MIT/LCS/TR-240

SAFETY AND OPTIMIZATION TRANSFORMATIONS FOR DATA FLOW PROGRAMS

Lynn Barbara Montz

This blank page was inserted to preserve pagination.

Safety and Optimization Transformations for Data Flow Programs

by

Lynn Barbara Montz

January, 1980

Copyright 1980 Massachusetts Institute of Technology

This research was supported in part by the National Science Foundation under research grant MCS75-04060 A01 and in part by the Lawrence Livermore Laboratory of the University of California under contract 8545403.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Laboratory for Computer Science

Cambridge Massachusetts 02139

This empty page was substituted for a blank page in the original document.

Safety and Optimization Transformations for Data Flow Programs

by

Lynn Barbara Montz

Submitted to the Department of Electrical Engineering and Computer Science
on January 31, 1980 in partial fulfillment of the requirements for
the Degree of Master of Science

ABSTRACT

The *data flow* concept of computation seeks to achieve high performance by allowing concurrent execution of instructions based on the availability of data. This thesis explores the translation of a subset of the high level language VAL to data flow graphs. The major problem in performing this translation for the target machine, the Dennis-Misunas data flow computer, stems from the restriction that graph execution sequences place at most one value on any given arc at any time. The *data/acknowledge arc pair transformation* is introduced as a means of implementing this required operational behavior. Its effect on data flow graph operation is subsequently explored as it relates to correctness and performance.

Though the arc transformation enables graphs to be executed without the possibility of deadlock, the resulting overhead and the potential loss of some concurrency represent significant costs. Two techniques aimed at minimizing these problems are developed for optimizing transformed graphs. The optimization to *eliminate unneeded acknowledge arcs* analyzes VAL constructs to identify arc pairs which may permit removal of their acknowledge arc. The optimization to *balance token flow* specifies a method of inserting identity operators into a graph for the purpose of pipelining input sets, and thereby increasing graph throughput. Though developed within the context noted, the translation and optimization issues described should prove applicable to other data flow architectures.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Keywords: data flow programming, data flow translation, optimization, asynchronous systems, Petri nets.

This empty page was substituted for a blank page in the original document.

ACKNOWLEDGEMENTS

Theodor Herzl said, אם תרצו אין זו אגדה, that is, "if you will it, it is no dream"; and though this thesis represents the completion of a personal endeavor, its accomplishment is owed in large part to several individuals to whom I am greatly indebted.

I wish to thank my thesis advisor, Jack Dennis, for offering me the opportunity to join the Computation Structures Group of the Laboratory for Computer Science, and for his subsequent guidance in the formulation and development of this research. I have gained much from my association with the members of his group, and am grateful for the very positive and warm working atmosphere which they have collectively created.

I would like to thank Clem Leung for providing direction and encouragement during the early stages of my work, and Dean Brock for his helpful editorial comments and suggestions concerning the technical content.

I am especially grateful to Bill Ackerman for countless invaluable discussions of the problems and ideas which arose throughout this research. His patience was truly remarkable, and his encouragement and confidence in my abilities should prove a permanent benefit.

Of course, I must express my special thanks to Chris Terman whose excellent preparation of the graphs in this thesis seems quite minor in comparison to the sincerity and value of his friendship.

Finally, I am most grateful to my parents and family, who have always provided me with support, encouragement, and love.

This empty page was substituted for a blank page in the original document.

TABLE OF CONTENTS

1. CHAPTER ONE 6

1.1 Introduction 6
1.2 Data Flow Graph Operation 7
1.3 Translation of VAL to Data Flow Graphs 9
1.4 Safety Transformations for Data Flow Graphs 12
1.5 Optimizing Transformed Data Flow Graphs 14
1.6 Structure of Thesis 15

2. CHAPTER TWO 16

2.1 The Safety Transformation 16
2.2 The Petri Net - Data Flow Graph Analogy 17
2.2.1 History and Analogy 17
2.2.2 Modelling Data Flow Graphs with Petri Nets 19
2.3 The Data/Acknowledge Arc Pair Transformation 22
2.3.1 Achieving Safe Data Flow Graph Operation 22
2.3.2 Preservation of Liveness 24

3. CHAPTER THREE 35

3.1 Balancing Token Flow 35
3.2 Formulating the Optimization 36
3.2.1 Identifying the Source of Bottleneck 36
3.2.2 Preview of a Solution 37
3.2.3 Analyzing Token Flow to Characterize the Solution 38
3.2.4 Observations 42
3.3 Full vs. Limited Buffering 43
3.3.1 Achieving Limited Buffering 43
3.3.2 Examples of Full vs. Limited Buffering 45
3.3.3 Additional Considerations 56

4. CHAPTER FOUR 57

4.1 Eliminating Unneeded Acknowledge Arcs 57
4.2 Considerations for Acknowledge Arc Removal 58
4.3 Analysis of the Conditional Construct 60
4.4 Analysis of the Iteration Construct 66
4.4.1 Acknowledge Arc Removal 66

4.4.2 Acknowledge Arc Removal in Iterative Programs 72

5. CHAPTER FIVE 79

5.1 Summary 79
5.2 Directions for Future Research 81

6. BIBLIOGRAPHY 83

CHAPTER ONE

1.1 Introduction

The short history of computing as a science is unique in its unparalleled rate of technological growth. In response to this, the demand for greater levels of computing power has risen as rapidly. Anticipating the continuation of this trend, research in the area of parallel computation seeks to achieve high performance by manipulating programs to exploit the parallelism inherent in many problems. Though this has led to the introduction of "do in parallel" constructs within certain languages, the sequential nature of conventional machine programming has proved to be a barrier to the formulation of an adequate and practical approach. The *data flow* concept of computation overcomes this difficulty by allowing the availability of data to determine the execution sequence, rather than a sequential instruction counter. In the data flow model, an operation is executed as soon as its required operands have been computed. The development of this concept has resulted in the proposal of several data flow machine architectures and associated data flow languages. This thesis addresses certain language translation problems which arise in translating the high level data flow language, VAL[2] for the Dennis-Misunas data flow machine[11].

The concept of data flow is best illustrated by *data flow graphs* which explicitly show the data dependencies of operations in a data flow program. The *operators* and *ares* of data flow graphs are viewed as an abstraction of the instruction cells and operand registers of the data flow machine and as such, provide a model for describing translation problems. The chapter proceeds with a more detailed look at the components and operation of data flow graphs, followed by a brief look at the high level data flow language, VAL and its translation into graph form. The major problem, termed *safety*, which arises in making the translation will be identified and discussed in section 1.4. While resolving the

safety issue is straightforward, the solution introduces a secondary, more subtle set of problems to the graphs. Section 1.5 identifies these along with several optimizations of the initial solution aimed at minimizing such problems, an expanded discussion of which forms a major portion of this thesis. The chapter concludes with a synopsis of the remainder of the thesis.

1.2 Data Flow Graph Operation

The basic components of directed *data flow graphs* are *operators* and *arcs* which join the operators. When an operator *fires*, it absorbs values or *tokens* from its input arcs and produces tokens on its output arcs. There are three operator types, and corresponding rules defining their operation or *firing behavior*. The graph in Figure 1.1 which represents the VAL conditional construct:

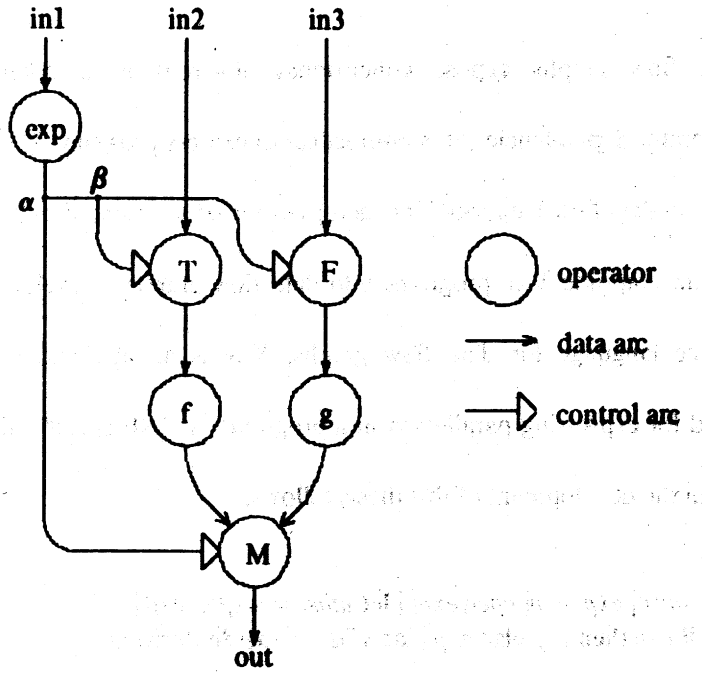
if *exp* **then** *f* **else** *g*

contains instances of each type. The *exp* node is an abbreviation for a VAL expression representing the predicate of the conditional. Thus, it should evaluate to a boolean value.

