ABSTRACT

DATA FLOW PROGRAM REDUCTION

by

Daniel M. Dias

In data flow programs, instructions execute when their operands become available. The arrival of data thus acts as the control as well and leads to models with a single graph representing the data and the control flow. In this thesis, we transform a data flow model to one with a separate data and control flow. This is motivated by 'reductions' that can be carried out on the transformed model and by the ease of implementing such models by conventional techniques.

Data flow models are networks consisting of operators and control nodes connected by data paths. The operator nodes carry out a functional operation on the input data while the control nodes route the data correctly. In directly implementing a data flow program, 'copy' operations are required for the control nodes. We specify conditions which allow the removal of these copy operations in the transformed model. The problem is significant since the control nodes often make up more than half of the data flow program.

The control flow is then simplified and it is demonstrated that this simplification can lead to an increase in parallelism.
Finally, the implications of the analysis to extant architectures is considered and it is shown how the results motivate a separation of the data and the control flow in implementing a data flow processor.
ACKNOWLEDGEMENTS

I sincerely thank Dr. J. R. Jump, my thesis advisor, for his guidance, interest, incisive comments and patience. Thanks to Dr. E. A. Feustel and Dr. M. M. Blattner for serving on my thesis committee. A special thanks to Timothy A. Gonsalves for useful discussions and to Gopal Agarwal for helping in the typing of the manuscript. Thanks to my fellow graduate students, Naresh Jotwani, Anand Jagannathan, Bart Sinclair and G. A. Merchant for helping to make graduate study a pleasure.

This research was supported by the National Science Foundation under grant DCR 74-14283.
# CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>INTRODUCTION</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.1 The motivation for data flow</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1.2 Data flow concepts</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1.3 Related work</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1.4 The motivation for the thesis</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td>1.5 The plan of the thesis</td>
<td>16</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>THE DATA FLOW MODEL</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2.1 The specification of the model</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>2.2 Determinate, live, safe, well-behaved and conflict free programs</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>2.3 Well formed schemas</td>
<td>28</td>
</tr>
<tr>
<td></td>
<td>2.4 Properties of well formed schemas</td>
<td>30</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>SPLITTING DATA AND CONTROL FLOW</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3.1 Preliminaries - Petri nets</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>3.2 The specification of the transformation</td>
<td>44</td>
</tr>
<tr>
<td></td>
<td>3.3 The simulation of an RDFS by a CFS-DFS</td>
<td>56</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>REDUCTION OF THE CONTROL AND THE DATA FLOW STRUCTURES</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4.1 Reduction of the data flow structure</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>4.1.1 General conditions for the removal of identity operators</td>
<td>67</td>
</tr>
<tr>
<td></td>
<td>4.1.2 The removal of identity operators for structured nets</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>4.2 Reduction of the control flow structure</td>
<td>109</td>
</tr>
</tbody>
</table>
CHAPTER 5 THE IMPLICATIONS OF THE ANALYSIS

5.1 The implications for extant architectures

5.2 Splitting the data and the control flow in the implementation

CHAPTER 6 SUMMARY AND CONCLUSIONS

6.1 Summary.

6.2 Suggestions for further work

REFERENCES
CHAPTER 1
INTRODUCTION

1.1 The motivation for data flow.

Data flow programs and data driven languages provide an attractive solution to the problem of programming highly parallel processors. They allow a simple representation of parallelism even down to the instruction level, guarantee determinacy and avoidance of deadlocks, and still have the expressive power of conventional programming languages. Finally, computer architectures have been proposed to execute data flow programs efficiently.

In data flow programs and programs written in one of the related data driven single assignment languages, instructions are executed as soon as the operands are available. Thus a large degree of parallelism may be represented. On the other hand, the parallel programming constructs of conventional programming languages only allow the representation of a limited parallelism which compels the use of look-ahead schemes in high performance machines like the CDC 6600/CDC 64/. The amount of parallelism achievable is limited by branches in the program and by the size of the 'window' examined/KELLER 75/. Further, a complex compile time analysis is generally required, increasing the overhead in detecting parallelism. Data flow programs express potential parallelism that is easily detected and programs in a single assignment language, which are easily translated to data flow programs, are easy to write.
The architectures proposed are tailored to suit the data flow programs and to execute them efficiently. Furthermore, unlike the implementation of other parallel programming constructs, these processors initiate parallel operations without any overhead making it feasible to represent a 'fine-grain' parallelism down to the instruction level.

Determinacy is generally defined as the property of producing identical results for all executions of a program with the same input. It is highly desirable, often imperative and is not easy to guarantee for parallel programs. Data flow programs with certain topological constraints are guaranteed to be determinate. This considerably simplifies the debugging and the proof of correctness of programs.

In this thesis we analyse a model for data flow programs. We then define a transformation to another model which simulates the data flow program. We derive necessary and sufficient conditions to eliminate certain operations in this model with the preservation of determinacy. We then consider the implications of this analysis on extant architectures and suggest schemes for further improvement.

1.2 Data flow concepts.

The idea behind data flow programs and other data driven languages is that instructions should not wait for the completion of instructions on which they do not depend. This dependance is expressed either explicitly, as in most graphical data flow models, by a predecessor-successor relationship in the graph, or implicitly as in single assignment languages where each variable is assigned a value
only once. A data flow model is described formally in chapter 2 while an informal description follows.

Most of the data flow models proposed are directed graphs with the nodes being called 'operators' and the arcs, 'links'. The links carry 'tokens' with associated values. The operator nodes are said to 'fire' when they carry out an operation on the values associated with tokens on their input links and place result tokens, with associated values, on the output links. They simultaneously remove the tokens on their input links. Operators fire according to some rule (eg. when tokens are present on all input links or when a subset of input links have tokens). The firing of operators is totally asynchronous and no assumption about the speed of the operators is made.

The data flow 'schema' works by starting at an 'initial state' with tokens on all 'input links' of the schema. Operators that are firable according to the rules, fire and their firing causes other operators to become firable. This process continues until tokens are placed in 'output links' of the schema and there are no enabled operators.

The schema is regulated by control operators to carry out a useful computation. These control nodes allow the specification of loops and conditionals and route intermediate results for use in several sub-computations. They regulate the 'flow' of tokens by routing them to different links depending on the values associated with tokens on the
input links, or by blocking the flow of tokens depending on the input, or by passing tokens from sets of input links. Various sets of control nodes have been proposed in the various models that have appeared in the literature and a few are discussed in section 1.3. It should be noted that these control nodes do not carry out an operation on the input tokens but merely route them. The consequence of this is explored in chapters 3 and 4.

The other operator nodes carry out an 'operation' on the 'input tokens', placing 'result tokens' on the output links. These operators may include simple 'function' operators which carry out simple arithmetic operations on input tokens or they may be 'procedure' operators which take the input tokens as inputs to another data flow program. These operators allow a hierarchical description of a program and can also allow recursion. There may be further operator nodes that carry out operations on 'structures', with input tokens to these operators being pointers to a structure memory.

The large degree of parallelism that can be expressed by such languages should be apparent from this discussion. The various models differ chiefly in the types of control operators and the information associated with tokens. Some models ensure determinacy without any topological constraints whereas others impose certain structural constraints to ensure determinacy. The properties of various proposed models and associated problems are discussed in the next
section.

At this point an example may serve to clarify some of the concepts presented. Fig. 1.1 depicts the operators and their firing rules for the data flow model proposed by Rumbaugh /RUMBAUGH 75/. Tokens on links are indicated by shaded circles. The token configurations for firable operators is shown as is the result of firing these operators. Fig. 1.2 is a simple program that computes the roots of a quadratic equation. The initial configuration is with tokens on the input links indicated as 'a', 'b', and 'c'. Appropriate values are associated with these tokens. Expressions have been inserted at various points to indicate intermediate values computed. The operation of the schema is self explanatory given the firing rules in fig. 1.1. Notice that the 'wyes', 'switches', and 'unions' are control nodes and merely route the input tokens correctly without modifying them. Also note the various operations carried out in parallel in the program.

1.3 Related work.

Numerous models for describing parallel computation have been proposed. Of these models, data flow programs and other data driven languages fall into the class where a single graph is used to specify both the data and the control flow. We will only discuss models in this category. The combining of data and control flow structures simplifies the writing of programs and makes determinacy easier to guarantee.
Fig. 1.1: operators and their firing rules  
(Rumbaugh's constructs /Rumbaugh 75/ )
Fig. 1.2: an elementary data flow program.
Some of the earliest single graph models proposed were the computation graphs of Karp and Miller /KARP 66, 68/ and the program graphs of Rodriguez /RODRIGUEZ 69/. These models are always determinate. The computation graphs guarantee determinacy by introducing queues at memory cells (which correspond to the links of section 1.2). This makes them difficult to implement. The program graphs of Rodriguez ensure determinacy by introducing four statuses of tokens on links and by allowing operators to fire only when there are no tokens on the output links. Again the complexity of the model reduces its practicality. Further, though determinacy is guaranteed unconditionally, 'hang-up' states (i.e. an operator waiting for two or more tokens which can never appear simultaneously) are possible which can only be avoided by topological constraints on the graph.

Sonnenburg /SONNENBURG 74/ proposed a model called 'computation graphs'. This model has a control and a data signal associated with each token and introduces 'memory read' and 'memory write' operations into a memory unit. The non-determinacy which arises with the use of these operations is difficult to avoid. Further, the explicit memory read and write operations do not buy any additional expressive power or facilitate the writing of programs.

Dennis /DENNIS 75/ and Rumbaugh /RUMBAUGH 75/ proposed similar models called data flow programs. These models have been used as the machine language for proposed data flow processors. Dennis' model ensures determinacy
and safeness (at most one token associated with a link). Safeness is guaranteed by requiring that there be no tokens on output links of an operator when it fires. However, like Rodriguez's model, it allows hang-up states and constraints must be introduced for clean behaviour. Also, the firing of the 'merge gate', which is a type of control node, is data content dependant (i.e. it fires depending on the arrival of the right type of operand). This complicates the control of the firing of operators. Rumbaugh's model allows unsafeness and a 'conflict' (i.e. the arrival of tokens at two link link inputs of a 'union module') for which the operation of the module is undefined. As with other models, deadlock (i.e. a set of operators each waiting for another operator in the set to fire) and hang up states are possible. The 'well formed' nature (i.e. the absense of these undesirable properties) can be determined by transforming the model to a 'free choice Petri net' (similar to the CFS in chapter 3) and applying criteria developed by Hack /HACK 72/. Such a checking procedure is not feasible for the writing of programs since not only are Hacks conditions complicated but it would involve a trial and error approach in writing clean programs.

To simplify the writing of programs and ensure that the resulting programs are determinate, hang up free and deadlock free, Dennis /DENNIS 72/ and Rumbaugh /RUMBAUGH 75/ introduce structured schemas call 'well formed' schemas by Dennis and 'well nested' programs by Rumbaugh. These schemas
are recursively defined and are composed of an acyclic interconnection of operators, 'loop subschemas' and 'conditional subschemas' which in turn have well formed or well nested schemas as their subschemas. These constraints simplify the writing of programs considerably. However, Jotwani /JOTWANI 77/ has shown that for his model of parallel computation, call 'formal parallel programs' structuring leads to an inherent loss in parallelism. This is probably true for data flow programs as well. If this loss is significant it may necessitate the writing of unstructured programs for frequently used programs. But, for 'user programs' and even complicated systems programs, the well known advantages of structured programs including modularity, top down design, ease of debugging and for parallel programs those of determinacy and deadlock free nature probably outweigh the loss in parallelism.

The models of Dennis and Rumbaugh also allow the specification of procedures and recursion. Operations on structures are also provided. Rumbaugh also proposes a 'for all' construct that permits the 'spawning' of a variable number of sub-processes in parallel.

A major criticism of these data flow models has been that they are not specified like conventional programming languages. It is here that 'single assignment languages' enter the picture. These languages assign a value to a variable only once, thus informing all instructions that use this variable of its availability. These
languages were proposed by Tesler and Enea /TESLER 68/ with a detailed implementation proposed by Chamberlin /CHAMBERLIN 71/. Single assignment languages that are easily translated into data flow form have been proposed /ARVIND 76/, /WENG 75/. Interpreters for these languages have also been developed /ARVIND 75/. The languages include loops, conditionals, for all constructs, procedures, recursion, structure operations and are easy to write being similar to conventional languages. This simplifies the writing of 'high level' programs which are easily translated to data flow form for an efficient execution on the 'data flow processors'.

The architectures proposed for the 'data flow processors' vary in the amount of segregation of the data and the control flow. Sonnenburg /SONNENBURG 74/ suggests using processors as the nodes of the data flow programs and passing tokens between them. This extreme is, of course, extremely wasteful of processors in that only a few nodes can fire in parallel. It is extremely expensive and non-modular.

Arvind and Gostelow /ARVIND 77/ present a somewhat similar architecture. In this system a processor is assigned to a data flow program operator node when a token arrives and it then waits for the other operands. On arrival of all the operands it executes the operation and ships a result package to a destination processor via a communication system. The latter has not been specified
but a hierarchical bus structure is envisaged. Again this system is too complex and inefficient in processor usage.

The architecture proposed by Dennis /DENNIS 77/ has an instruction cell memory, a distribution network, a set of processors (functional and decision units), a control network and an arbitration network. The instruction cell block is a complex piece of hardware. Not only does it store instructions with the information about successor instructions but it also has hardware to detect when instructions have received their operands and it then forwards these instructions to the distribution network which routes them to available processors. The processors carry out the indicated operation on the data presented to them with the instructions and forward the results to successor instructions in the instruction cell memory via the arbitration and control networks. Realizing the expense of the complex instruction cell memory it is proposed to use a separate conventional instruction memory with an instruction cell 'cache' memory /DENNIS 74/. Patil is examining an implementation using a kind of programmable logic array /MISUNAS 77/. The architecture has been studied extensively with performance analyses, structure processing /MISUNAS 75/, procedure representation /MISUNAS 76/, complexity and performance of the interconnection network, and simulation studies /MISUNAS 77/ being done.

The 'data flow machine' proposed by Rumbaugh /RUMBAUGH 75/ is more conventional. It consists of a
'program memory' holding a representation of entire data flow programs, a 'scheduler', a 'swap memory', a set of 'activation processors' and a 'structure memory and controller'. The scheduler runs procedures of which the programs comprise, on activation processors (AP). The AP consists of an 'enabling count memory', a 'token memory', an 'instruction memory' and an 'execution pipeline' which removes enabled instructions from memory and routes them to the processors and then uses the results to update the enabling count and token memories. The enabling count represents the number of operands yet to arrive before an instruction is enabled and is decremented for each operand that arrives. The token memory corresponds to the values associated with tokens in the data flow program. The instruction memory specifies the operands in the token memory, the operation to be performed and the destination instructions. A nested procedure is invoked by carrying out the operation in parallel on another activation processor and routing the results to the calling procedure. A count is maintained of the number of active instructions in a procedure running on an AP and if this goes to zero when waiting for results from an invoked procedure, the calling procedure is suspended and transferred to a 'swap memory' via a 'swap network' so as to free the 'activation processor'.

An architecture similar to that of Rumbaugh has been proposed by a group in Tolouse, France /eg.SYRE 77/. The
processor consists of an 'instruction memory', a 'control instruction memory' and a 'data memory' which correspond to Rumbaugh's instruction memory, enabling count and token memories respectively. The difference in operation is that here enabled instructions are detected by status tags in the control instruction memory. Enabled instructions are fetched and assigned to an elementary processor which then requests the operands from the data memory subsystem, executes the instruction and returns results to the data memory. The data memory has pointers to instructions the data is used in and the status tags of these instructions are then updated. Rumbaugh's scheme, on the other hand, has a decoder that takes active instructions from an 'activity list', obtains the operands and creates a 'packet' that is routed to an appropriate processor. Result packets are received from the processors by an 'updater' which places results in the token memory and updates the 'enabling counts' of successor instructions. If any counts go to zero, the instructions are placed in the activity list. This scheme reduces memory contention but limits parallelism by a full 'execution pipeline'. Notice also that using pointers from data to successor instructions requires allocated space for all the tokens in a procedure precluding the reuse of token memory for links that do not contain tokens simultaneously.

Dennis' and Rumbaugh's architectures use 'packet communication' in which a stream of independant unordered
packets is routed to a set of processors. The efficient scheduling of the processors under these assumptions has been studied by Bain /BAIN 78/. The above schemes use lisp like structures which make array and vector processing rather inefficient. The 'stream processing' proposed by Ahuja /AHUJA 77/ can be incorporated into the architecture to improve vector processing.

Various other languages, interpreters and simulators have been developed several of which are summarized in the 'workshop on data flow computer and program organization' /MISUNAS 77/. We have discussed a few systems in some detail to provide a background for chapter 5 where the implications of the analyses in chapters 3 and 4 to some of these architectures is discussed.

1.4 The motivation for the thesis.

In this thesis we examine the elimination of data flow operations for the control nodes of a data flow program. The motivation for the work was provided by Bryant's observation /MISUNAS 77/ that in translating medium sized Fortran programs to data flow form, the data flow version had approximately twice as many control operations as arithmetic operation. Further, the control operations do not modify the values associated with input tokens but merely route them to or from the correct operators. This suggests that no data flow operations need be carried out for the control operators. However, it turns out that an indiscriminate removal of data flow operations gives rise
to non-determinacy.

We specify a data flow model and transform it to data flow and control flow structures. We examine necessary and sufficient conditions to eliminate data flow operations for the control nodes. The implications of this analysis to some of the architectures described in section 1.3 are then examined.

It is further observed that the control flow structure can be simplified considerably after the removal of the unnecessary data flow operations. It is shown that the degree of parallelism can be increased by removing superfluous transitions. Further, branching decisions in extant architectures are made by the processors. We propose a separate control that is constructed from the simplified control flow structure that makes branching decisions, thus speeding up the processor.

1.5 The plan of the thesis.

In chapter 2 we specify a data flow model and examine some of its properties.

Chapter 3 specifies a transformation of the data flow model to an 'extended Petri net' representation of the control flow structure and a data flow structure. We then show the equivalence of the two models with regard to the computation performed and the parallelism exhibited.

In chapter 4 we examine necessary and sufficient conditions that permit the elimination of operations in the data flow structure corresponding to control operators,
concentrating on the case of structured data flow programs. Finally we look at simplifications of the control flow structure that accrue to the elimination of data flow operations.

Chapter 5 discusses the implications of the analysis on extant architectures and presents the case for a separate control in the implementation.

Finally, chapter 6 summarizes the results and suggests areas for further work.
CHAPTER 2

THE DATA FLOW MODEL

In this chapter we formally specify the data flow model. We use the constructs proposed by Rumbaugh
/RUMBAUGH 75/ and hence designate it as a 'Rumbaugh data flow schema' or an 'RDFS'. We define the properties of 'determinacy', 'liveness', 'safeness' and 'well behaved nature' for these schemas. 'Well formed' schemas are then introduced and such schemas are shown to be determinate, live, safe and well-behaved.

2.1 The specification of the model.

Definitions.*

An \((m,n)\) Rumbaugh data flow schema (RDFS) 'S' is a bipartite directed graph with the two classes of nodes being links and actors. There are two types of link nodes—data link nodes and control link nodes. Arcs between actors and links are called data arcs or control arcs according to the type of link node. An \((m,n)\) RDFS 'S' has 'm' input nodes and 'n' output nodes which are link nodes of 'S'. No arc of S terminates on an input node and no arcs emanate from output nodes.

There are three classes of actors called 'operators', 'deciders', and control actors as indicated in fig. 2.1.

Operators are Boolean or data operators. Each has an ordered set of 'k' input arcs \((k>1)\) and an ordered set

* The definitions are similar to those of Dennis /DENNIS 72/.
Fig. 2.1: Links and actors.
of 'n' output arcs (n>1). Boolean operators have control arcs as inputs and outputs and have a letter 'b' from a set of Boolean operation letters 'B' associated with each such operator. Data operators have data arcs as input and output arcs and have a letter 'f', from a set 'F' of function letters associated with them. All operators having the same associated letters have the same number of input and output arcs.

Deciders are similar to operators but have data arcs as inputs and output control arcs. They have a letter 'p' from a set of predicate letters 'P' associated with them.

The control nodes are 'switches', 'unions', and 'wyes' as depicted in fig. 2.1.

The wyes have a single input arc (control or data arc) and a set of output arcs of the same type (i.e.a control or a data arc).

The union nodes have two ordered sets of inputs, designated as 'ports', and an ordered set of output arcs. Each port has the same number of arcs incident on it. Further, if there are 'n' arcs incident on each port, then the i^th arcs (1≤i≤n) must be of the same type at each port.

The switch nodes have an ordered set of inputs and two output ports. The 'k' ordered input arcs have integers one through 'k' associated with them. The output ports have at most 'k' arcs and each arc has a number from one
through 'k' associated with it with no two arcs of the same port assigned the same integer. Arcs incident on the node with the same associated integer must be of the same type.

**Interpretations:** An interpretation of an RDFS with function letters in F, predicate letters in P and Boolean operator letters in B is:

(i) A domain \( \mathfrak{X} \) of values, \( \mathcal{B} = \{\text{True}, \text{False}\} \)

(ii) An assignment of a total function

\[
\phi_f : \mathfrak{X}^k \rightarrow \mathfrak{X}^n
\]

to each \( f \) in F, where each operator with the function letter \( f \) has \( k \) input arcs and \( n \) output arcs.

