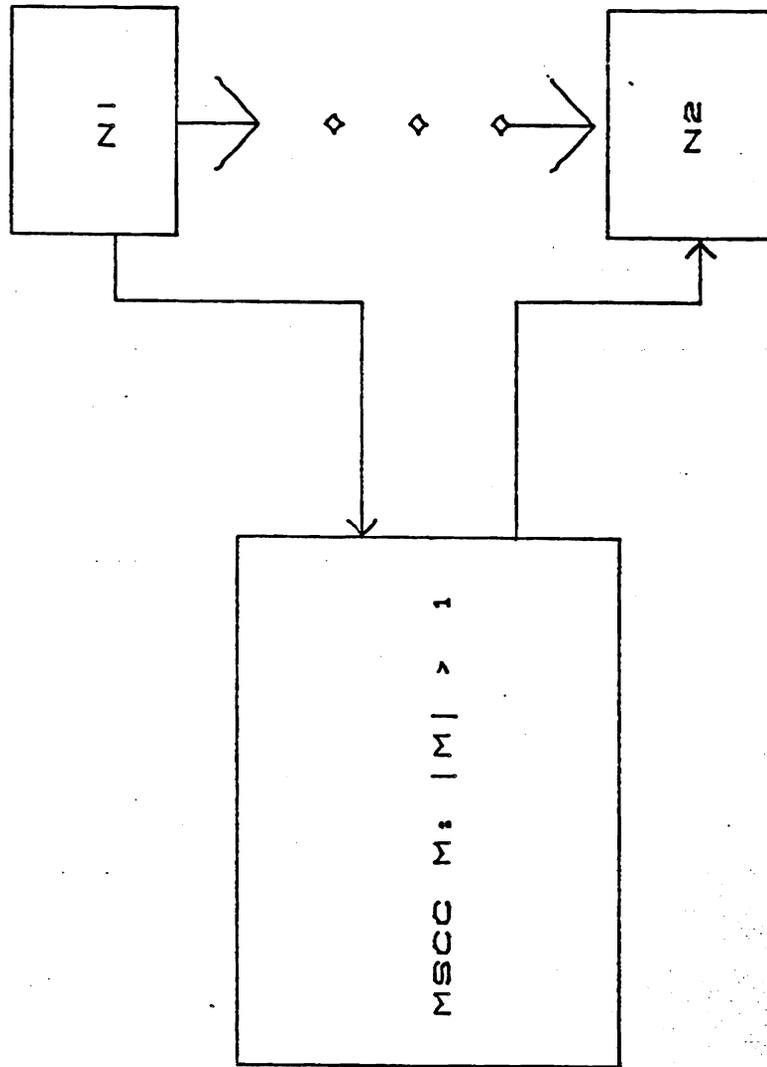


FIGURE 6.40 EDGES BETWEEN MSC AND SINGLE NODE COMPONENTS

Next, consider the effect on the component graph of adding edges so that for each edge in the chain from $N1 \rightarrow N2$, a new edge is added in the reverse direction. This creates a new chain from $N2 \rightarrow N1$. If Case 1 above applies, then adding such a chain will not result in a cycle between the new MSCC containing $\{N1, N2\}$ and M . New edges are only added between nodes in single node components, so a new edge cannot be added from $N1$ to M . Similarly for Case 2, creating an MSCC containing $\{N1, N2\}$ does not cause a cycle between the new MSCC and M .

However, Case 3 poses a problem. If a chain is added back from $N2$ to $N1$, then since there is already an edge from $N1$ to M , a new path is created from $N1$ back to itself, and from $N2$ back to itself. A new MSCC is created which contains both M and the chain $N1 \rightarrow N2$. An MSCC which is to be part of a parallel block is merged with an MSCC for an iterative block. Omitting any one edge in the chain from $N2$ back to $N1$ is sufficient to prevent this condition. The scheduler (arbitrarily) omits the edge from the successor of $N1$ back to $N1$.

VI.5.2 Adding Edges To The Component Graph

`SCHEDULE_GRAPH` first finds the MSCC's of the component graph, as in the simple scheduling algorithm. The procedure then scans each component, starting with components with no successors and ending with components with no predecessors. Let $N0$ be a node of a component C of G , where $|C| = 1$. The scheduler forms the predecessor set of C , $P(C)$. $P(C)$ consists of all other components $C2$ of G such that

1. $|C_2| = 1$
2. There is an edge in the array graph from the node N_2 in C_2 to the node N in C
3. Adding an edge to the component graph $N \rightarrow N_2$ will not cause N and N_2 to become part of one of the original multi-node MSCC's.

From $P(C)$, the scheduler forms a set of candidate components $I(C)$ for inclusion with C in the component being formed. A predecessor C_2 is added to $I(C)$ if all edges from the node in C_2 , N_2 , to the node in C , N , have a Type 1 subscript expression from one or more unscheduled dimensions D_{2i} in N_2 to the corresponding unscheduled dimensions D_i in N . In addition, each pair D_i and D_{2i} must belong to the same range set R . C_2 is said to be a candidate for inclusion in a common component with C for the range R .

A node has zero or more dimensions. Each dimension belongs to a range set. Let the set of range sets associated with a node N be called R_N . Let R_C refer to the set of range sets associated with a component C . In general,

$$R_C = \text{Union}(R_{N_i}), \text{ where } N_i \text{ is in } C.$$

In our case, $R_C = R_N$ when N is in C , because we only consider components C with $|C|=1$. Each member of $I(C)$, C_j , has a set of range sets R_{C_j} associated with C_j . Let

$$\underline{R_{C_j}} = \text{Intersection}(R_{C_j}, R_C).$$

That is, $\underline{R_{C_j}}$ contains those range sets which are in both R_{C_j} and in R_C . There may be cases in which there exists a range set R_1 such that

R_l in RC_i ; R_l not in RC_j

C_i, C_j in $I(C)$.

For example,

Assertion 4: $A(I, J, K) = X(I, K) + Y(K, J)$

Let R_X and R_Y refer to the set of range sets associated with the components containing X and Y respectively. Let C be the component containing Assertion 4. Then R_X contains the range sets of I and K ($\{I, K\}$), and R_Y contains the range sets of J and K ($\{J, K\}$). R_X and R_Y overlap. The scheduler must resolve this overlap. Assertion 4 cannot be in a common component with X for the range of I and at the same time in a common component with Y for the range of J . Doing so would place Y in a common component with X for the range of I . Since Y does not have a dimension whose range set is the range of I , it would be impossible to schedule Y in the block with range I .

This overlap occurs when

$\text{Intersection}(RC_j) \subset \text{Union}(RC_i)$.

To resolve the situation, the scheduler computes $\text{Intersection}(RC_j)$. It then retains in $I(C)$ those components C_l such that

$\text{Intersection}(RC_j)$ is contained in RC_l ,

and retains as the common ranges only those ranges in $\text{Intersection}(C_j)$. For the example above, the common set of range sets is $\{K\}$, and the candidates for inclusion with C in a common component

for the range of K are {X,Y}.

The scheduler then adds edges to the component graph. Adding edges changes the connectivity properties of the graph. In particular adding edges causes new multi-node MSCC's to appear in the component graph. Parallel blocks in the data flow template are generated from these new multi-node MSCC's.

For each component C2 in the inclusion set $I(C)$, the scheduler adds an edge from N in C to N2 in C2. The edge type indicates that this is a "back" edge, that is, an edge back from the target of an existing edge to the source of the edge. Adding this edge creates a new cycle in G. The subscript expressions for the dimensions in common are marked as Type 2 (I-1), so that they can be recognized in later processing as a pseudo rather than true cyclic dependency.

Once the new edges are added, the graph can be decomposed once more into MSCC's. If back edges were added, then new MSCC's will have been formed. Each component is now given as parameter to SCHEDULE_COMPONENT. This procedure is the same as described for the simple scheduling algorithm, with two exceptions. SCHEDULE_COMPONENT now recognizes "back" edges and creates a parallel block from a component which contains these edges.

The other change to SCHEDULE_COMPONENT is that data structure simplification, the second optimization goal, is also performed in this procedure. Once a new block has been created (Case 2 of SCHEDULE_COMPONENT), data nodes in the component from which the block

is formed are examined to locate virtual subscript dimensions.

The component graph for EXAMPLE obtained from the new SCHEDULE_GRAPH is as follows:

Component 1 has nodes: OUTFILE

Component 2 has nodes: INTERIM.X0

Component 3 has nodes: AASS220 INFILE1.A, INFILE1.INREC1,
INFILE2.B, INFILE2.INREC2, INTERIM.X,
AASS230, OUTFILE.C, OUTFILE.OUTREC

Component 4 has nodes: INFILE2

Component 5 has nodes: INFILE1

Component 6 has nodes: EXAMPLE

VI.5.3 The Revised SCHEDULE_GRAPH

The component enlargement analysis is performed in SCHEDULE_GRAPH. The revised SCHEDULE_GRAPH performs the following steps:

1. Input to the procedure is a component graph. The format of the component graph is as described in Chapter 5.
2. Build a list of predecessors for each component. If there is an edge E from Component C2 to Component C, then C2 is a member of CPREDS(C), where CPREDS(i) consists of the predecessors of Component i.

Remove from CPREDS components for whom adding an edge back from C to C2 would cause the new MSCC to be merged with an MSCC in the original

A predecessor is added to Candidate if 1) the predecessor forms a single node MSCC, and 2) adding an edge from C to the predecessor will not cause a merger of the new component with a multi-node component from the original component graph. ndim is the number of node dimensions. range_num is the range set number for the dimension. one_cand is a bit vector. A true value for an element of one_cand indicates that the indexed component is a candidate for inclusion in a parallel block with the current component for the indicated range. The Candidate structure is built as follows:

```
For each dimension I of N in C, the node being processed, do
{ for each member Ci of CPREDS do
  { Let Ni be the name of the node in Ci.
    Locate a dimension I' in Ni with the
      same range as I, R(I).
    If such a dimension should exist then do
    { Look at the edges from Ni to NO.
      If all edges Ei have a Type 1 subscript expression
        at the I' dimension then add Ci to
        Candidate at the dimension I.
    }
  }
}
```

4. Form the intersection of the candidate ranges. The intersection data structure consists of two parts.

```
dcl 01 intersec,
    02 i_ranges (ndim) fixed bin,
    02 i_sec (comp_cnt) bit(1);
```

This data structure is built as follows:

```
{For i ranging from 1 to ndim,  
  {for j ranging from 1 to comp_cnt,  
    if candidate(i).one_cand(j) = '1'b then  
      if, for all k ranging from 1 to ndim,  
        candidate(k).one_cand(j) = '1'b  
      then  
        {set i_ranges(i)=candidate(i).range_num;  
          set i_sec(j)='1'b;  
        }  
      else set i_ranges(i)=0;  
    }  
  }  
}
```

5. Check to see whether any members of `intersec` must be discarded because of the partial order relation over range sets. Suppose there is an edge from N_i to N , where N_i is in C_i and C_i is a member of `intersec`, and where the edge contains a Type 1 subscript expression for dimension I' of N_i corresponding to I of N . Now suppose that I'' is another dimension of N_i and the range set corresponding to I'' , $R(I'')$, precedes the range set corresponding to I' , $R(I')$. That is, suppose $R(I'')$ must be defined before $R(I')$ can be known. This would occur if the `SIZE` or `END` qualifiers were used to define I' , and I'' were an argument to the `SIZE` or `END` expression. If I'' cannot be scheduled before I , then C_i must be removed from `intersec` for the dimension I . This information is gathered by examining the `ralp` field of the `LOCAL_SUB` corresponding to I' . `ralp` points to a list of other dimensions of the node N_i which must precede the dimension I' . The following check is performed for each entry in the `ralp` list. The dimension referenced in the `ralp` entry precedes (in the partial order on range sets) the dimension being processed .