The most generalized operator type is the functional operator, represented in the figure by nodes *f* and *g*. These operators may perform simple arithmetic operations such as addition or multiplication, or more complex functions such as square root. The firing behavior rule for functional operators specifies that a token be present on each input arc for the operator to fire, at which time all inputs are absorbed, the appropriate function is computed and a result token is produced on each of the operator's output arcs.

The true and false control gates represented in Figure 1.1 by the T; and F nodes form a second operator type. Each of these operators requires a control and a data input to fire, and operates according to the following rule: If the control input matches the gate type, the data input is transmitted to the gate's output arc, otherwise the input data token is absorbed and no output is produced. Thus, a

Figure 1.1. Data flow graph of the VAL expression 'if *exp* then *f* else *g*'



T gate (F gate) will transmit its input data token to its output arc if and only if it receives a true (false) input control token.

The remaining operator type is the M gate, or merge control gate, which has three inputs; a control input, and two data inputs corresponding to true and false control input values. To fire, an M gate requires an input control token and corresponding input data token which is then transmitted to the gate's output arc. A value present on the input data arc not selected, is unaffected by the gate's firing. Appropriately, the M gate merges two paths in the graph. Thus, Figure 1.1 models the conditional construct behavior by allowing an input token to flow through either the T or F gate, (based on the evaluation of *exp*), to the M gate which merges the true and false paths to produce a result token on the graph output port.

1.3 Translation of VAL to Data Flow Graphs

While data flow graphs expose concurrency inherent in a computation by explicit representation of operator dependencies, it is impractical to express programs in this form. Instead, we introduce the high level data flow language VAL, acronym for value-oriented *a*lgorithmic *l*anguage, and a translation algorithm mapping VAL programs into data flow graphs. Developed by Ackerman and Dennis[2] as a source language for data flow graphs, VAL is an applicative language containing constructs well suited for expressing parallelism in a program. A BNF specification of the syntax of a subset of VAL, used in the development of this thesis follows.

exp ::= *id* | *const* | *exp*, *exp* | *oper*(*exp*) | **let** *idlist* = *exp* **in** *exp* |
if *exp* **then** *exp* **else** *exp* | **for** *idlist* = *exp* **do** *iterbody*

iterbody ::= *exp* | **iter** *exp* | **let** *idlist* = *exp* **in** *iterbody* |
if *exp* **then** *iterbody* **else** *iterbody*

id ::= "programming language identifiers"

idlist ::= *id* {, *id*}

const ::= "programming language constants"

oper ::= "programming language operators"

The recursive translation algorithm mapping VAL expressions into their data flow graph implementations, defined by J. D. Brock[3], consists of the functions $\mathbf{T}$ and $\mathbf{T_I}$ which respectively map VAL expressions and iteration bodies into their graph implementations. Both functions produce graphs which have an input port for each free variable in the expression or iteration body being translated. $\mathbf{T}[\![exp]\!]$ has an output port for each value returned by the expression; $\mathbf{T_I}[\![iterbody]\!]$ has two sets of output ports, I and R, used respectively to re-iterate or return a set of values, and an output port *iter?* to signal which possibility has occurred. Translations of the conditional and iteration expressions are used

extensively in this thesis, and are shown in Figures 1.2 and 1.3 respectively.

Functioning of the conditional expression in Figure 1.2 should be clear from the discussion of Figure 1.1. Evaluation of $\mathbf{T}[exp_1]$ should produce an input control value for all gates in the graph, allowing tokens to flow through either the T or F gates, enabling computation of the graph represented by $\mathbf{T}[exp_2]$ or $\mathbf{T}[exp_3]$ respectively. The iteration expression of Figure 1.3 is formed by using M gates to merge the values resulting from evaluation of *exp*, with the iteration, I, outputs of $\mathbf{T}_\mathbf{I}[iterbody]$. The control input port of each M gate is connected to the *iter?* output of $\mathbf{T}_\mathbf{I}[iterbody]$, initialized with a false token to ensure that selection of the first set of data values is from $\mathbf{T}[exp]$. A set of data values will be iterated as long as successive *iter?* outputs are true and will be returned at the first instance of a false *iter?* output, which reinitializes the M gates. A more detailed explanation of the application of the translation algorithm to the conditional and iteration expressions, as well as to the remaining expressions specified in the VAL subset defined above, can be found in [3].

Figure 1.2. $\mathbf{T}[$**if** exp_1 **then** exp_2 **else** exp_3 **end**$]$

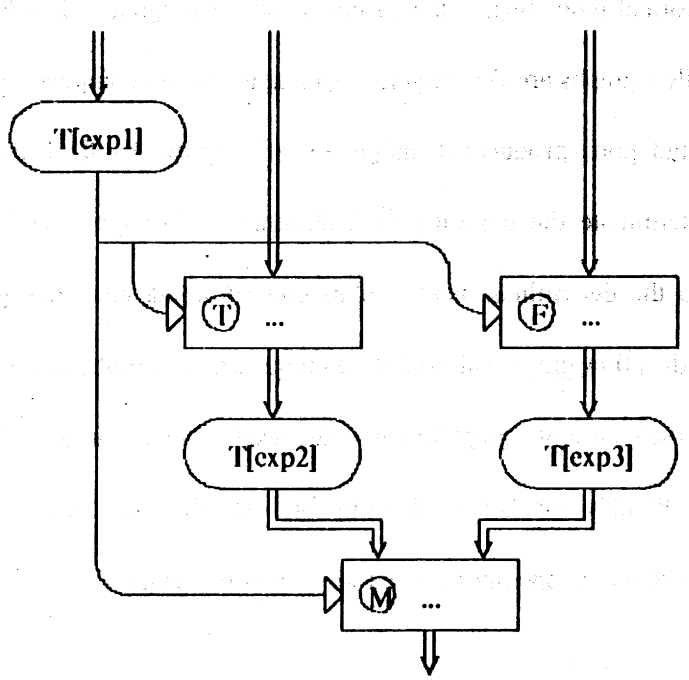
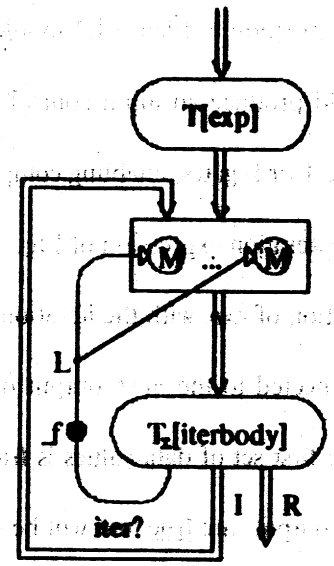


Figure 1.3. T[for *idlist* = *exp* do *iterbody* end]

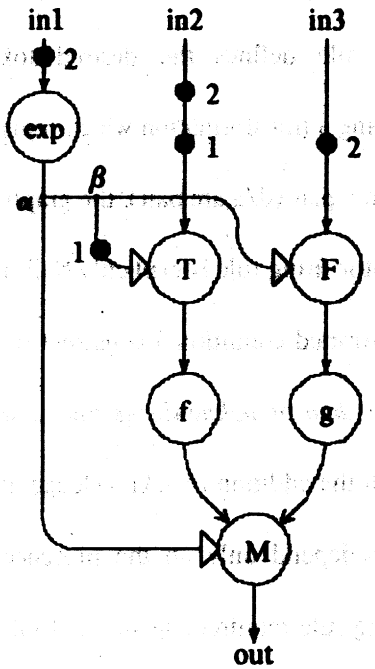


A major concern in generating data flow graph implementations of VAL expressions is ensuring correct modelling of the semantics of each high level construct. In fact, the translation algorithm is part of a two step process giving the operational semantics for the VAL subset: The *operational semantics* of a data flow program is a formal modelling of the execution of the program's data flow graph. The operators composing data flow graphs are *determinate*, meaning that every complete set of inputs to an operator (one for each input port) produces a unique set of outputs. Patil[25] proved that if the operators of a graph are determinate, the graph itself is determinate. Developing operational semantics for VAL is possible due to the determinate nature of its corresponding data flow graphs. Thus, a complete set of inputs to a data flow graph will produce a unique set of outputs, making it necessary to examine only one execution sequence of a graph to derive the result of its execution. The graphs in this thesis are generated from Brock's translation algorithm and are therefore assumed to be *correct* semantic representations based on the operational semantics developed in [3].