(iii) An assignment of a total predicate

\[
\Pi_p : \mathfrak{X}^k \rightarrow \mathcal{B}^n
\]

to each \( p \) in P, where each decider bearing the predicate letter \( p \) has \( k \) input arcs and \( n \) output arcs.

(iv) An assignment of a total function

\[
\beta_b : \mathcal{B}^k \rightarrow \mathcal{B}^n
\]

to each \( b \) in B, where each Boolean operator bearing the letter \( b \) has \( k \) inputs and \( n \) outputs.

**Configurations:** A configuration of an RDFS 'S' for an interpretation with domain \( \mathfrak{X} \) is:

(i) An association of a value in \( \mathfrak{X} \) or the symbol \text{null} with each data link of 'S'.

(ii) An association of one of the symbols \{true,false,null\} with each control link of 'S'.

Let \( \Gamma \) be the set of configurations,

\( \mathcal{L} \) be the set of link nodes.
Let \( \text{VALUE}: (\tau, L) \rightarrow \mathbb{S} \cup \{\text{true}, \text{false}, \text{null}\} \)

be a function that specifies the value associated with each link node for each configuration.

A configuration of an RDFS is depicted by drawing a solid circle on each link node having a non-null value and writing a value denoting symbol alongside. These circles are called data tokens, true tokens or false tokens according to the associated value.

**Firing rules:** An RDFS progresses through a sequence of configurations through the firing of nodes. Fig. 2.2 indicates the firing rules for each node in an RDFS. Enabled nodes are indicated on the left and the action produced by firing them, on the right. For the union and switch nodes, the arcs on the three ports are numbered with values from 1 through 'r' for an 'r' input node. On firing, the tokens are passed to the link 'fed' by the arc with the corresponding number on the output port.

We introduce some notation to express this formally: Let \( S \) be an RDFS and \( O, D, B, L, U, W, Sw \) be the sets of operators, deciders, Boolean operators, link nodes, union nodes, wyes and switches respectively. Let \( A \) be the set of actors. Then for \( \alpha \) in \( A \), let \( (\alpha)^j \) denote the set of input links at the \( j^{th} \) port and \( (\alpha)^k \) denote the \( k^{th} \) link of port \( j \). Similarly denote the sets of output arcs as \( (\alpha^*)^m \) and \( (\alpha^*)^n \). For nodes with a single input or output port, the superscript is omitted. For switch nodes let \( (\alpha)^C \) denote the control input link.
Fig. 2.2: firing rules.
We illustrate the formal specification of the firing rules by those for a union node:

For \( u \in U \), \( u \) is 'enabled' for configuration \( \gamma \) if

\[
\forall 1 \in \{u\}^1 \quad (\text{i.e. port 1}), \quad \text{VALUE}(\gamma, 1) = \text{Null}
\]
or

\[
\forall 1 \in \{u\}^2 \quad (\text{i.e. port 2}), \quad \text{VALUE}(\gamma, 1) = \text{Null}, \quad \text{but not both}.
\]

'Firing' an enabled union node \( u \in U \) leads to configuration \( \gamma' \) with \( \forall 1 \in L - ( (\{u\}^1 \cup \{u\}^2) \cup u') \),

\[
\text{VALUE}(\gamma, 1) = \text{VALUE}(\gamma', 1).
\]

\[
\forall 1 \in (\{u\}^1 \cup \{u\}^2) - u', \quad \text{VALUE}(\gamma', 1) = \text{Null} \quad \text{and}
\]

\[
\text{VALUE}(\gamma', (u')^k) = \text{VALUE}(\gamma, (u)^k), \quad k=1, 2, ..., |u'|,
\]

where \( i = 1 \) or \( 2 \) to produce a non-null value.

The rules for the other nodes are expressed similarly. Note that tokens on both input ports of a union node do not allow the node to fire.

**Execution sequences:** Let \( 'S' \) be an RDFS. Then for some interpretation and initial configuration an execution sequence is a sequence

\[
\tau : \eta \rightarrow \Lambda, \quad \text{where } \eta = \{0, 1, ..., \}, \quad \Lambda \text{ is the set of actors. } \tau \text{ defines a sequence of configurations of } 'S'
\]

\[
\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \ldots \rightarrow \gamma_k
\]

where, if \( \tau(i) = x \) then the node 'x' of 'S' is enabled for configuration \( \gamma_i \) and \( \gamma_{i+1} \) is the result of firing node 'x'.

**Markings:** For a configuration \( \gamma \) of an RDFS 'S', the corresponding marking is a function that associates an element of \{mark, null\} with each data link of 'S' and an element of \{true, false, null\} with each control link.
of 'S'. A marking is identical to the corresponding configuration except that values in a domain Σ are replaced by a single element 'mark'.

Fig. 2.2 also specifies the firing rules for markings. Deciders and Boolean operators may fire in two ways - placing a 'true' token or a 'false' token on their output arcs. All other enabled nodes fire depending on the tokens at the input. The possible markings of 'S' depend on the initial marking and are independent of the interpretation.

Control sequences: Let 'S' be an RDFS and O, D, B, L, U, W, Sw be the sets of operators, deciders, Boolean operators, links, unions, wyes and switches. Then the alphabet of actions for 'S' is the set

\[ V = \{o : o \in O\} \cup \{d : d \in D\} \cup \{b : b \in B\} \cup \{u : u \in U\} \cup \{w : w \in W\} \cup \{s : s \in Sw\} \].

A control sequence of 'S' for an initial marking \( M_0 \) is a sequence

\[ \delta : \eta \to V \] where \( \eta = \{0, 1, \ldots\} \) that defines a sequence of markings of 'S'

\[ M_0 \xrightarrow{\delta(0)} M_1 \xrightarrow{\delta(1)} M_2 \rightarrow \cdots \xrightarrow{\delta(k-1)} M_k \]

such that if \( \delta(i) = x \) then the node of 'S' corresponding to \( x \) is enabled for marking \( M_i \) and \( M_{i+1} \) is the result of firing node 'x'. Note that for deciders, the values placed at output links are specified by the control sequence.

Final configurations and final markings are
configurations and markings respectively in which no actors are firable.

2.2 Determinate, live, safe, well behaved and conflict-free programs.

Definition: An RDFS is **determinate** if for any particular interpretation and initial configuration \( Q_0 \), the sequence of values associated with each link is the same for all execution sequences.

i.e. let \( \gamma_0, \gamma_{1i}, \gamma_{2i}, \ldots, \gamma_{ni} \) and \( \gamma_0, \gamma_{1j}, \gamma_{2j}, \ldots, \gamma_{mj} \) be two sequences of configurations generated by two arbitrary execution sequences \( \tau_i \) and \( \tau_j \). Then in a determinate RDFS the following condition must hold:

Let \( l \in L \), the set of link nodes, and let

\[
\begin{align*}
\nu^1_0 & , \nu^1_{1i}, \nu^1_{2i}, \ldots, \nu^1_{ni} \\
\text{and} \quad \nu^1_0 & , \nu^1_{1j}, \nu^1_{2j}, \ldots, \nu^1_{mj}
\end{align*}
\]

be the sequences of values associated with link \( l \) corresponding to the configuration sequences in (2-1).

In (2-2) delete all entries which are null or are identical to their previous entry in the sequence (i.e. if \( \nu^1_{ki} = \nu^1_{(k-1)i} \), or if \( \nu^1_{ki} = 'null' \), then delete \( \nu^1_{ki} \)). Then for any particular \( l \) in \( L \), the link value sequences must be identical.

Definition: An RDFS is **safe** if for an initial marking with one token on each input link and for all control sequences no node is firable when there are tokens on any of its output links.
**Definition:** A well behaved RDFS.

Let 'S' be an (m,n) RDFS with an initial marking $M_0$ in which each input link carries a token as in fig. 2.3. Here $M'_0$ denotes the marking of the schema excluding the input and output links, while $M_0$ is that of the entire schema. RDFS 'S' is well behaved for marking $M_0$ if and only if each finite control sequence starting from the marking in fig. 2.3 (a) reaches the marking in (b) with the same marking $M'_0$ for the part of the schema excluding the input and output links.

**Definition:** An RDFS 'S' that is well behaved for marking $M_0$ is live for marking $M_0$ if, for the initial marking in fig. 2.3(a) and any actor 'x', there exists a control sequence in which 'x' fires.

**Definition:** An RDFS 'S' is said to be conflict free if for an initial marking with tokens on all input links there is no reachable marking with tokens on both input ports of a union node in 'S'. i.e. there is no control sequence $\delta = \delta_1 \cdot \delta_2$ ('\cdot\ ' denotes concatenation of strings) such that $\forall \delta_1, \delta_2. M_0 \xrightarrow{\delta_1} M$ and there are tokens on both ports of some union node in 'S' for M.
2.3 Well formed schemas.

Definition: A well formed RDFS denoted by 'WFRDFS' is an acyclic interconnection of operators, conditional subschemas and loop subschemas. (The latter two are defined recursively below.)

![Diagram of a well-formed schema]

Fig. 2.4 A conditional subschema.

Conditional subschemas: Given two WFRDFS's 'S1' and 'S2' with the same number of output nodes a conditional subschema may be constructed as in fig. 2.4 provided that at the switch and at the union, corresponding link nodes are of the same type (i.e. both data links or both control links).

![Diagram of a loop subschema]

Fig. 2.5 A loop subschema
Loop subschemas: Given two WFRDFS's 'S_1' and 'S_2' satisfying constraints on the number of input and output links as in fig. 2.5, a loop subschema can be constructed as in fig. 2.5. Again corresponding links incident on the switch and on the union should be of the same type. 'S_1' is called the 'forward subschema' and 'S_2', the 'feedback subschema' of the loop subschema.

A WFRDFS imposes a partial order on the loop and conditional subschemas in it. The ordering relation is that of 'containment' denoted by '<'. That is, for two loop or conditional subschemas 'S_1' and 'S_2', \( S_1 \leq S_2 \) iff \( S_1 \) is a subgraph of \( S_2 \).

**Definition:** Let \( \mathcal{S} \) denote the set consisting of 'S' and the loop and conditional subschemas of a WFRDFS 'S'. Then '<' is an ordering relation on \( \mathcal{S} \) such that

\[
\forall S_1, S_2 \in \mathcal{S}, S_1 \leq S_2 \Rightarrow S_1 \text{ is a subgraph of } S_2.
\]

**Lemma 2.1:** For a WFRDFS 'S' with loop and conditional subschemas and 'S' in \( \mathcal{S} \), the relation '<' defines a partial ordering on \( \mathcal{S} \). Further, for all \( S_1, S_2 \) in \( \mathcal{S} \), \( S_1 \) and \( S_2 \) are either related by '<' or are disjoint. Also, if the WFRDFS 'S' is included in \( \mathcal{S} \) then the associated Hasse diagram for the partial ordering of \( \mathcal{S} \) is a rooted tree with 'S' as the root.

**Proof:** The partial ordering as subgraphs follows for any set of subgraphs of a graph. The statement that all schemas in are either related or disjoint follows by an easy induction on the depth of the Hasse diagram of the partial
order and the recursive definition of a WFRDFS. Finally, 
\[ \forall S_1, S_2, S_3 \in \mathcal{S}, S_1 \leq S_2, S_1 \leq S_3 \Rightarrow S_2 \leq S_3 \text{ or } S_3 \leq S_2 \]
because \( S_3 \) and \( S_2 \) are not disjoint since both contain \( S_1 \).
Also \( \forall S_1 \in \mathcal{S}, S_1 \leq S \). Hence the Hasse diagram is a rooted tree. **

Definition: For a WFRDFS 'S' with loop and conditional subschemas and 'S' in \( \mathcal{S} \), a subschema \( S_1 \) in \( \mathcal{S} \) is said to be an innermost subschema if it corresponds to a leaf of the associated Hasse diagram. The depth of 'S' is the depth of the Hasse diagram (i.e. the maximum path length from the root to leaves of the tree).

2.4 Properties of WFRDFS's.

In this section we show that WFRDFS's are determinate, live, safe, well behaved and conflict free for an initial configuration of a token on each input link of the schema. We then develop some properties of WFRDFS's that will be used in succeeding chapters.

Lemma 2.2: A WFRDFS is well behaved for an initial marking \( M_0 \) with a token on each input link, and no tokens on non-input links of the schema. *

Proof: We prove this by an induction on the depth of a WFRDFS.

Basis step: A WFRDFS of depth '0' is an acyclic interconnection of operators. We prove that this is well

* We will assume henceforth that in the context of WFRDFS's, the initial marking is with a token on each input link of the schema and no tokens on non-input links.
behaved by contradiction. First note that in an acyclic interconnection of operators, each operator fires exactly once because if not, then tokens must appear at its input links more than once. Thus its predecessor operator fires more than once. An easy induction on the length of a path from an input link to this operator then shows that input nodes must fire more than once. Similarly, operators fire at least once else input operators never fire. Now suppose that an acyclic interconnection of operators is not well behaved. Then there is a finite execution sequence from marking $M_0$ leading to a final marking with tokens at some non-output link. But the successor operator of this link node fires once. Thus the predecessor operator of such a link node must fire more than once, a contradiction.

A WFRDFS of depth '1' is an acyclic interconnection of operators and loop and conditional subschemas, the latter being derived from subschemas of depth '0'. The topology of the conditional subschemas (figs. 2.4 and 2.5) and the firing rules (fig. 2.2) show that these subschemas must also be well behaved. Also, an acyclic interconnection of well-behaved schemas is also well behaved because they behave exactly like operators - they accept input tokens at input links and put out tokens at output links after an arbitrary delay. Thus WFRDFS's of depth '1' are well behaved.

**Induction step:** Suppose that WFRDFS's of depth 'n' or less are well behaved. Consider a WFRDFS of depth (n+1). The
subschemas of schemas at depth '1' are well behaved, by hypothesis and by an identical argument for WFRDFS's of depth '1', these WFRDFS's of depth (n+1) are also well behaved.

***

**Corollary:** In a loop sub-schema, when the union node fires, there are no tokens in the WFRDFS's $S_1$ and $S_2$ (fig. 2.5).

**Proof:** Follows directly from the well behaved nature of WFRDFS's $S_1$ and $S_2$, the topology of the loop subschema (fig. 2.5) and the firing rules (fig. 2.2).

***

**Lemma 2.3:** A WFRDFS is safe.*

**Proof:** The proof is analogous to lemma 2.2 - by an induction on the depth of WFRDFS's.

The basis step follows from the proof of lemma 2.2 since each operator in an acyclic interconnection of operators fires exactly once. The proof for WFRDFS's of depth 1 and the induction step follow by analogous arguments to lemma 2.2.

***

**Lemma 2.4:** A WFRDFS is determinate.

**Proof:** The proof is again by an induction on the depth of WFRDFS's.

**Basis step:** A WFRDFS of depth '0' is an acyclic interconnection of operators. Suppose such a schema is not determinate. Then there is an interpretation for which there exists a link node and execution sequences which produce differing value sequences for this link node. Since

---

* See footnote on preceding page.
the schema is safe and the sequence of values placed by
the immediate predecessor operators is not unique, the
sequence of values in at least one input link of this
operator must differ for the two execution sequences. An
easy induction on the length of a path from an input link
to this link then shows that input values must differ for
the execution sequences. Thus an acyclic interconnection
of operators is determinate.

The case for a WFRDFS of depth '1' and
the induction step is then analogous to lemma 2.2.

**Lemma 2.5:** A WFRDFS is live.

**Proof:** The proof is analogous to that of the preceding
lemmas. The basis follows since as proved earlier each
operator fires precisely once in an acyclic interconnection
of WFRDFS's. The induction step follows by arguments
similar to those in lemma 2.2. 

**Lemma 2.6:** A WFRDFS is conflict free.

**Proof:** This again follows by an easy induction of the
depth of a WFRDFS.

**Theorem 2.1:** A WFRDFS is determinate, live, safe, well
behaved and conflict free for an initial configuration of
a token on each input link of the WFRDFS.

**Proof:** Follows directly from lemmas 2.2, 2.3, 2.4, 2.5,
and 2.6.

**Lemma 2.7:** Consider a WFRDFS 'S' composed of an acyclic
interconnection of WFRDFS's $S_1$, $S_2$, $S_n$. (Operators in
this acyclic interconnection are also included in $S_1, S_2, S_n$.)
Suppose that tokens are present at either some input link or within WFRDFS \( S_i \) \((1 \leq i \leq n)\). Then no token can exist either on a link or within a WFRDFS that is on a path from the output links of \( S_i \) to an output link of \( S \).

**Proof:** Each of the component WFRDFS's are well behaved (theorem 2.1) and hence behave exactly like operators. Thus the arguments of the proof of lemma 2.2 are valid and tokens are placed at the output links of each WFRDFS \( S_i, \ldots, S_n \) exactly once. So if tokens are present within or at the input links of \( S_i \) \((1 \leq i \leq n)\) and either on a link '1' or within a WFRDFS \( S_j \) \((1 \leq j \leq n)\) on a path from an output link of \( S_i \) to an output link of \( S \), then an easy induction on the length of the path from \( S_i \) to 1 or \( S_j \) (with WFRDFS's \( S_k \) \((1 \leq k \leq n)\) taken as nodes) shows that \( S_i \) must have placed tokens on its output links earlier. Thus \( S_i \) would place tokens on its output links more than once - a contradiction which establishes the lemma.  

***

**Lemma 2.8:** In a WFRDFS there is a path from each node to an output node and from an input node to each node.

**Proof:** Follows by an easy induction on the depth of a WFRDFS.

***

**Lemma 2.9:** Consider two link nodes \( l_1, l_2 \in L \) in a WFRDFS 'S'. If there is a path from \( l_1 \) to \( l_2 \) that does not pass through a union node, then \( l_1 \) and \( l_2 \) cannot have tokens simultaneously.

**Proof:** We prove this by an induction on the depth of a WFRDFS.
Basis step: For WFRDFS's of depth zero, we have a special case of lemma 2.7 since there is a path from $1_2$ to an output node by lemma 2.8.

Induction step: Suppose that the lemma is true for all WFRDFS's of depth 'n' or less. Then consider a WFRDFS 'S' of depth (n+1). The cases that occur are depicted in fig. 2.6. We refer to the cases as (a),(b) and (c). For case (a), $S_1$ may either be a loop or a conditional subschema. If $S_1$ is a loop subschema then $1_1$ and $1_2$ may both be in the 'forward subschema' or $1_1$ may be in the 'forward subschema' and $1_2$ in the feedback subschema of $S_1$. For the case of both $1_1$ and $1_2$ in the forward subschema, the path from $1_1$ to $1_2$ must be entirely in the forward subschema since union nodes are not permitted to be on the path. This forward subschema is at most of depth n. Thus from the inductive hypothesis, the statement holds. For the case of $1_1$ in the forward subschema and $1_2$ in the feedback subschema of $S_1$, $1_1$ and $1_2$ cannot have tokens simultaneously since tokens cannot exist simultaneously in the forward and feedback subschemas of a loop subschema. If $S_1$ is a conditional subschema, $1_1$ and $1_2$ must be in the same well formed subschema of $S_1$ (from the construction of a conditional subschema). This subschema is at most of depth n and the statement follows.

Cases (b) and (c) are special cases of lemma 2.7. Thus the lemma is true for WFRDFS's of depth (n+1) and the lemma is proved.
Fig. 2.6: the cases for lemma 2.9.
CHAPTER 3

SPLITTING DATA AND CONTROL FLOW

In this chapter we describe the transformation of an RDFS to a model with a separate data and control flow called the data flow and control flow structures and denoted by 'DFS' and 'CFS' respectively. The CFS is an 'extended Petri net' /PATIL 75/ and the DFS consists of a directed bipartite graph of 'locations' and 'operators'. This model is shown to simulate the source RDFS.

3.1 Preliminaries - Petri nets *

The control flow structure (CFS) of the new model is expressed as an extended Petri net. In this section we define Petri nets and summarise some of their relevant properties.

Definitions: A Petri net is a triple G=(T,P,E) where

T is a non-empty set of transitions,

P is a non-empty set of places,

and \( E \subseteq (T \times P) \cup (P \times T) \) is a set of edges.

Thus a Petri net is a bipartite directed graph.

A marking is a function \( M: P \rightarrow N \), where \( N \) is the set of non-negative integers.

Notation: \( x' \) denotes the set \( \{ y \mid (x,y) \in E \} \)

and \( 'x \), the set \( \{ y \mid (y,x) \in E \} \)

Firing rules: A Petri net progresses through a sequence of markings by the firing of transitions. A transition

* The definitions here are similar to those of Jotwani /JOTWANI 77/.
't' is **firable** for marking M if \( \forall p \in t, M(p) > 0 \). Firing 't' leads to a marking M' where

\[
M'(p) = \begin{cases} 
M(p) + 1, & \text{if } p \in t^* \setminus t, \\
M(p) - 1, & \text{if } p \notin t^* \setminus t, \\
M(p), & \text{otherwise}.
\end{cases}
\]

This is denoted by \( M \stackrel{t}{\longrightarrow} M' \).