For each "ralp" entry in the list, do the following:

```
{ If the dimension in the ralp entry has been
  scheduled then continue.
  If the dimension has not been scheduled, and the
  range corresponding to that dimension is not a
  member of intersec, then discard the node and exit.
}
```

If all dimensions in the ralp list pass,
then keep the node in the intersection set.

6. For each candidate which is left in intersec, insert a new edge from the node N to Ni, the single node in Ci. This "back" edge will cause an MSCC to be formed which will include nodes N and Ni. The MSCC will be the basis for a parallel block.

7. Once back edges have been inserted for each eligible node, the array graph is again divided into MSCC's. Each MSCC is then submitted to the procedure SCHEDULE_COMPONENT.

8. SCHEDULE_COMPONENT adds a member M to the current block description entry. If M is of type "block", SCHEDULE_COMPONENT calls FindVirtual(M) (described below) to mark virtual dimensions of data nodes defined in M.

9. When all components have been scheduled, SCHEDULE_GRAPH return the composite schedule.

VI.5.4 Locating Virtual Dimensions

The procedure FindVirtual is called by SCHEDULE_COMPONENT to locate virtual dimensions of data nodes. Input to FindVirtual is the block description entry B of the block member constructed by SCHEDULE_COMPONENT. Let R be the range for B. FindVirtual performs two functions.

1. If the block type of B is iterative, look for virtual dimensions of data nodes defined in B.

For each data node defined in B, let D be the name of the data node. Mark the dimension of D corresponding to R as virtual if each edge from D (source) to an assertion (target) is in the following form:

- 1a. The edge has a subscript expression of Type 1, 2 or 3 in the distinguished dimension and the target is in B,

and, optionally,

- 1b. The edge has a Type 4 subscript expression in the distinguished dimension, and the subscript expression is SIZE.<name of D>. This edge indicates that the target depends only on the last element of D.

2. Construct a table, Local, of local data nodes. Each entry of the table has two fields: Node_id, the data node id; and Block_ix, the index of the block to which Node_id is local.

For each data node defined in B, let D be the name of the data node. D is a local data node if for each edge from D to an assertion, the assertion is also a member of B. If D is found to be a local data

node, then do the following:

If every dimension of D has been scheduled, and each edge from D to an assertion has a Type 1 subscript expression in every dimension, then add (D, Block Number of B) to Local.

VI.6 CONCLUSION

This concludes the discussion of efficiency considerations in scheduling for a data flow machine. We have described algorithms which enlarge the scope of generated blocks and which simplify the structure of data. The topic of the next chapter is code generation, the problem of generating from the data flow template a program in the MaD language.

CHAPTER VII

CODE GENERATION FOR THE MANCHESTER MACHINE

VII.1 INTRODUCTION

The data flow template produced by the scheduler represents a language- and machine-independent form of the Model specification. The template is input to the code generation phase in which the template is translated to a specific language and machine. In this work, the data flow template is translated to the Manchester Data flow language (MaD) for the Manchester data flow machine. The MaD language is described in Chapter 3, and the Manchester data flow machine in Chapter 2.

The generated program is divided into three parts: global declarations, "assignment" statements, and a return statement. MaD follows the requirements of Pascal in compulsory data declaration and strong typing [Jens79]. The MaD program header resembles a Pascal function declaration. The program name is followed by a list of input

parameters and their types. Next is the data type of the result returned by the program. After the program header, the variables used in the program and their data types are listed in the global declarations section of the program. In this section, the variables used to hold the value returned by the program (called the output parameters in the following discussion); interim variables; and any special variables used in the specification (such as END or SIZE variables) are declared. After the data declarations come the "assignment" statements. These statements define the values of the variables. An assignment statement may have as the right hand side a simple arithmetic expression or a more complicated MaD block. The assignment statements are generated systematically from the block description section of the template. If a member of the block being processed is an assertion, an assignment statement of the simple sort is generated. If the block member is itself of type block, an assignment statement with a right hand side of consisting of a MaD block is generated. A block member of type block is then processed recursively. In this way, blocks nested to an arbitrary level may be generated. The final section of the MaD program generated by the Model Processor is the return statement. A MaD program must return a value upon termination. The value returned may be simple or composite. The return value is composed of the values computed by the assignment statements for the output parameters of the program. The form of the generated program is shown in Figure 7.1.

- I. Declarations
 - A. The Program Header
 - 1. Program Name
 - 2. Input Parameter Names and Types
 - 3. Result Type
 - B. Global Declarations
 - 1. Interim and Special Variable Names and Types
 - 2. Output Parameter Names and Types
- II. Assignment Statements
 - A. Simple Assignments
 - B. Nested Blocks
- III. Return Statement

Figure 7.1 The Generated Program Structure

To demonstrate the correspondence between data flow template and MaD program, the structure of the MaD program generated for the EXAMPLE template (Figure 6.2) is shown in Figure 7.2. The EXAMPLE template contains the following information:

- 1) The file description section of the template contains two input files containing repeating fields A and B respectively, one output file containing repeating field C, and an interim file containing the repeating field X.
- 2) There are two blocks in the block description section of the template. The first is the outer block, representing the entire program. The second block is nested in the outer block. The two assertions are nested within the second block.

I. Declarations

A. The Program Header

1. Program Name: EXAMPLE
2. Input Parameter Names and Types:
Integer streams A and B
3. Result Type: Integer stream

B. Global Declarations

1. Interim Variable Names and Types:
None (since X is local to
a nested block)
2. Output Parameter Names and Types: Integer stream C

II. Assignment Statements

A. Simple Assignments: None

- B. Nested Blocks: There is one nested block.
It contains assignment statements for X and C.

III. Return Statement:

the value of C is returned as the program result.

Figure 7.2 Structure of the EXAMPLE Program

The following sections describe the algorithms used to transform the data flow template into a MaD language program. The next section describes limitations of the MaD language which are more restrictive than the Model language. Then, an overview of the code generation phase is presented. Each part of code generation is discussed: generating the data declarations, the assignment statements, and the return statement.

VII.2 RESTRICTIONS OF THE MAD LANGUAGE

MaD has several limitations which are more restrictive than those of the Model System. MaD permits dimensions of a multi-dimensional structure to be constructed only in a hierarchical sequence. The least significant dimension must be defined first, followed by the next least significant dimension, and so on. If $M(\text{dim1}, \text{dim2})$ is a matrix, the restriction dictates that every element of the first row of M must be defined before an element of the second row is defined. This restricts the flexibility of algorithm construction rather than the scope of algorithm which can be defined in MaD. The Model data flow scheduler chooses any unscheduled dimension of a component from which to construct a block. The choice may be limited by precedence relations among range sets. However, if a selection is not limited by such precedence relationships, the scheduler does not restrict the nested block structure to be a hierarchical definition of node dimensions as does MaD. It does not require that a node's dimension be defined from least significant to most. Therefore, some valid data flow templates produced by the scheduler may not be translatable to MaD.

A second restriction is that the input and output parameters of a program may not be of generalized tree structure. There can be only one leaf node in the tree structure for the parameter. Thus, although a parameter may be multi-dimensional, it must be of elementary base type. Because of this restriction, Model specifications which require

input and output data formatted as generalized trees cannot be translated into MaD.

The third limitation of MaD which affects translation is that a complete dimension of a multi-dimensional structure must be defined in one expression. Individual elements may not be defined separately. For example, if A is a one-dimensional array, a definition of the form "A(6) := 15" is not permitted. Instead, the definition must take the form "A := <expression>", where the <expression> evaluates to a stream, the MaD equivalent of a Model one-dimensional array. Because of this restriction, Model assertions with generalized subscript expressions on the left hand side cannot be translated into MaD.

This restriction also means that an entire record or group must be defined by a single expression. A data flow template in which all the components of a record or group are not defined within the same block is not directly translatable to MaD. Therefore, structured interim data is transformed to an equivalent but simpler form in the generated MaD program. The simple tree structure used for interim variables is the same as the structure required of input and output parameters in MaD (described above).

VII.3 ORGANIZATION OF THE CODE GENERATION PHASE

The procedure Codegen in the Model Processor is responsible for generating a MaD program from the data flow template. Data structures used by Codegen include the data flow template, the table Local

created by the scheduler, and the node attribute table or dictionary created prior to scheduling. Codegen calls on three auxiliary procedures to generate the data declarations, the assignment statements, and the return statement. The procedure GenDcl handles the declarations. Input to GenDcl are 1) the data description section of the data flow template, 2) the table Local, and 3) the node attribute table. GenDcl produces all the global declarations. The procedure GenBlk generates the assignment statements, both simple statements and statements which contain nested blocks. Input to GenBlk are 1) the block description of the data flow template, 2) the table Local, and 3) the node attribute table. GenBlk uses several procedures to generate parts of the assignment statements. Procedure GenAssr generates a simple assignment statement from an assertion. Procedure LocalVar generates local variable declarations in nested blocks. Procedure ForEach generates the body of a parallel block. Procedure Iter generates the body of an iterative block. The procedure Ret, called by GenBlk, ForEach, and Iter, generates a Return statement.

VII.4 GENERATING DATA DECLARATIONS

Declarations are generated for the program header, the interim variables, and the output parameters. The program header is generated first.

VII.4.1 The Program Header

The program header consists of the program name, input parameters, and output result type.

```
<programheader> ::= PROGRAM <program-id>
                  [ <parameterlist> ] <result> ';'
<parameterlist> ::= '(' <parmlist>
                  [ ';' <parmlist> ]* ')'
<parmlist>       ::= <parmid> [ ',' <parmid> ]* ':'
                  [ [STORED] STREAM [STREAM]* ]
                  <typeid>
<result>         ::= '[' <result> [ ',' <result> ]* ']'
                  | [ [STORED] STREAM ] <typeid>
```

Example:

```
PROGRAM FIBONACCI(N:INTEGER): STREAM INTEGER;
```

The program name is FIBONACCI. There is one input parameter, N, of type integer. The output parameter is of type stream, with the base type of the stream as integer.

The name for the program is taken from the name of the specification. Declarations for the input and output parameters are generated from the data description section of the data flow template. Data description entries referring to files of type SOURCE are used to generate the input parameter list. TARGET file entries are used to generate the output parameter list.

A file node is the root of a generalized tree. Recall from Chapter 4 that each data node has attributes SON1, the node number of the leftmost descendant of the node, and BROTHER1, the node number of the sibling to the immediate right of this node. By following the SON1 field of a file node and the BROTHER1 field of descendants of the file node, all the fields contained in a file may be accessed. The input and output parameters are generated from the descendants of the source and target file nodes respectively. In the restricted form of tree used for input and output parameters, each data node has zero or one son and zero brothers.