1.4 Safety Transformations for Data Flow Graphs

Though we accept the data flow graphs generated by the translation algorithm discussed in the previous section as theoretically correct, their arcs are assumed to be infinite queues -- this prevents their realization. While it might be possible to implement the graphs using sufficiently large finite buffers, this solution may not be acceptable. To examine the problem, consider the state of the graph shown in Figure 1.4. The token configuration shown can be reached by assuming that the graph occurs within an iteration construct which recycles the output of the construct. The second set of inputs shown could therefore have been generated in response to the output resulting from the first set of inputs. Assuming that the output of this first set was produced by propagating tokens through the false branch of the graph, it would be possible for the corresponding T gate inputs (tokens labelled 1) to still be present when the second set of tokens arrives, creating the computation state shown.

Figure 1.4. Unsafe token configuration resulting from infinite queue arcs



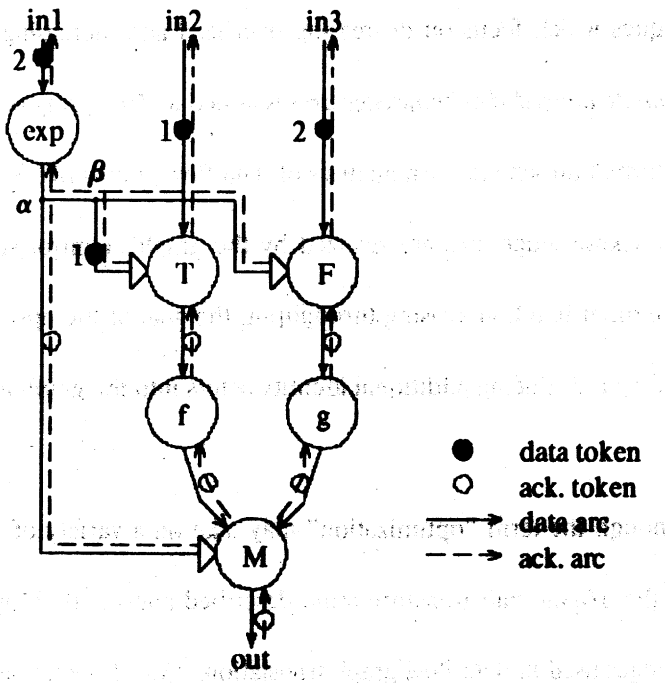
While an implementation of graph arcs as buffers of some constant size (greater than one) could accommodate this configuration, the design of a number of data flow architectures, including that of the Dennis-Misunas data flow machine, cannot support this: The correspondence of graph arcs to machine registers in such designs makes it necessary to consider only those execution sequences which place *at most one token on any given arc at any time.* In the Dennis-Misunas data flow machine, the consequences of placing more than one token on an arc or correspondingly, computing a successive register value before it can be stored, are possible nondeterminism, and deadlock as a result of values queueing up in its distribution network and blocking other values from reaching their destinations[24]. Meeting the one-token operational requirement involves preventing data flow operators from producing new tokens until their output arcs are empty. This behavior is achieved by defining the following firing rule for all graph operators:

> **Operator Firing Rule:** An operator is enabled to fire when all of its needed inputs are present and all of its output arcs are empty.

Application of this rule prevents the Figure 1.4 state from occurring.

While the operator firing rule defines the desired token behavior, the problem of implementation remains. By performing a transformation which replaces each arc of a data flow graph by an appropriate data/acknowledge arc pair (d/a arc pair), the graph's infinite queues are replaced by buffers of capacity one, and the operator firing rule is explicitly built into the graph. This is illustrated in Figure 1.5, which shows the transformed conditional construct of Figure 1.4. The transformation creates arc pairs which hold either a *data* or *acknowledge* token, where the later indicates that its corresponding data arc is empty. With the addition of acknowledge arcs and tokens, firing rules revert to their original specifications which depend only on the presence of tokens on input, including acknowledge, arcs: The operator firing rule requirement that output arcs be empty is ensured by the

Figure 1.5. Transformed Figure 1.4



enabling condition that acknowledge inputs be present.

The keyword used in describing this transformation is *safety*, where the underlying idea and the terminology is rooted in Petri net theory. Chapter 2 discusses the analogy between data flow graphs and Petri nets, and the influence of Petri net theory on the *safety transformation*. Included in the same chapter is a more detailed description of the transformation, and a consideration of its effect on the correctness of graphs.

1.5 Optimizing Transformed Data Flow Graphs

While the transformation of data arcs to d/a arc pairs enables the implementation of data flow graphs, it is imperative to question the cost of the acknowledging scheme and determine the inefficiencies, if any, that are introduced. In fact, there is much to say concerning these issues. Aside from the obvious overhead involved in incorporating acknowledge arcs and tokens, the constraints

which they impose on graph operation may cause bottlenecks. In response to this, we have developed optimization techniques which focus on decreasing overhead and increasing graph throughput. The optimization to *eliminate unneeded acknowledge arcs* is aimed at decreasing overhead, thereby reducing the cost of the transformation scheme. An analysis of data flow graphs of VAL constructs indicates that the effect of certain acknowledge arcs are realized by the graph's control structure, making the arcs unnecessary. On the other hand, increasing throughput, the goal of the optimization to *balance token flow*, is accomplished by introducing additional identity actors into the graph and consequently creating more d/a arc pairs.

Note that though the term "optimization" may take on a variety of meanings, our use of the word is confined to the d/a arc pair transformation described above: Both optimizations consider the number of acknowledges used in data flow graph translations. We do not consider program dependent optimizations which might typically involve modification of a graph's structure, i.e., removal of unnecessary data arcs or operators. This latter form of optimization is analogous to standard optimization techniques for conventional sequential programs and, though not yet fully explored, should prove readily adaptable to data flow.

1.6 Structure of Thesis

Having established a foundation, we proceed to consider the main tasks identified. Chapter 2 expands on the safety transformation introduced in section 1.4, and discusses related relevant theory. Chapters 3 and 4 respectively contain a development of the optimizations to balance token flow, and eliminate unneeded acknowledge arcs. Conclusions are presented in chapter 5 along with suggested areas for future research.

CHAPTER TWO

2.1 The Safety Transformation

The aim of the data/acknowledge arc pair transformation of data flow programs is to implement the operator firing behavior, defined in chapter 1, and restated here:

> **Operator Firing Rule:** An operator is enabled to fire when all of its needed inputs are present and all of its output arcs are empty.

This rule reflects the correspondence of data flow graph arcs to machine registers, which requires that the occurrence of more than one token on any arc be prevented. Restricting data flow graph behavior in this manner is necessary to ensure determinate and deadlock free execution for the architecture assumed. The analogy between the data flow graph characteristics of determinacy and deadlock and the Petri net theory properties of *safety* and *liveness* suggests the use of Petri net theoretical results to formulate and verify the d/a arc pair transformation. In fact, the strategy taken in developing the safety transformation is to extract relevant Petri net concepts and redefine them for data flow graphs.

This chapter proceeds with a closer look at the data flow graph - Petri net analogy, particularly focusing on the possibility of modelling the former with the later. Section 2.3 expands on the safety transformation and its effect in guaranteeing determinate (safe) and deadlock free (live) operation. While showing the existence of the former is straightforward, a significant question concerns whether or not the restrictions imposed to ensure safety affect liveness.

2.2 The Petri Net - Data Flow Graph Analogy

2.2.1 History and Analogy

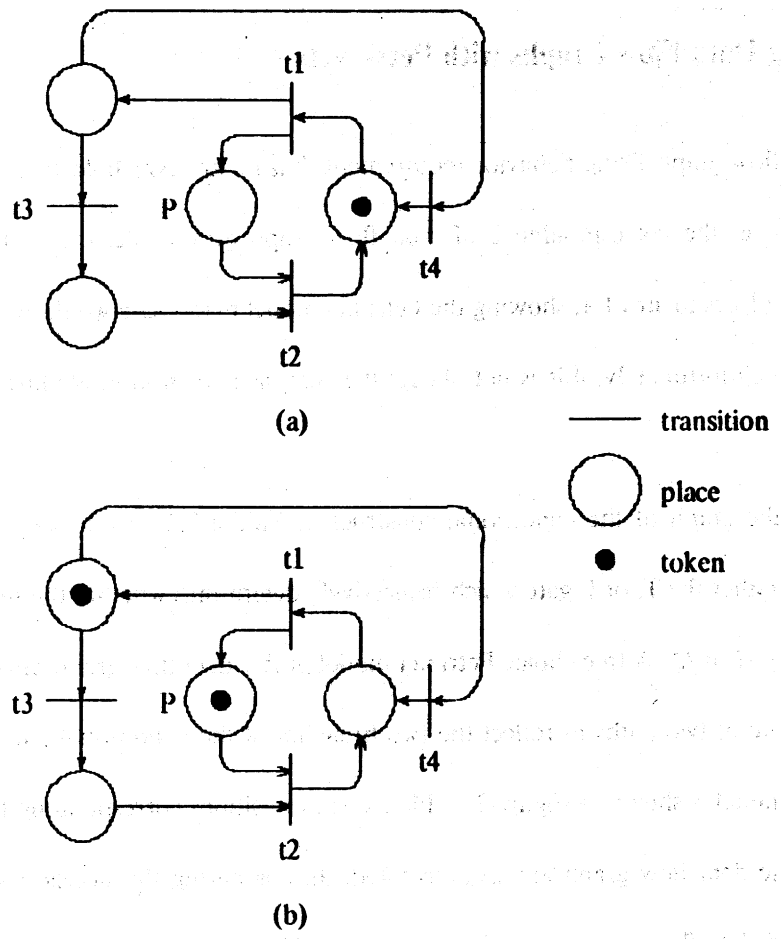
The major contribution of Petri nets is to aid in understanding systems. A closer look at the components of Petri nets seems an essential first step. As shown in the Figure 2.1 example, a Petri net is a graph composed of *transitions* and *places* with an initial *marking* determining the number of *tokens* (pieces of data) residing on each place. The transitions and places correspond respectively to data flow graph operators and arcs. A token must reside on each input place to a transition for it to be *enabled* for firing, where *firing* the transition causes a token on each input place to be removed, and one to appear on each output place. Figures 2.1(a) and (b) respectively show the Petri net token configuration before and after firing transition t1. The operation of a Petri net is *safe* if it behaves according to the following definition:

