REPRESENTATION OF ALGORITHMS BY MAXIMALLY PARALLEL SCHEMATA

by

John Blake Fosseen

S.B., Yale University (1968)

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE at the MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1972

Signature of Author

Department of Electrical Engineering, May 18, 1972

Certified by ________________________________ Thesis Supervisor

Accepted by ________________________________ Chairman, Departmental Committee on Graduate Students
REPRESENTATION OF ALGORITHMS BY MAXIMALLY PARALLEL SCHEMATA

BY

John Blake Fosseen

Submitted to the Department of Electrical Engineering on May 19, 1972 in partial fulfillment of the requirements for the Degree of Master of Science.

ABSTRACT

In this thesis we define a class of schemata which are called data flow schemata. Data flow schemata are determinate schemata in which the flow of data within the schema determines the eligibility of schema operators to fire. Well formed data flow schemata, a subset of data flow schemata, are defined, and it is shown that given any flowchart schema, it can be transformed into an equivalent well formed data flow schema.

We then show that the equivalence problem for well formed data flow schemata is undecidable. Free well formed data flow schemata are defined, and it is shown that whereas the equivalence problem of primary data arcs in a well formed data flow schema is undecidable, this same equivalence problem is decidable in free well formed data flow schemata.

In conclusion we define what it means for a data flow schema to be in maximum parallel form and show that all data flow schemata are in maximum parallel form.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Electrical Engineering
CONTENTS

CHAPTER I: INTRODUCTION 5

CHAPTER II: DATA FLOW SCHEMATA

Definition of Data Flow Schemata 10
Determinacy of Data Flow Schemata 22

CHAPTER III: WELL FORMED DATA FLOW SCHEMATA

Definition of Well Formed Data Flow Schemata 31
Flowchart Schemata and Well Formed Data Flow Schemata 39

CHAPTER IV: WELL FORMED DATA FLOW SCHEMATA AND THE EQUIVALENCE PROBLEM

Undecidability Results 59
Dadep Graphs and Well Formed Data Flow Schemata 64

3
### CONTENTS (Continued)

**CHAPTER V: FREE WELL FORMED DATA FLOW SCHEMATA**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Equivalence Problem Revisited</td>
<td>76</td>
</tr>
<tr>
<td>Maximally Parallel Data Flow Schemata</td>
<td>95</td>
</tr>
<tr>
<td>Conclusion</td>
<td>97</td>
</tr>
</tbody>
</table>

**BIBLIOGRAPHY**  
99
CHAPTER I

INTRODUCTION

Presently there is a great deal of interest in allowing parallel processing within computer programs so that execution of computer programs could be accomplished in less time than if the computer were to execute the instructions of the program serially. A major goal of parallel processing is to allow parallel processing within an algorithm while maintaining the determinacy of that algorithm. In other words, the results generated by the algorithm should be the same regardless of the order in which the instructions allowed to execute concurrently are actually executed.
Bernstein (2) has shown that the problem of whether two blocks of a serial program may be executed in parallel without destroying the determinacy of the program reduces to the Turing Machine halting problem and is therefore undecidable. As a result the search for methods of transforming programs into maximum parallel form has led to the study of program schemata which operate on unstructured data.

Both Slutz (15) and Keller (11) have studied the problem of representing a form of Karp and Miller's (10) parallel program schemata in maximum parallel form. In parallel program schemata (PPS) there is a data flow graph and a control graph. The data flow graph consists of a set of memory locations and a set of operators such that each of these operators is associated with a set of domain and range locations. The control portion of the PPS controls the order in which the operators in the data flow graph are allowed to execute. It is the control graph which allows concurrent execution of the operators in the data flow graph. Keller and Slutz both define what it means for a PPS to be in maximum parallel form, and Keller shows that his definition of maximum parallel form is equivalent to Slutz's. Keller also shows that there exists a PPS as defined by Karp and Miller for which there is no equivalent PPS which is in maximum parallel form and which has a finite state control.