**Firing sequences:** For an initial marking \( M_0 \), a firing sequence is a sequence \( \sigma : \eta \rightarrow T, \quad \eta = 0, 1, 2, \ldots \) which defines a sequence of markings

\[
M_0 \xrightarrow{\sigma(0)} M_1 \xrightarrow{\sigma(1)} M_2 \xrightarrow{\sigma(2)} \cdots
\]

where if \( \sigma(i) = t \), then transition t is enabled for marking \( M_i \) and \( M_i \xrightarrow{t} M_{i+1} \).

We extend the notation to say that \( M_0 \xrightarrow{\sigma(0) \ldots \sigma(n)} M_{n+1} \) or denoting \( \sigma(0) \ldots \sigma(n) \) by \( \sigma_n \), \( M_0 \xrightarrow{\sigma_n} M_{n+1} \).

**Definition:** A transition t in G is **live** iff, for any firing sequence \( \sigma \) of G, another sequence \( \sigma_t \) exists such that t is firable under the marking M' where \( M_0 \xrightarrow{\sigma \cdot \sigma_t} M' \).

The net G is **live** iff all its transitions are live.

A place p in G is **safe** iff, for any firing sequence \( \sigma \) of G, \( M'(p) \leq 1 \) where \( M_0 \xrightarrow{\sigma} M' \). The net G is **safe** iff all its places are safe.

This completes the basic definitions of Petri nets. We now discuss some special classes of Petri nets since they are used later in the development.

**Definition:** A **subnet** (or subgraph) of a net \( G = (T, P, E) \) is a net \( G' = (T', P', E') \) s.t. \( T' \subseteq T, P' \subseteq P \), and \( E' \)
is the restriction of $E$ to $(T' \times P' U P' \times T')$. A subnet $G'$ is said to be $t$-complete iff for every transition $t$ in the subnet, the set $\{ t U t \}$ is a subset of $P'$. A subnet $G'$ is $p$-complete iff $\forall p' \in P', \{ p' U p' \} \subseteq T'$.

**Definition:** A Petri-net $G$ is said to be a state-machine if $\forall t \in T$, $|t| = |t'| = 1$ (i.e. every transition has a single input place and a single output place).

**Definition:** A Marked graph is a Petri net $G = (T, P, E)$ in which $\forall p \in P$, $|p'| = |p| = 1$ (i.e. every place has a single input transition and a single output transition.

**Definition:** A Petri net $G = (T, P, E)$ is said to be free-choice if $\forall p \in P$, $|p'| > 1 \Rightarrow \{ (p') \} = \{ p \}$ (i.e. every place that is an input place to more than one transition is the unique input place to its successor transitions).

The problem of determining if a Petri net is live and safe has been solved by Hack /HACK 75/. We summarize Hack's results as modified by Jotwani /JOTWANI 77/.

**Definition:** A free choice Petri net (FCP net) is said to be well formed iff there exists a marking $M$ on it under which the net is live and safe.

**Notation:** Denote by $FC(G)$, the set of places in an FCP net $G = (T, P, E)$ with more than one successor transition. i.e. $FC(G) = \{ p | p \in P$ and $|p'| > 1 \}$. These are called free-choice places.

**Definition:** An MG-allocation $AMG$ on an FCP net $G = (T, P, E)$ is a function $AMG: FC(G) \rightarrow T$ satisfying $\forall p \in FC(G)$, $AMG(p) \in p'$. 
Definition: Consider an FCP net $G = (T, P, E)$ with an MG allocation, 'AMG'. The set of strongly connected marked graph components (SCMG components) of $G$ produced by AMG, is the set of mutually disjoint subnets of $G$ found by the following reduction procedure.

(i) Delete all the edges of $G$ in the set $\text{FC}(G) \times (\text{FC}(G)^{-} - \text{AMG}(\text{FC}(G)))$ i.e. all the 'unallocated output edges' of FC places.

(ii) Find the maximal strongly connected components of the resulting graph.

(iii) Of the components found in (ii), delete those that are not $t$-complete. The components that remain are the SCMG components of $G$ produced by allocation AMG.

Theorem 3.1: An FCP net $G = (T, P, E)$ is well-formed iff, for any MG allocation AMG on $G$, the following conditions hold:

(i) The set of SCMG components of $G$ produced by AMG is not null.

(ii) Each SCMG component is a marked graph.

The theorem was proved in its original form by Hack /HACK 75/ and stated in the form above by Jotwani /JOTWANI 77/.

We now summarize a dual reduction procedure in terms of state machine subnets of an FCP net.

Definition: An SM-allocation ASM on an FCP net $G = (T, P, E)$ is a function $\text{ASM}: T \to P$ satisfying $t \in T$, $\text{ASM}(t) \in \ 't$.

Definition: Consider an FCP net $G = (T, P, E)$ with an SM
allocation ASM. The set of strongly connected state machine components (SCSM components) of G produced by ASM is the set of mutually disjoint subnets found by the following reduction procedure:

(i) Delete all the edges of G in the set (P-ASM(T)) XT, i.e. the 'unallocated input edges' of transitions.

(ii) Find the maximal strongly connected components of the resulting graph.

(iii) Of the components found in (ii) delete those that are not p-complete. The components that remain are the SCSM components of G produced by allocation ASM.

Definition: A 1-SCSM component is an SCSM component of a well-formed FCP net G which has exactly one token on it.

Theorem 3.2: An FCP net G=(T,P,E) is live and safe under marking M iff, for any SM-allocation ASM on G the following conditions hold:

(i) The set of SCSM components of G produced by ASM is not null.

(ii) Each SCSM component is a state machine with at least one token on it under marking M.

(iii) Each place p in G is on a one SCSM component of G.

Again this theorem is Jotwani's /JOTWANI 77/ restatement of Hack's criteria /HACK 75/.

This completes our discussion of Petri nets and their properties relevent to this thesis. We now discuss the 'extended Petri nets' of Patil /PATIL 75/. These nets enhance the modelling power of Petri nets by allowing the
selective firing of two simultaneously firable transitions in a Petri net. Some of the transitions have Boolean variables associated with them and a transition is firable only if its associated Boolean variable is true in addition to the normal firing rules.

Definitions: An extended Petri net $G$ is a triple $G=(T,P,E)$ where $T,P$ and $E$ are defined as for Petri nets.

A marking (as before) is a function $M: P \rightarrow N$, $N = \{0,1,\ldots\}$.

A configuration includes a marking $M$ and a function $\text{CONTROL}: T \rightarrow \{\text{TRUE, FALSE}\}$.

Firing rules: The extended Petri net progresses through a sequence of configurations by the firing of 'enabled' transitions. However, configurations may also change by an alteration of the function $\text{CONTROL}$ (by conditions external to the Petri net). A transition $t$ is firable for configuration $C$ with $M$ and $\text{CONTROL}$ as the associated marking and firing function if $\forall p \in t$, $M(P) > 0$ and $\text{CONTROL}(t) = \text{TRUE}$. Firing $t$ then leads to the marking $M'$ where

$$M'(p) = \begin{cases} M(p) + 1, & \text{if } p \in t \setminus t' \\ M(p) - 1, & \text{if } p \in t' \setminus t \\ M(p), & \text{otherwise.} \end{cases}$$

The new configuration $C'$ is specified by marking $M'$ and the same function $\text{CONTROL}$.

The definitions of liveness and safeness are absolutely analogous to those for Petri nets.
Notation: —— = Transitions;  O = places

(a) Transforming operator nodes.

(b) Transforming wye nodes.

(c) Transforming union nodes.

(d) Transforming switch nodes.

For a ∈ Ω U B U D:
include t in T with
\[ \tau = \mathcal{I}(\alpha_a), \tau' = \mathcal{I}(\alpha'_{\tau}) \]

For a ∈ W:
include t in T with
\[ \tau = \mathcal{I}(\alpha_a), \tau' = \mathcal{I}(\alpha'_{\tau}) \]

For a ∈ U:
include \( t_1 \) and \( t_2 \) in T with
\[ \tau_1 = \mathcal{I}(\alpha_{a_1}), \tau_1' = \mathcal{I}(\alpha'_{\tau_1}) \]
\[ \tau_2 = \mathcal{I}(\alpha_{a_2}), \tau_2' = \mathcal{I}(\alpha'_{\tau_2}) \]

For a ∈ Sw:
include \( t_1 \) and \( t_2 \) in T with
\[ \tau_1 = \mathcal{I}(\alpha_a), \tau_1' = \mathcal{I}(\alpha'_{\tau_1}) \]
\[ \tau_2 = \mathcal{I}(\alpha_a), \tau_2' = \mathcal{I}(\alpha'_{\tau_2}) \]

Fig. 3.1: obtaining the CFS from an RDF.
Note that thus far we have not mentioned how the function control may change. (Without this, there is no change in the modelling power.) We stipulate that the function CONTROL is changed by a system external to the net.

3.2 The specification of the transformation.

In this section we transform an RDFS to a model with separate data and control flow. The control flow structure (CFS) is modelled by an extended Petri net. The data flow structure (DFS) consists of a bipartite graph with the nodes called locations and operators. The CFS and the DFS interact to perform a computation. The firing of transitions in the CFS cause operators in the DFS to 'fire' by using operand values in their 'input' locations and placing result values in 'output' locations. Further, some of the locations in the DFS control the firing of some of the transitions in the CFS. This allows conditional branches depending on values in locations in the DFS.

Definitions: the control flow structure (CFS) is an extended Petri net \( \mathcal{C} = (T, P, E) \) derived from an RDFS \( S \) as follows.

Let \( O, D, B, L, U, W, Sw \) be the sets of operators, deciders, Boolean operators, link nodes, union nodes, wyes, and switches in \( S \). Let \( A \) be the set of actors.

Define \( \chi \) to be a bijection from link nodes in \( S \) to places in \( \mathcal{C} \), i.e. \( \chi : L \leftrightarrow P \). Thus there is a one to
one correspondence between link nodes in 'S' and places in C.

For each actor in 'S', consider the subgraph consisting of the actor and its direct predecessor and successor nodes. i.e. $\forall a \in A$, consider 'S' restricted to 'a U a U a'. For $P' = \chi('a U a')$, interconnect $P'$ as in fig. 3.1. For operator nodes this is expressed as: if $a \in O$, the set of operator nodes in 'S', then include a transition 't' in 'T' with $t = \chi('a)$ and $t' = \chi(a')$. Fig. 3.1 specifies these transformation rules both graphically and formally.

The transformation is illustrated by the example in fig. 3.3. Fig. 3.3 (a) is an Algol like version of a program with a data flow representation of it in fig. 3.3 (b). The CFS is shown in fig. 3.3 (c), with the function $\chi$ specified in fig. 3.3 (f). Note the one-to-one correspondence between links in the WFRDFS and places in the CFS.

The marking functions and the function CONTROL of the extended Petri net C are detailed later.

The data flow structure (DFS) $\Xi$ is a bipartite directed graph, $\Xi = (LOC, OP, E_p)$ with

- LOC being a set of locations,
- OP, a set of operators,

and $E \subseteq (LOC \times OP) \cup (OP \times LOC)$.

It is obtained from an RDFS as follows.

Define $\psi$ to be a bijection from link nodes in 'S'
Notation: \( \Rightarrow \) = locations; \( \bigcirc \) = operators

For \( a \in O \cup U \cup U \cup D \)
include \( op \) in OP with
\[
\text{op} = \gamma(\{a\}), \quad (\text{op})^* = \gamma(a^*)
\]

(a) Transforming RDFS operator nodes.

For \( a \in W \)
include \( |a^*| \) I operators
with \( |a^*| = n \), denote the
I operators as \( I_1^*, \ldots, I_n^* \)
Then \( |(I_i^*)^*| = 1 \) and
\[
(I_i^*)^* = \gamma(a^*)_i
\]

(b) Transforming wye nodes.

For \( a \in U \)
include \( 2^* |a^*| \)
I operators.
If \( |a^*| = n \),
denote the I's as
\( I_1^1, I_2^1, \ldots, I_n^1 \) and
\( I_1^2, I_2^2, \ldots, I_n^2 \).
and let
\[
(I_i^j)^* = \gamma(\{a\}_i^j)
\]
\[
(I_i^j)^* = \gamma(\{a^*\}_i^j)
\]
\[
\forall i=1,2,\ldots,n; \forall j=1,2.
\]

(c) Transforming Union nodes

For \( a \in Sw \)
include \( |(a^*)^T| + |(a^*)^F| \)
I operators.
Denote them as
\( I_1^T, \ldots, I_n^T \) & \( I_1^F, \ldots, I_m^F \)
and let
\[
(I_1^T)^* = \gamma(\{a\}_1^T)
\]
\[
(I_i^F)^* = \gamma(\{a\}_i^F)
\]
\[
\forall i=1,2,\ldots,n \text{ & } \forall j=1,2,\ldots,m.
\]

(d) Transforming switch nodes.

Fig. 3.2: obtaining the DFS from an RDFS.
to locations in $S$. i.e. $\psi : L \leftrightarrow LOC$. There is thus a one-to-one correspondence between link nodes in $'S'$ and locations in $S$.

For each actor in $'S'$, consider the subgraph consisting of the actor and its direct predecessor and successor nodes. i.e. $\forall a \in A$, consider $'S'$ restricted to $\{ a U a U a \}$. For $LOC' = \psi ( \{ a U a \} )$, interconnect $LOC'$ as in fig. 3.2. i.e. operators are included in $S$ and are interconnected to $P'$. For operator nodes, the associated symbol (in $B U F U P$) is also associated with the new operator included in $S$. The transformation is illustrated both graphically and algebraically in fig. 3.2.

For the example in fig. 3.3, fig (d) shows the DFS obtained by transforming the WFRDFS of fig. 3.3 (b). The bijection $\psi$ is given in fig. 3.3 (g).

**Notation:** for $op \in OP$, let $\{ op \}$ denote the set of predecessor locations of $op$ and $\{ op^* \}$ denote the successor locations of $op$.

**Interpretations:** an interpretation of a CFS-DFS is in terms of the interpretation of the source RDFS. i.e. it has the same domain $S$ of values, $B = \{ True, False \}$, and functions $\phi_f$, $\pi_p$, and $\beta_b$ (as defined in chapter 2). Also included is a reserved function 'identity' (copy) 'I' which does not appear in $\phi_f U \pi_p U \beta_b$.

**Configurations:** a configuration of a DFS-CFS for an interpretation with domain $S$ is:

(i) A marking function, $M : P \rightarrow N$, $N = \{0,1,... \}$
(ii) A function \( \text{VALUE}' : \text{LOC} \rightarrow \mathbb{D} \cup \{\text{True}, \text{False}, \text{Null}\} \), such that \( \text{VALUE}'(\text{LOC}) = \text{NULL} \) iff \( M(\chi(\chi^{-1}(\text{LOC}))) = 0 \).

i.e. a null value is associated with a location iff the corresponding place is blank. Furthermore, if \( \chi^{-1}(\text{LOC}) \) is a control link then \( \text{VALUE}'(\text{LOC}) \in \{\text{True}, \text{False}, \text{Null}\} \).

(iii) A function \( \text{CONTROL} : \text{T} \rightarrow \{\text{True}, \text{False}\} \) such that

\[
\forall t \in \text{T} \text{ s.t. } (\chi^{-1}(\text{t}))(\not\in \text{Sw}, \text{CONTROL}(t) = \text{True} \text{ and } \\
\forall t \in \text{T} \text{ s.t. } (\chi^{-1}(\text{t}))(\in \text{Sw}, \text{if } \text{t}^* = (((\chi^{-1}(\text{t}))(\not)))^T)
\]

(i.e. if \( t \) corresponds to the 'T' output of a switch) then \( \text{CONTROL}(t) = \text{VALUE}'(\psi((((\chi^{-1}(\text{t}))(\not)))(\not))) \) (i.e. it is the value associated with the location corresponding to the control input of the switch.

Similarly, if \( t^* = (((\chi^{-1}(\text{t}))(\not)))(\not) \) (i.e. if \( t \) corresponds to the 'F' output of a switch) then

\[
\text{CONTROL}(t) = \overline{\text{VALUE}'}(\psi((((\chi^{-1}(\text{t}))(\not)))(\not)))
\]

where \( \overline{\text{VALUE}'} \) is the Boolean complement. (i.e. it is the complement of the value associated with the location corresponding to the control input of the switch.

The function \( \text{CONTROL} \) for the example in fig.3.3 is given in fig. 3.3 (j). Note that all transitions except those corresponding to switches have their 'CONTROL' as True.

Thus these nodes fire as in a normal Petri net. The transitions \( t_{10} \) and \( t_{11} \) correspond to switch node \( A_9 \).

The firing of these transitions is controlled by the value in location \( \text{loc}_{12} \) (which corresponds to the control link input of the switch node \( A_9 \)). Note that \( \text{CONTROL}(t_{10}) = \text{VALUE}'(\text{loc}_{12}) \) and \( \text{CONTROL}(t_{11}) = \overline{\text{VALUE}'}(\text{loc}_{12}) \). Thus
though \( t_{10} \) and \( t_{11} \) have the same input places, they are never firable simultaneously and which of the two fires is controlled by the value associated with \( \text{loc}_{12} \).

**Transforming configurations from the RDFS to the CFS-DFS.**

Given a configuration in the RDFS, the corresponding configuration in the CFS-DFS is obtained by marking places corresponding to links with non-null associated values and associating these values with corresponding locations.

Formally: let \( \gamma \in \Gamma \) be a configuration in the RDFS.

Then \( \forall \gamma \text { s.t. } \text{VALUE}(\gamma,1) \in \mathbb{D} \cup \{\text{True, False}\} \),

let \( M(\gamma(1)) = 1, \text{VALUE}'(\psi,1) = \text{VALUE}(\gamma,1) \).

\( \forall \gamma \text { s.t. } \text{VALUE}(\gamma,1) = \text{Null} \) let \( M(\gamma(1)) = 0, \text{VALUE}'(\psi(1)) = \text{Null} \).

An 'initial configuration' for the example in fig. 3.3 is indicated in fig. 3.3 (b). There is a token on the input link with an associated value \( 'v_{\text{in}}' \). The corresponding configuration in the CFS-DFS has a token in place \( p_1 = \gamma(1_1) \) \((1_1 \text { is the input place}) \) and a value \( v_{\text{in}} \) associated with location \( \text{loc}_1 = \psi(1_1) \). All other places are blank and all other locations have a 'null' value associated with them.

**Notation:** let \( \Gamma \) and \( \Gamma' \) be the sets of configurations in the RDFS and corresponding configurations in the CFS-DFS respectively. Let \( \xi : \Gamma \rightarrow \Gamma' \) be the above configuration transformation.

We wish to provide a simpler notation to indicate the correspondence between transitions in the CFS and
operators in the DFS. A glance at Figs. 3.1 and 3.2 should make the correspondence apparent.

**Definition:** let FIRE be a function from operators in the DFS to transitions in the CFS. i.e. FIRE: OP \rightarrow T. s.t \( \forall \text{op} \in \text{OP}, \text{if} \ a \in A = (\gamma^{-1}(\text{op})) \) then

if \( a \in \{0, U, B, D, W, U\} \) (i.e. the operator in the DFS corresponds to an operator, binary operator, decider, wye or a union node in the RDFS) then \( \text{FIRE}(\text{op}) = (X(\gamma^{-1}(\text{op}))) \) (i.e. the corresponding transition). If \( a \in \text{Sw} \) (i.e. op corresponds to a switch operator) then \( \text{FIRE}(\text{op}) = (Y(\gamma^{-1}(\text{op}'))) \) (i.e. the corresponding transition).

The function 'FIRE' for the example in fig.3.3 is specified in fig. 3.3 (h). Note that 'FIRE' merely specifies the transition that fires an operator as the following firing rules show.

**Firing rules:** The CFS-DFS progresses through a sequence of configurations through the 'firing' of transitions in which cause the corresponding operators in the DFS to fire simultaneously.

The firing rules of the CFS are exactly that of extended Petri nets as specified on page 3.5. When a transition \( t \in T \) fires, the operators in \( \text{FIRE}^{-1}(t) \) fire simultaneously. i.e. if a transition 't' is firable for configuration \( \gamma' \) with marking \( M' \) in the CFS and values associated with locations as in \( \text{VALUE}' \) then firing \( t \) leads to configuration \( \gamma'' \) with marking \( M'' \) derived as in section 3.1 and \( \text{VALUE}'' \) obtained as follows:
loc \in \text{LOC}, \text{VALUE}''(\text{loc}) = \begin{cases} f_i(\text{VALUE}'(\text{op})) & \text{if } \text{loc} = (\text{op})_{i}^* \\
\text{Null} & \text{if } \text{loc} = (\text{FIRE}(\text{t})) - (\text{FIRE}(\text{t}))^* \\
\text{VALUE}'(\text{loc}) & \text{otherwise} \end{cases}

(\text{Note that we have extended the notation so that } \text{op}^* \text{ and } \text{op} \text{ denote the \underline{ordered} set of input and output locations of op. The ordering is the same as the corresponding operator in the RDFS. i.e. if } |\text{op}| > 1 \text{ then consider } a = (\psi^{-1}(\text{op}))^* \text{ and let } (\text{op})_{i}^* = \psi((a)_{i}^*). \text{ Similarly } (\text{op}^*)_{i}^* = \psi((a')_{i}^*). )

The firing of transition 't' as above is denoted by \( \gamma' \xrightarrow{t} \gamma'' \).