> Definition. For a marking M, a Petri net is *safe* if for every marking M' that can be reached by a sequence of firings from M, there is at most one token on any place.

This is precisely the behavior that we would like data flow graphs to satisfy. Note that the Figure 2.1 graph is, in fact, not safe since the sequence of transition firings: t1; t4; t1 will place two tokens on place P.

We briefly survey the evolution of Petri nets to introduce the theoretical results that could prove applicable to data flow. Petri nets were initially presented by Petri in 1962 [26] and modified by Holt in 1968 [15]. Extensive study of safety and liveness for Petri nets of the *marked graph* and *state machine* varieties has been done by Holt and Commoner [16]. Each of these classes form a particular subset of *free choice* Petri nets. This work has been extended by Michel Hack [14] to include free choice Petri nets. Hack introduces production schemas, similar to data flow graphs, and asserts that

Figure 2.1. Petri net token configuration before and after transition t1 firing



every production schema can be represented by a free choice Petri net. A major result known as the liveness-and-safeness theorem states circumstances under which a free choice net displays these properties. We explore the possibility of using such a result in producing determinate and deadlock free data flow graphs. Guaranteeing safety for free choice Petri nets involves ensuring that every place is part of some directed cycle containing one token. This fact should prove useful in determining if a data flow graph is safe, or in modifying it to be safe: We seek a modelling of data flow graphs by free choice Petri nets which allows us to conclude that a data flow graph is safe and live if its corresponding

Petri net is safe.

2.2.2 Modelling Data Flow Graphs with Petri Nets

The data flow graph firing behavior requirement that no arc ever hold more than one token, forces us to focus on the correspondence of data flow graph arcs to Petri net places. Were the correspondence of places to arcs 1-1, showing the Petri net model places safe would prove the data flow graph arcs "safe". Unfortunately, this is not always the case, as is seen in modelling data flow graph control structures.

Consider the graph of the conditional construct in Figure 2.2. Evaluation of the predicate results in enabling either the T, or F gate which respectively determines whether the input data value x will be processed by *f1* or *f2*. A free choice Petri net model of this data flow graph must enable a token to procede down one of two paths to reflect the two branches of the conditional and must merge the paths. A possible model is shown in Figure 2.3. Places and transitions corresponding to particular arcs and operators in the data flow graph are so designated. In comparing the decision structures of the Petri net model and data flow graph, note that place aa' in Figure 2.3 represents two arcs in the data flow graph. Although the mapping between places and arcs is clearly not 1-1, the Petri net decision structure presented is essential for allowing a token to take one of two paths. Unfortunately, this makes it more difficult to determine how properties of *place* aa' correspond to those of *arcs* a and a'.

A significant difference in the actual control structure is the absence of specific places and transitions in the model to represent the data flow graph predicate and its output control arcs. Whereas the decision concerning which branch of the conditional construct will be executed is uniquely determined by the output of the predicate, the Petri net is *nondeterministic*, providing a model for all possible decisions: Though each token arriving at place aa' will cause only one path of the Petri net to

Figure 2.2. Conditional construct data flow graph

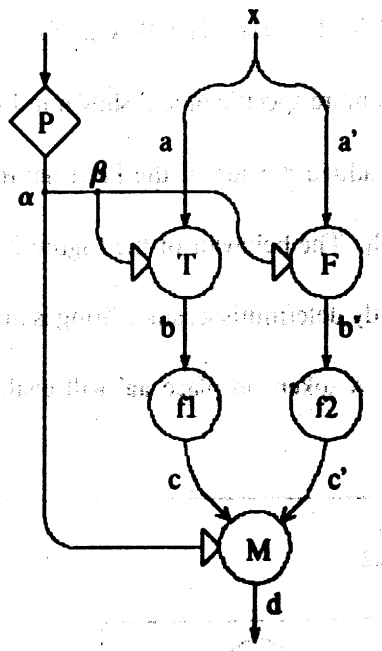
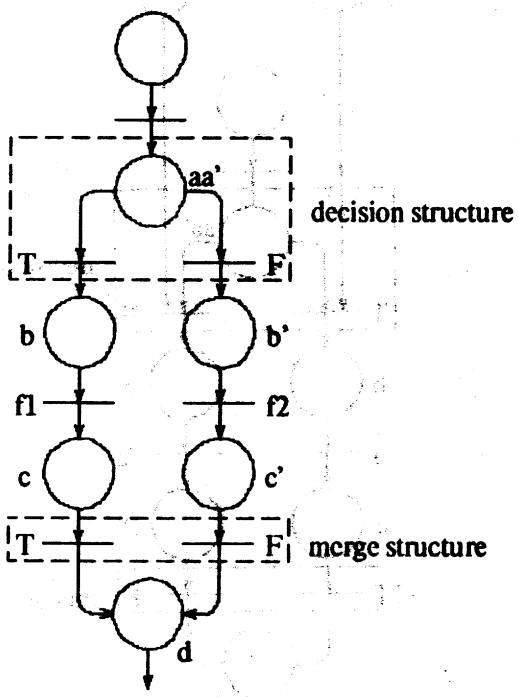
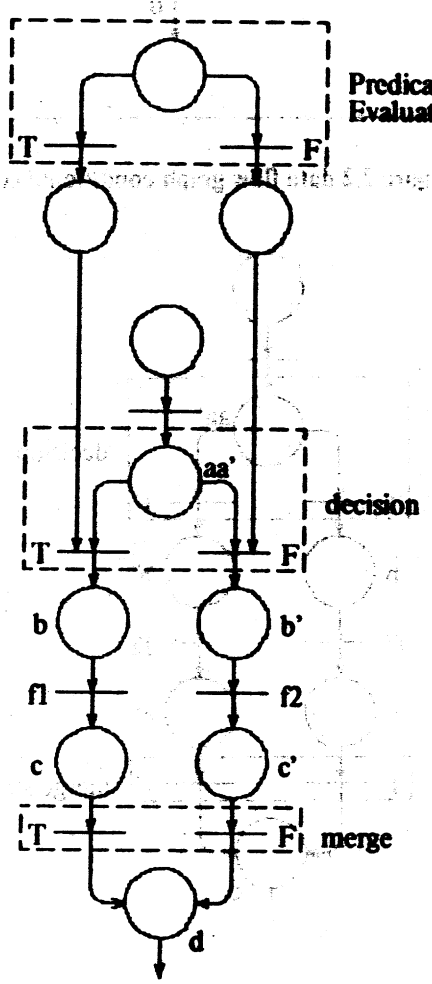


Figure 2.3. Petri net model of Figure 2.2 data flow graph conditional construct



become active, both paths are potential candidates. This situation emphasizes the use of Petri nets as general models for specific systems - in this case, data flow graphs [22]. To remedy the modelling problems of the Figure 2.3 Petri net, a more specific model shown in Figure 2.4 is built which attempts to localize the nondeterminism in an added portion of the Petri net meant to represent the predicate and control arcs of the data flow graph. The behavior of the Figure 2.4 transitions modelling the data flow graph T and F gates is consequently deterministic, since firing is now dictated by the portion of the net labelled "predicate evaluation". A token on place 'aa' will enable either the T or F transition,

Figure 2.4. Petri net model of Figure 2.2



thereby determining its path.