A MaD input parameter declaration is generated as follows:

The parameter name is the name of the leaf node of the tree whose root is an input file. The base type of the parameter is the data type associated with the leaf node. The dimensionality of the leaf node determines the number of "STREAM" prefixes to which the base type is appended. For example, for the Model data declaration

```
MAT IS RECORD (ROW (10));  
ROW IS GROUP (COL (10));  
COL IS FIELD INTEGER;
```

the MaD parameter name is COL; the base type is integer; and, since the dimensionality of COL, the leaf node, is two, there are two "STREAM" prefixes. The parameter declaration is as follows:

```
COL:  STREAM STREAM INTEGER
```

If a node dimension is virtual, then the STREAM prefix for that node

dimension is omitted in the parameter declaration. For example, if the Model declaration for an array used to compute the factorial of a number is

```
FAC IS FIELD (*) (FIXED BINARY);
```

and dimension 1 of FAC is found by the scheduler to be virtual, then the MaD declaration

```
FAC: INTEGER;
```

is generated instead of

```
FAC: STREAM INTEGER;
```

The output parameters are formed in the same way. However, only the data type of the output parameter is specified, not the parameter name. This is similar to the function result declaration in Pascal. If there is more than one TARGET file, then the output parameter list is enclosed in brackets, for example, [STREAM INTEGER, REAL]. This notation allows the program to return a composite result, a record. The first field of the record is a STREAM INTEGER; the second field is a REAL number.

VII.4.2 Data Declarations For Global Variables

After the program header is generated, GenDcl generates declarations for global variables. Variables in the interim "file" of the file description section of the template are processed. If such a variable is not local to a nested block, a MaD declaration for the variable is generated. Following these interim variables come

declarations for the variables which hold the program result. The target file descriptions in the template are processed to obtain these variables. The syntax of variable declarations in Mad is shown below. The <typedefn> refers to the data type of the variable.

```
<blockdeclarations> ::= <id> [ ',' <id> ]* :  
                        <typedefn> ';' ;  
                        [ <id> [ ',' <id> ]* :  
                        <typedefn> ';' ]*
```

VII.4.2.1 Interim Variables -

There is one entry in the data description portion of data flow template for the interim "file." Any variables declared in the Model specification which are not part of a source or target file are members of the interim "file." The data structure of an interim item in Model is also a generalized tree. However, the MaD STRUCT construct, which is provided to describe a variable whose data structure is a generalized tree, is not used in code generation. This is because there are several restrictions in MaD on definition and usage of fields in a data structure declared with a STRUCT data type. Instead, the tree for the interim variable is transformed to a set of restricted (as opposed to generalized) trees. Each restricted tree has the form of the tree for an input or output parameter, that is, a zero or greater level tree with a single leaf node. A data declaration is generated for each restricted tree using the method outlined above for input and output parameters. An example of data

declaration for interim variables is as follows:

Example:

Let an interim variable in a Model specification be declared as follows:

```
SCORES IS GROUP (100) (SCORE_1, SCORE_2);  
SCORE_1 IS FIELD (FIXED BINARY);  
SCORE_2 IS FIELD (FIXED BINARY);
```

The data structure of SCORES is illustrated in Figure 7.3a. This data structure is transformed into the data structure illustrated in Figure 7.3b. The MaD declaration generated for this example is as follows:

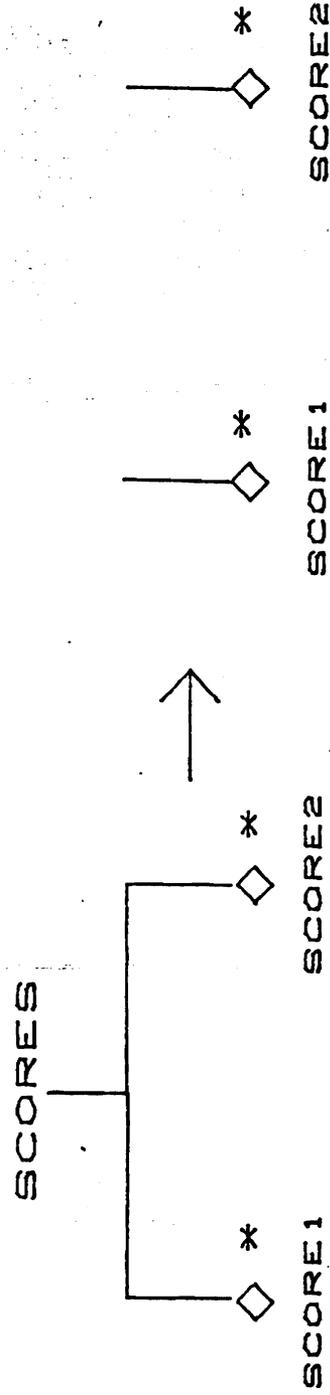
```
SCORE_1: STREAM INTEGER;  
SCORE_2: STREAM INTEGER;
```

Declarations for interim data which the scheduler found to be local to a particular block are not generated in the outer block. Instead, these declarations are generated in the block in which the data is defined. Given the template for specification EXAMPLE, the following declarations are generated in MaD:

```
PROGRAM EXAMPLE(  
    A: STREAM INTEGER;  
    B: STREAM INTEGER  
):STREAM INTEGER;  
  
DECLARE C: STREAM INTEGER;
```

A declaration is not generated for X, because X is a local data node.

GENERALIZED TREE TWO RESTRICTED TREES



7.3 A

7.3 B

* INDICATES MULTIPLE INSTANCES

FIGURE 7.3 TRANSFORMING A GENERALIZED TREE TO A SET OF RESTRICTED TREES

VII.4.2.2 Variables To Hold The Program Result -

The output parameters, which correspond to the target files in the specification, are declared next. Only the data types of these variables are declared in the <result> portion of the program header. For example, if a record for the target file in a Model specification is declared as follows:

```
RESULT IS RECORD (RES(50));
```

```
RES IS FIELD (INTEGER);
```

then the MaD output parameter declaration is `STREAM INTEGER`, and the declaration in the <blockdeclarations> is

```
RES: STREAM INTEGER
```

The output parameters are declared using the same method as is used to generate input parameters. The name associated with the leaf node is used as the variable name. The data type is generated from the leaf node data type as for the input parameters.

VII.5 GENERATING THE ASSIGNMENT STATEMENTS

The next part of the MaD program consists of a definition for each of the variables declared in the global declarations. This definition section is generated as one or more assignment statements, a MaD <let> clause. The format of a <let> clause is

```
<let> ::= LET <lhs> := <expression>  
        [ ; <lhs> := <expression> ]*
```

```
<lhs> ::= <variable id> |
```

```
'[ <variable id> [ ',' <variable id> ]* ]'
```

The second form of <lhs> is a composite definition. If this form is used, the expression on the right hand side must evaluate to a list of results. Each result in the list must match the data type of each component of the list on the left hand side. For example, if I1 is of type integer and R1 is of type real, then the following composite definition is correct:

```
[I1, R1] := [15, 0.9];
```

The <let> clause is generated from the block description portion of the data flow template. Each block description entry consists of two lists. The first is a list each element of which has two components, the data node defined in the block and which dimension of the data node has been defined. The second is a list of members of the block. A member can be either an assertion or another, nested block. If a block member is an assertion, then a MaD definition is generated from the assertion. Procedure GenAssr transforms the assertion into a simple assignment statement.

VII.5.1 Procedure GenAssr

GenAssr modifies the text of the assertion to conform to MaD syntax. For example, instead of the Model '=' separating the left and right hand sides of an assertion, a ':=' is used in the MaD form. The name of a variable used in the Model specification may be modified in

the MaD program. The MaD version of the variable's name might have fewer subscript qualifiers than the Model version. This would occur if one or more of the variable's dimensions were found by the scheduler to be virtual. Another difference is that a qualified variable is not permitted on the left hand side of a MaD definition. Therefore, subscripts are omitted from the variable on the left hand side of the assertion when the assertion is output as a MaD definition. MaD permits a qualifier 'NEW' for the variable being defined. The 'NEW' qualifier is used for variables defined in an iterative block. At each iteration, a new instance of the variable is defined. For example, if a variable FAC is defined by an assertion

```
FAC(I) = IF I = 1 THEN 1 ELSE I * FAC(I-1)
```

and dimension 1 of FAC is virtual, then the MaD equivalent is

```
NEW FAC := IF I = 1 THEN 1 ELSE I * FAC
```

VII.5.2 Procedure GenBlk

GenBlk generates a MaD block from a block description entry in the template. The MaD program outer block is constructed from the first block description entry. A MaD block is constructed in two steps. Simple assignment statements are generated first for data nodes defined directly in the block. Then, definition statements are generated for data nodes defined in nested blocks. In the latter form of assignment statement, the right hand side is a nested MaD block. Procedure Genblk, given a block description entry, constructs the MaD

VII.5.3 Procedure LocalVar

LocalVar is called by GenBlk with parameters Lhs, the list of data nodes defined in a block, and Level, the nesting level. This procedure generates the data declarations for the nested block on the right hand side of the definition.

Let D be an element of Lhs. Examine the table Local constructed by FindVirtual during scheduling. If D is in Local, then omit the declaration for D unless Level is equal to the dimensionality of D. This is done so that a local data node is only declared in the most deeply nested block in which it is produced and used.

Now consider the dimensionality of D. D is a leaf data node of dimensionality n. Level represents the most significant dimension of D which will be used to generate the declaration. For example, if Level is 2 and D has three dimension, D(I,J,K), then the second and third dimensions of D, those represented by J and K, are used to generate the declaration. The dimension corresponding to Level, that of J, is the most significant dimension. The dimensionality of the local copy of D is at most $(n - \text{Level}) + 1$. Let the dimensions of D be numbered from most significant to least significant. Examine the node subscripts of D from dimension # Level to dimension # n. If a dimension of D among those examined is virtual, the dimensionality of the local copy is reduced by one. For example, let Level = 1 and a data node FAC have dimensionality 2. The maximum dimensionality of the local copy of FAC is $(n - \text{Level}) + 1 = (2 - 1) + 1 = 2$. However, if the

most significant dimension of FAC is virtual, the dimensionality of the local copy of FAC is 1. The dimensionality of the variable determined by this calculation is reduced by one if the variable is not in the table Local.

The algorithm for LocalVar is as follows:

For each D in Lhs which is either non-local or is local to the current block,

Output the name of D followed by a ':'.

Initialize the dimension count to 0.
Examine each dimension of D,
starting at the most significant
dimension Level. If the dimension
is not virtual, add 1 to the
dimension count.

If D is not in the table Local, decrement
the dimension count of D by 1.

Output the data type of D, for example,
'INTEGER' or 'REAL'.

VII.5.4 Procedure ForEach

This procedure generates the "FOR EACH" statement for a parallel block. The call parameter to ForEach is the parallel block Block.

The form of the MaD <foreach> statement used is

```
<foreach> ::= 'FOR EACH' <variable name> 'IN' <stream> 'DO'  
           <lhs> ':=' <expression>  
           [ ';' <lhs> ':=' <expression> ]+  
           'RETURN' <expression>
```

ForEach first generates the block header, the first line of the

<foreach> definition above. The <variable name> is the <range set name> of the range set for Block. The <stream> consists of a sequence of indices, from 1 to the range set maximum. It is constructed by a standard procedure PROLIFERATE, which, given a maximum integer, returns a stream of integers from 1 to the maximum. The range set maximum can be defined in Model in one of three ways. The maximum can be a constant; it can be defined by the end-of-file condition; or it can be defined by a range array. If the maximum is constant, the constant is the input to PROLIFERATE. If the maximum is defined by end-of-file, then the MaD function SIZE(<stream id>) is used. SIZE, given the name of a stream, returns the number of elements in the stream. If the maximum is defined by the Model range array SIZE, then the range array is used as input to PROLIFERATE. If the maximum is defined by the Model range array END, then the input to PROLIFERATE is the number of elements in the dimension of the END array corresponding to the Block range set.