We illustrate the firing rules for the configuration in fig. 3.3. For the configuration given, the actor \( A_1 \) (the union node) is firable in the WFRDFS and transition \( t_1 \), in the CFS. Note that transition \( t_1 \) corresponds to actor \( A_1 \). Firing \( A_1 \) in the WFRDFS 'S' places a token in link \( l_2 \) with associated value \( v_1 \). Firing \( T_1 \) in the CFS-DFS causes the operator \( \text{op}_1 = \text{FIRE}^{-1}(t_1) \) to fire simultaneously. The configuration reached has a token in place \( p_2 \) and a value \( v_1 \) associated with location \( \text{loc}_2 \). Notice that the configurations reached in 'S' and the CFS-DFS correspond. i.e. they are related by \( \xi \).

\textbf{Execution sequences:} let \( \gamma_0, \gamma_1, \ldots, \gamma_k \) be a sequence
BEGIN;
READ IN;
DO WHILE (P(IN) & Q(H(IN)) ) ;
IN ← H(F(IN)) ;
END;
IF P(IN)
THEN OUT ← IN ;
ELSE OUT ← H(IN) ;
PRINT OUT;
END;

(a) A program in a typical block structured language

(b) An equivalent WFRDFS

Fig. 3.3: Deriving a CFS-DFS from a WFRDFS
Fig. 3.3 (contd.)
(e) Notation: Denote
the set of actors in the WFRDFS by \( A = \{A_1, \ldots, A_n\} \);
the set of links in the WFRDFS by \( L = \{l_1, \ldots, l_m\} \);
the set of places in the CFS by \( P = \{p_1, \ldots, p_n\} \);
the set of transitions in the CFS by \( T = \{t_1, \ldots, t_n\} \);
the set of locations in the DFS by \( \text{LOC} = \{\text{loc}_1, \ldots, \text{loc}_m\} \);
the set of operators in the DFS by \( \text{OP} = \{\text{op}_1, \ldots, \text{op}_n\} \).

(f) The function \( \psi: L \rightarrow P; \forall i \in \{1, \ldots, 2l\}, (l_i) = p_i \)

(g) The function \( \psi: L \rightarrow \text{LOC}; \forall i \in \{1, \ldots, 2l\}, (l_i) = \text{loc}_i \)

(h) The function \( \text{FIRE}: \text{OP} \rightarrow T; \) (Operator \( \text{op}_i \) is
  fired in the DFS when transition \( \text{FIRE}(\text{op}) \) fires
  in the CFS)

\[
\begin{array}{cccccccccccc}
\text{op} & \text{OP} & \text{op}_1 & \text{op}_2 & \text{op}_3,4,5 & \text{op}_6 & \text{op}_7 & \text{op}_8,9 \\
\text{FIRE}(\text{op}) & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{op} & \text{OP} & \text{op}_{10,11} & \text{op}_{12} & \text{op}_{13} & \text{op}_{14,15,16} & \text{op}_{17} \\
\text{FIRE}(\text{op}) & t_7 & t_8 & t_9 & t_{10} & t_{11} \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
\text{op} & \text{OP} & \text{op}_{18} & \text{op}_{19} & \text{op}_{20} & \text{op}_{21} & \text{op}_{22} \\
\text{FIRE}(\text{op}) & t_{12} & t_{13} & t_{14} & t_{15} & t_{16} \\
\end{array}
\]

(i) The function \( \psi: T \rightarrow A; \) (Transition \( t_i \) cor-
  responds to actor \( (t_i) \). For corresponding execution
  sequences, \( t_i \) fires in the CFS and \( (t_i) \) in the
  RDFS.)

\[
\begin{array}{cccccccccccc}
t_1 & t_{1,2} & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 & t_{10,11} \\
(t_i) & A_1 & A_2 & A_3 & A_4 & A_5 & A_6 & A_7 & A_8 & A_9 \\
\end{array}
\]

\[
\begin{array}{cccccccccccc}
t_1 & t_{12} & t_{13,14} & t_{15,16} \\
(t_i) & A_{10} & A_{11} & A_{12} \\
\end{array}
\]

\text{Fig. 3.3} (contd.)
(j) The function $\text{CONTROL}: T \to \{\text{True, False}\}$. (For transition $t_i$ to be 'firable', all its input places must have tokens and $\text{CONTROL}(t_i)$ must be 'true'.)

<table>
<thead>
<tr>
<th>$t_i$</th>
<th>$t_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{CONTROL}(t_i)$</td>
<td>$\text{True}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$t_i$</th>
<th>$t_{11}$</th>
<th>$t_{13}$</th>
<th>$t_{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{CONTROL}(t_i)$</td>
<td>$\text{VALUE}(\text{loc}_{12})$</td>
<td>$\text{VALUE}(\text{loc}_{16})$</td>
<td>$\text{VALUE}(\text{loc}_{16})$</td>
</tr>
</tbody>
</table>

Fig. 3.3 (contd.)
of configurations of a CFS-DFS for some interpretation. Let \( T \) be the set of transitions in the CFS. Then the execution sequence \( \omega \) is defined as follows:

\[ \omega : \eta \to T \quad \text{where} \quad \eta = \{0, 1, \ldots\} \]

and

\[ y_0 \xrightarrow{t_0} y_1 \xrightarrow{t_1} y_2 \xrightarrow{t_2} \ldots \xrightarrow{t_{k-1}} y_k \]

where if \( t_i = x \) then transition \( x \) in \( T \) is firable for configuration \( y_i \) and \( y_{i+1} \) is the result of firing transition \( x \).

**Determinacy:** A CFS-DFS is determinate iff for any particular interpretation and initial configuration \( y'_0 \), the sequence of values associated with each location is the same for all execution sequences. This can be formalized analogously to that for RDFS's as in section 2.2.

3.3 The simulation of an RDFS by a CFS-DFS.

In this section we show that the CFS-DFS obtained from an RDFS, as detailed earlier, simulates the source RDFS. By 'simulation' we mean that the new model performs the same computation and exhibits the same parallelism as the model it simulates.

**Definition:** A CFS-DFS, obtained by the transformation of section 3.2, from an RDFS, is said to simulate the source RDFS if for any interpretation and corresponding initial configurations \( y_0 \) and \( y'_0 \) in the RDFS and the CFS-DFS respectively, there is a one to one correspondence between execution sequences in the models. Further, for corresponding execution sequences, the sequences of configurations also correspond. i.e. if \( \mathcal{E} \) and \( \mathcal{E}' \) are sets of
execution sequences in the RDFS and the CFS-DFS respectively for some initial configuration \( Y_0 \) and \( Y_0' = \xi(Y_0) \) then there exists a bijection

\[ \chi: \mathcal{E} \rightarrow \mathcal{E}' \text{ s.t. } \forall e \in \mathcal{E}, \chi(e) \in \mathcal{E}' \]

if \( e = a_1, a_2, \ldots, a_i \in A, \chi(e) = t_1, t_2, \ldots, t_i \in T \)

\[ \forall i \in \{1, 2, \ldots\} \text{ then } \forall n \in \{1, 2, \ldots\} \]

if \( \gamma_0 \xrightarrow{a_1 \cdot a_2 \cdot \ldots \cdot a_n} \gamma_n \) and \( \gamma_0' \xrightarrow{t_1 \cdot t_2 \cdot \ldots \cdot t_n} \gamma_n' \)

then \( \gamma_n' = \xi(\gamma_n) \)

In the definition above, the transform related configurations ensure that the models perform the same computation and the correspondence between execution sequences ensures the same parallelism in the models.

We now show that a CFS-DFS derived from a safe RDFS simulates the source RDFS for an initial configuration with non-null tokens only on input links of the RDFS.

**Notation:** The proof uses the correspondence between transitions in the CFS and actors in the RDFS. Let \( \alpha: T \rightarrow A \) be a function from transitions 'T' in the CFS to actors 'A' in the RDFS s.t. \( \forall t \in T, \alpha(t) = (\gamma^{-1}(t))' \). (A study of fig. 3.1 shows that \( \alpha(t) \) is a unique actor.)

**Theorem 3.3:** The CFS-DFS derived by the transformation of section 3.2, from a safe and conflict-free RDFS, simulates the source RDFS for an initial configuration of one token on each input link of the source RDFS.

**Proof:** Let the initial configuration with a token on each input link of the RDFS be \( Y_0 \). \( \xi(Y_0) \) is the corresponding configuration in the CFS-DFS. Let \( e' \in \mathcal{E}' \) be an execution
sequence in the CFS-DFS. We show that if $e' = t_0t_1\ldots\ldots$, 
$\lambda(e') = \alpha(t_0) \alpha(t_1) \ldots\ldots$ is a sequence in the RDFS that
is an execution sequence in $E$. Furthermore, it satisfies the requirement that
$\forall n = 1, 2, \ldots$ if
$\gamma_0 \xrightarrow{t_0t_1\ldots\ldots t_{n-1}} \gamma_n$ and $\gamma_0 \xrightarrow{t_0t_1\ldots\ldots t_{n-1}} \gamma_n$
then $\gamma_n = \mathcal{E}(\gamma_n)$.

We show this by an induction on the length of execution sequences $e' \in E'$.

**Basis step:** The basis of execution sequences of length '0' is trivially true since $\gamma_0 = \mathcal{E}(\gamma_0)$ by hypothesis.

**Induction step:** Suppose that the statement is true for execution sequences $e'$ of length $n$. i.e. $\gamma_n = \mathcal{E}(\gamma_n)$.

A study of the transformation (figs. 3.1 and 3.2) then shows that an actor 'a' in the RDFS is firable iff a transition $t \in \alpha^{-1}(a)$ is also firable in the CFS-DFS.

(Example: if $\alpha(t) \in \mathcal{O}$, the set of operator nodes in the
RDFS 'S', then $t$ firable $\Rightarrow \forall p \in 't$, $M(p) > 0$
$\Rightarrow \forall l \in \mathcal{L}^{-1}('t)$, VALUE(1) $\neq$ NULL $\Rightarrow \alpha(t)$ is firable in 'S' for $\gamma_n$. Similarly if 'a' is firable in 'S' for $\gamma_n$, $a \in \mathcal{O}$, then $\alpha^{-1}(t)$ is also firable in the CFS-DFS for $\gamma_n$. Note however that if $a \in \mathcal{U} \mathcal{U} \mathcal{Sw}$ (a switch or a
union node) then $|\alpha^{-1}(t)| = 2$ and by the construction exactly one of the transitions in $\alpha^{-1}(t)$ is firable if 'S' is safe and conflict free.)

Furthermore firing 'a' in the RDFS and the firable transition $t \in \alpha^{-1}(a)$ in the CFS-DFS leads to configurations $\gamma_{n+1}$ and $\gamma_{n+1}'$ in the RDFS and the CFS-DFS respectively
such that $\gamma_{n+1}' = \xi(\gamma_{n+1})$. Again, this follows from a study of the transformation and firing rules. (Example: if $a \in 0$, an operator node in the RDFS, and 'a' is firable for $\gamma_n$. Then for $\gamma_n \xrightarrow{a} \gamma_{n+1}$, if $f \in F$ is the letter associated with 'a', then $\text{VALUE}(a')_i = f_i(\text{VALUE}(a))$, $\forall i=1,\ldots,|a'|$. Thus for $\xi(\gamma_{n+1})$ the corresponding configuration, $\text{VALUE}'(\gamma(a')_i) = \text{VALUE}(a')$. But the same values are obtained by firing the transition $\alpha^{-1}(t)$ since $\text{FIRE}(t) = 'a'$ and $(\text{FIRE}(t)) = \gamma(a) and for $\gamma_n$ and $\gamma_n'$, $\text{VALUE}(a')_i = \text{VALUE}'(\text{FIRE}(t))_i \forall i=1,\ldots,|a'|$. Also the same function letter is associated with 'a' and $\text{FIRE}(t)$. Similar arguments hold for the other actors.)

Thus the statement is true for execution sequences $e'$ of length $(n+1)$ and the induction is complete.

We now show that $\lambda: \mathcal{E}' \rightarrow \mathcal{E}$ is a bijection.

Given two execution sequences $e_1'$ and $e_2' \in \mathcal{E}'$ $e_1' \neq e_2' \Rightarrow \lambda(e_1') \neq \lambda(e_2')$. This follows because if $e_1' = t_1^1 t_2^1 t_3^1 \ldots \neq e_2' = t_1^2 t_2^2 t_3^2 \ldots$ then $\exists k \text{ s.t. } \forall n < k, t_n^1 = t_n^2, t_k^1 \neq t_k^2$. Thus $t_k^1$ and $t_k^2$ are both fireable for the same configuration. So (from the proof of the induction step above) $\alpha(t_k^1)$ and $\alpha(t_k^2)$ are also fireable for the corresponding configuration in the RDFS. But $\alpha(t_k^1) \neq \alpha(t_k^2)$ if the RDFS is conflict free (because otherwise tokens exist on both input ports of a union node). Thus $\lambda(e_1') \neq \lambda(e_2')$ and $\lambda$ is one to one.

Given an execution sequence $e \in \mathcal{E}$, $e = a_1 a_2 \ldots$, we can find a valid execution sequence $e' \in \mathcal{E}' \in \alpha^{-1}(a_1) \alpha^{-1}(a_2) \ldots$
by choosing the enabled transitions corresponding to switches. This follows directly from the statement (proved in the induction step) that an actor 'a' in the RDFS is firable iff a transition $t \in \alpha^{-1}(a)$ is also firable. Further $e = \lambda(e')$. Thus $\forall e \in \mathcal{E}, \exists e' \in \mathcal{E}$ s.t. $\lambda(e') = e$. Thus $\lambda$ is onto.

This proves that $\lambda$ is a bijection and the proof is complete.

**Corollary 1:** A CFS-DFS obtained by the transformation of section 3.2 from a safe, conflict-free and determinate RDFS is also determinate.

**Proof:** From theorem 3.3, there is a one to one correspondence between execution sequences in the RDFS and the CFS-DFS (from initial configurations $\gamma_0$ and $\xi(\gamma_0)$). Also for corresponding execution sequences the sequences of configurations also correspond. Thus for corresponding execution sequences, the sequences of values in link nodes and related locations are identical. Since the RDFS is determinate, these sequences are identical for all execution sequences. Thus the CFS-DFS is also determinate.

**Corollary 2:** A CFS-DFS derived from a WFRDFS, simulates the source WFRDFS and is determinate.

**Proof:** From theorem 2.1 a WFRDFS is safe, conflict-free and determinate. Hence theorem 3.3 and corollary 1 above apply and the statement follows.

* Henceforth an initial configuration with one token on each input link of the RDFS is assumed. Initial configurations in the CFS-DFS are assumed to be mapped by from initial configurations in the RDFS.
The firing rules for the CFS-DFS that we have specified earlier require that when an operator fires, null values be placed in the operand locations (that are not result locations as well). Such a 'clearing of operands' is required in a data flow program because the presence of operands is used to 'fire' actors. However, in the CFS-DFS, the control function is carried out by the CFS. This makes it unnecessary to place null values in operand locations of an operator. The operator merely needs to read values from these locations. We wish to make this change because it allows several operators to 'read' the same value from a location without constraining them to fire simultaneously. This is required later in the development when identity operators are removed from the DFS. This is only possible if several actors are allowed to read the same value.

The problem with making this change in the firing rules is that configurations do not correspond as defined earlier. That is, if the starting configurations are \( \gamma_0 \) and \( \mathcal{E}(\gamma_0) \) and corresponding transitions in the CFS and actors in the DFS are fired, the resulting configurations are no longer related by \( \mathcal{E} \) (as defined earlier). We modify the definition of \( \mathcal{E} \) so that configurations in the RDFS and the CFS-DFS are related by \( \mathcal{E} \) if the non-null values associated with links are the same as those associated with corresponding locations. With these changes theorem 3.3 and its corollaries go through again.
Definition: \( \xi \) is a relation between configurations 
\( \forall \xi \in \Gamma \) in the RDFS 'S' and \( \forall \xi' \in \Gamma' \) in the CFS-DFS derived from the RDFS (by the transformation of section 3.2) s.t.

\[(\xi, \xi') \in \xi \text{ iff} \]

\( \forall l \in L \) (the set of links in 'S') s.t.

\[\text{VALUE}(\xi, l) \in \{\text{True, false}\}, \]

\[M(\xi(1)) = 1, \text{VALUE}'(\xi(1)) = \text{VALUE}(\xi, 1)\]

and \( \forall l \) s.t. \( \text{VALUE}(\xi, l) = \text{Null}, \)

\[M(\xi'(1)) = 0, \text{VALUE}'(\xi'(1)) \in \{\text{Null, True, False}\} \cup \mathbb{R}. \]

i.e. for null tokens associated with links, the corresponding locations have any associated value.

Modification: Change the firing rules defined earlier in this section so that \( \text{VALUE}' \) is changed to:

\[\forall \text{loc} \in \text{LOC}, \text{VALUE}''(\text{loc}) = \begin{cases} 
 f_i(\text{VALUE}'('op')) & \text{if } \text{loc}=(\text{op}')_i \\
 \text{VALUE}'(\text{loc}) & \text{otherwise.} 
\end{cases} \]

Theorem 3.3*: with the modification above, theorem 3.3 holds for initial configurations related by \( \xi \) and with \( \xi \) being used in the definition of simulation instead of \( \xi \).

Proof: Follows analogously to theorem 3.3.

Corollary: The corollaries to theorem 3.3 hold after the above modifications.

Proof: the corollary follows as before after noting that in defining determinacy, null values are deleted from the
sequence of values in locations.

**Lemma 3.1**: Consider a CFS-DFS 'C-D' derived from an RDFS 'S'. Let 'x' and 'y' be two link nodes in 'S'. Then there exists a path 'P' from x to y in S iff there is a path p' from K(x) to K(y) in the CFS. Furthermore, if p passes through link nodes x, l_1, l_2, ..., l_n, y then p' passes through places K(x), K(l_1), ..., K(l_n), K(y).

**Proof**: Follows by a trivial induction on the length of paths by examining the transformation rules of fig. 3.1.
CHAPTER 4

REDUCTION OF THE CONTROL AND DATA FLOW STRUCTURES

In this chapter we simplify the CFS-DFS derived, from an RDFS, by the transformation of section 3.2. In the CFS-DFS thus obtained, the subgraphs for wyes, unions and switches contain identity operators. As pointed out earlier, these node types often make up more than half of an RDFS. Further, for the most part, these identity operators are unnecessary for producing a determinate result. On the other hand, an indiscriminate removal of them can lead to non-determinacy or a change in the computation performed. We derive conditions which allow the removal of identity operators with the preservation of determinacy and of the correctness of the computation.

When identity operators are removed from the DFS, some transitions in the CFS neither initiate the firing of any operators in the DFS nor are required for synchronization of the program. In other words, the CFS can be simplified by removing these superfluous transitions. We specify an algorithm to do this and prove that the computation performed is unchanged. Further, we show that the removal of some transitions can actually increase the parallelism obtained.

4.1 Reduction of the data flow structure.

In this section we specify conditions that allow the removal of identity operators from the DFS while retaining the correctness of the computation performed.
We first derive general sufficient conditions that allow the removal of identity operators. We then show that, for a CFS-DFS derived from a WFRDFS, the identity operators corresponding to wyes and switches in the source WFRDFS can be removed directly. Conditions are then derived for removing identity operators corresponding to union nodes.

**Definition:** The operation of the 'removal' of an identity operator 'I' from the DFS is as follows:

(i) Fig. 4.1 indicates graphically how the DFS is modified. Let \( \text{LOCA} = 'I' \) and \( \text{LOCB} = 'I' \). (Necessarily, from the construction, \(|'I'| = |I'| = 1\)). Locations \( \text{LOCA} \) and \( \text{LOCB} \) are deleted from the set of location 'LOC' in the DFS and a new location \( \text{LOCAB} \) is inserted in LOC s.t.

\[
(\text{LOCAB}) = (\text{LOCA}) \cup (\text{LOCB}) \quad \text{and} \quad (\text{LOCAB}') = (\text{LOCA}') \cup (\text{LOCB}').
\]

'I' is deleted from the set of operators 'OP' in the DFS.

(ii) Modify the domain of the function \( \text{FIRE: OP} \rightarrow T \), from operations in the DFS to transitions in the CFS, so that 'OP' is now the new set of operators with 'I' deleted.

(iii) Modify the function \( \psi: L \rightarrow \text{LOC} \), from links in the RDFS to locations in the DFS so that

\[
\forall l \text{ s.t. } \psi(l) \in \{\text{LOCA, LOCB}\}, \quad \psi(1)|_{\text{new}} = \text{LOCAB}.
\]

i.e. all link nodes that corresponded to \( \text{LOCA} \) or \( \text{LOCB} \) are made to correspond to the new location \( \text{LOCAB} \).

**Notation:** \( \psi_{\text{loc}} \in \text{LOC} \), the set of locations in the DFS,

\[
|\text{loc} = \psi(\psi^{-1}(\text{loc})). \quad \text{i.e. it is the set of places in the CFS corresponding to the location 'loc' in the DFS.}
\]
Fig. 4.1: The 'removal' of an 'I' operator from the DFS.
*/loc/* denotes the subgraph of places in /loc/ and transitions with at least one input place and one output place in /loc/ and the edges in the CFS interconnecting them.