Though this Petri net modelling of the conditional construct more accurately captures the data flow graph behavior, the portion of the net representing the T and F gates violates the structure defining the *free choice* subset of Petri nets: If a transition following a particular place is firable at a marking M, then all transitions following that place are firable at M. Informally, the definition of a free choice Petri net states that every arc from a place must be either the unique output of the place or unique input to a transition. Thus, the configuration involving place aa' and the T and F transitions in Figure 2.4 violates the free choice property. Since free choice nets form the largest subset of Petri nets for which a developed theory of liveness and safety exists, there is no advantage to pursuing this modelling route. For this reason we change directions, attempting to accomplish our goals more directly by extracting the relevant concepts of Petri net theory and redefining them for data flow.

2.3 The Data/Acknowledge Arc Pair Transformation

2.3.1 Achieving Safe Data Flow Graph Operation

Since the Petri net properties of safety and liveness reflect the behavior we want data flow graphs to display, we attempt to redefine these terms for data flow via the correspondence of arcs and operators to places and transitions.

> **Definition.** For an initial configuration of tokens, a data flow graph is *safe* if every configuration of tokens that can be reached from the initial configuration contains at most one token on any individual arc.
>
> **Definition.** An initialized data flow graph is *live* if a complete set of inputs will eventually cause a complete set of values to appear on the output arcs of the graph.

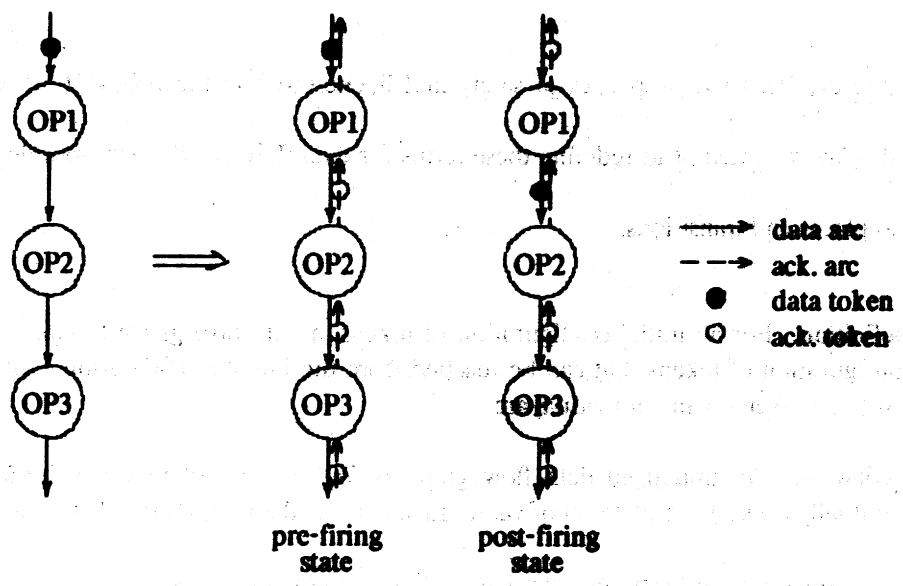
To ensure safe operation in Petri nets, every transition in the net must be part of a one-token directed cycle. Adapting this for data flow is accomplished by introducing initialized *data/acknowledge arc pairs*

(d/a arc pairs) and ensuring that every arc in a data flow graph is part of such a pair.

The mechanics of the transformation illustrated in Figure 2.5 involves replacing each full data arc with an arc pair composed of a full data arc and empty acknowledge arc; and each empty data arc with an arc pair composed of an empty data arc and full acknowledge arc. Alternatively, Brock's T algorithm can be modified to produce graphs with d/a arc pairs, rather than infinite queue arcs. We distinguish the two by terming such an algorithm $T_{d/a}$, as opposed to T_{∞}. The Figure 2.5 graph segment labelled, "pre-firing state" represents the transformation of the graph segment to its left. Having defined this transformation we must verify that, in fact, it accomplishes its intended function - to ensure the safety and liveness of data flow graphs.

An initially transformed graph is potentially safe since each of its arc pairs holds only one token. What must be shown is the preservation of this property under firing. In the pre-firing state of the Figure 2.5 graph segment, OP1 is the only enabled operator since it is the only operator which has

Figure 2.5. D/A arc pair transformation



tokens present on each of its input arcs. Firing OP1 produces the post-firing state shown. The firing action results in the absorption of a token from each of OP1's input arcs and the production of a token on each of its output arcs. Consequently, OP1 is disabled, and OP2 becomes the only enabled operator. More importantly, OP1 cannot be reenabled until it receives both a data, and an acknowledge input, where the appearance of the later is dependent on firing OP2: Firing OP2 will absorb its input data token and produce an acknowledge token, input to OP1. Thus, OP1's output data arc must be empty for it to fire a successive time, producing a new data output. This reasoning, shows the firing behavior dictated by the data/acknowledge arc pair transformation is safe.

2.3.2 Preservation of Liveness

Verifying liveness of data flow graphs under the d/a arc pair transformation is more difficult. Due to its determinate nature, a result obtained from a $T_{d/a}$ graph will match that of its corresponding T_{∞} graph: Any $T_{d/a}$ graph firing sequence is a legal firing sequence in the T_{∞} graph. The question to address is therefore, whether the firing rule constraint causes some $T_{d/a}$ graph to deadlock that would not have done so in its T_{∞} version.

The intuitive feeling that T_{∞} graphs and their corresponding $T_{d/a}$ graphs produce the same results is established via the theorem stated below. Its proof consists of a structural induction on the size of data flow graph expressions. By asserting an induction hypothesis for expression subgraphs, we show that the liveness property holds for $T_{d/a}$ graphs composed of acyclic interconnections of *exp* subgraphs, or graphs whose top level is a conditional or iteration expression.

In analyzing the $T_{d/a}$ iteration expression, we have to make some assumption about the behavior of its *iterbody* operator which represents an iteration subgraph. Recall that the T_1 translation function produces iterative graphs which have one set of input ports and two sets of output ports

through which values can be iterated or returned, as well as a control output port to signal which of the two occurs. The behavior of the ports of an iterative subgraph within a well-formed live T_{∞} graph can be characterized as follows: When presented with n sets of inputs, the subgraph will produce n *iter?* control values-- k true ($0 \leq k \leq n$) and n-k false; and correspondingly, k sets of I data values and n-k sets of R data values for a total of n data output sets. To prove liveness for a $T_{d/a}$ graph containing an *iterbody* operator, we must first show that the port behavior of $T_{d/a}$ iterative subgraphs is the same as that displayed by T_{∞} iterative subgraphs. This will allow us to assume the desired *iterbody* port behavior, an essential step in proving the expression live.

Proving the correct port behavior for $T_{d/a}$ iterative subgraphs consists of a subproof occurring within the larger inductive proof. Since the iteration expression contains the only instance of an *iterbody* operator, the subproof should naturally appear just prior to proving the $T_{d/a}$ iterative expression live. However, to stem confusion only a statement of the assumed *iterbody* operator port behavior will be made: An outline of the subproof follows the inductive proof. Finally, inherent in this discussion is the assumption that the equivalence of T_{∞} and corresponding $T_{d/a}$ graphs is being shown for graphs which are *well-formed*, where this term is defined as follows:

> **Definition.** A *well-formed* data flow graph is derived from a syntactically correct VAL program using the T_{∞} translation algorithm.

We proceed with the liveness theorem.

> **Theorem:** A well-formed live data flow graph will remain live under the d/a arc pair transformation.

Stated in operational terms:

Any $T_{d/a}$ graph corresponding to a well-formed live T_{∞} graph, when presented with n complete input sets will either:

(1) have produced n complete output sets and absorbed n acknowledge sets on its output d/a arc pairs, and emitted n acknowledge sets on its input d/a arc pairs, or

(2) contain some enabled operator.

Proof:

Basis: A data flow graph consisting of a single functional operator will remain live under the d/a arc pair transformation.

An initialized functional operator is shown in Figure 2.6. On receipt of a complete input set, the operator will be enabled and when fired, will produce an output token absorbing the acknowledge token on its output arc pair, and emit acknowledge tokens on its input arc pairs. Since the operator's output arc pair is the graph output arc pair, within finite time the output token will be absorbed and a corresponding acknowledge token supplied reinitializing the graph. If an nth set of inputs has been presented to the operator and an nth output has not appeared, then the acknowledge arcs of the input arc pairs must have seen their nth acknowledges, n-1 of which were produced by firing operator *f*. This implies that the state of the output d/a arc pair is one of the following: The data arc has its n-1st data

Figure 2.6. Initialized data flow graph of a functional operator

value and the acknowledge arc is empty but has seen n-1 acknowledge tokens; the data arc is empty and the acknowledge arc is holding its nth acknowledge token. In the first case, within finite time the n-1st data value will be absorbed and an nth acknowledge token produced reenabling the operator. In the second case the operator is enabled.