For example, let I be the name associated with a range set. Let the maximum for I be the number of elements in a file A consisting of a sequence of integers. Then the first part of the <foreach> statement in MaD is as follows:

```
FOR EACH I IN PROLIFERATE(SIZE(A)) DO
```

In this example, SIZE is the MaD function which returns the number of elements in A. PROLIFERATE creates a stream of indices from one to the number of elements in A.

The body of the <foreach> is generated by Genblk. The procedure ForEach calls Genblk(B) to generate the definition statements which constitute the body of B and the return statement.

VII.5.5 Procedure Iter

This procedure is called with input parameter Block, a block description entry for an iterative block. Iter generates the 'WHILE' statement for the iterative block. The form of the 'WHILE' statement used is as follows:

```
<while> ::= 'INIT' <range set name> ':= 1;'  
          'WHILE' <range set name> '<=' <range max> 'DO'  
          [ 'NEW' ] <lhs> := <expression>  
          [ ';' <lhs> := <expression> ]+  
          'RETURN' <expression>
```

The <range max> is found in the same way as described in ForEach above. The body of the iteration is generated by Genblk(B).

VII.6 GENERATING THE RETURN STATEMENT

Procedure Ret constructs the Return statement for a block. This procedure is called with Lhs, the list of data nodes generated in a block, and Level, the nesting level. For each element D of Lhs, Ret does the following:

1. If D is a local data node, then skip it.

2. If the dimension of D corresponding to Level is not virtual, then output 'ALL'.
3. Output the name of the local copy of D.

VII.7 CODE GENERATION FOR EXAMPLE

Generating the MaD assignment statements and the return statement from the data flow template is illustrated with the EXAMPLE template. The block description entries for EXAMPLE are shown in Figure 7.4.

Block Description:

```
Block 1: Type=Simple Level=0 Range=0
  Data Nodes: None
  Block Members: {(m_name=Block 2 m_type=block)}

Block 2: Type=Parallel Level=1 Range=1
  Data Nodes: {(d_name=X d_dim=1) (d_name=C d_dim=1)}
  Block Members: {(m_name=Assertion 1 m_type=assertion)
                  (m_name=Assertion 2 m_type=assertion)}
```

Figure 7.4 Block Description Entries for EXAMPLE

Procedure Genblk is called with parameter Block 1. There are no data nodes defined in Block 1, so Step 1 of Genblk is skipped. There is one member of Block 1 of type "block", so Step 2 is invoked. FindLhs is called with parameters Lhs=null and Block= Block 2. Findlhs returns a list Lhs (X,C) as the data nodes defined in Block 2. The left hand side of the MaD definition for Block 2 is generated from Lhs. The lhs is

C:=

X is not part of the lhs since it is a local data node.

Next, the right hand side of the definition is generated. "I" is the name of the subscript associated with the range set for Block 2.

```
DECLARE I: INTEGER;
```

Local declarations are generated for each entry in Lhs:

```
X: INTEGER; C: INTEGER;
```

Since Block 2 is of type parallel, Procedure ForEach is called with parameter Block 2. This procedure generates

```
FOR EACH I IN PROLIFERATE(100) DO
```

ForEach then calls Genblk recursively with Block=Block 2. Genblk now generates the definitions for the two assertions in Block 1 (Step 1 of Genblk). The output of Genblk:

```
X := A[I] + B[I];
```

```
C := X * X;
```

```
RETURN ALL C;
```

ForEach returns to Genblk. Genblk generates the RETURN statement for Block 1:

```
RETURN C;
```

The complete MaD program is shown below:

```
PROGRAM EXAMPLE(  
    A: STREAM INTEGER;  
    B: STREAM INTEGER  
): STREAM INTEGER;
```

```
DECLARE  
    C: STREAM INTEGER;
```

```
C := DECLARE
  I: INTEGER;
  X: INTEGER;
  C: INTEGER;
  FOR EACH I IN PROLIFERATE(100) DO
    X := A[I] + B[I];
    C := X * X;
  RETURN ALL C;

RETURN C

END
```

VII.8 CONCLUSION

This concludes the discussion of code generation to MaD. We have shown how the data flow template is translated to a MaD language program. The data description section of the template is used to generate the program header and global variable declarations. The block description section is processed recursively to generate the "assignment" statements, either simple assignments or nested blocks.

CHAPTER VIII

CONCLUSION

VIII.1 SUMMARY OF CONTRIBUTIONS

VIII.1.1 Desirability Of Nonprocedural Languages For Data Flow

In this investigation, we have demonstrated the use of a nonprocedural language as a very high level programming language for data flow computers. A nonprocedural language seems desirable as a way to specify a problem because such a language provides a powerful vehicle for concise description of problems. A nonprocedural language seems particularly well suited to data flow because, in a data flow environment, a schedule for a specification can be derived almost immediately from the dependency graph representation of the specification. However, by using the Manchester Emulator as a tool to test scheduling strategies, we have been able to develop optimizations which produce more efficient schedules than the one immediately available from the dependency graph.

VIII.1.2 Scheduling The Array Graph For Data Flow

The approach in scheduling has been to partition the array graph into Maximally Strongly Connected Components (MSCC). Iterative blocks are generated whenever possible for MSCC's derived from the original array graph. Remaining nodes with common ranges in corresponding dimensions are merged into graph components. From these components, parallel blocks are generated. Each parallel block can be expanded into multiple incarnations which can execute concurrently.

After the array graph has been partitioned, the Scheduler searches for dimensions of data nodes which can be virtual. Data nodes local to a block and data nodes produced by iterative blocks can potentially be reduced in dimension. Dimension reduction results in a more efficient data flow program.

VIII.1.3 Generating Data Flow Programs

We have shown how the template generated by the Scheduler may be used to generate programs to one specific data flow language, MaD. The generated program can be compiled and run on the Manchester Emulator.

VIII.2 FUTURE RESEARCH

Several promising avenues have presented themselves in the course of this investigation. We summarize these areas:

1. The current Model system is embedded in the PL/I programming environment. Creating a data flow version of the Model language with constructs more suited to a data flow environment would provide a good tool for programming data flow machines in a nonprocedural language.
2. In this work, we produce a data flow program partitioned into blocks. An interesting area of study would be the static allocation of these blocks to processors. Processor allocation would depend on data generation and usage. The array graph contains information which would facilitate this analysis.
3. Another interesting study would be a performance comparison of sequential vs. data flow programs produced by the same specifications.
4. Developing applications in Model suited to data flow is another area of research. Such algorithms as graph-theoretic problems and discrete event simulation could be written in Model and run on a data flow machine.

5. Model produces iterative and parallel blocks from the specification. An enhancement to Model currently being implemented is to produce whole specifications as blocks and to analyze the data dependencies among specifications. [ShiY82] describes this distributed processing version of Model.

APPENDIX A

MAD BNF

```
<programme> ::= PROGRAM <program-id> [ <parameterlist> ]
               <header> ';'
               [ <typedeclaration> ]
               [ <funcdefs> ]
               [ <expression> ]
               END
               [ <assembly-code> ]

<parameterlist> ::= '(' <parmlist> [ ';' <parmlist> ]* ')'
<parmlist>      ::= <parmid> [ ',' <parmid> ]* ':'
               [ [STORED] STREAM [STREAM]* ] <typeid>

<header>       ::= '[' <header> [ ',' <header> ]* ']' |
               [ [STORED] STREAM ] <typeid>

<typedeclaration> ::= TYPE <typeid> '=' <typedefn> ';'

<typedefn>    ::= [STORED] <structyp> |
               [STORED] <structypeid> |
               <typeid>

<structyp>    ::= STREAM [ <structypsys> ]* <structyp> |
               STREAM <structypeid> |
               STRUCT <gen> [ ';' <gen> ]* ENDSTRUCT

<structypsys> ::= STREAM | STRUCT | SET
```

```
<gen> ::= <generatorid>
        [ '(' <typ> [ ',' <typ> ]* ')' ]

<typ> ::= <structypsys> <structyp> | <typeid>

<funcdefns> ::= FUNCTION <funcid> <parameterlist> ':'
              <header> ';'
              [ <funcdefns> ] [ <expression> ] ';'

<expression> ::= DECLARE <block> |
               IF <condexp> |
               CASE <casexp> |
               <basicexp>

<block> ::= <id> [ ',' <id> ]* : <typedefn> ';'
           <legalblock>

<legalblock> ::= <id> [ ',' <id> ]* : <typedefn> ';' |
                <let> |
                <initforwhile>

<let> ::= LET [ <lhs> := <expression> ';' ]+
        RETURN <expression>

<lhs> ::= <id> | '[' <lhs> [ ',' <lhs> ]* '['

<initforwhile> ::= [ INIT [ <lhs> := <expression> ';' ]+ ]
                 FOR EACH <id> IN <streamid>
                 [ ';' EACH <id> IN <streamid> ]*
                 DO
                 [ WHILE <expression> DO ]
                 [ [ NEW ]
                   <lhs> := <expression> ';' ]+
                 RETURN <expression>

<condexp> ::= <expression> THEN <expression> ELSE <expression>

<casexp> ::= <id> OF
           [ <genid> [ '(' <parameters> ')' ]
             ':' <expression> ';' ]+
           ENDCASE

<parameters> ::= <id> [ ',' <id>* ]

<basicexp> ::= <all-remainder> |
              <simpleexp> [ <relop> <simpleexp> ]

<all-remainder> ::= ALL <basicexp> [ BUT <basicexp> ] |
```

REMAINDER [<id>]

<simpleexp> ::= ['-'] <term> <simpleops> <term>

<term> ::= <factor> <termops> <factor>

<simpleops> ::= '+' | '-' | OR | MAX | MIN

<termops> ::= '*' | DIV | MOD | AND

<factor> ::= <generatorid> ['(' <parmid> [',' <parmid>]* ')'] | <simpleid> <qualifier> | <function> | '(' <basicexp> ')' | <constvalue> | '[' [<basicexp> [',' <basicexp>]*]' | NOT <factor> | LAMBDA | <reductionops> '!' <factor>

<reductionops> ::= '*' | '+' | AND | OR | MAX | MIN

<qualifier> ::= '[' <b-or-col> [',' <b-or-col>]+ [':' [<basicexp>]]]'

<b-or-col> ::= <basicexp> | ':'

<function> ::= <userdef> | <standard>

<userdef> ::= <funcid> '(' <basicexp> [',' <basicexp>]* ')'

<standard> ::= <cons> '(' <basicexp> , <basicexp> ')' | <streamop> '(' <basicexp> ')' | CONCAT '(' <basicexp> ',' <basicexp> ')' | <oneargfns-i> '(' <basicexp> ')' | <oneargfns-r> '(' <basicexp> ')' | <exponentiation> '(' <basicexp> ',' <basicexp> ')' |