4.1.1 General conditions for the removal of identity operators:

We first specify a sufficient condition for the removal of an identity operator with the preservation of the correctness of the computation. This condition essentially is that the places that correspond to input and output location of the identity operator must not contain tokens simultaneously. An additional constraint is then imposed to obtain a stronger sufficient condition that is 'almost necessary'.

Definition: Consider a CFS-DFS 'C-D' derived from a conflict free, well behaved, live and safe RDFS 'S' and the CFS-DFS C'-D' obtained from C-D by the removal of an arbitrary number of 'I' operators. Then C-D and C'-D' are said to perform the same computation iff for all interpretations and initial configurations \( \gamma_0 \) (with a token on each input link of the RDFS), and for all finite execution sequences, the final value at each location in C-D mapped by \( \gamma \) from an output link of 'S' is identical to the value at the corresponding location (mapped by \( \gamma_{\text{new}} \)) in C'-D'.

Lemma 4.1: Consider a CFS-DFS 'C-D' derived from a well behaved, live, safe and determinate RDFS 'S' followed
by the removal of an arbitrary number of 'I' operators. A value is placed in a location $x \in \text{LOC}$ (the set of locations) when a token is placed in the set of places $/x/$ by any transition in $\{t \mid t \in T \text{ and } \text{FIRE}^{-1}(t) \neq \phi\}$. Further, a value in $x$ is 'used' by a successor operator of $x$ when a token is removed from $/x/$ by a transition in $\{t' \mid t' \in T \text{ and } \text{FIRE}^{-1}(t') \neq \phi\}$. Hence no operator in $x'$ fires when there is no token in $/x/$.

**Proof:** By an induction on the number of 'I' operators removed from the CFS-DFS.

**Basis step:** For no 'I' operators removed from the CFS-DFS, there is a one-to-one correspondence between places in the CFS and locations in the DFS. The lemma follows trivially from the transformation rules (fig. 3.1 and 3.2) and the firing rules (section 3.2).

**Induction step:** Suppose that the lemma is true after removing $n$ 'I' operators from the CFS-DFS. Consider removing an $(n+1)^{st}$ 'I' operator as in fig. 4.1. Now $\forall \text{ LOCA, LOCB } \in \text{ LOC, if } \text{ LOCA} = \text{ LOCB} \text{ then } /\text{ LOCA}/ \text{ and } /\text{ LOCB}/$ are disjoint. (This can be proved by an easy induction on the number of 'I' operators removed.) Thus, from the induction hypothesis, a token is placed in $/\text{ LOCA}/$ or $/\text{ LOCB}/$ when a value is place in $\text{ LOCA}$ or $\text{ LOCB}$ by a transition in $\{t \mid t \in T \text{ and } \text{FIRE}^{-1}(t) \neq \phi\}$; a value in $\text{ LOCA}$ or $\text{ LOCB}$ is used by an operator when a token is removed from a place in $/\text{ LOCA}/$ or $/\text{ LOCB}/$ by a transition in $\{t' \mid t' \in T \text{ and } \text{FIRE}^{-1}(t') = \phi\}$; no successor operator of
/LOCA/ or /LOCB/ can fire when no place in /LOCA/ U /LOCB/ has a token. But /LOCAB/ = /LOCA/ U /LOCB/. Thus the statement is true for (n+1) 'I' operators removed and the induction is complete.

**Lemma 4.2**: sufficient condition 1 for the removal of an identity operator.

Consider a CFS-DFS 'C-D' derived from a well-behaved, live, safe and determinate RDFS, 'S'. Then if the following condition is satisfied, 'I' operators can be removed recursively (i.e. one 'I' operator is removed at a time and the new CFS-DFS obtained. The condition must be satisfied for the new CFS-DFS to remove another 'I' operator.) with no change in the computation performed; for LOCA and LOCB as defined in fig. 4.1, the sets of places /LOCA/ and /LOCB/ must not contain tokens simultaneously. i.e. ∀ p_a ∈ /LOCA/ and p_b ∈ /LOCB/, p_a and p_b must not contain tokens simultaneously.

**Proof**: We show that if the above condition is satisfied then the sequence of values in all locations other than LOCA and LOCB are unchanged by the removal of an 'I' operator. Further if LOCA (or LOCB) is the mapping of an output link, then the final value LOCAB contains is the same as LOCA (or LOCB) for all finite execution sequences.

The conditions of the lemma require that /LOCA/ and /LOCB/ do not contain tokens simultaneously. From lemma 4.1, no successor operator of LOCA ( LOCB ) fires when there is a token in /LOCB/ ( /LOCA/ ). Furthermore,
a 'useful' value does not exist in /LOCA/ and /LOCB/ simultaneously (i.e. one that will be 'used' by an operator as an operand when it fires). Thus the values placed in locations other than LOCA and LOCB are unchanged by the merging of LOCA and LOCB to LOCAB as in fig.4.1. Thus, the values in output locations other than LOCA and LOCB are unchanged for all finite execution sequences.

Now LOCA and LOCB (fig. 4.1) cannot both be output locations if there are any finite execution sequences. If both were output locations with a finite execution sequence existing, since the source RDFS is well behaved by hypothesis, final values would appear in both simultaneously. So tokens would appear in /LOCA/ and /LOCB/ simultaneously.

The sequence of values in location 'LOCAB' is the 'time intersection' of values in LOCA and LOCB. i.e. a new value is placed in LOCAB whenever a token appeared in either of locations LOCA or LOCB. Thus if LOCA or LOCB is mapped by \( \forall \) from an output link, the final value in LOCAB is the same as it was for the output location LOCA or LOCB. Thus the computation is unchanged and the proof is complete.

***

The conditions of lemma 4.1 are not necessary since it is possible for /LOCA/ and /LOCB/ to have tokens simultaneously but for LOCA and LOCB to have the same value when this occurs. For this case too, the 'I' operator can be removed with no change in the computation. Further,
this is not a pathological case but occurs for the 'I' operators corresponding to wyes and switches in a CFS-DFS derived from a WFRDFS.

**Lemma 4.3:** Sufficient condition 2 for the removal of an identity operator preserving the computation.

In a CFS-DFS derived, by the transformation of section 3.2, from a well-behaved, live, safe and determinate RDFS, S, 'I' operators can be removed recursively * with preservation of the computation if (see fig. 4.1):

- either (a) /LOCA/ and / LOCB/ do not contain tokens simultaneously
- or (b) For all interpretations and initial configurations, LOCA and LOCB hold identical values when /LOCA/ and / LOCB/ have tokens simultaneously.

**Proof:** For case (a), lemma 4.1 holds directly. The arguments of lemma 4.1 also hold for case (b). ***

* See lemma 4.1.
4.1.2 The removal of identity operators for structured nets.

The conditions developed in section 4.1.1 are rather difficult to apply for general live and safe nets. We now examine the reduction of the data flow structure for a CFS-DFS derived from a WFRDFS. We first show that if the identity operators corresponding to union nodes are retained then all other identity operators can be directly removed from the CFS-DFS with the preservation of the correctness of the computation. We prove this by showing that the sufficient conditions of section 4.1.1 are satisfied. We then examine the removal of identity operators corresponding to union nodes.

Theorem 4.1: Consider a CFS-DFS, 'C-D', derived from a WFRDFS 'S' by the transformation of section 3.2. If the identity operators corresponding to union nodes in the RDFS are retained, then all other 'I' operators can be removed from the CFS-DFS.

i.e. recalling that FIRE: OP → T and ≅: T → A are functions from operators to transitions and from transitions to actors, ∀ identity operators i ∈ OP, if ≅(FIRE(i)) ∈ U, the set of union nodes, then these 'I' operators are retained in the DFS otherwise the 'I' operator is removed from the CFS-DFS.

Proof: The identity operators that we claim are removable directly are those corresponding to 'wyes' and 'switches' in the source RDFS. The transformation rules for these
operators are repeated in fig. 4.2. We show that for identity operators of these types, the assumptions of lemma 4.3 are satisfied if the 'I' operators corresponding to union nodes are retained.

We first show, by an induction on the number of such 'I' operators removed, that /LOCA/* and /LOCB/* (fig. 4.1) have a unique input place *

**Basis step:** The basis of no 'I' operators removed is trivial since in the initial CFS-DFS there is a one-to-one correspondence between locations and places and

\[ |/LOCA/| = |/LOCB/| = 1. \]

**Induction step:** Suppose it is true for n 'I' operators removed. Consider removing another 'I' operator with LOCA and LOCB as in fig. 4.1. Then, by hypothesis, /LOCA/ and /LOCB/ have unique input places, say, p_LOCA and p_LOCB. Now p_LOCB has an input transition corresponding to the 'I' operator being examined for removal. Further, this must be the unique input transition of p_LOCB since, as figs. 3.1 and 3.2 show, the only places that have more than one input transition correspond to the output links of union nodes and the 'I' operator being examined for removal cannot correspond to a union.

---

* An **input place** of a subgraph is a place in the subgraph that has an input transition not in the subgraph or is an input place of the (entire) schema (i.e. it has no input transition).
(a) Transforming a 'wye' node.

(b) Transforming a 'switch' node.

Fig. 4.2: transformation rules for 'wyes' and 'switches' (repeated from figs. 3.1 and 3.2).
Also, the transition corresponding to the 'I' being removed (i.e. FIRE(I)) must be from a place in /LOCA/ to a place in /LOCB/ since 'I' is from LOCA to LOCB and \( \mathcal{L}(\mathcal{N}^{-1}(I)) \in \mathcal{L}(\text{FIRE}(I)) \) (definitions in section 3.2).

Now /LOCAB/ = /LOCA/ U /LOCB/ and thus /LOCAB/ has \( p_{LOCA} \) as its unique input place.

It can be shown by an easy induction on the number of identity operators removed, that in /LOCA/* (or /LOCB/*) there is a path in /LOCA/* (or /LOCB/*) from its unique input place \( p_{LOCA} \) (or \( p_{LOCB} \)), to every other place in /LOCA/* (or /LOCB/*).

Now from lemma 3.1 we conclude that there exists a path in the RDFS from \( \mathcal{L}^{-1}(p_{LOCA}) \) (or \( \mathcal{L}^{-1}(p_{LOCB}) \)) to the link nodes that correspond to places in /LOCA/ (or /LOCB/). Furthermore, these paths do not pass through union nodes since there are no transitions corresponding to unions on the corresponding paths in the CFS-DFS (which are in /LOCA/* (or /LOCB/*)). Thus lemma 2.9 is applicable and tokens cannot exist simultaneously in \( \mathcal{L}^{-1}(p_{LOCA}) \) (or \( \mathcal{L}^{-1}(p_{LOCB}) \)) and any other place in \( \mathcal{L}^{-1}(/LOCA/) \) (or \( \mathcal{L}^{-1}(/LOCB/) \)).

Recall from theorem 3.3 that there is a one to one correspondence between execution sequences in the WFRDFS 'S' and the CFS-DFS C-D. Further, the configuration sequences are related by \( \mathcal{L} \) and for \( \mathcal{L} \) related configurations a place \( 'p' \) has a token iff the link \( \mathcal{L}^{-1}(p) \) also has a token. Thus there exists no execution sequence which
has tokens in $p_{\text{LOCA}}$ (or $p_{\text{LOCB}}$) and another place in
/LOCA (or /LOCB/ ) simultaneously.

Observe that from a configuration with no tokens in
/LOCA (or /LOCB/ ), tokens can only enter /LOCA/
( or /LOCB/ ) by a token being placed in $p_{\text{LOCA}}$ (or $p_{\text{LOCB}}$)
(since this is the only place with an input transition
not in the subgraph). From the above reasoning, a new
token can only appear in $p_{\text{LOCA}}$ after all tokens have
been removed from /LOCA/ (or /LOCB/ ). These statements
also hold for /LOCA/ U /LOCB/ since a similar development
holds for ( /LOCA/ U /LOCB/ ) $^*$ . But /LOCAB/ =
/LOCA/ U /LOCB/ (from definitions). Thus, a new token
can only appear in $p_{\text{LOCA}}$ after all tokens have been
removed from /LOCAB/. i.e. a new value is placed in
LOCA only after the old value in LOCA and LOCB has been
removed (see proof of lemma 4.1). Further, since LOCA
and LOCB are connected by an 'I' operator, the value in
LOCA and LOCB is the same when they have tokens simul-
taneously. The conditions of lemma 4.3 are thus satisfied
and the lemma follows.

***

We illustrate the simplification of the DFS that results
from the 'removal' of identity operators corresponding
to 'wyes' and 'switches' by removing these 'I' operators
for the example of fig. 3.3. The new CFS-DFS that results
is given in fig. 4.3. Note the vast simplification of
the DFS. Note that the transitions that corresponded to
wyes earlier, now serve no purpose whatever. The
Fig. 4.3: the simplification of the DFS by removing 'I' operators corresponding to 'wyes' and 'switches'.

(a) The CFS (unchanged from fig. 3.3 (c))

(b) The reduced DFS (after removing 'I' operators corresponding to 'wyes' and 'switches').
(c) The function \( \psi : L \rightarrow P \) is unchanged. i.e.
\[ \forall i \in \{1, \ldots, 21\}, \ \psi(i) = p_i. \]

(d) The function \( \psi : L \rightarrow \text{LOC} \). (Note that this is no longer a bijection.)

| \( i \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 1 \) | \( 2 \) | \( 3 \) | \( 4 \) | \( 5 \) | \( 15 \) | \( 18 \) | \( 16 \) | \( 8 \) | \( 9 \) | \( 16 \) |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| \( \psi(i) \) | \( \text{loc}_1 \) | \( \text{loc}_2 \) | \( \text{loc}_3 \) | \( \text{loc}_7 \) |

<table>
<thead>
<tr>
<th>( i )</th>
<th>( 1 )</th>
<th>( 1 )</th>
<th>( 1 )</th>
<th>( 1 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 1 )</th>
<th>( 2 )</th>
<th>( 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \psi(i) )</td>
<td>( \text{loc}_7 )</td>
<td>( \text{loc}_{21} )</td>
<td>( \text{loc}_{12} )</td>
<td>( \text{loc}_{14} )</td>
<td>( \text{loc}_{20} )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(e) The function \( \text{FIRE} : \text{OP} \rightarrow \text{T} \)

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{op}_1 )</th>
<th>( \text{op}_2 )</th>
<th>( \text{op}_6 )</th>
<th>( \text{op}_7 )</th>
<th>( \text{op}_{12} )</th>
<th>( \text{op}_{13} )</th>
<th>( \text{op}_{18} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{FIRE(op}_i) )</td>
<td>( t_1 )</td>
<td>( t_2 )</td>
<td>( t_4 )</td>
<td>( t_5 )</td>
<td>( t_8 )</td>
<td>( t_9 )</td>
<td>( t_{12} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( i )</th>
<th>( \text{op}_1 )</th>
<th>( \text{op}_{21} )</th>
<th>( \text{op}_{22} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{FIRE(op}_i) )</td>
<td>( t_{15} )</td>
<td>( t_{16} )</td>
<td></td>
</tr>
</tbody>
</table>

Note that \( t_3, t_6, t_7, t_{10}, t_{11}, t_{13}, t_{14} \) do not initiate any data flow operation.

(f) The function \( \prec \text{T} \rightarrow \text{A} \) is unchanged from that in fig. 3.3 (i).

(g) The function \( \text{CONTROL} : \text{T} \rightarrow \{\text{True, False}\} \);
simplification of the CFS that can be carried out as a consequence of this is examined in section 4.2. The changes in the functions \( \psi \), FIRE and CONTROL are also indicated in fig.4.3. (The change in the function CONTROL is implicit in that CONTROL is defined in terms of \( \psi \) which changes explicitly by the removal of 'I' operators.)

We now examine the removal of identity operators in the DFS (of a CFS-DFS derived from a WFRDFS) which correspond to union nodes in the source WFRDFS. We will assume that all 'I' operators corresponding to wyes and switch nodes in the WFRDFS have already been removed as detailed earlier. The problem we will examine is that of deriving conditions that allow the removal of identity operations corresponding to union nodes with the preservation of the computation.

We first specify an algorithm for finding a subset of the set of SCMG components of the CFS derived from a WFRDFS. In this context we note that the CFS interpreted as a normal Petri net is not free choice. However, it is easily shown that applying Hack's MG reduction algorithm (section 3.1) does produce strongly connected marked graphs that cover the CFS. It is also easily shown that the set of SCMG components, obtained from the algorithm we specify, is a subset of the set of SCMG components produced by Hack's procedure. However, this is not of much consequence to the present discussion.
We next show that the application of the sufficient conditions of lemma 4.2 to the sets of places restricted to the SCMG components obtained is equivalent to applying them to the entire CFS. We show how the conditions are applied to strongly connected marked graphs.

Finally, we consider a special case that is handled easily.

**Definition:** Consider a CFS, C-D, derived from a well-behaved RDFS. The modified CFS corresponding to C-D (M-CFS) is obtained from the CFS as indicated in fig. 4.4.

The reason for 'closing the net' as indicated is to make the M-CFS live in the Petri net sense of the term. This also enables us to get strongly connected marked graph components that 'cover' the CFS.

**Algorithm 4.1:** Obtaining 'basic SCMG components' of a WFRDFS.

Consider a WFRDFS 'S' and the CFS-DFS 'C-D' obtained from 'S' by the transformation of section 3.2. Let \( \zeta \) be the bijection from links in 'S' to places in 'C'.

1. Start with a null set of 'basic SCMG components' of 'S'.
2. Construct the Hasse diagram 'H' for 'S'(as in section 2.3) with 'S' as the root of 'H' and all 'component' loop and conditional subschemas as nodes in the rooted tree. (Recall that 'H' is a rooted tree with root 'S'.)
3. For each internal node \( n \) in 'H' consider the associated WFRDFS \( S_n \). In \( S_n \) replace all subschemas that
correspond to sons of n in 'H' by uniquely labelled operators with the same number of input and output links as the replaced subschema.

4. **Claim 1**: each node in 'H' now represents a WFRDFS of a type in fig. 4.4. (i.e. it is an acyclic interconnection of operators or a loop or conditional schema with subschemas consisting of acyclic interconnections of operators.)

**Claim 2**: For each node n in 'H', the associated WFRDFS S\_n has link nodes in 'S' (since no link nodes are added).

For each node 'n' in 'H' and associated WFRDFS S\_n use the transformation of section 3.2 to obtain a CFS C\_n. Use the same function \(\kappa\) restricted to links in S\_n to obtain places in C\_n so that the set of places in C\_n is a subset of the set of places in C (the CFS obtained from S). Also let the function \(\phi\) (from transitions in the CFS to actors in the WFRDFS) be unchanged from that in the transformation of 'S' to C-D except for the new operators introduced. i.e. the set of transitions in C\_n are a subset of the set of transitions in C except for the new transitions corresponding to the new operators.

5. For each leaf in the Hasse diagram 'H', consider its SCMG components obtained as in fig. 4.5.

* We prove these later.
5.1 If the Hasse diagram is merely 'S' (i.e. a single node) then add all the SCMG components to the set of basic SCMG components of 'S'. The input places of basic SCMG components correspond to (i.e. are mapped by * from) input links of 'S'.

5.2 If the Hasse diagram is not merely S then do the following.

5.2.1 Add all SCMG components of leaves which do not contain places corresponding to their input links (i.e. obtained from a loop schema as in fig. 4.5 (b) (iii) ) to the set of basic SCMG components. The input places of these SCMG components are the places corresponding to the output links of the loop union (marked p₁,...,pₙ in fig. 4.5 (b) (ii) ).

5.2.2 Augment the SCMG components of ancestors of leaves by 'replacing' transitions corresponding to operators added in (2) (to represent WFRDFS's) by SCMG components (of the corresponding leaf node) which include their input links. The operation of 'replacing' transitions by SCMG components is as in fig. 4.6. (There may be many such SCMG components of leaves and all combinations of 'replacement' are used to augment the SCMG components of ancestors of leaves.)

5.3 Delete the leaves from the Hasse diagram and repeat (4) if the Hasse diagram is not blank.
(a) A CFS derived from a well behaved RDFS.  
(b) Modified CFS corresponding to (a).

**Fig. 4.4**: Obtaining a modified CFS.

(a) The single basic SCMG component obtained by transforming an acyclic interconnection of operators.

**Fig. 4.5**: obtaining basic SCMG components.
(i) RDFS

(ii) Modified CFS

(iii) Basic SCMG component 1.

(iv) Basic SCMG component 2.

(b) Basic SCMG components of the CFS obtained from a loop subschema with subschemas of acyclic interconnections of operator nodes.

Fig. 4.5 (contd.)
(i) RDFS.

(ii) Modified CFS.

(iii) Basic SCMG component 1.

(iv) Basic SCMG component 2.

(c) Basic SCMG components of the CFS obtained from a conditional subschema with subschemas \( S_1 \) and \( S_2 \) composed of acyclic interconnections of operator nodes.

Fig. 4.5 (contd.)
(a) To replace a transition 't' in an SCMG component 'S' by SCMG component $S_1$.