Induction Hypothesis: In response to an nth complete input set, an *exp* operator (expression subgraph) will either:

(1) have produced an nth complete output set and absorbed an nth acknowledge set on its output d/a arc pairs, and emitted an nth acknowledge set on its input d/a arc pairs, or

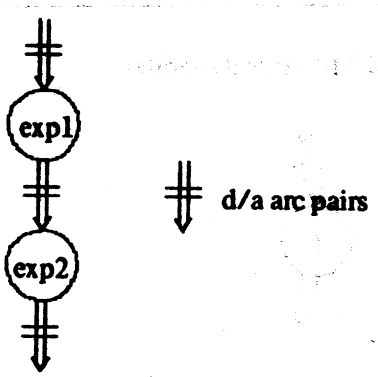
(2) contain some enabled operator.

Acyclic Interconnection of exp operators

Assume that the Figure 2.7 graph has been presented with an nth set of inputs and that it has not produced an nth output set. We will show that the graph must contain an enabled operator.

Suppose the graph has produced j output sets where j<n, and the output arc pairs have had their jth data values absorbed, and are holding their j+1st acknowledge tokens. This implies that exp_2 must have seen at least j input sets. Three possibilities arise.

Figure 2.7. Acyclic interconnection of expression subgraphs



Suppose exp_2 has not yet seen its j+1st input set. Then by the induction hypothesis, since exp_1 has seen its nth input set and only emitted j output sets where j<n, exp_1 contains an enabled operator.

Suppose exp_2 has seen part of its j+1st input set. Then by the induction hypothesis since exp_1 has seen its nth input set and not yet emitted a complete j+1st output set where $j+1 \leq n$, exp_1 contains an enabled operator.

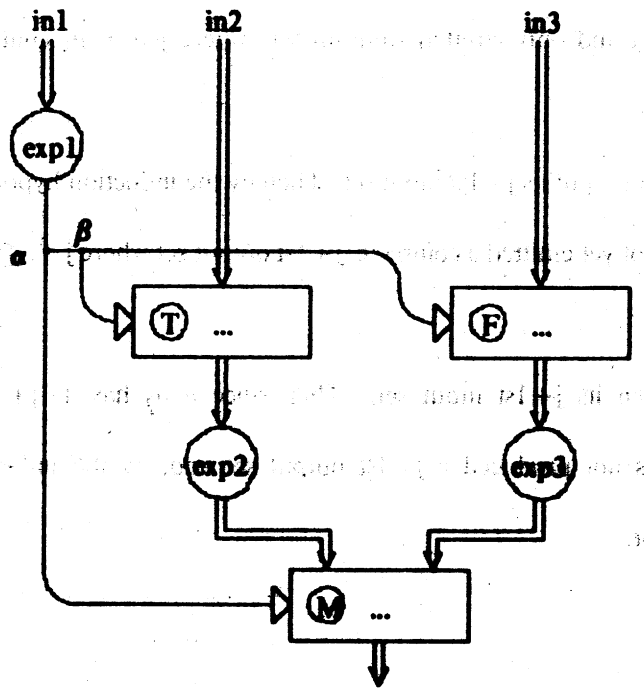
Suppose exp_2 has seen its j+1st input set. Then since exp_2 has its j+1st set of input acknowledges available, it has not produced a j+1st output set and, by the induction hypothesis contains some enabled operator.

Conditional Expression

The conditional expression is shown in Figure 2.8. In its T_∞ form, when presented with n inputs, exp_1 will produce n boolean outputs; k true where $0 \leq k \leq n$ and n-k false. In response to this, the M gates will see a total of n data input sets -- k on their true data input arcs and n-k on their false arcs. These are merged to produce the graph outputs according to the n M gate control inputs (k true; n-k false) which correspond to the M gate data inputs.

An important consequence of the d/a firing restriction is that once a control input value is presented to the M gate, a successive control input cannot appear on that control arc (between α and the M gate) until the M gate fires to absorb the previous value and emit an acknowledge token. The implication of this is that α is prevented from firing a successive time to reenable any gates in the graph before the output set corresponding to the previous control value has been produced. This in turn implies that only one input set will be within the branches of the conditional expression at any time.

Figure 2.8. Conditional construct data flow graph



Assume the graph has received an nth set of inputs. Assume further, that no operator is enabled within exp_1. By the induction hypothesis exp_1 must have produced an nth output set. The d/a arc pair between α and the M gate can be in one of two states. Either the arc pair is holding its n-1st control value, or it is holding an nth acknowledge token. Assume the arc pair is holding its n-1st control value. By the functioning of the graph described above, this implies that the n-1st input set is being processed. Since the graph has received its nth input set, this implies that the T and F gates must have emitted an n-1st set of acknowledges by firing in response to their n-1st set of inputs. We can assume as a result, that either exp_2 or exp_3 becomes enabled. By the induction hypothesis, within finite time we will see the n-1st output set on the appropriate *exp* output data arcs and an nth set of acknowledges on the *exp* input arc pairs. This action enables the M gates which when fired will produce an n-1st set of graph outputs and emit acknowledge tokens along its data and control input arcs. At this point, the arc pair between α and the M gate is in its second possible state-- holding its nth

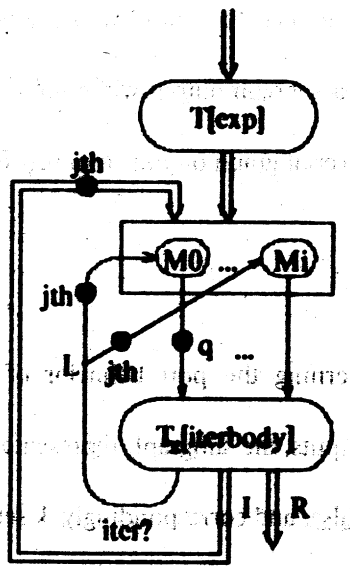
acknowledge. Note that if α is fired, which is now possible, the graph will be in the state it was in when the arc pair between α and the M gate held its n-1st control value. Since within finite time the n-1st set of graph outputs will be absorbed and each graph output will hold an nth acknowledge, we can repeat the above reasoning to show that an nth set of graph outputs is produced.

Iterative Expression

We assert the following concerning the port behavior of the *iterbody* operator: When presented with an nth complete set of inputs, the subgraph represented by *iterbody* will either produce n *iter?* control values -- k true and n-k false, and correspondingly, k sets of I data values and n-k sets of R data values or; will contain some enabled operator.

The iterative data flow graph is shown in Figure 2.9. We can make the following observations concerning the functioning of the graph in its $T_{d/a}$ form. Note that firing copy operator L causes each of the M gates to be presented with the next control input. The implication of this is twofold: Operator L cannot fire until every M gate has fired, absorbing its previous control input and emitting acknowledge tokens; the number of input sets processed by each M gate is either equal to, or one less than the number of control inputs that have been presented to each M gate. The operation of an iterative graph is such that a set of input values will be iterated in response to true *iter?* outputs until *iterbody* produces a false *iter?* output which signals return of the values. We consider these two stages of $T_{d/a}$ graph behavior -- iterating values and returning values, separately. Since the synchronizing affect of copy operator L prevents any interesting overlapping of graph input sets, it suffices to show that when presented with one complete input set the graph will produce an output set without deadlocking. We begin with the return case.

Figure 2.9. Iterative data flow graph



Assume *iter?* produces a false value. By the first implication above, L cannot present the M gates with this value until each has fired to acknowledge L and produce a data input to *iterbody*. Thus *iterbody* must see a complete set of inputs for the M gates to be reinitialized. The stated behavior of *iterbody* dictates that within finite time a complete set of return values will be produced in correspondence with the false *iter?*. Thus if the M gates are reinitialized, a set of outputs is guaranteed without the possibility of deadlocking. The possible ways of a deadlock occurring are considered in the iterative path argument which follows.

We proceed to show that a deadlock does not occur within the iterative path of the graph by assuming the opposite and reaching a contradiction, supporting the conclusion that an enabled operator exists within the graph. Assume that there exists some well-formed live iterative data flow graph which deadlocks under the d/a arc pair transformation. To see how the deadlock occurs we apply the same

sequence of computation steps to a T_{∞} graph and its corresponding $T_{d/a}$ graph, until we reach a state where there exists some operator which is enabled in the T_{∞} graph and not enabled in the $T_{d/a}$ graph. The cause of deadlock must be that an operator in the $T_{d/a}$ graph has its inputs available, but cannot fire due to the presence of a token on its output arc. We attempt to locate this operator, which must be an M gate or a gate within *iterbody*. We proceed to consider each case.

Assume merge operator Mo is in such a state, and that it has its jth set of iteration inputs available. The token on its output arc, labelled q, must be used in producing the I iterative input value of some other M gate, say Mi. Since the $T_{d/a}$ graph is deadlocked, one of two situations must exist:

(1) The path taken by token q through *iterbody* to the I input of gate Mi is blocked (every arc is full).