<cons> ::= CONSL | CONS

<streamop> ::= FIRST | REST | GET | EMPTY | SIZE

<oneargfns-i> ::= ISQRT | ABS | EVEN | ODD

<oneargfns-r> ::= SIN | COS | ARCSIN | ARCCOS | ALN | LN |
 SQRT | ROUND | TRUNC

<exponentiation> ::= EXR | EXI

APPENDIX B

MODEL BNF

```

<MODEL_SPECIFICATION> ::= [ <MODEL_BODY_STMTS> ] *
                           <MODEL_SPECIFICATION>
<MODEL_BODY_STMTS> ::= MODULE <MODULE_NAME_STMT>
                       | SOURCE <SOURCE_FILES_STMT>
                       | TARGET <TARGET_FILES_STMT>
                       | @#_END#@
                       | <DCL_DESCRIPTION>
                       | <OLD_FILE_STMT>
                       | <SIMPLE_ASSERTION>
<DCL_DESCRIPTION> ::= 1 <DATA_SPEC>
                    [, <INTEGER> <DATA_SPEC> ] * <ENDCHAR>
<DATA_SPEC> ::= <DCL_MVAR> [( <OCCSPEC> )] [ <IS> ]
               <ATTR_SPEC>
<ATTR_SPEC> ::= <FILE> <FILE_DESC> <STORAGE_DESC>
               | <RECORD>
               | <FIELD_STMT>
               | [ <GROUP> ]
<SIMPLE_ASSERTION> ::= <MVAR>
                    =
                    <BOOLEAN_EXPRESSION> <ENDCHAR>
<SUB_VARIABLE> ::= <VAR>
                  [(
                    <BOOLEAN_EXPRESSION> [,
                    <BOOLEAN_EXPRESSION> ] *
                    ) ]
<SUB_VARIABLE1> ::= <VAR>
                  [(

```

```

    <BOOLEAN_EXPRESSION> [,
    <BOOLEAN_EXPRESSION>]*
  ) ]
<BOOLEAN_EXPRESSION> ::= <COND_EXP> |
    <BOOLEAN_TERM>
    [ <OR> <BOOLEAN_TERM> ]*

<COND_EXP> ::= IF <BOOLEAN_EXPRESSION>
    THEN <BOOLEAN_EXPRESSION>
    [ ELSE <BOOLEAN_EXPRESSION> ]

<OR> ::= |
<BOOLEAN_TERM> ::= <BOOLEAN_FACTOR>
    [ _ <BOOLEAN_FACTOR> ]*
<BOOLEAN_FACTOR> ::= <CONCATENATION>
    [ <RELATION> <CONCATENATION> ]*
<RELATION> ::= = | = | < | <= | > | >=
<CONCATENATION> ::= <ARITH_EXP>
    [ <CONCAT> <ARITH_EXP> ]*
<CONCAT> ::= ||
<ARITH_EXP> ::= [ <SIGN> ]
    <TERM> [ <OPS> <TERM> ]*
<TERM> ::= <FACTOR>
    [ <MOPS> <FACTOR> ]*
<FACTOR> ::= [ ] <PRIMARY>
    [ <EXPON> <PRIMARY> ]*
<EXPON> ::= **
<PRIMARY> ::= <IS_PRIM>
<IS_PRIM> ::= ( <BOOLEAN_EXPRESSION> )
    | <NUMBER>
    | <STRING_FORM>
    | <FUNCTION_CALL>
    | <SUB_VARIABLE1>
<STRING_FORM> ::= ' [ <STRING> ]
    ' [ B ]
<FUNCTION_CALL> ::= <FUNCTION_NAME>
    [ ( <BOOLEAN_EXPRESSION>
    [ , <BOOLEAN_EXPRESSION>
    ]* ) ]
<MVAR> ::= ( <SUB_VARIABLE>
    [ , <SUB_VARIABLE> ]* )
    | <SUB_VARIABLE>
<VAR> ::= <NAME>
    [ . <NAME> ]* /STR_CON/
<DCL_MVAR> ::= ( <VAR> [ , <VAR> ]* )
    | <VAR>
<OPS> ::= + | -
<MOPS> ::= * | /
```

```
<MODULE_NAME_STMT> ::=          : <NAME>
    <ENDCHAR>
<SOURCE_FILES_STMT> ::=          [ <FILE_KEYWORD> ] :
    <SOURCE_FILELIST>          <ENDCHAR>
<FILE_KEYWORD> ::= FILES | FILE
<SOURCE_FILELIST> ::= <NAME>
    [ , <NAME> ] *
<TARGET_FILES_STMT> ::=          [ <FILE_KEYWORD> ] :
    <TARGET_FILELIST>          <ENDCHAR>
<TARGET_FILELIST> ::= <NAME>
    [ , <NAME> ] *
<DATA_DESC_STMT> ::= <DATA_DESCRIPTION> <ENDCHAR>
<DATA_DESCRIPTION> ::=
    <FILE_STMT>
    | <RECORD_STMT>
    | <GROUP_STMT>
    | <FIELD_STMT>
    | <SUB_STMT>
<SUB_STMT> ::= <SUBSCRIPT> [ ( <OCCSPEC> ) ]
<SUBSCRIPT> ::= SUB | SUBSCRIPT | SUBSCRIPTS
<FILE> ::= FILE | REPORT | FILES | REPORTS
<RECORD_STMT> ::= <RECORD> [ ( ) <ITEM_LIST> ( ) ]
<RECORD> ::= REC | RECORD | RECORDS
<ITEM_LIST> ::= <ITEM> [ [ , ] <ITEM> ] *
<ITEM> ::= <NAME> [ . <NAME> ] * [ ( <OCCSPEC> ) ]
<OCCSPEC> ::= <STAR> | <MINOCC> [ <MAXOCC> ]
<STAR> ::= *
<MINOCC> ::= <INTEGER>
<MAXOCC> ::= [ : ] <INTEGER>
    |
    <INTEGER>
<GROUP_STMT> ::= <GROUP> [ ( ) <ITEM_LIST> ( ) ]
<GROUP> ::= GRP | GROUP | GROUPS
<FIELD_STMT> ::= <FIELD> <FIELD_ATTR>
<FIELD> ::= FLD | FIELD | FIELDS
<FIELD_ATTR> ::= [ ( ) <TYPE> [ <LENG_SPEC> ] ( ) ]
<LENG_SPEC> ::= ( <MIN_LENGTH> [ <MAX_LENGTH> ] )
    | <MIN_LENGTH> [ <MAX_LENGTH> ]
<MIN_LENGTH> ::= <INTEGER>
<TYPE> ::= <STRING_SPEC> | <NUM_SPEC>
<STRING_SPEC> ::= <STRING_TYPE>
<STRING_TYPE> ::= CHAR | CHARACTER | BIT | NUM | NUMERIC
<NUM_SPEC> ::= <NUM_TYPE> [ <FIXFLT> ]
<NUM_TYPE> ::= BIN | BINARY | DEC | DECIMAL
<FIXFLT> ::= FIX | FIXED | FL | FLOAT | FLT
<MAX_LENGTH> ::= [ : ] <INTEGER>
    | , <SINTGR>
    | <INTEGER>
<SINTGR> ::= - <INTEGER> | <INTEGER>
```

```
<SIGN> ::= + | -
<RECG> ::= <RECORD> | <GROUP>
<ENDCHAR> ::= ;
<END_CHAR> ::= ;
<IS> ::= IS | = | ARE
<FILE_STMT> ::= <FILE> <SON_DESC>
                <FILE_DESC> <STORAGE_DESC>
<SON_DESC> ::= ( <ITEM_LIST> )
                | <RECG> [NAME] [<IS>] [(] <ITEM> [)]
<OLD_FILE_STMT> ::= <FILE> [NAME] [<IS>]
                <DCL_MVAR>
                <RECG> [NAME] [<IS>] [(] <ITEM> [)]
                <ENDCHAR>
```

APPENDIX C

EXAMPLES

This Appendix contains examples of the translation of Model specifications to MaD. For each example the following reports are reproduced below:

- Listing of Specification
- Block Description
- MaD Program

C.1 MATRIX MULTIPLY

This example is the familiar matrix multiply program from Chapter 1. A and B are the two 10x10 input matrices to be multiplied. C is the 10x10 output matrix result. In a new dialect of Model under development, even the two assertions defining the multiplication will not be needed. The dialect supports matrix operations at the source level, so that the operator `!*` indicates matrix multiplication [LiuW82].

It should be noted that the generated MaD program does not use a "transpose" function as does the Id function for matrix multiply in Chapter 1. The matrix B input to the Id version of the program is transposed and the transposed array is input to function `mmt`. Doing so creates a stream, the column of B. The inner product of a row of A with a column of B can then be computed within a simple loop. In the Mad implementation a loop is not needed in order to compute the inner product. The partial products are computed in `INTERIM_X`, and the reduction operator `+` is used to add all the partial products. In addition, MaD stores each dimension after the first of a multi-dimension structure as a random access permanent structure in the Matching Store. Each element can be accessed repeatedly with no cost other than the cost of selecting a single element. The entire array need not be duplicated. Therefore, the generated MaD program does not use a transpose function.

Note that the variable INTERIM_X is local to block 3. It is not declared in the global declaration section of the program.

```
1  MODULE: MM;
2
3  SOURCE: INFILE1, INFILE2;
4  TARGET: OUTFILE;
5
6  INFILE1 IS FILE (INREC1);
7  INREC1 IS RECORD (IN1(10));
8  IN1 IS GROUP (A(10));
9  A IS FIELD (NUMERIC);
10
11 INFILE2 IS FILE (INREC2);
12 INREC2 IS RECORD (IN2(10));
13 IN2 IS GROUP (B(10));
14 B IS FIELD (NUMERIC);
15
16 OUTFILE IS FILE (OUTREC);
17 OUTREC IS RECORD (OUT1(10));
18 OUT1 IS GROUP (C(10));
19 C IS FIELD (NUMERIC);
20
21 X0 IS GROUP (X1(10));
22 X1 IS GROUP (X2(10));
23 X2 IS GROUP (X(10));
24 X IS FIELD (NUMERIC);
25
26 I IS SUBSCRIPT (10);
27 J IS SUBSCRIPT (10);
28 K IS SUBSCRIPT (10);
29
30 X(I,J,K) = A(I,K) * B(K,J);
31 C(I,J) = SUM(X(I,J,K),K);
32
33 END MM;
```

Block Description:

Block 1
SIMP Level: 0 Range: 0 # Data nodes: 2 # Block
Members: 1
Data Nodes:
INTERIM.X for dimension 0 in block 1
OUTFILE.C for dimension 0 in block 1
Block Members:
2 BLOCK

Block 2
PARA Level: 1 Range: 1 # Data nodes: 2 # Block
Members: 1
Data Nodes:
INTERIM.X for dimension 3 in block 1
OUTFILE.C for dimension 2 in block 1
Block Members:
3 BLOCK

Block 3
PARA Level: 2 Range: 2 # Data nodes: 2 # Block
Members: 2
Data Nodes:
INTERIM.X for dimension 2 in block 1
OUTFILE.C for dimension 1 in block 2
Block Members:
4 BLOCK

Block 4
PARA Level: 3 Range: 3 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.X for dimension 1 in block 1
Block Members:
AASS300 ASSERTION

Block 5
PARA Level: 3 Range: 3 # Data nodes: 1 # Block
Members: 1
Data Nodes:
OUTFILE.C for dimension -2 in block 1
Block Members:
AASS310 ASSERTION
Local Data Nodes