(b) SCMG component obtained by the 'replacement'.

Fig. 4.6: replacing transitions in an SCMG component by another SCMG component.
Fig. 4.7 : example of obtaining the basic SCMG components.
(d) SCMG components of $S_1$.

(e) SCMG components of $S_2$ (places $p_{18}$ and $p_{19}$ mapped from links $l_{18}$ and $l_{19}$ are used to distinguish the components).
(i) Basic SCMG component obtained from (c) by substituting SCMG components of (d)(i) for $S_1$ and (e)(ii) for $S_2$ in (c).

(ii) Basic SCMG component obtained from (c) by substituting SCMG component (d)(i) for $S_1$ and (e)(i) for $S_2$ in (c).

(iii) Basic SCMG component corresponding to the 'feedback choice' in $S_2$. (Same as (d)).

Fig. 4.7 (contd.)
Lemma 4.4: the claims 1 and 2 in step (4) of algorithm 4.1 are valid.

Proof: Each node in the Hasse diagram corresponds to a WFRDFS which is either an acyclic interconnection of WFRDFS's or loop or conditional subschemas with subschemas that are acyclic interconnections of WFRDFS's. Step 3 of the algorithm replaces each such sub-WFRDFS by an operator leading to loop or conditional schemas with subschemas that are acyclic interconnections of operators or merely to an acyclic interconnection of operators. These are the types of schemas in fig. 4.5 and claim 1 is true. Claim 2 is trivially true since link nodes are not added to the subgraphs.

We illustrate algorithm 4.1 by applying it to the example in fig. 3.3. The Hasse diagram, SCMG components associated with nodes of the Hasse diagram and the basic SCMG components obtained by applying algorithm 4.1 are shown in fig. 4.7. The substitution of SCMG components associated with nodes of the Hasse diagram into the SCMG components associated with their 'father' node in the Hasse diagram is also indicated. Note that the components obtained are indeed strongly connected marked graphs. Further, they cover the CFS (i.e. each place and each transition in the CFS is contained in some basic SCMG component). Finally, note that the input places of the basic SCMG components correspond either to the output links of a loop union or to input places of the schema.
We show that these observations are true in general.

**Lemma 4.5**: consider a WFRDFS 'S' and the CFS-DFS, C-D, derived from 'S'. Then the basic SCMG components obtained from 'S' by algorithm 4.1 are strongly connected marked graphs and cover the CFS, 'C' (i.e. each place and transition in 'C' is contained in some basic SCMG component).

**Proof**: By an induction on the depth of the source WFRDFS.

**Basis step**: The statement is trivially true for WFRDFS's of depth '0' since these are acyclic interconnections of operators and the basic SCMG component is the entire modified CFS as in fig. 4.5 (a) (ii). The transformation rules of fig. 3.1 and 3.2 clearly show that for operator nodes the corresponding transitions have input places with a single output transition and output places with a single input transition. Thus an acyclic interconnection of operators leads to a marked graph. The modified CFS is strongly connected because there is a path from an input node to each operator and from each operator to an output node in the RDFS. Thus lemma 3.1 asserts that this is also true for the CFS. Further, in the modified CFS there is a path from each output place to each input place. Thus the M-CFS is strongly connected.

The lemma is also true for loop and conditional schemas with their subschemas consisting of an acyclic interconnection of operators. Fig. 4.5 (b) and (c) show the basic SCMG components for such schemas. Now from the
above reasoning there are paths from each input place of $S_1'$ and $S_2'$ (fig. 4.5 (b),(c)) to transitions and places within them and from all places and transitions to output nodes. Furthermore, all places have a single input and a single output transition. Fig. 4.5 (b),(c) then show that strongly connected marked graphs are obtained and that they cover the CFS.

**Induction step:** Suppose the lemma is true for WFRDFS's of depth $n$ or less. Consider a WFRDFS of depth $(n+1)$. In step 3 of the algorithm, all WFRDFS's are converted to the type considered in the basis step and the SCMG components are strongly connected marked graphs. From the inductive hypothesis, the SCMG components obtained after the 'replacement' of step 5 of algorithm 4.1 into components associated with nodes at a depth of 1 in 'H' are strongly connected marked graphs which cover the sub-WFRDFS's. The 'replacement' into components associated with the root of 'H' then produce strongly connected marked graphs which cover 'S' as is easily seen from the replacement rules of fig. 4.6.

**Lemma 4.6:** the input places of the basic SCMG components correspond either to the output links of a loop union or to input places of the schema.

**Proof:** follows by an easy induction on the depth of the WFRDFS.

**Lemma 4.7:** let 'C-D' be a CFS-DFS derived from a WFRDFS 'S' and $p_1$ and $p_2$ be any two distinct places in 'C'.
Let $l_1$ and $l_2$ be the links in $S$ that correspond to $p_1$ and $p_2$. Then there does not exist any basic SCMG component containing both $p_1$ and $p_2$ iff:

either (i) there is a conditional subschema in $S$ with $l_1$ in one of its subschemas and $l_2$ in the other subschema

or (ii) there is a loop subschema with $l_1(l_2)$ in its feedback subschema and $l_2(l_1)$ outside the loop subschema.

Proof:

(a) To show that if the conditions (i) or (ii) are satisfied then there does not exist a basic SCMG component containing $p_1$ or $p_2$. This follows rather trivially from the algorithm and fig. 4.4.

(b) To show that if there is no basic SCMG component containing $p_1$ and $p_2$ then either of conditions (i) or (ii) must be satisfied. We prove this by an induction on the depth of the WFRDFS.

Basis step: For WFRDFS's of depth '0', the statement is trivially true since the basic SCMG component is the entire M-CFS which contains all places in 'S'.

For WFRDFS's consisting of a single loop or conditional schema with subschemas of depth '0', the statement is true as a study of fig. 4.4 reveals.

Induction step: Suppose that the statement is true for RDFS's of depth $n$ or less. Then consider a WFRDFS 'S' of depth $(n+1)$. Let $\mathcal{S}$ be the set of WFRDFS's with 'S' as an
acyclic interconnection of WFRDFS's in \( \hat{\mathcal{S}} \). A WFRDFS \( \hat{S} \) in \( \hat{\mathcal{S}} \) is either an operator or a WFRDFS of depth \( n+1 \). For loop or conditional schemas in \( \hat{\mathcal{S}} \), their subschemas are of depth \( n \) or less. Now consider the various cases.

**Case 1:** if \( p_1 \) and \( p_2 \) are both in some WFRDFS \( \hat{S} \) in \( \hat{\mathcal{S}} \) and furthermore are in the same subschema of \( \hat{S} \) then the statement is true since the subschema of \( \hat{S} \) is of depth \( n \) or less and all basic SCMG components of this subschema are either basic SCMG components of 'S' or subgraphs of basic SCMG components of 'S' (as easily follows by an induction on the depth of a schema and the replacement rules of fig. 4.6).

**Case 2:** if \( p_1 \) and \( p_2 \) are both in some WFRDFS \( \hat{S} \) in \( \hat{\mathcal{S}} \) and furthermore are in different subschemas of \( \hat{S} \) then

- **Case 2.1** if \( \hat{S} \) is a loop subschema then there is a basic SCMG component containing \( p_1 \) and \( p_2 \) (of the form of fig. 4.5.(b) (iii)) and the statement is true.
- **Case 2.2** if \( \hat{S} \) is a conditional subschema, then there is no SCMG component containing \( p_1 \) and \( p_2 \) (since the SCMG components are formed as in fig. 4.5 (c) (iii), (iv) ) and condition (i) of the lemma is satisfied. Thus the statement (b) of the lemma is satisfied.

**Case 3:** if \( p_1 \) and \( p_2 \) are in (i.e. within or input or output places of ) different subschemas \( \hat{S}_1 \) and \( \hat{S}_2 \) in \( \hat{\mathcal{S}} \). The statement follows easily, as in the previous case, for the subcases of \( \hat{S}_1, \hat{S}_2 \) being conditional, loop subschemas or operators.
This completes the cases in the induction step and proves the lemma.

Lemma 4.8: with the assumptions of lemma 4.6, suppose that $p_1$ and $p_2$ are two distinct places in some basic SCMG component $SCMG_i$. Then $p_1$ and $p_2$ can have tokens simultaneously in the entire CFS-DFS for some interpretation and initial configuration iff $p_1$ and $p_2$ can have tokens simultaneously in $SCMG_i$ with an initial marking of a token on each input place of $SCMG_i$.

Proof: follows by an easy induction on the depth of a WFRDFS.

Notation: Let 'S' be a WFRDFS and 'C-D', the CFS-DFS derived from 'S'. For a location $LOCX$ in the DFS 'D' and the set of places /$LOCX$/ in the CFS 'C' and a basic SCMG component $SCMG_i$, /$LOCX$/ is the set of places of /$LOCX$/ in $SCMG_i$.

Theorem 4.2: Let 'C-D' be the CFS-DFS obtained from a WFRDFS 'S' by the transformation of section 3.2 followed by the removal of identity operators corresponding to wyes and switches in 'S' (as described earlier in this section). Let 'I' be an identity operator in C-D and let $LOCA$ and $LOCB$ be as in fig. 4.1. Then /$LOCA$/ and /$LOCB$/ do not contain tokens simultaneously for all interpretations and all initial configurations with a token on each input link of 'S' and for all execution sequences iff for all basic SCMG components $SCMG_i$, /$LOCA$/ and /$LOCB$/ do not contain tokens simultaneously for the
initial marking of SCMG\textsubscript{i} being with a token on each input place of SCMG\textsubscript{i} and with SCMG\textsubscript{i} interpreted as an ordinary (i.e. not an extended) Petri net.

Furthermore, if identity operators that satisfy the latter condition are removed from 'C-D' recursively (i.e. one 'I' operator is removed and a new CFS-DFS obtained. The condition must be satisfied for this new CFS-DFS to remove another 'I' operator.) then the above conditions are still equivalent.

**Proof:** The theorem is stated in the form \( A \Leftrightarrow B \).

(a) The statement \( A \Rightarrow B \) follows rather trivially by observing that \( /\text{LOCX}_i / \) is a subset of \( /\text{LOCX} / \) and from lemma 4.8.

(b) The statement \( B \Rightarrow A \) will be proved as \( \overline{A} \Rightarrow \overline{B} \). This gets paraphrased as: If \( /\text{LOC}_{A}/ \) and \( /\text{LOC}_{B}/ \) contain tokens simultaneously then there exists a basic SCMG component SCMG\textsubscript{i} which has tokens simultaneously in \( /\text{LOC}_{A}/_i \) and \( /\text{LOC}_{B}/_i \). (The further stipulations of 'for some interpretation' etc. have been omitted for clarity.)

We will prove this by contradiction. i.e. assume that \( \overline{A} \not\Rightarrow \overline{B} \) i.e. assume \( \overline{A} \cdot \overline{B} \). This paraphrases as: assume that \( /\text{LOC}_{A}/ \) and \( /\text{LOC}_{B}/ \) contain tokens simultaneously but there does not exist a basic SCMG component with tokens simultaneously in \( /\text{LOC}_{A}/_i \) and \( /\text{LOC}_{B}/_i \).

By hypothesis there exists a place \( p_A \) in \( /\text{LOC}_{A}/ \) and a place \( p_B \) in \( /\text{LOC}_{B}/ \) which have tokens simultaneously.
From lemma 4.8, if $p_A$ and $p_B$ are both in some basic SCMG component $SCMG_i$, then $p_A$ and $p_B$ have tokens simultaneously in $SCMG_i$ contradicting the hypothesis that such an SCMG component does not exist. Thus there is no basic SCMG component containing both $p_A$ and $p_B$. From lemma 4.7, $p_A$ and $p_B$ must be places corresponding to links in different subschemas of a conditional schema or one is in the feedback schema of a loop schema and the other is outside the loop schema. But for the former condition, $p_A$ and $p_B$ could not contain tokens simultaneously. Thus $p_A(p_B)$ is in the feedback path of a loop schema and $p_B(p_A)$ is outside the loop subschema.

We now consider the cases of identity operator $\ast I$ corresponding to loop or conditional unions. The cases are handled similarly and we will prove the lemma for the case of $\ast I$ operators corresponding to unions at the input of a loop subschema (fig. 4.8).

From the above reasoning, there exist places $p_A$ in /LOCA/ and $p_B$ in /LOCB/ with $p_A(p_B)$ in the feedback path of some loop subschema $\mathcal{L}$ and $p_B(p_A)$ outside this loop subschema.

It is easily shown (by an induction on the number of $\ast I$ operators removed) that /LOCA/* and /LOCB/* are connected.

The cases that arise are illustrated in fig. 4.9. Fig. 4.9 (a) shows the case of the $\ast I$ operator (being considered for removal) being outside the feedback
(i) Loop Schema.

(ii) Corresponding CFS.

(iii) The DFS indicating 'I' operators to be 'removed'.

Fig. 4.8 : removing 'I' operators corresponding to union nodes.
(a) Case with 'I' operator outside the feedback subschema of the loop subschema ' '.

(b) Cases with 'I' operator inside the loop schema ' '.

Fig. 4.9: Removing the 'I' operators corresponding to the input to a loop schema.
subschema of the loop schema ' $\mathcal{X}$ '. Since $\text{/LOCA/}*$ and $\text{/LOCB/}*$ are connected, if $p_B(p_A)$ is within the feedback subschema of $\mathcal{X}$, then $\text{/LOCA/} \ (\ /\text{LOCB/})$ must have a place $p_B'(p_A')$ in the forward subschema of $\mathcal{X}$ as indicated in fig. 4.9(a). But the firing within the loop subschema is totally asynchronous with that outside the loop and thus $p_B'(p_A')$ can have a token simultaneously with $p_A(p_B)$.

Further, there exists a basic SCMG component including $p_A(p_B)$ and $p_B'(p_A')$ (from lemma 4.7) which contradicts the initial hypothesis proving (b) for this case.

The cases in fig. 4.9(b) are for the 'I' operator within the feedback subschema of the loop subschema. For the case in fig. 4.9(b)(i), $p_B(p_A)$ is in the feedback subschema of $\mathcal{X}$ while $p_A(p_B)$ is outside $\mathcal{X}$ with $\text{/LOCA/}*$ ( $\text{/LOCB/}$ ) passing out of $\mathcal{X}$ through the transition corresponding to loop input identity operators which means that an 'I' operator corresponding to the loop input was 'removed' earlier. Thus by previous arguments, there is a place $p_A'(p_B')$ in the forward schema which has a token simultaneously with $p_A(p_B)$ and there is a basic SCMG component containing $p_A(p_B)$ and $p_A'(p_B')$. However, for $p_A'(p_B')$ as indicated in fig. 4.9(b)(i), the loop input identity operator of $\mathcal{X}$ which allowed $\text{/LOCA/}*$ ( $\text{/LOCB/}$ ) to 'pass' outside the loop could not have been eliminated anyway. Thus $p_A$ cannot be outside loop $\mathcal{X}$ and we have a contradiction.

For the case of fig. 4.9(b)(ii), a similar argument
to the preceding shows that the path(s) to \( p_A(p_B) \) from output places of loop \( \mathcal{L} \) cannot pass through transitions corresponding to unions. The correspondence between execution sequences in the CFS-DFS and the RDFS (theorem 3.3) and lemma 2.9 then assert that \( p_A(p_B) \) cannot have tokens when there are tokens in loop schema \( \mathcal{L} \) - a contradiction.

These contradictions establish (b) for the case of 'I' operators corresponding to the inputs of loops. The other cases follow by similar reasoning.

This completes the proof.

We illustrate the application of theorem 4.2 by using it to simplify the DFS of fig. 4.3. The CFS-DFS that results is shown in fig. 4.10 where the conditions of theorem 4.2 allow the removal of all save one identity operator. The figure also indicates a basic SCMG component (in bold lines) which does not satisfy the conditions of the theorem when attempting to remove the identity operator \( 'OP_{22}' \).

Theorem 4.2 simplifies the problem to that of determining if sets of places in a strongly connected marked graph can have tokens simultaneously. This problem is much easier than a similar problem for general or free-choice Petri nets. We show that two places do not have tokens simultaneously in a basic SCMG component (with an initial marking of one token on each input place of the SCMG component) iff there is no SCSM component (section 3.1)
Notation:
indicates places in /loc./
indicates places in /loc./

P_4 and P_7 can have tokens simultaneously.

(a) The CFS.

(b) The DFS.

Fig. 4.10: simplification of the DFS by removing 'I' operators corresponding to unions.
(c) The function $\chi : L \rightarrow P$ is unchanged. i.e. $i \in 1, \ldots, 21$ $(1_i) = p_i$.

(d) The function $\psi : L \rightarrow \text{LOC}$.

<table>
<thead>
<tr>
<th>$l_i$</th>
<th>$l_1, 2, 3, 4, 5, 14, 15, 18, 20$</th>
<th>$l_6, 8, 9, 16$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1_i)$</td>
<td>$\text{loc}_1$</td>
<td>$\text{loc}_6$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$l_i$</th>
<th>$l_7, 10, 11, 13, 17, 19$</th>
<th>$l_12$</th>
<th>$l_12$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1_i)$</td>
<td>$\text{loc}_7$</td>
<td>$\text{loc}_{21}$</td>
<td>$\text{loc}_{12}$</td>
</tr>
</tbody>
</table>

(e) The function $\text{FIRE} : \text{OP} \rightarrow T$

<table>
<thead>
<tr>
<th>$\text{op}_i$</th>
<th>$\text{op}_6$</th>
<th>$\text{op}_7$</th>
<th>$\text{op}_{12}$</th>
<th>$\text{op}_{13}$</th>
<th>$\text{op}_{18}$</th>
<th>$\text{op}_{22}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{FORE(}\text{op}_i\text{)}$</td>
<td>$t_4$</td>
<td>$t_5$</td>
<td>$t_8$</td>
<td>$t_9$</td>
<td>$t_{12}$</td>
<td>$t_{16}$</td>
</tr>
</tbody>
</table>

(f) The function $\alpha : T \rightarrow A$ is unchanged from fig. 3.3(i).

(g) The function $\text{CONTROL} : T \rightarrow \{\text{True, False}\}$ is unchanged from fig. 4.3(g).

Fig. 4.10: (contd.)
of the SCMG component containing both places.

Lemma 4.9: The SCSM components of the basic SCMG components (obtained from the SCMG components as in section 3.1) are cycles.

Proof: Trivially true since in the basic SCMG components, places have single input and single output transitions and in the SCSM components transitions have single input places and unique output places.

Lemma 4.10: the basic SCMG components are of the forms in fig. 4.5(b)(iii),(iv) or 4.4(c)(iii),(iv) (with \( S^*_1 \) and \( S^*_2 \) as acyclic marked graphs) or fig. 4.5(b) (with \( S^*_1 \) as an acyclic marked graph) with \( p_1, \ldots, p_n \) or \( p_{i_1}, \ldots, p_{i_n} \) as the input places.

Proof: follows by an easy induction on the depth of a WFRDFS (similar to lemma 4.5).

Theorem 4.3: consider a basic SCMG component, \( \text{SCMG}_i \) derived from a WFRDFS 'S'. Let \( p_1 \) and \( p_2 \) be any two places in \( \text{SCMG}_i \). Then \( p_1 \) and \( p_2 \) do not have tokens simultaneously for an initial marking with one token on each input place of \( \text{SCMG}_i \) and all firing sequences iff \( p_1 \) and \( p_2 \) are both on some SCSM component of \( \text{SCMG}_i \) (with the SCSM components obtained using Hack's procedure of section 3.1).

Proof:

(a) Suppose \( p_1 \) and \( p_2 \) are both on some SCSM component \( \text{SCSM}_j \) of \( \text{SCMG}_i \). Lemma 4.9 asserts that \( \text{SCSM}_j \) is a cycle and lemma 4.10 says that \( \text{SCMG}_i \) is of one of the forms of
figures 4.4(b)(iii) or (iv) or 4.4(c)(iii) or (iv) (with $S_1'$ and $S_2'$ in these cases being acyclic marked graphs) or fig. 4.5(b) (with $S_1$ as an acyclic marked graph). It follows rather trivially from the acyclic nature of $S_1'$, $S_1'$ and $S_2'$ that $p_1$ and $p_2$ do not have tokens simultaneously.

(b) Suppose $p_1$ and $p_2$ are not on any SCSM component of $SCMG_i$. It easily follows from the strongly connected nature of $SCMG_i$ and the fact that $S_1'$, $S_2'$ and $S_1$ (as above) are acyclic marked graphs, that $p_1$ and $p_2$ are within $S_1'$ or $S_2'$ or $S_1$ and that there is no path in $S_1'$ or $S_2'$ or $S_1$ from $p_1$ to $p_2$ or vice versa (otherwise an SCSM containing $p_1$ and $p_2$ is easily constructed). It then follows that $p_1$ and $p_2$ can contain tokens simultaneously (since $S_1'$, $S_2'$ and $S_1$ are acyclic marked graphs with $p_1, ..., p_n$ or $p_1', ..., p_n'$ as the input places).