(2) Token q is input to some operator which lacks some input and therefore is not enabled.

Assume (1). Recall from our preliminary discussion of iterative graph operation, that if token q was produced as a result of the j-1st input set, it will be used to produce the jth I input of some M gate which, according to the assumption, is blocked. Thus, the token currently residing on the I input to that M gate must be part of the j-1st input set or some set previous to the j-1st set. This implies that the M gate has not yet fired j-1 times. But from our knowledge of iterative graph operation, this is not possible since firing copy operator L to present each M gate with a jth control input required the prior firing of each M gate a j-1st time sending j-1st acknowledges to L -- a contradiction.

Assume (2). Since firing L a jth time is only possible if each M gate has fired j-1 times, it must be that a complete set of inputs to *iterbody* is available contradicting the assumption that some input is not present.

Assume the disabled operator occurs as a result of *iterbody* and that its output arc is an I output arc. If the disabled operator has a jth set of inputs available, then they will be used to produce the j+1st I input of some M gate. The token on its output arc, must therefore be a jth I input of that M gate. By the twofold implication stated above, the fact that the disabled operator has its jth inputs available implies that every M gate was presented with a jth control input and has fired either j or j-1 times. Thus the M gate which has its jth I input available, must have fired j-1 times. If we can show that this M gate is enabled, then within finite time it will fire, sending an acknowledge to the blocked operator. Consequently, in finite time there will be an enabled operator within *iterbody*.

We know that the M gate has its inputs available, so it can only be disabled if its output arc is full. Assuming this situation, the token on its output arc must be from the j-1st input set and will be used to produce the jth input of some other M gate. But then we know that within finite time the operator to which this token is input will fire since by the twofold implication, every M gate has fired j-1 times. This simultaneously ensures that the operator has its inputs available and has an empty output arc. The acknowledge necessary to enable the M gate will be sent as a result of firing the operator. Thus, within finite time, the M gate and subsequently the blocked operator in *iterbody* will be enabled.

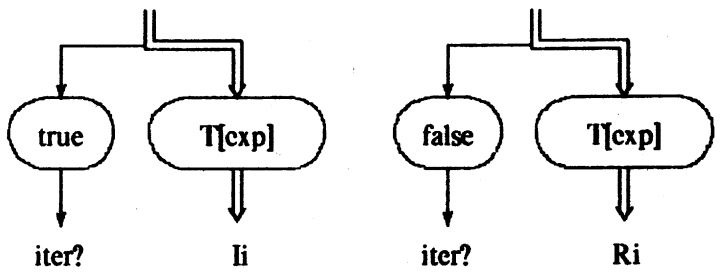
It follows that if the T_{∞} graph is well-formed and live, the corresponding $T_{d/a}$ graph is well-formed and live. Q.E.D.

The subproof concerning port behavior for iterative subgraphs is also inductive in that it must assume a behavior for iterative operators within subgraphs and then prove the behavior for the top level structures defining iterative subgraphs. The behavior to be shown has been stated above at the start of the section of the proof dealing with the iterative expression.

The simplest iterative structures, *exp* and **iter** *exp*, are shown in Figure 2.10. Since the iterative subgraph proof is within the inductive proof above, the induction hypothesis concerning *exp* subgraphs is valid. As a consequence, proving that the Figure 2.10 graphs satisfy the stated behavior is trivial. Establishing this fact for the conditional iteration body, **if** *exp* **then** $iteration_1$ **else** $iteration_2$, is tedious and will not be presented.

Having developed the data/acknowledge arc pair transformation and shown T_∞ and $T_{d/a}$ graphs equivalent, the task of determining the quality of this solution remains. Major concerns to investigate focus on cost and efficiency. Chapters 3 and 4 address these issues and present optimizations of the solution subsequently developed. Example graphs in the remainder of this thesis are assumed to have been produced by algorithm $T_{d/a}$. Therefore, though not explicitly shown, all arcs represent d/a arc pairs unless otherwise stated.

Figure 2.10.



(a) *exp* (b) iter *exp*

This empty page was substituted for a blank page in the original document.

CHAPTER THREE

3.1 Balancing Token Flow

The optimization to balance token flow discussed in this chapter addresses certain inefficiencies introduced by the acknowledging scheme presented in chapter 2. Though the d/a arc pair transformation prevents the occurrence of more than one token on an arc at any time, the firing restrictions it imposes are severe, and may significantly curtail concurrency. Specifically, the requirement that an operator receive acknowledge signals on each of its output ports before refiring, unnecessarily delays computation of successive input sets. While ensuring the safe operation of the graph is essential, it is possible to identify which output arcs are potential bottlenecks, and modify each so that it can be safely implemented as a fixed size buffer. The purpose of this change is to effectively enable arcs to hold more than one token, thereby eliminating bottlenecks by allowing computation of successive sets of inputs to "pipeline" through the graph. Safe implementation of these buffers involves the use of identity operators which, when inserted along an arc, act as place holders. Identifying arcs within a graph that may cause bottlenecks, and determining the extent to which they should be buffered are prerequisites to their modification. While the former of these tasks is straightforward, deciding on a buffering strategy is subject to a number of considerations including graph configuration and cost of buffering.

A simple example is presented in section 3.2 which clearly illustrates the problem addressed in this chapter, and serves to motivate the subsequent optimization. This discussion is formalized in an algorithm which produces optimized graphs. The section concludes by pointing out certain subtleties of graph operation and factors not accounted for in formulating the proposed solution. In response to this, section 3.3 introduces a modified version of the section 3.2 algorithm, along with several

comparative studies of graphs in their limited and fully buffered configurations.

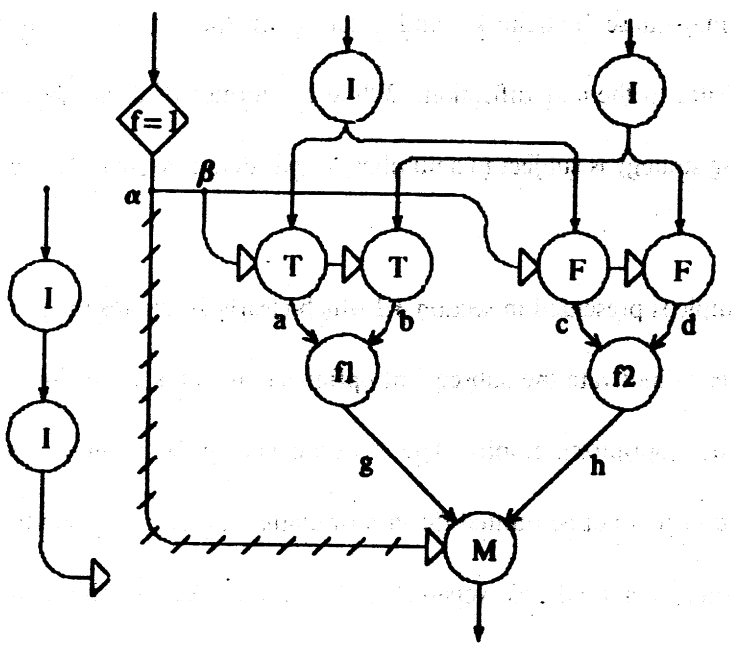
3.2 Formulating the Optimization

3.2.1 Identifying the Source of Bottleneck

The goal of the optimization to balance token flow through a graph is to increase throughput by modifying a graph to allow for maximum pipelining. The bottleneck problem, and therefore application of the optimization, arises in acyclic segments of a data flow graph. A clear illustration of the problem is shown in Figure 3.1, the graph translation of the VAL expression:

if f=1 **then** *f1* **else** *f2*

Figure 3.1. Buffering for a conditional expression



The interesting and problematic issues arise when considering the consequence of presenting the graph with multiple input sets. Hopefully, processing of a second set of inputs can begin before outputs of the previous set appear, with the optimum situation being one in which sets of inputs pipeline through the graph. Unfortunately, the control structure of the graph dictates that the overlap in processing of successive sets of inputs be minimal: Only one set of values may be within the branches of the outer conditional at any time. Referring to Figure 3.1, we see that in order for a second set of values to enter the branches of the conditional, both α and β must fire a second time presenting the sets of T and F gates with new control inputs. However, α cannot fire a second time until the M gate to which it also sends a control input has fired to emit an acknowledge. Thus, the d/a arc connecting α and the M gate (marked in Figure 3.1 by slashes), prevents sets of values from pipelining through the graph, creating a bottleneck whose severity depends on the depth of the computation performed within the branches of the conditional.

Eliminating this undesirable behavior so that successive sets of values may pipeline through the graph involves finding a method of enabling node α sooner, consequently allowing the slashed arc to hold more than one token. The ideal situation would be one in which the arc could hold as many tokens as the number of sets of values that could be pipelined through the graph.