Node INTERIM.X is local to block 3

MaD Program

```
PROGRAM MM(
INFILE1_A: STREAM STREAM INTEGER
;INFILE2_B: STREAM STREAM INTEGER
):
OUTFILE_C: STREAM STREAM INTEGER
;
OUTFILE_C := DECLARE
OUTFILE_C: STREAM INTEGER ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(10) DO
OUTFILE_C := DECLARE
INTERIM_X: INTEGER ;
OUTFILE_C: INTEGER ;
I_2: INTEGER;
FOR EACH I_2 IN PROLIF(10) DO
INTERIM_X := DECLARE
INTERIM_X: INTEGER ;
I_3: INTEGER;
FOR EACH I_3 IN PROLIF(10) DO
INTERIM_X := INFILE1_A[I_1,I_3]*INFILE2_B[I_3,I_2];
RETURN ALL INTERIM_X ;
OUTFILE_C := +!INTERIM_X[SUB3,SUB2];
RETURN ALL OUTFILE_C ;
RETURN ALL OUTFILE_C ;
RETURN OUTFILE_C ;
END.
```

C.2 TRAPEZOIDAL INTEGRATION

In this example, the area under a curve in an interval $[A,B]$ is computed using the trapezoidal method. Input parameters are the coefficients of the quadratic function to be integrated; the interval limits A and B ; the width of each subinterval H ; and the number of subintervals N . The interim array X holds the X values for each subinterval. The array F holds the corresponding function values. The area is computed in array S . This specification, the FFT, and the explanation of the FFT were written by Mr. Chi-Ming Chen of the University of Pennsylvania. I am grateful for his help.

```
1  MODULE: INTEG;
2
3  SOURCE: INFILE;
4  TARGET: OUTFILE;
5
6  INFILE IS FILE (INREC);
7  INREC IS RECORD (IN(6));
8  IN IS FIELD (NUMERIC);
9  A IS FIELD (NUMERIC);
10 B IS FIELD (NUMERIC);
11 H IS FIELD (NUMERIC);
12 N IS FIELD (NUMERIC);
13 COEFF1 IS FIELD (NUMERIC);
14 COEFF2 IS FIELD (NUMERIC);
15 COEFF3 IS FIELD (NUMERIC);
16
17 S1 IS GROUP (S(1:100));
18 S IS FIELD (NUMERIC);
19
20 F1 IS GROUP (F(1:100));
21 F IS FIELD (NUMERIC);
22
23 X1 IS GROUP (X(*));
```

```
24      X IS FIELD (NUMERIC);
25      FA IS FIELD (NUMERIC);
26
27      OUTFILE IS FILE (OUTREC);
28      OUTREC IS RECORD (OUT);
29      OUT IS FIELD (NUMERIC);
30
31      I IS SUBSCRIPT(100);
32
33      A=IN(1);
34      H=IN(2);
35      N=IN(3);
36      COEFF1=IN(4);
37      COEFF2=IN(5);
38      COEFF3=IN(6);
39      FA = COEFF1 * A * A + COEFF2 * A + COEFF3;
40      S(I) = IF I = 1 THEN (FA + F(N))/2
41      ELSE S(I-1) + F(I-1);
42      F(I) = COEFF1 * X(I) * X(I) + COEFF2 * X(I) + COEFF3;
43      X(I) = IF I = 1 THEN A + H
44      ELSE X(I-1) + H;
45      OUT = S(N);
46
47      END INTEG;
```

Block Description:

Block 1
SIMP Level: 0 Range: 0 # Data nodes: 11 # Block
Members: 11
Data Nodes:
INTERIM.COEFF3 for dimension 0 in block 1
INTERIM.COEFF2 for dimension 0 in block 2
INTERIM.COEFF1 for dimension 0 in block 3
INTERIM.N for dimension 0 in block 4
INTERIM.H for dimension 0 in block 5
INTERIM.A for dimension 0 in block 6
INTERIM.FA for dimension 0 in block 7
INTERIM.X for dimension 0 in block 8
INTERIM.F for dimension 0 in block 9
INTERIM.S for dimension 0 in block 10
OUTFILE.OUT for dimension 0 in block 11
Block Members:
AASS380 ASSERTION
AASS370 ASSERTION
AASS360 ASSERTION
AASS350 ASSERTION
AASS340 ASSERTION
AASS330 ASSERTION
AASS390 ASSERTION
2 BLOCK
3 BLOCK
4 BLOCK
AASS450 ASSERTION
Block 2
ITER Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.X for dimension 1 in block 1
Block Members:
AASS430 ASSERTION
Block 3
PARA Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.F for dimension 1 in block 1
Block Members:
AASS420 ASSERTION
Block 4
ITER Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.S for dimension 1 in block 1

Block Members:
AASS400 ASSERTION
Local Data Nodes

Node INTERIM.N is local to block 1
Node INTERIM.H is local to block 1
Node INTERIM.PA is local to block 1
Node INTERIM.COEFF3 is local to block 1
Node INTERIM.COEFF2 is local to block 1
Node INTERIM.COEFF1 is local to block 1
Node INTERIM.A is local to block 1

MaD Program

```
PROGRAM INTEG(
INFILE_IN: STREAM INTEGER
):
OUTFILE_OUT: INTEGER
;
DECLARE
INTERIM_X: STREAM INTEGER;
INTERIM_S: STREAM INTEGER;
INTERIM_F: STREAM INTEGER;
INTERIM_B: INTEGER;
INTERIM_COEFF3 := INFILE_IN[6];
INTERIM_COEFF2 := INFILE_IN[5];
INTERIM_COEFF1 := INFILE_IN[4];
INTERIM_N := INFILE_IN[3];
INTERIM_H := INFILE_IN[2];
INTERIM_A := INFILE_IN[1];
INTERIM_FA := INTERIM_COEFF1*INTERIM_A*INTERIM_A+
              INTERIM_COEFF2*INTERIM_A+INTERIM_COEFF3;
INTERIM_X := DECLARE
INTERIM_X: INTEGER ;
I_1: INTEGER;
INIT
I_1:=1;
INTERIM_X:=0;
WHILE I_1<=100 DO
NEW I_1:=I_1+1;
NEW INTERIM_X := IF I_1=1 THEN INTERIM_A+INTERIM_H
                ELSE INTERIM_X+INTERIM_H;
RETURN ALL NEW INTERIM_X ;
INTERIM_F := DECLARE
INTERIM_F: INTEGER ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(100) DO
INTERIM_F := INTERIM_COEFF1*INTERIM_X[I_1]*INTERIM_X[I_1]
            +INTERIM_COEFF2*INTERIM_X[I_1]+INTERIM_COEFF3;
RETURN ALL INTERIM_F ;
INTERIM_S := DECLARE
INTERIM_S: INTEGER ;
I_1: INTEGER;
INIT
I_1:=1;
INTERIM_S:=0;
WHILE I_1<=100 DO
NEW I_1:=I_1+1;
NEW INTERIM_S := IF I_1=1 THEN
                ( INTERIM_FA+INTERIM_F[INTERIM_N] )/2
```

```
ELSE INTERIM_S+INTERIM_F[I_1-1];  
RETURN ALL NEW INTERIM_S ;  
OUTFILE_OUT := INTERIM_S[INTERIM_N];  
RETURN OUTFILE_OUT ;  
END.
```

C.3 FAST FOURIER TRANSFORM

This module specifies the computation of the Fourier Transform of N=16 sample data taken from the function

$$f(t) = (e^{-t}) \sin(t)$$

with the sampling interval $T = 0.375$. The FORTRAN version of the program can be found in Appendix B of [Stea75].

The purpose of the FFT is to calculate the DFT using $N \log N$ multiplications rather than N^2 multiplications, where N is the number of sample data points.

Let $f(t)$ be the time function for the sample data and f_n be the n th sample. Then the DFT of the data is given by

$$F_m = \sum_{n=0}^{N-1} f_n * e^{(-j)*(2*\pi*m*n/N)} \quad m=0 \dots N-1$$

$$= \sum_{n=0}^{N-1} f_n * WN(mn) \quad \text{where } WN = e^{(-j)*(2*\pi/N)}$$

Because of the cyclic property of the exponential function

$$WN^k = WN^{(c-N+k)} \quad \text{where } c \text{ is any integer}$$

We can find $\{F_m \mid 0 \leq m \leq N-1\}$ together by q decompositions if we choose $N = 2^q$. For simplicity, the method is explained using $N = 2^3$. The method described is that of computing the FFT using time decomposition with input bit reversal.

In order to reduce the number of multiplications, the sequence of inputs should be reordered, for example by the input bit reversal method. This is illustrated in Figure C.1.

input	0	1	2	3	4	5	6	7
pattern	000	001	010	011	100	101	110	111
reversal	000	100	010	110	001	101	011	111
result	0	4	2	6	1	5	3	7

Figure C.1 Input Bit Reversal

Using the obtained sequence, we can calculate $\{F_m\}$. Each F_m is calculated through 2 intermediate stages, G_{i1} and G_{i2} , $i=1 \dots 8$.

For example,

$$G_{11} = f_0 + W_{80} f_4$$

$$G_{21} = f_0 + W_{84} f_4$$

$$G_{31} = f_2 + W_{80} f_6$$

$$G_{41} = f_2 + W_{84} f_6$$

.....

and the result for the second stage is given by

$$G_{12} = G_{11} + W_{80} * G_{31}$$

$$G_{22} = G_{21} + W_{82} * G_{41}$$

$$G_{32} = f_2 + W_{84} * G_{31}$$

$$G_{42} = f_2 + W_{86} * G_{41}$$

.....

That is, the node is obtained by two nodes in the previous stage and some power of WN .