Theorem 4.3 provides us with a method of finding if sets of places in basic SCMG components can contain tokens simultaneously. In particular with the terminology of theorem 4.2, with $SCMG_i$ as a basic SCMG component and $/LOCA/_{i}$ and $/LOCB/_{i}$ as defined there, $/LOCA/_{i}$ and $/LOCB/_{i}$ do not contain tokens simultaneously iff $\forall p_{A_j} \in /LOCA/_{i}$ and $p_{B_j} \in /LOCB/_{i}$, $p_{A_j}$ and $p_{B_j}$ are both in some SCSM component of $SCMG_i$.

There is one special case that arises often, which violates the sufficient condition we have used and which still allows the identity operator to be 'removed'. This occurs when $LOCA = LOCB$ (fig. 4.1). This occurs when there
Fig. 4.11: Example illustrating self loops in the DFS.
(d) The DFS that results from removing all the 'I' operators from the DFS in (c).

Fig. 4.11 (contd.)
is a loop constant. An example is in fig. 4.11. Fig. 4.11(c) shows the DFS and indicated in bold lines are two 'I' operators for which $\text{LOCA} = \text{LOCB}$ (with $\text{LOCA}$ and $\text{LOCB}$ as in fig. 4.1). Thus the values in $\text{LOCA}$ and $\text{LOCB}$ are identical when they contain tokens simultaneously (trivially true since $\text{LOCA} = \text{LOCB}$) and the sufficient condition of lemma 4.2 is valid.

**Lemma 4.11**: in removing an identity operator from the DFS if $\text{LOCA} = \text{LOCB}$ (fig. 4.1) then the identity operator can be removed.

**Proof**: The sufficient condition of lemma 4.2 is directly satisfied. 

***
4.2 The reduction of the control flow structure.

We have derived conditions that allow the simplification of the DFS by the elimination of superfluous identity operators. This leads to the presence of transitions in the CFS which do not initiate any data flow operations when they fire. We now state an algorithm that eliminates these transitions from the CFS. We show that the CFS-DFS thus obtained performs the same computation as the source CFS-DFS. Further, it has at least as much parallelism as the source CFS-DFS. We show by an example that there may be an increase in parallelism when the redundant transitions are eliminated as specified.

A transition which does not initiate any data flow operations is eliminated by making its direct successor places, successors of its ancestor transitions. Consider the CFS-DFS of fig. 4.10. Here transitions $t_1, t_2, t_3, t_6$, $t_7$ and $t_{15}$ neither initiate any data flow operation nor are they required for synchronization in the CFS. Thus they may be eliminated from the CFS. In eliminating transition $t_6$, its successor places, $p_8$ and $p_9$ are made successors of its ancestor transition, $t_4$. i.e. arcs are introduced from $t_4$ to $p_8$ and to $p_9$, and $t_6$ and $p_6$ are eliminated from the CFS.

Algorithm 4.2: simplification of the CFS for a CFS-DFS derived from a WFRDFS.

Consider a CFS-DFS obtained by transformation from
a WFRDFS followed by a simplification of the DFS (as in section 4.1). Then the CFS is simplified as follows:

(1) Define the set of 'removable transitions' $T_r$ as

$$T_r = \{ t \in T \mid \text{FIRE}^{-1}(t) = \emptyset \text{ and } \alpha(t) \notin \text{Sw} \}$$

where $T$ is the set of transitions in the CFS, $\text{Sw}$ is the set of 'switch nodes' in the WFRDFS and $\emptyset$ is the null set.

i.e. the 'removable transitions' are those that do not initiate the firing of any operator in the DFS and that do not correspond to switch nodes in the WFRDFS. (Switch nodes are required for routing data correctly.) Thus these transitions can only correspond to wyes and unions.

(1.1) Select $t \in T_r$ s.t. $\alpha(t) \in U$. (i.e. a transition corresponding to a union node.)

Claim: $|t| = |t'|$ * (since the number of arcs incident at different ports of a union node are the same.)

(1.1.1) If $p \in t$, and $p = \chi((\alpha(t))_j^i)$ (i.e. $p$ corresponds to the $j^{th}$ input link at input port 'i' of the corresponding union node) then if $p' = \chi(((\alpha(t))_j^i))$ (i.e. the place corresponding to the $j^{th}$ output link of the same union).

Insert $p'$ into ('p').

Change $\chi$ s.t. if $1 = \chi^{-1}(p)$, then $\chi(1)_{\text{new}} = p'$. 

* We prove this later.
Remove p from the CFS. (i.e. delete p from P and remove all edges in (pxT U Txp) and hence, p from 't.)

Repeat (1.1.1) till no places remain in 't.

(1.1.2) Delete t from the CFS. (i.e. delete all edges in (P x t U t x P) and remove t from T.)

(1.1.3) Repeat 1.1 till there are no t ∈ Tᵣ s.t. α(t) ∈ U.

At this point all transitions corresponding to union nodes that have no corresponding data flow operation are 'removed' from the CFS.

(1.2) Change I to be a relation \( I_{new}^{\infty} (L \times P) \) s.t.

\[
(1, I(1)) \notin I_{new}^{\infty} \quad \text{(We do this because in step(1.2.1) \( I \) is modified so that it no longer is a function.)}
\]

(1.2.1) Select t ∈ Tᵣ s.t. α(t) ∈ W (i.e. a transition corresponding to a wye node.)

Claim: \(|'t'| = 1 \ast\).

Let p = 't.

Insert t' into ('p)' . (i.e. make all the output places of t as successors of all the input transitions of p.)

Change \( I \) s.t. \( \forall 1 \) with \( (1, p) \notin I, \forall p' \in t' \) insert \( (1, p') \) into \( I_{new}^{\infty} \).

Delete p from the CFS.

Delete t from the CFS.

Repeat (1.2) until there are no t ∈ Tᵣ s.t. α(t) ∈ W.

* We prove these later.
Fig. 4.12: Simplification of the CFS.
We illustrate the simplification of the CFS by algorithm 4.2 by applying it to the CFS-DFS in fig. 4.10. Fig. 4.12 illustrates the steps of the algorithm. In fig. 4.12 (a), the transitions corresponding to the union nodes in the WFRDFS are removed. Following this, the transitions \( t_3, t_6 \) and \( t_7 \) that correspond to 'wyes' in the WFRDFS are eliminated. Fig. 4.12 (a) also indicates the subgraphs removed with bold lines. The simple CFS that results is depicted in fig. 4.12(b).

There are several aspects of algorithm 4.2 that need to be proved. First, it was assumed in the algorithm that for transitions 't' corresponding to unions, \(|t'_t| = |t|\) even after removing an arbitrary number of such transitions. Similarly, for transitions, t, corresponding to wyes, that \(|t| = 1\). These assumptions are easily proved by an induction on the number of 'I' operators removed.

We next show that the resulting (modified) CFS is live and safe and that it carries out the same computation as the source CFS-DFS. Finally, we show that the resulting CFS-DFS has at least as much parallelism as the source CFS-DFS. In this context, note that applying algorithm 4.2 removes transitions. Thus there is a distinct possibility of a decrease in the number of execution sequences. However, this should not be viewed as a loss in parallelism because the firing of removed transitions is unnecessary. Rather, we should compare the execution sequences with the removed transitions deleted.
That is, the execution sequences in the source CFS-DFS should be split into equivalence classes such that members in the same class are identical when the removed transitions are deleted. We use the number of equivalence classes thus obtained to compare the parallelism of the source and 'target' CFS-DFS's. We will produce an example that shows that, with this measure of parallelism, there may actually be an increase in the parallelism. This occurs because, when transitions corresponding to unions are removed, the synchronization performed by these unions is eliminated. This will be clarified later by an example.

**Lemma 4.12:** in step 1.1 of algorithm 4.2, with transition $t \in T_r$, the set of removable transitions, and $x(t) \in U$, $|t| = |t'|$.

**Proof:** The lemma follows by an easy induction on the number of transitions eliminated in step 1.1 of algorithm 4.2.

**Basis step:** For no transitions removed it is trivially true (from the transformation rules and since the number of arcs incident on each port of a union node is the same.)

**Induction step:** Eliminating a transition in step 1.1 does not change the number of input or output places of any transition. Thus if the lemma is true for $n$ transitions removed, it is also true for $(n+1)$ transitions removed in step 1.1. ***
Lemma 4.13: In step 1.2 of algorithm 4.2, with transition \( t \in T_r \), the set of removable transitions, and \( \alpha(t) \in W \), \(|t| = 1\).

Proof: The lemma follows by an induction similar to that in lemma 4.12 after noting that step 1.2 does not change the number of input places of a transition. 

Algorithm 4.2 also specifies a change in the function \( \psi: L \rightarrow P \) (from links in the RDFS to places in the CFS). The reason for doing so is to be able to specify the initial configuration in the new CFS-DFS i.e. a configuration that corresponds to one in the RDFS with one token on each input link. We wish to transform a configuration with tokens at the input links of a union in a WFRDFS to one in the new CFS-DFS with a token at each place that now corresponds to the union. Similarly, configurations in the WFRDFS with a token at the input link of a wye transform to configurations in the CFS-DFS with a token on each place that, earlier corresponded to the output places of wyes.

Transforming an initial configuration in a WFRDFS to one in the (new) CFS-DFS: (This is analogous to transforming configurations as defined in section 3.2.)

Let \( \gamma_0 \) be an 'initial configuration' in the WFRDFS 'S' with a token on each input link. Let \( L_I \) denote the set of input links. Then \( \forall l \in L_I \):

(a) \( \forall p \text{ s.t. } (l, p) \in \psi \), let \( M(p) = 1 \).
(b) let \( \text{VALUE}'(\psi(l)) = \text{VALUE}(\gamma_0, 1) \).
∀p s.t. ∃l_I ∈ L_I with (l_I, p) ∈ L, let M(p) = 0.
i.e. mark all places that are \( \cdot \)-related to some input link. Values are inserted in locations that are \( \cdot \)-related to input links.

Lemma 4.14: Consider a CFS-DFS, 'C-D' derived by the transformation of section 3.2 from a WFRDFS 'S' followed by the simplification of the DFS as in section 4.1. Let C'-D' be the CFS-DFS obtained by applying algorithm 4.2 to C-D. Then for an initial configuration \( \Psi_0 \) in the WFRDFS (with a token on each input link of the WFRDFS) let \( \Psi_0' \) and \( \Psi_0'' \) be the corresponding configurations in C-D and C'-D' respectively. Then C'-D' is safe and well behaved for initial configuration \( \Psi_0'' \) and performs the same computation as C-D.

Proof: We prove the lemma by (separate) inductions on the number of transitions removed in steps 1.1 and 1.2 of algorithm 4.2.

(a) Induction on the number of transitions removed in step 1.1.

Basis step: For no transitions removed C-D and C'-D' are identical and they perform the same computation. C-D is safe and well behaved (since the CFS-DFS derived directly from the WFRDFS simulates the WFRDFS (theorem 3.3) and the CFS is unchanged in obtaining C-D). Thus C'-D' is safe and well behaved.

Induction step: Suppose the CFS-DFS C'-D' obtained after removing n or fewer transitions in step 1 of the
algorithm is safe and performs the same computation as C-D. Then consider removing another transition corresponding to a union node. The transition corresponds to one of the three types - $T_1$, $T_2$, or $T_3(T_4)$ in fig. 4.13 (a),(b). Eliminating these transitions results in the CFS-DFS's as indicated in fig. 4.13 (a),(b). Note that $G_1'$ and $G_2'$ in the figure may already have some transitions removed. The arguments are analogous and we illustrate them for the case of removing transition $T_2$ of fig. 4.13 (a)(ii). The input and the output places of $T_2$ cannot hold tokens simultaneously. This is easily shown by the following argument. $G_1'$ is safe and well behaved. Tokens are initially placed at $p_1,\ldots,p_n$. If $T_1$ is not removed then this occurs when $T_1$ fires and tokens appear simultaneously at $p_1,\ldots,p_n$. If $T_1$ was removed earlier, then tokens may not appear at $p_1,\ldots,p_n$ simultaneously. But $G_1'$ is safe and well behaved since it corresponds to a WFRDFS and has at most 'n' transitions removed. Also, the asynchronous appearance of input tokens does not matter since the successor transitions of the input places can fire asynchronously. Thus there is an execution sequence such that these transitions fire exactly as they are allowed by the asynchronous arrival of inputs. (i.e. a successor transition of an input place which does not have a token cannot fire but even if it had its tokens, it need not fire for an arbitrarily long time.) Thus, when tokens are placed at the output
(i) Loop schema  (ii) CFS of (i).  (iii) CFS of (ii)
(a) Removing transitions corresponding to a union of a loop schema.

(b) Removing transitions corresponding to a union of a conditional schema.

(i) Conditional schema  (ii) CFS of (i).  (iii) CFS of (ii) with $T_1$ & $T_2$ removed.

(i) A 'wye' node.  (ii) CFS of (i).  (iii) CFS of (ii) with $T_5$ removed.

(c) Removing transitions corresponding to wyes.

Fig. 4.13: Removing transitions from the CFS.
places of $G_1'$, there are no tokens either at $p_1, \ldots, p_n$
or within $G_1'$. (Tokens cannot reappear from outside the
subgraph because the firing of $G_1'$ is totally asynchrone-
ous and the outside schema has no means of knowing when
$G_1'$ has 'finished'. Safeness requires that the statement
be true.) Now, for an execution sequence in which $T_2$
fires, tokens are placed at the input places of the well
behaved and safe subgraph $G_2'$ which puts out tokens to
the input places of $T_2$ when there are no tokens at its
output places. Thus merging the input and output places
of $T_2$ and removing $T_2$ (as step 1 of the algorithm
accomplishes) results in a safe CFS-DFS. The computation
is unchanged since $T_2$ does not initiate any operation in
the DFS and since (as reasoned earlier) the asynchronous
arrival of tokens does not affect the operation of $G_1'$.

The cases for the removal of transitions of the type
of $T_2$ and $T_3(T_4)$ of fig. 4.13 follow by similar arguments.
Thus the lemma is true for $(n+1)$ transitions removed in
step 1 and the induction is complete.

(b) Induction on the number of transitions removed in
step 1.2.

The induction follows rather trivially by observing,
as in fig. 4.13(c), that the rest of the CFS-DFS is
unaffected by the change.

Lemma 4.15: Consider the CFS-DFS, 'C-D', derived by the
transformation of section 3.2 from a WFRDFS 'S' followed
by the simplification of the DFS as in section 4.1. Let
C'-D' be the CFS-DFS obtained by applying algorithm 4.2 to C-D. Then every execution sequence in C-D with the transitions eliminated (in algorithm 4.2) deleted, is an execution sequence in C'-D'.

Proof: The lemma is easily proved by an induction on the number of transitions eliminated in steps 1.1 and 1.2 of the algorithm 4.2.

(a) Induction on the number of transitions removed in step 1.1.

Basis step: Trivially true since for no transitions removed C-D is the same as C'-D'.

Induction step: Suppose the lemma is true for at most 'n' transitions removed in step 1.1. Then consider removing another transition in step 1.1. The cases are the same as in lemma 4.14 and we again illustrate the argument with the case of removing T_2 in fig. 4.13(a)(ii). Suppose that the transitions eliminated are t_1,...,t_n. Then by the induction hypothesis any execution sequence in C-D with t_1,...,t_n deleted is also an execution sequence in the CFS-DFS with t_1,...,t_n eliminated. Call this CFS-DFS (C-D)*. Thus we need to show that any execution sequence in (C-D)* with T_2 deleted is also an execution sequence in the CFS-DFS, (C-D)**, with t_1,...,t_n and T_2 eliminated. Now any execution sequence in (C-D)* which does not include T_2 is definitely an execution sequence in (C-D)** since the rest of the graph is unchanged. For execution sequences including T_2 in (C-D)*, T_2 is firable when all
its input places are marked. Let $\omega$ be a partial execution sequence in $(C-D)^*$ for which a token is placed on each input place of $T_2$ and such that $T_2$ does not appear in $\omega$. Then $\omega$ is also a valid (partial) execution sequence in $(C-D)^{**}$ since it is a sequence in which $T_2$ does not appear. But for $\omega$ in $(C-D)^{**}$ tokens appear directly on places corresponding to the output places of $T_2$ (fig. 4.13(a)(iii)). This occurs in $(C-D)^*$ after (the firable transition) $T_2$ fires. This argument can be used in an easy induction on the number of occurrences of $T_2$ in an execution sequence to show that all execution sequences in $(C-D)^*$ which include $T_2$ are also execution sequences in $(C-D)^{**}$ when the occurrences of $T_2$ are deleted.

Similar arguments apply for the cases of $T_2$ and $T_3(T_4)$ of fig.4.13.

(b) For transitions removed in step 1.2, the induction follows easily by observing the graphical interpretation (fig. 4.13(c)) of the algebraic statement of step 1.2.

Lemmas 4.14 and 4.15 establish that the application of algorithm 4.2 results in a CFS-DFS that carries out the same computation with at least as much parallelism as the source CFS-DFS. Fig. 4.14 depicts a case when there is actually an increase in the parallelism obtained. In this example, $op_3$ and $op_1$ can fire in parallel in the CFS-DFS, $C'-D'$, of fig. 4.14(d),(e) whereas these operators cannot fire in parallel in the CFS-DFS, $C-D$, of fig. 4.14(b),(c). As proven in lemma 4.15 every execution
(b) The CFS derived from (a)  

(c) DFS derived from (a) with 'I' operators corresponding to wyes and switches removed.

Fig. 4.14: example illustrating an increase in parallelism.
(d) The CFS of (b) simplified by the removal of transitions corresponding to wyes and switches.

(e) The DFS of (c) simplified by the removal of 'I' operators corresponding to unions.

Fig. 4.14 (contd.)
sequence in C-D is also a valid execution sequence in
C'-D'. We now specify an interpretation and an initial
configuration and produce a finite execution sequence
that is a valid execution sequence in C'-D' but not in
C-D.

Let the domain of values $\mathcal{X} = \mathbb{R}$, the field of real
numbers. Let the functions $\text{op}_1^*$, $\text{op}_2^*$, $\text{op}_3^*$, and $D$ be:

$$(\text{op}_1^*)_1 = (\text{op}_1)_1 + (\text{op}_1)_2;$$
$$(\text{op}_1^*)_2 = (\text{op}_1)_1 - (\text{op}_1)_2;$$
$$(\text{op}_2^*)_1 = (\text{op}_2)_1 + (\text{op}_2)_2;$$
$$(\text{op}_2^*)_2 = (\text{op}_2)_1 - (\text{op}_2)_2;$$
$$(\text{op}_3^*) = (\text{op}_3) + 1;$$
$$(D^*) = ((D) \geq 10).$$

The input links are indicated in fig. 4.2.3 and
are designated as links 1, 2, and 3.

Consider the initial configuration $\mathcal{Y}_0$ with
$\text{VALUE}(\mathcal{Y}_0, 1) = 2; \text{VALUE}(\mathcal{Y}_0, 2) = 3; \text{VALUE}(\mathcal{Y}_0, 3) = 4$.
Let the corresponding initial configurations in C-D and
C'-D' be $\mathcal{Y}_0'$ and $\mathcal{Y}_0''$.

The finite execution sequence

$$e'' = t_5 t_4 t_6 t_8 t_4 t_9 t_5 t_6 t_7$$
in C'-D' (with $t_4 (\text{op}_1)$

firing before $t_9 (\text{op}_3)$ ) has no corresponding execution
sequence in C-D. i.e. there is no execution sequence in
C-D which is identical to $e''$ when the removed transitions
$(t_1, t_2, t_3)$ are deleted from the sequence. This is
easily seen from fig. 4.14 (b) which shows that, in C-D,
after $t_8$ fires, $t_9$ must fire before $t_2$ fires. $t_2$, in turn, fires before $t_4$ does. Thus $t_9$ fires before $t_4$ fires again. Thus in C-D the firing order '$t_8 t_4 t_9$' is impossible. Thus we have demonstrated that the application of algorithm 4.2 may result in an increase in parallelism.

We note that the cause of this increase in parallelism is the removal of the synchronization provided by the union nodes. i.e. in a WFRDFS, the presence of union nodes in a loop subschema causes all tokens to be removed from the loop feedback subschema before tokens are placed in the loop forward subschema. Removing the transitions corresponding to unions removes this constraint and allows the simultaneous firing of transitions in the forward and feedback subschemas.
CHAPTER 5
THE IMPLICATIONS OF THE ANALYSIS.

In this chapter we discuss the application of the preceding analysis to existing architectures and we show how it motivates modifications to them. We then discuss the splitting of the data and the control flow in the implementation.

5.1 The implications for extant architectures.

We have analysed the data flow model proposed by Rumbaugh /RUMBAUGH 75/. The results thus carry over directly to the implementation he proposes. The CFS-DFS obtained after the reductions of chapters 4 and 5 may be directly transformed into machine language instructions of Rumbaugh's 'data-flow-machine' (DFM). The removal of identity operators from the DFS during the 'reduction' results in the elimination of copy operations during the execution of the program on the DFM. The reduction of the CFS decreases the number of machine instructions and can increase the parallelism in the program.