3.2.2 Preview of a Solution

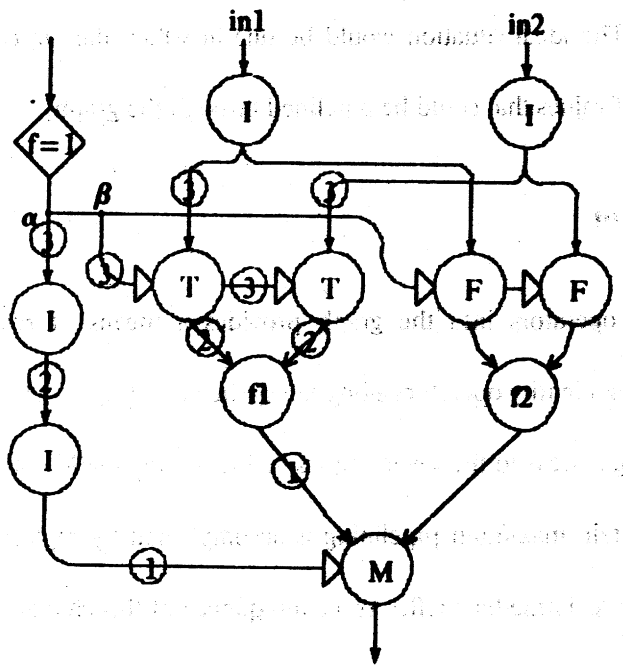
Introducing identity operators into the graph provides a means of realizing the desired behavior. Specifically, inserting identity operators along the slashed arc (Figure 3.1) would break it into d/a arc pair segments, allowing node α to fire several times before forcing the M gate to fire. Using this technique on Figure 3.1 to attain maximum pipelining is accomplished by replacing the slashed arc with the arc segment shown to its immediate left. As a consequence of this change, the state shown in

Figure 3.2 in which three sets of tokens are pipelining through the graph, can be reached. (The token sets have been numbered accordingly for clarity.) Thus the introduction of identity nodes has eliminated the bottleneck. Generalizing this optimization technique requires a determination of the ideal number and location of buffers to be inserted. To respond to such considerations, we attempt to analyze how tokens flow through the graph.

3.2.3 Analyzing Token Flow to Characterize the Solution

Though the data flow computer operates asynchronously and data flow programs nonsequentially, we can model optimum token flow through the graph by assuming a somewhat synchronous behavior. To do this, we analyze the firings within the graph in terms of time units where during any given unit of time all enabled actors must fire and produce a result. This assumption attempts to approximate optimum behavior by preventing an enabled actor from remaining enabled

Figure 3.2. Token configuration allowed by buffering scheme



and thereby slowing up processing for any length of time. Recalling that our aim is to pipeline computation through the graph, we wish to develop a method of modifying the graph so that under this "synchronous behavior" assumption it displays maximum pipelining and consequently, best throughput.

Referring back to Figure 3.1, we note that every input set to the graph results in the production of a token on the control (slashed) arc, and tokens that will either be processed by *f1* or *f2*. While under the synchronous behavior assumption the tokens being processed by these functional operators can move one step through the graph during every time unit, the control token on the slashed arc cannot, and must remain stationary until its corresponding tokens propagate through the graph to enable the M gate. As previously seen, the inability of the control arc to accept a second token prevents any tokens in a successive input set from being pipelined. The dependency between the control arc and the branches of the conditional, and the consequent need to equalize their buffering capacities to attain maximum pipelining has been recognized by the addition of identity nodes shown in Figure 3.2. An algorithm to equalize buffering along graph paths must be able to identify dependencies within a graph and pipeline their paths. This can be accomplished by an arc numbering scheme which compares and equalizes buffering capacities of dependent paths, recognized by identifying functional operators or gates which join two or more paths. An illustration of the algorithm which performs this optimization follows its presentation.

Algorithm to Maximize Pipelining -- I

Starting from each graph input, descend through the graph assigning consecutive numbers to arcs joining successive sets of operators until a multi-input operator is encountered. Compare the arc numbers on the input arcs of the operator and:

(a) if equal, continue the arc numbering process

(b) if not equal, balance the arcs by inserting identity operators into the lower numbered arcs. Renumber the modified arcs and continue the arc numbering process.

Note that if the operator is an M gate, the comparison and balancing described above must involve all three input arcs, using the highest numbered arc as the goal.

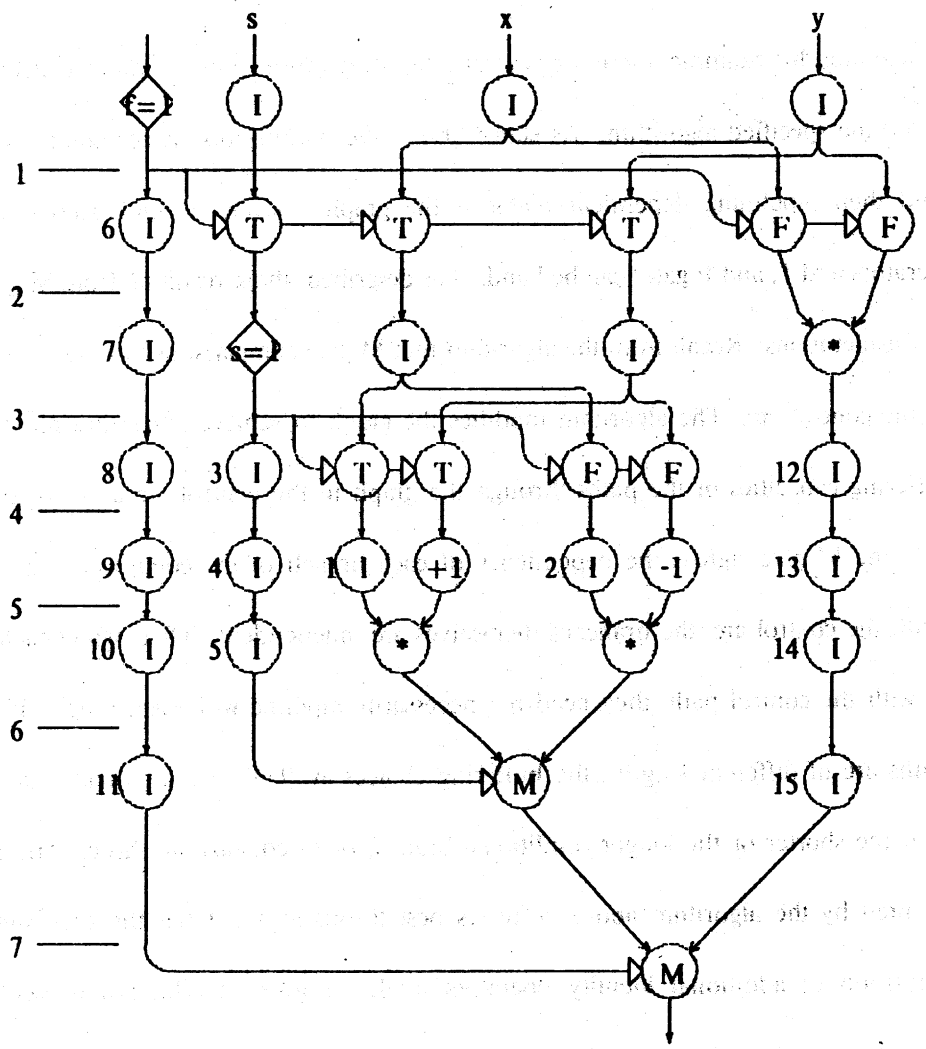
The result of applying this algorithm to the graph translation of the following program segment is shown in Figure 3.3:

if f=1 then if s=1 then x*(y+1) else x*(y-1) end else x*y end

For reference purposes, the added identity nodes have been numbered. The seven numbers shown at the extreme left of the graph result from the arc numbering process, and apply respectively to appropriate arcs moving horizontally across the graph. Nodes I1 and I2 have been added in response to the imbalances which occur when comparing arc numbers on the input arcs to the multiplication operators. I3 through I5 are added in response to the comparison of the input arcs to the inner M gate. Note that, as specified in the algorithm, arc number comparisons involve all three M gate input arcs. Finally, operators I6 through I15 are introduced as a result of comparing input arcs to the outer M gate.

One essential question to ask is whether or not the addition of identity operators changes the functionality of a data flow graph. This can be answered by recognizing that the essence of the change resulting from the application of Algorithm I is to replace some of the one-token arcs of a graph with queues of a given finite length. Since successive identity operators along the arc are separated by d/a

Figure 3.3. Example of maximal pipelining



arc pairs, the graph remains deterministic; and since an identity actor merely passes its input to its output arc, the functionality of the graph is unaffected. These observations ensure the functional