In the MODEL specification, $BRS(I)$ gives the input bit reversal sequence. $WR(I)$ and $WI(I)$ give the real and imaginary parts respectively of WN^k , $k = 0, 1, \dots, N-1$. $WP(J,I)$ gives the power k in WN^k for the I th node in the J th stage. $GP1(J,I)$ and $GP2(J,I)$ give the first and second node numbers respectively from the $(J-1)$ th stage for the I th node in the J th stage. $GR(J,I)$ and $GI(J,I)$ give the real and imaginary parts respectively of the values for the I th node at stage J . Then, finally, $GR(\log N, I)$ and $GI(\log N, I)$ are the real and imaginary parts of the Fourier transform FI .

```
1  MODULE: FFTMOD;
2  TARGET : OUTFILE1, OUTFILE2;
3  OUTFILE1 IS FILE(OUTREC1(16));
4  OUTREC1 IS RECORD (OUTR);
5  OUTFILE2 IS FILE(OUTREC2(16));
6  OUTREC2 IS RECORD (OUTI);
7  (OUTR,OUTI) IS FIELD (BIN FIXED);
8
9  GO IS GROUP (GOREC(16));
10 GOREC IS GROUP (T,F,ARG,WR,WI,FR,BRS);
11 (T,F,ARG,WR,WI,FR) IS FIELD (BINARY FLOAT);
12 BRS IS FIELD (BIN FIXED);
13
14 G1 IS GROUP (G1REC1(4));
15 G1REC1 IS GROUP (P,Q);
16 (P,Q) IS FIELD (BIN FIXED);
17
18 G2 IS GROUP (G2REC1(4));
19 G2REC1 IS GROUP (G2REC2(16));
20 G2REC2 IS GROUP (BR,WP,GP1,GP2,GR,GI);
21 (BR,WP,GP1,GP2) IS FIELD (BIN FIXED) ;
22 (GR,GI) IS FIELD (BINARY FLOAT) ;
23
24 (NB,N,NH) IS FIELD (BIN FIXED);
25
26 I IS SUBSCRIPT (16);
27 J IS SUBSCRIPT (4) ;
28
29 NB=4;
30 N=2**NB;
31 NH=N/2;
32 P(J)=2**J;
33 Q(J)=2**(J-1);
34 T(I)=0.375*(I-1);
35 F(I)=EXP(-T(I))*SSIN(T(I));
36 BR(J,I)= IF J=1 THEN MOD(I-1,2)*NH
37           ELSE (MOD(I-1,P(J))/P(J-1))*(N/P(J)) ;
38 BRS(I)=SUM(BR(J,I),J) ;
39 FR(I)= F(BRS(I)+1) ;
40 ARG(I)=3.1415926535*(1-I)/NH ;
41 WR(I)= IF I <= NH THEN CCOS(ARG(I)) ELSE -WR(I-8) ;
42 WI(I)= IF I <= NH THEN SSIN(ARG(I)) ELSE -WI(I-8) ;
43 WP(J,I)= MOD((I-1),P(J))*(N/P(J))+1;
44 GP1(J,I)= IF MOD(I-1,P(J)) < Q(J) THEN I
45           ELSE I-Q(J) ;
46 GP2(J,I)= GP1(J,I) + Q(J) ;
47 GR(J,I)= IF J=1 THEN IF MOD(I,2) = 1 THEN FR(I) +
FR(I+1)
```

```
48             ELSE FR(I-1) - FR(I)
49     ELSE GR(J-1,GP1(J,I))+WR(WP(J,I))*GR(J-1,GP2(J,I))
50             -WI(WP(J,I))*GI(J-1,GP2(J,I)) ;
51 GI(J,I)= IF J=1 THEN 0.
52     ELSE GI(J-1,GP1(J,I))+WR(WP(J,I))*GI(J-1,GP2(J,I))
53             +WI(WP(J,I))*GR(J-1,GP2(J,I));
54     OUTR(I)= GR(NB,I);
55     OUTI(I)= GI(NB,I);
56
57 END FFTMOD;
```

Block Description:

Block 1
SIMP Level: 0 Range: 0 # Data nodes: 20 # Block
Members: 13
Data Nodes:
INTERIM.T for dimension 0 in block 1
INTERIM.F for dimension 0 in block 1
INTERIM.NB for dimension 0 in block 2
INTERIM.N for dimension 0 in block 3
INTERIM.NH for dimension 0 in block 4
INTERIM.ARG for dimension 0 in block 5
INTERIM.WR for dimension 0 in block 6
INTERIM.WI for dimension 0 in block 7
INTERIM.Q for dimension 0 in block 8
INTERIM.P for dimension 0 in block 8
INTERIM.WP for dimension 0 in block 8
INTERIM.GP1 for dimension 0 in block 8
INTERIM.BR for dimension 0 in block 9
INTERIM.BRS for dimension 0 in block 9
INTERIM.FR for dimension 0 in block 9
INTERIM.GP2 for dimension 0 in block 10
INTERIM.GR for dimension 0 in block 11
INTERIM.GI for dimension 0 in block 11
OUTFILE1.OUTR for dimension 0 in block 12
OUTFILE2.OUTI for dimension 0 in block 13
Block Members:
2 BLOCK
AASS290 ASSERTION
AASS300 ASSERTION
AASS310 ASSERTION
3 BLOCK
4 BLOCK
5 BLOCK
6 BLOCK
9 BLOCK
12 BLOCK
14 BLOCK
17 BLOCK
18 BLOCK
Block 2
PARA Level: 1 Range: 1 # Data nodes: 2 # Block
Members: 2
Data Nodes:
INTERIM.T for dimension 1 in block 1
INTERIM.F for dimension 1 in block 2
Block Members:
AASS340 ASSERTION

AASS350 ASSERTION

Block 3

PARA Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.ARG for dimension 1 in block 1

Block Members:

AASS400 ASSERTION

Block 4

ITER Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.WR for dimension 1 in block 1

Block Members:

AASS410 ASSERTION

Block 5

ITER Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.WI for dimension 1 in block 1

Block Members:

AASS420 ASSERTION

Block 6

PARA Level: 1 Range: 2 # Data nodes: 4 # Block
Members: 4

Data Nodes:

INTERIM.Q for dimension 1 in block 1

INTERIM.P for dimension 1 in block 2

INTERIM.WP for dimension 2 in block 3

INTERIM.GP1 for dimension 2 in block 4

Block Members:

AASS330 ASSERTION

AASS320 ASSERTION

7 BLOCK

8 BLOCK

Block 7

PARA Level: 2 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.WP for dimension 1 in block 1

Block Members:

AASS430 ASSERTION

Block 8

PARA Level: 2 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.GP1 for dimension 1 in block 1

Block Members:

AASS440 ASSERTION

Block 9

PARA Level: 1 Range: 1 # Data nodes: 3 # Block
Members: 3

Data Nodes:

INTERIM.BR for dimension 1 in block 1

INTERIM.BRS for dimension 1 in block 2

INTERIM.FR for dimension 1 in block 3

Block Members:

10 BLOCK

11 BLOCK

AASS390 ASSERTION

Block 10

PARA Level: 2 Range: 2 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.BR for dimension 2 in block 1

Block Members:

AASS360 ASSERTION

Block 11

PARA Level: 2 Range: 2 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.BRS for dimension -1 in block 1

Block Members:

AASS380 ASSERTION

Block 12

PARA Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.GP2 for dimension 1 in block 1

Block Members:

13 BLOCK

Block 13

PARA Level: 2 Range: 2 # Data nodes: 1 # Block
Members: 1

Data Nodes:

INTERIM.GP2 for dimension 2 in block 1

Block Members:

AASS460 ASSERTION

Block 14

ITER Level: 1 Range: 2 # Data nodes: 2 # Block
Members: 2

Data Nodes:

INTERIM.GR for dimension 2 in block 1

INTERIM.GI for dimension 2 in block 2

Block Members:

15 BLOCK

16 BLOCK
Block 15
PARA Level: 2 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.GR for dimension 1 in block 1
Block Members:
AASS470 ASSERTION
Block 16
PARA Level: 2 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
INTERIM.GI for dimension 1 in block 1
Block Members:
AASS510 ASSERTION
Block 17
PARA Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
OUTFILE1.OUTR for dimension 1 in block 1
Block Members:
AASS540 ASSERTION
Block 18
PARA Level: 1 Range: 1 # Data nodes: 1 # Block
Members: 1
Data Nodes:
OUTFILE2.OUTI for dimension 1 in block 1
Block Members:
AASS550 ASSERTION
Local Data Nodes

Node INTERIM.NH is local to block 1
Node INTERIM.NB is local to block 1
Node INTERIM.N is local to block 1
Node INTERIM.BR is local to block 9
Node INTERIM.WP is local to block 1
Node INTERIM.GP1 is local to block 1
Node INTERIM.GP2 is local to block 1
Node INTERIM.Q is local to block 1
Node INTERIM.T is local to block 2
Node INTERIM.ARG is local to block 1
Node INTERIM.BRS is local to block 9

MaD Program

```
PROGRAM FFTMOD(
):[
OUTFILE1_OUTR: STREAM INTEGER
;OUTFILE2_OUTI: STREAM INTEGER
];
DECLARE
INTERIM_GR: STREAM STREAM REAL;
INTERIM_GI: STREAM STREAM REAL;
INTERIM_P: STREAM INTEGER;
INTERIM_F: STREAM REAL;
INTERIM_WR: STREAM REAL;
INTERIM_WI: STREAM REAL;
INTERIM_FR: STREAM REAL;
INTERIM_F := DECLARE
INTERIM_T: REAL ;
INTERIM_F: REAL ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
INTERIM_T := 0.375*(I_1-1);
INTERIM_F := EXP((-INTERIM_T))*SSIN(INTERIM_T);
RETURN ALL INTERIM_F ;
INTERIM_NB := 4;
INTERIM_N := 2**INTERIM_NB;
INTERIM_NH := INTERIM_N/2;
INTERIM_ARG := DECLARE
INTERIM_ARG: REAL ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
INTERIM_ARG := 3.1415926535*(1-I_1)/INTERIM_NH;
RETURN ALL INTERIM_ARG ;
INTERIM_WR := DECLARE
INTERIM_WR: REAL ;
I_1: INTEGER;
INIT
I_1:=1;
INTERIM_WR:=0;
WHILE I_1<=16 DO
NEW I_1:=I_1+1;
NEW INTERIM_WR := IF I_1<=INTERIM_NH THEN CCOS(INTERIM_ARG[I_1])
ELSE -INTERIM_WR[I_1-8];
RETURN ALL NEW INTERIM_WR ;
INTERIM_WI := DECLARE
INTERIM_WI: REAL ;
I_1: INTEGER;
INIT
I_1:=1;
```

```
INTERIM_WI:=0;
WHILE I_1<=16 DO
NEW I_1:=I_1+1;
NEW INTERIM_WI := IF I_1<=INTERIM_NH THEN SSIN(INTERIM_ARG[I_1])
ELSE -INTERIM_WI[I_1-8];
RETURN ALL NEW INTERIM_WI ;
[INTERIM_Q, INTERIM_P, INTERIM_WP, INTERIM_GP1] := DECLARE
INTERIM_Q: INTEGER ;
INTERIM_P: INTEGER ;
INTERIM_WP: STREAM INTEGER ;
INTERIM_GP1: STREAM INTEGER ;
I_2: INTEGER;
FOR EACH I_2 IN PROLIF(4) DO
INTERIM_Q := 2**(I_2-1);
INTERIM_P := 2**I_2;
INTERIM_WP := DECLARE
INTERIM_WP: INTEGER ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
INTERIM_WP := MOD((I_1-1), INTERIM_P)*( INTERIM_N/INTERIM_P)+1;
RETURN ALL INTERIM_WP ;
INTERIM_GP1 := DECLARE
INTERIM_GP1: INTEGER ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
INTERIM_GP1 := IF MOD((I_1-1), INTERIM_P)<INTERIM_Q THEN I_1 ELSE
I_1-INTERIM_Q;
RETURN ALL INTERIM_GP1 ;
RETURN [ ALL INTERIM_Q, ALL INTERIM_P, ALL INTERIM_WP, ALL
INTERIM_GP1];
INTERIM_FR := DECLARE
INTERIM_BR: INTEGER ;
INTERIM_BRS: INTEGER ;
INTERIM_FR: REAL ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
INTERIM_BR := DECLARE
INTERIM_BR: INTEGER ;
I_2: INTEGER;
FOR EACH I_2 IN PROLIF(4) DO
INTERIM_BR := IF I_2=1 THEN MOD((I_1-1),2)*INTERIM_NH
ELSE (MOD((I_1-1), INTERIM_P[I_2])/
INTERIM_P[I_2-1])*(INTERIM_N/INTERIM_P[I_2]);
RETURN ALL INTERIM_BR ;
INTERIM_BRS := +!INTERIM_BR[SUB1];
INTERIM_FR := INTERIM_F[INTERIM_BRS+1];
RETURN ALL INTERIM_FR ;
INTERIM_GP2 := DECLARE
```

```
INTERIM_GP2: STREAM INTEGER ;
I_1: INTEGER;
FOR EACH I_1 IN PROLIF(16) DO
  INTERIM_GP2 := DECLARE
  INTERIM_GP2: INTEGER ;
  I_2: INTEGER;
  FOR EACH I_2 IN PROLIF(4) DO
    INTERIM_GP2 := INTERIM_GP1[I_2,I_1]+INTERIM_Q[I_2];
  RETURN ALL INTERIM_GP2 ;
  RETURN ALL INTERIM_GP2 ;
  [INTERIM_GR, INTERIM_GI] := DECLARE
  INTERIM_GR: REAL ;
  INTERIM_GI: REAL ;
  I_2: INTEGER;
  INIT
  I_2:=1;
  INTERIM_GR:=0;
  INTERIM_GI:=0;
  WHILE I_2<=4 DO
    NEW I_2:=I_2+1;
    NEW INTERIM_GR := DECLARE
    INTERIM_GR: REAL ;
    I_1: INTEGER;
    FOR EACH I_1 IN PROLIF(16) DO
      INTERIM_GR := IF I_2=1 THEN IF MOD(I_1,2)=1
        THEN INTERIM_FR[I_1]+INTERIM_FR[I_1+1]
        ELSE INTERIM_FR[I_1-1]-INTERIM_FR[I_1]
      ELSE INTERIM_GR+INTERIM_WR[INTERIM_WP[I_2,I_1]]
        *INTERIM_GR-INTERIM_WI[INTERIM_WP[I_2,I_1]]
        *INTERIM_GI[INTERIM_GP2[I_2,I_1]];
    RETURN ALL INTERIM_GR ;
    NEW INTERIM_GI := DECLARE
    INTERIM_GI: REAL ;
    I_1: INTEGER;
    FOR EACH I_1 IN PROLIF(16) DO
      INTERIM_GI := IF I_2=1 THEN 0
        ELSE INTERIM_GI+INTERIM_WR[INTERIM_WP[I_2,I_1]]
          *INTERIM_GI+INTERIM_WI[INTERIM_WP[I_2,I_1]]
          *INTERIM_GR[INTERIM_GP2[I_2,I_1]];
    RETURN ALL INTERIM_GI ;
  RETURN [ ALL NEW INTERIM_GR, ALL NEW INTERIM_GI];
  OUTFILE1_OUTR := DECLARE
  OUTFILE1_OUTR: INTEGER ;
  I_1: INTEGER;
  FOR EACH I_1 IN PROLIF(16) DO
    OUTFILE1_OUTR := INTERIM_GR[INTERIM_NB,I_1];
  RETURN ALL OUTFILE1_OUTR ;
  OUTFILE2_OUTI := DECLARE
```

```
OUTFILE2_OUTI: INTEGER ;  
I_1: INTEGER;  
FOR EACH I_1 IN PROLIF(16) DO  
OUTFILE2_OUTI := INTERIM_GI[INTERIM_NB,I_1];  
RETURN ALL OUTFILE2_OUTI ;  
RETURN [OUTFILE1_OUTR,OUTFILE2_OUTI];  
END.
```