We first outline Rumbaugh's implementation (in some more detail than that presented in chapter 1). Figs. 5.1 and 5.2 /RUMBAUGH 75/ are block diagrams of the system. It consists of a 'program memory', a 'scheduler', a 'swap memory', a set of 'activation processors', a 'structure memory' and 'structure controller' and a 'peripheral processor'. The program memory holds a representation of an entire data flow program. The scheduler runs procedures,
Fig. 5.1: Block diagram of Rumbaugh's Data flow machine (adapted from /RUMBAUGH 75/)
Fig. 5.2: an activation processor (adapted from /RUMBAUGH 75/ )
of which programs comprise, on an activation processor. The structure memory and controller are involved with operations on structures. An activation processor consists of an 'enabling count memory', a 'token memory' an 'instruction memory' and an 'execution pipeline' (fig. 5.2). An enabling count is associated with each instruction and represents the number of operands that are yet to arrive before the instruction can be executed. The 'initial enabling count' is the total number of operands. Instructions have five fields as indicated in fig. 4.3. The execution pipeline consists of an 'updater', an 'activity list', a 'decoder', a set of 'functional units' (add, multiply etc.) and modules for structure operations, and procedure calls and returns. The activity list contains pointers to instructions that are ready for execution. The decoder receives these pointers, pulls out the instructions and the operands and ships a 'packet' of this information to the functional units. These functional units perform the necessary operations and ship 'result packets' to the updater. The updater then places the results in the indicated result token memory locations and decrements the enabling counts of successor instructions. If any of these counts goes to zero, all the operands of this instruction have arrived and a pointer to the instruction is placed on the activity list. For structure and procedure instructions, the appropriate units are invoked. However, the operation of these units
is irrelevant to the application of the preceding analysis.

The activation processors also have an activity counter which maintains a count of the number of active instructions in a procedure running on a processor. If this count goes to zero, either when a procedure terminates or while it is waiting for results from an invoked procedure or the structure controller, the procedure is suspended and transferred to a 'swap memory' via a 'swap network' so as to free the activation processor.

We now describe how the reduced CFS-DFS obtained by the methods of the preceding chapter is transformed into instructions of the form of fig. 4.3 for execution on the data flow machine.

Definitions:

The data flow machine (DFM) consists of a set of token memory locations 'TM' and a set of instructions 'I'. It is obtained, from a CFS-DFS in turn derived from a WFRDFS followed by the reduction of chapter 4, as follows:

Let $\alpha'$ be an onto function from transitions $T$ in the CFS to instructions in the DFM. i.e. $\alpha': T \rightarrow I$.

$\alpha'$ is one to one except for transitions that correspond to switches in the source RDFS.

i.e. $\forall t \in T$, if $|('t')'| = 1$ (i.e. not a switch node) then $|\alpha'^{-1}(\alpha'(t))| = 1$ else $\forall t_1, t_2 \in ('t')'$,

$\alpha(t_1) = \alpha(t_2)$ and $|\alpha'^{-1}(\alpha'(t))| = |('t')'|$.

Let $\psi'$ be a bijection from locations LOC in the DFS to token memory locations 'TM' in the DFM.
Interpretations: An interpretation of a DFM is in terms of the interpretation of the source CFS-DFS and includes the same domain $\mathcal{D}$ of values, $\mathcal{B} = \{\text{True, False}\}$ and functions $\phi_f, \pi_p$, and $\beta_b$ and the (reserved) identity function 'I'.

Configurations: A configuration of the DFM for an interpretation with domain $\mathcal{D}$ is:

(i) A function $\text{OPCODE:} I \rightarrow \{\text{F U P U B U } \{I\}\}$, from instructions to function, predicate, Boolean operator letters and the identity operator (indicating the operation to be performed).

(ii) A function $\text{OPERAND:} I \rightarrow (\text{TM})^n$, from instructions to an ordered set of $n$ token memory locations with variable 'n' (indicating the operand locations).

(iii) A function $\text{RESULT:} I \rightarrow (\text{TM})^k$, from instructions to an ordered set of 'k' token memory locations with variable 'k' (indicating the locations results are to be put in).

(iv) A function $\text{SUCCESSOR:} I \rightarrow (I \times \mathcal{H})^m$, where $\mathcal{H} = \{1, 2, \ldots\}$ and $m$ is variable (indicating the successor instructions and the number of operands provided for them).

(v) A function $\text{INITIAL_COUNT:} I \rightarrow \{0, 1, 2, \ldots\}$, from instructions to the set of non-negative integers (specifying the number of operands of an instruction).

(vi) A function $\text{VALUE*:} \text{TM} \rightarrow \mathcal{D} \cup \{\text{True, False, null}\}$, from token memory to the domain of values (indicating values associated with token memory locations).
(vii) A function ENABLING_COUNT : I → \{0, 1, 2, ...\}, \text{from instructions to the non-negative integers (indicating the number of operands that are to arrive before the instruction becomes 'firable').}\n
**Transforming configurations in the CFS-DFS to configurations in the DFM:** Given a configuration in the CFS-DFS, the corresponding configuration in the DFM is obtained by specifying the functions as follows:

(i) \(\forall i \in I, \text{if } |\alpha^{-1}(i)| > 1, \text{ (i.e. 'i'is a switch instruction) then } \text{_OPCODE}(i) = \text{SWITCH}, \text{OPERAND}(i) = \gamma'(\gamma((\gamma^{-1}(t')))^c))\)

i.e. the operand is the (unique) token memory location that corresponds to the location that controls the firing of the set of transitions (section 3.2).

(Note that we are starting from a reduced CFS-DFS derived from a WFRDFS and hence there are no identity operators corresponding to switch nodes.)

RESULT \((i) = \phi\) (since no data flow operations correspond to the switch nodes).

With \(T' = \{t \mid t \in \alpha'^{-1}(i) \text{ and } \text{CONTROL}(t) = \text{True}\}\),

SUCCESSOR \((i) = \{(i', n) \mid t \in T' \text{ and } i' \in \alpha'(t')', \text{ } n=\text{number of places 'p' in t' s.t. } \alpha'(p') = i'\}\)

\(\text{INITIAL_COUNT}(i) = |\gamma'(\alpha'^{-1}(i))|\) (the number of input places of the corresponding transition).

(ii) \(\forall i \in I, \text{if } |\alpha^{-1}(i)| = 1 \text{ and } |\text{FIRE}^{-1}(\alpha'^{-1}(i))| = 1 \text{ (i.e. it does not}
correspond to a 'switch' instruction and corresponds to only one operator in the DFS indicating an operator in the WFRDFS) then

\[ \text{OPCODE}(i) = \text{the function letter associated with} \]
\[ \text{FIRE}^{-1}(\alpha'^{-1}(i)). \]
\[ \text{OPERAND}(i) = \gamma'((\text{FIRE}^{-1}(\alpha'^{-1}(i)))) \]
\[ \text{RESULT}(i) = \gamma'((\text{FIRE}^{-1}(\alpha'^{-1}(i))))' \]

(i.e. the ordered set of token memory locations corresponding to the ordered predecessor and successor locations of the corresponding operator in the DFS.)

\[ \text{SUCCESSOR}(i) = \{ (i',n) | i' \in \alpha''(((\alpha'^{-1}(i)))') \} \]
\[ n = \text{number of places } p \text{ in} \]
\[ (\alpha'^{-1}(i))' \text{ s.t. } \alpha''(p') = i' \]
\[ \text{INITIAL\_COUNT}(i) = \gamma'(\alpha'^{-1}(i)) \]

(as before).

(iii) \( \forall i \in I \) if \( |\alpha'^{-1}(i)| = 1 \) and \( |\text{FIRE}^{-1}(\alpha'^{-1}(i))| > 1 \)

(This occurs when several identity operators correspond to a transition in the CFS and occurs for transitions corresponding to union nodes in the WFRDFS when the 'I' operators mapped from the union node cannot be removed from the DFS.)

If \( |\text{FIRE}^{-1}(\alpha'^{-1}(i))| = n \) (i.e. the number of 'I' operators corresponding to a union after the reduction).

Order the elements of \( \text{FIRE}^{-1}(\alpha'^{-1}(i)) \) from 1 to n and let \( \text{FIRE}^{-1}(\alpha'^{-1}(i))_j \), \( 1 \leq j \leq n \) denote the \( j \)th element. Then

\[ (\text{OPERAND}(i))_j = \gamma'((\text{FIRE}^{-1}(\alpha'^{-1}(i)))_j') \]
\((\text{RESULT}(i))_j = \psi'(((\text{FIRE}^{-1}(\alpha'^{-1}(i)))_j)^*)\)

where \(1 \leq j \leq n\) and \((\text{OPERAND}(i))_j\) and \((\text{RESULT}(i))_j\)
denote the \(j^{th}\) operand and result token memory locations.

\[ \text{OPCODE}(i) = I'_n \] where \(I'_n\) is a new reserved function
letter and \(\phi_{I'_n}\) is a function that has 'n' ordered
operands and 'n' ordered results and which copies the
\(j^{th}\) operand \((1 \leq j \leq n)\) to the \(j^{th}\) result location.

(iv) \(\forall \, t_m \in \text{TM}, \text{VALUE}''(t_m) = \text{VALUE}'(\psi'^{-1}(t_m))\) (i.e. the
value associated with a token memory location in the
DFM is the same as that of the corresponding location in
the DFS).

(v) \(\forall \, i \in I, \text{ENABLING\_COUNT}(i) = \text{number of places 'p'}
in \((\alpha'^{-1}(i))\) with \(M(p) = 1\) (i.e. the number of marked
predecessor places of the corresponding transition).

Notation: Let \(\mathcal{T}''\) be the set of configurations of the
DFM and \(\mathcal{T}'\), the set of configurations of the CFS-DFS.
Denote the above transformation as \(\xi: \mathcal{T}' \rightarrow \mathcal{T}''\).

Firing rules: The DFM progresses through a sequence of
configurations by the 'firing' of instructions that are
'enabled'.

Any instruction with \(\text{ENABLING\_COUNT}(i)=0\) for
configuration \(\chi'' \in \mathcal{T}''\) is 'enabled' for \(\chi''\). Any enabled
instruction \(i \in I\) may fire resulting in a new configu-
ration \(\chi*'\) denoted by \(\chi'' \xrightarrow{i} \chi*'\) and obtained as
follows: \((\text{RESULT}(i)) = \phi_{\text{OPCODE}(i)}(\text{OPERAND}(i))\)
(if \(\text{OPCODE}(i) \neq F\) or replace \(\phi\) by \(\pi\) or \(\beta\) if \(\text{OPCODE}(i)\) is in
P or B).

\[ \forall (j,n) \in \text{SUCCESSOR}(i), \text{ENABLING\_COUNT}(j) | y^* = \text{ENABLING\_COUNT}(j) | y^* - n. \]

(i.e. the enabling count of all successor instructions is decremented.)

**Execution sequences**: Let \( \gamma_0^", \gamma_1^", \gamma_2^", \ldots \) be a sequence of configurations of the DFM for some interpretation. Then the execution sequence \( \omega^" \) is defined by:

\[ \omega^": \eta \rightarrow I \quad \text{with} \quad \eta = \{0, 1, \ldots\} \]

such \( \gamma_0^" \rightarrow \gamma_1^" \rightarrow \gamma_2^" \rightarrow \ldots \rightarrow \gamma_k^" \)

where if \( i_j = x \) then instruction \( x \in I \) is 'enabled' for configuration \( \gamma_j^" \) and \( \gamma_j^" \rightarrow \gamma_{j+1}^" \) is the result of firing 'x'.

**Simulation**: A DFM obtained by the above transformation from a CFS-DFS is said to simulate the CFS-DFS if, for all interpretations and initial configurations \( \gamma_0^\prime \) and \( \gamma_0^" \) in the CFS-DFS and the DFM respectively, there is a one-to-one correspondence between execution sequences in CFS-DFS and the DFM and if \( t_0, t_1, t_2, \ldots \) and \( i_0, i_1, i_2, \ldots \) are corresponding execution sequences then the configuration sequences correspond. i.e. if \( \mathcal{L}' \) and \( \mathcal{L}^" \) are sets of execution sequences in the CFS-DFS and the DFM respectively, then there exists a bijection

\[ \chi': \mathcal{L}' \rightarrow \mathcal{L}^" \quad \text{such that} \]

\[ \forall e', e' \in \mathcal{L}' \quad \text{and} \quad \chi(e') \in \mathcal{L}^" , \]

if \( e' = t_1, t_2, \ldots \) with \( t_j \in T, \)

\( \chi'(e') = i_1, i_2, \ldots \) with \( i_j \in I \)
\[ \forall j = 1, 2, \ldots, \forall n = 1, 2, \ldots \text{ if } \gamma_0^{t_1 \ldots t_n} \rightarrow \gamma_n \]

\[ \gamma_0^{t_1 \ldots t_n} \rightarrow \gamma_n \text{, then } \gamma_n^* = \mathcal{E}(\gamma_n^*) \]

**Theorem 5.1**: Consider a CFS-DFS 'C-D' derived from a WFRDFS and let C'-D' be the CFS-DFS obtained from C-D by the reduction of chapter 4. Then the DFM obtained by the transformation from C'-D' simulates C'-D'.

**Proof**: The theorem follows by analogous arguments to those used in proving theorem 3.3. ***

**Definition**: Consider a DFM derived from a (reduced) CFS-DFS, C-D. The DFM and C-D are said to perform the same computation if for all interpretations and initial configurations, the values in output locations and corresponding token memory locations are identical for all finite execution sequences.

**Corollary**: The DFM and the source CFS-DFS perform the same computation.

**Proof**: Follows directly from theorem 5.1 and the definition of simulation. ***

We illustrate the transformation by applying it to the example in fig. 5.3. Fig. 5.3(a) shows the reduced CFS-DFS of figs 4.10 and 4.12 and fig. 5.3(b) shows the corresponding DFM. Note particularly that when an instruction fires the enabling count of a successor instruction is decremented by the number of predecessor places of the corresponding transition that are marked as a result of the firing.

It should be apparent that the DFM specified is a
(a) The reduced CFS-DFS of figs. 4.10 and 4.12.

<table>
<thead>
<tr>
<th>Location in DFS</th>
<th>loc_1</th>
<th>loc_6</th>
<th>loc_7</th>
<th>loc_21</th>
<th>loc_12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location in TM</td>
<td>tm_1</td>
<td>tm_6</td>
<td>tm_7</td>
<td>tm_21</td>
<td>tm_12</td>
</tr>
</tbody>
</table>

(b) Function \( f \) from locations in the DFS to locations in TM.

Fig. 5.3: transforming a CFS-DFS to a DFM.
<table>
<thead>
<tr>
<th>Transition</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>INSTRUCTION #</strong></td>
<td><strong>INITIAL COUNT</strong></td>
</tr>
<tr>
<td>( t_4 )</td>
<td>( i_p )</td>
</tr>
<tr>
<td>( t_5 )</td>
<td>( i_h )</td>
</tr>
<tr>
<td>( t_8 )</td>
<td>( i_q )</td>
</tr>
<tr>
<td>( t_9 )</td>
<td>( i_{&amp;} )</td>
</tr>
</tbody>
</table>
| \( t_{10}, t_{11} \) | \( i_{sw1} \) | 4 | SWITCH | \( t_{m_6} \) | - | If true then \((i_f, 1)\)  
If false then \((i_{sw2}, 3)\) |
| \( t_{12} \) | \( i_f \) | 1 | \( f \) | \( t_{m_7} \) | \( t_{m_1} \) | \((i_{sw1}, 1)\), \((i_p, 1)\), \((i_h, 1)\) |
| \( t_{13}, t_{14} \) | \( i_{sw2} \) | 3 | SWITCH | \( t_{m_{12}} \) | - | If true then  
If false then \((i_i, 1)\) |
| \( t_{16} \) | \( i_i \) | 1 | \( I_1 \) | \( t_{m_7} \) | \( t_{m_1} \) |

(c) The instructions.

*Fig. 5.3 (contd.)*
formalization of Rumbaugh's implementation. The operations carried out by the execution pipeline is described by the firing rules.

5.2 Splitting the data and the control flow in the implementation.

In chapter 1 we mentioned the advantages of the single graph representation of data and control in data flow programs. This simplifies the writing of parallel programs and makes determinacy easier to guarantee. The question we now address is whether the merging of the data and control flow should be carried over to the implementation of the data flow processor.

The architectures described in section 1.3 vary in the amount of segregation of the data and the control flow. The architectures proposed by Sonnenburg /SONNENBURG 74/, Arvind and Gostelow /ARVIND 77/ and Dennis /eg. DENNIS 77/ have a 'distributed' control. There is specialized control hardware at each 'node' representing an actor of the program that detects the arrival of operands and either carries out the operation on the data and routes results to other 'nodes' or sends information packets of the data, the operation to be performed and the successor nodes to 'functional units'. On the other hand, the architectures of Rumbaugh /RUMBAUGH 75/ and the group in Tolouse, France /eg. SYRE 77/, split data and control by having a separate 'token memory' that holds values associated with tokens in the data flow.
program and a separate instruction memory. However, even in these architectures, decisions of branching in the program are made by the processors.

There are various pros and cons for both types of architectures. For example, having a distributed control requires 'intelligent' nodes and a complex interconnection network. However, this allows a high memory-processor bandwidth. The schemes segregating the data and control use conventional components. However, the memory-processor interface is the bottleneck in the execution pipeline.

We view the developments in this thesis as a point in favour of a segregation of the data and control flow in the implementation. We have seen how this separation can lead to the elimination of data flow operations for most of the control nodes. This saving is significant since control nodes often comprise more than half of a data flow program. Further, simplifying the data flow structure allows a reduction of the control and the parallelism achievable can actually be increased in the process.

It is observed that in the scheme proposed by Rumbaugh, the branching decision at 'switch' instructions is made by the processors. We have shown that for structured programs, the synchronization at unions is unnecessary. This leaves the switch nodes as the only synchronization demarcating several iterations in a loop.
This means that delaying the execution of the switch instruction by routing it to the processors can delay several parallel operations in the next iteration of a loop. This suggests that switch instructions should be given a high priority, possibly by placing them at the head of queues at various points in the execution pipeline (described in section 5.1).

It was also shown that no data flow 'copy' operations are necessary for switch nodes. Thus there is no necessity to route switch instructions to the processors if the branching decision is made by the control. A possible implementation is one that uses the 'storage cum logic arrays' (SLA's) proposed by Patil /PATIL 78/. Patil has also shown how such SLA's can implement an extended Petri net /HARTENSTEIN 75/ and he proposes a very large scale integration (VLSI) implementation of the SLA's.

The SLA is programmable and can be 'loaded' to represent different extended Petri nets. It puts out a signal representing that a transition is firable and this can be used to pull out the appropriate instruction(s) that correspond to the transition. The large number of signals between the instruction memory and the SLA suggests that they be placed on the same chip.

The point we wish to make is that a further splitting of the data and the control flow can lead to a further improvement in performance. Simulation studies are
necessary to assess how much improvement is obtained and at what cost.
CHAPTER 6
SUMMARY AND CONCLUSIONS

6.1 Summary.

In this thesis we have examined the problem of the elimination of data flow operations for the control nodes of a data flow program. The problem is significant since the control nodes often make up more than half of the program.

We specified a data flow model and transformed it to a model with a separate data flow and a control flow structure (CFS-DFS). We showed that the two models are equivalent with respect to the degree of parallelism and the computation performed. We then derived general sufficient conditions for the removal of identity operators (corresponding to the control nodes) in the data flow structure. It was shown that for structured programs identity operators corresponding to wyes and switches can be removed directly if the identity operators resulting from union nodes are retained. Methods for the removal of the identity operators corresponding to union nodes were then developed. The control structure was simplified by removing transitions that do not initiate any data flow operation. It was demonstrated that this can lead to an increase in parallelism. Finally we showed how the simplified CFS-DFS can be transformed into instructions for Rumbaugh's data flow machine /RUMBAUGH 75/. We also indicated how the results motivate a separation of
the data and control flow in implementing a data flow processor.

6.2 Suggestions for further work.

We have analysed the data flow model proposed by Rumbaugh. Similar analyses can be carried out for the other data flow models.

The major results are for structured data flow programs. We mentioned the advantages of structured programs which makes them highly desirable. However, Jotwani's results /JOTWANI 77/ for his model of parallel programs indicate that there is an inherent loss in parallelism due to structuring. Similar studies can be carried out for data flow programs. If the loss in parallelism due to structuring is significant it may motivate the writing of unstructured programs for often used programs and the analysis of this thesis will then need to be extended to unstructured programs.

We have indicated how the results motivate a separation of the data and the control flow in the implementation. We proposed a separate control implemented using SLA's. Simulation studies need to be carried out to assess the speed advantage that accrues to such schemes.

Several aspects of the proposed architectures need further work. Most of the schemes use lisp-like structures. Vector processing can be rather inefficient in this environment. Schemes like the stream processing
proposed by Ahuja /AHUJA 77/ can be incorporated into the architectures to improve vector processing.

Various improvements to schemes using conventional techniques can be explored, simulated and analysed. For instance, in Rumbaugh's architecture, the instruction and token memories can be the bottleneck in the 'execution pipeline'. It is possible to employ cache memories for each of these. An instruction can be placed in the cache when it receives its first operand since it is likely that the other operands will arrive soon. Similarly, tokens produced by instructions can be placed in a cache since they will probably be used by successor instructions in a short time. Other schemes to improve the memory-processor bandwidth for implementations with separate data and control can also be examined.
REFERENCES