BIBLIOGRAPHY

[Acke78] Ackerman, W., "A Structure Memory for Data Flow Computers", MIT Lab for Computer Science, TR-156.

[Acke79] Ackerman, W., "VAL- A value oriented algorithmic language, Preliminary Reference Manual", Lab for Computer Science, TR-218, 1979.

[Acke82] Ackerman, W., "Data Flow Languages", IEEE Computer, February 1982, pp. 15-23.

[Arvi78] Arvind, Gostelow, K., and Plouffe, W., "An Asynchronous Programming Language and Computing Machine", T.R. #114A, Univ. of Calif., Irvine, Dec. 1978.

[Arvi80] Arvind, Thomas, Robert E., "I-Structures: An Efficient Data Type For Functional Languages", T.R. #178, MIT, September 1980.

[Ashc77] Ashcroft, E. A. and Wadge, W. W., "Lucid, a Nonprocedural Language with Iteration", CACM July 1977, pp. 519-526.

[Bowe81] Bowen, D. L., Implementation of Data Structures on a Data Flow Computer, Ph.d. dissertation, Manchester University, April 1981.

[Cheh79] Che, Her-daw, "Static Scheduling in Computer Systems", Ph.d dissertation, CIS, U. of Pennsylvania, 1979.

[Chen82] Chen, Chi-Ming, "An Experience of Using the Manchester Prototype Dataflow System Emulator", unpublished paper, University of Pennsylvania, May 1982.

[Davi79] Davis, A. L. and Drongowski, P. J., "Dataflow Computers: A Tutorial and Survey", UUCS-80-109, University of Utah Technical Report, Sept. 1979, revised July 1980.

[Denn74] Dennis, Jack B., "First Version of a Data Flow Procedure Language", Lecture Notes in Computer Science, v. 19, 1974, pp. 362-376.

[Denn77] Dennis, Jack B., Leung, C., and Misunas, D., "A Highly Parallel Processor Using a Data Flow Machine Language", MIT Computation Structures Group Memo 134-2, Jan 1977.

[Denn80] Dennis, Jack B., "Data Flow Computer Architecture", MIT Computation Structures Group Memo 198, July 1980.

[Gajs82] Gajski, D. D., Padua, A., Kuck, D. J., "A Second Opinion on Data Flow Machines and Languages", IEEE Computer, February 1982, pp. 58-68.

[Gost80] Gostelow, Kim P., and Thomas, Robert E., "Performance of a Simulated Dataflow Computer", IEEE Transactions on Computers, Vol. C-29 No. 10, October 1980, pp. 905-918.

[Gree81] Greenberg, R. G., Simultaneous Equations in the Model System with an Application to Econometric Modelling, Master's thesis, University of Pennsylvania, December 1981.

[Gurd81] Gurd, J. R., et. al., "Generation of Data Flow Object Code for the Lapse Programming Language", CONPAR 81, Lecture Notes in Computer Science, vol. iii, Springer-Verlag, June 1981, pp. 155-168.

[Kirk82] Kirkham, C. C., "The Manchester Prototype Data Flow System-Basic Programming Manual", April 1982.

[Kosi79] Kosinski, Paul R., "Denotational Semantics of Determinate and Non-Determinate Data Flow Programs", Ph.d dissertation, MIT/LCS/TR-220, 1979.

[Lock82] Lock, Evan, Prywes, N. S., and Szymanski, Boleslaw, "Automatic Verification and Debugging of a Program Specification", T. R. #149, University of Pennsylvania, 1982.

[Luka81] Lu, Kang-Sen, Program Optimization Based on a Non-Procedural Specification, Ph. D. dissertation, University of Pennsylvania, December, 1981.

[Luka82] Lu, Kang-Sen, MODEL Program Generator: System and Programming Documentation, T.R. #144, University of Pennsylvania,

December, 1982.

[McGr80] McGraw, J. R., "Data Flow Computing- Software Development", IEEE Transactions on Computers, v. C-29, No. 12, Dec 1980, pp. 1095-1103.

[Pryw82a] Prywes, Noah, and Pnueli, Amir, "Compilation of Nonprocedural Specifications into Computer Programs", T.R. #147, University of Pennsylvania, 1982.

[Pryw82b] Prywes, Noah, and Pnueli, Amir, "Automatic Program Generation in Distributed Cooperative Computation", T.R. #146, University of Pennsylvania, 1982.

[Saub80] Sauber, W., ed., "A Dataflow Architecture Implementation", Texas Instruments, 1980.

[Shas78] Shastry, S. K., Verification and Correction of Non-Procedural Specifications in Automatic Generation of Programs, Ph. D. dissertation, University of Pennsylvania, 1978.

[ShiY82] Shi, Yuan, "Distributed Processing with the Model System", Dissertation Proposal, University of Pennsylvania, 1982.

[Stan75] Stanley, W.D., Digital Signal Processing, Reston, 1975.

[Stea75] Stearns, S. D., Digital Signal Analysis, Hayden, 1975.

[TITE80] The TI DDP Machine: Test Bed Software Design Specification, Texas Instruments Incorporated, August 1980.

[Wats79] Watson, Ian, And Gurd, John, "A Practical Data Flow Computer", IEEE Computer, February, 1982, pp. 51-57.

INDEX

- Arvind, 17
- Atomic storage unit, 35
- Cert laboratory, 17
- Code generation
 - assertions, 157
 - assignment statements, 156
 - block generation, 158
 - data declarations, 149
 - example, 164
 - for each, 161
 - gendcl, 152
 - interim variables, 153
 - iter, 163
 - let clause, 157
 - local variables, 154
 - organization, 148
 - overview, 143
 - restrictions, 147
 - result variables, 156
- Cycles, 118
- Data dependencies, 11
- Data flow, 2
- Data flow graph, 14
- Data flow model, 11
 - asynchrony, 14
 - lack of address, 14
 - parallelism, 14
 - referential transparency, 15
- Data flow template
 - block description table, 94
 - data description table, 93
- Data structure simplification, 119
- Data structures, 34
- DDML, 27
- Dennis, 17
- Denotational semantics, 16
- Dimension, 74
- Dimension reduction, 119
- Directed arcs, 11
- Distinguished dimension, 101
- End array, 66
- Findvirtual, 141
- Fortran, 46
- Functional languages, 47
- Hanging, 105
- I-structure, 37
- Id, 48
- Instructions, 17
 - group packet, 22
 - in the Lau system, 22
 - in the TI DDP, 20
 - scheduling of, 19
- Label generation, 114
- Languages
 - model, 3
 - nonprocedural, 3
- Languages for data flow
 - conventional languages, 45
 - Fortran, 46
 - functional languages, 47
- MaD, 50
 - bnf, 51
- new constructs
 - forall, 48

- iter, 48
- stream data type, 48
- Lau programming language, 48
- Lau system, 22
- Local variables, 154

- MaD, 9, 48
 - bnf, 51
 - matrices, 54
 - multidimensional structures, 53
 - restrictions, 147
 - stored streams, 53
 - structure types in, 52
 - structured loops, 56
 - the basic expression, 57
- Manchester, 9
- Manchester data flow machine, 40
 - data and instruction formats, 43
 - machine layout, 40
- Matching store, 22
- Matching unit, 22
- Matrix, 54
- Merging components, 118
 - criteria for, 125
 - valid candidates, 125
- Model
 - array graph, 74
 - description, 72
 - edge data structure, 83
 - end array, 66
 - iterative loops, 61
 - node dimensionality, 80
 - objectives, 61
 - physical and virtual dimensions, 87
 - precedence, 81
 - range, 81
 - range sets, 85
 - size array, 65
 - underlying graph, 73
- Model, 3
- Msc, 91

- Node store, 22
- Nonprocedural languages, 58
 - common properties, 58
 - environment, 59
 - Lucid, 60
 - Model, 60
 - semantics, 59

- Partitioning, 33
- Physical dimensions, 87
- Problem areas in data flow, 29
 - data structures, 33
 - partitioning the program, 32
 - reentrancy of graph, 30
- Processing unit, 22
- Processor network topology, 24
 - DDML, 27
 - TI DDP, 25

- Range, 74
- Referential transparency, 2

- Schedule_component, 100
- Schedule_graph, 99
- Scheduling, 90
 - a simple algorithm, 97
 - asserion node, 101
 - component graph, 91
 - cycles in the array graph, 105
 - description, 91
 - distinguished dimension, 101
 - efficiency, 112
 - hanging, 105
 - initialization, 98
 - Msc, 91