Rodriguez (14) introduced a graph model for parallel computa-
tion which is quite different in structure from Karp and Miller's model. In Rodriguez's model the data flow and control of the schema are combined into one graph, and in this graph the execution of the operators is governed by control information which flows through the graph along with the data for the operators. This thesis presents a similar form of schema which will be called data flow schemata. As in Rodriguez's model there is no separate data flow graph and control graph, and the flow of data through this model will determine the eligibility of an operator to fire. A data flow schema consists of operator nodes, deciders, and control gates. The operator nodes take data values from their inputs, compute a result and place that result on their output. Deciders, like operators, take data values from their inputs and compute a control value which is either T or F and place that result on their output arc. Control gates use the information computed by the deciders to route the flow of data values. The second chapter of this thesis formally defines the meaning of data flow schemata and proves that all data flow schemata are determinate. A data flow schema is in maximum parallel form in the sense that once all input data for an operator has been computed it can be eligible to fire; however, an operator is often not allowed to fire until certain control information or control values, which are output values of deciders, have been computed. The problem of whether a change in the outcome of a decider would change an operator node's data input values is related to the equivalence problem.
of data flow schemata.

In order to study the equivalence problem of data flow schemata a subset of DFS is defined in the third chapter, and this subset of data flow schemata is termed well formed data flow schemata. Paterson (12) has introduced a form of serial schemata which he calls program schemata. Ashcroft and Manna (1) have developed an algorithm whereby any GO TO program (i.e., a program which contains conditional transfer and assignment statements) in normal form can be transformed into an equivalent program which contains only conditional statements of the form IF boolean expression THEN program block ELSE program block; do while statements of the form DO program block WHILE boolean expression; and assignment statements. Cooper (3) provides an algorithm for transforming an arbitrary flowchart or GO TO program into an equivalent GO TO program in normal form. Both the transformations of Cooper and Ashcroft and Manna can be applied to flowchart schemata by introducing only the boolean logical and, and negation operators. Thus it can be shown that given any program schema S there is an equivalent WF-DFS. Paterson introduced program schemata to prove some undecidability results by simulating a two headed Turing Machine with these schemata. These undecidability results can be shown to apply to arbitrary WF-DFS. Thus a more restricted class of well formed data flow schemata is defined, and this class of WF-DFS is called free WF-DFS. The definition of freedom as it applies to WF-DFS is similar to freedom as it applies to Paterson's
program schemata. However, a free flowchart schema when transformed according to Ashcroft and Manna's algorithm does not always result in a free WF-DFS. Equivalence problems which were undecidable in the general WF-DFS are shown to be decidable in the free WF-DFS. As a result algorithms exist to transform a free WF-DFS into a free productive WF-DFS. Thus it can be decided whether a control value must be computed before the execution of an instance of an operator takes place. The fifth chapter formally defines what it means for a WF-DFS to be in maximum parallel form and shows that a free WF-DFS can be transformed into maximum parallel form. This maximum parallel form is not unique and therefore is not a canonical form for free WF-DFS. The definition of maximum parallel form as expressed in the fifth chapter could be made stronger in the sense that additional requirements could be added for a free WF-DFS to be in maximum parallel form. Further discussion of possible future research in this area is presented in the last chapter.
CHAPTER II

DATA FLOW SCHEMATA

As was mentioned in the previous chapter the data flow schema incorporates the data flow graph and control graph such as is used by Karp and Miller (10) into one graph. The flow of data within this graph determines the flow of control within the data flow schema. Given a data flow schema with property P then one can state that property P holds for every program which that data flow schema represents. This chapter will discuss how a given data flow schema represents a set of programs. Then it will be shown that all data flow schemata represent sets of programs such that any one of the programs in this set is determinate.
In order to formally discuss data flow schemata one must first make some preliminary definitions.

Def: A function symbol set is the set \( F = \{ f_1, f_2, \ldots, f_k \} \).

Def: A decider symbol set is the set \( D = \{ d_1, d_2, \ldots, d_j \} \).

Given a decider symbol set and a function symbol set a data flow schema is a directed graph such that each node is one of nine node types defined below, each directed arc is one of two directed arc types defined below, and arcs are connected to nodes as directed by the connection conventions outlined below. To facilitate definition of connection conventions each node will consist of input and output terminals to which directed arcs are connected according to the connection conventions.

A directed arc in a data flow schema may be either:

(a) A data arc which is denoted by \[ \begin{array}{c}
\text{----} \\
\rightarrow
\end{array} \]

(b) A control arc which is denoted by \[ \begin{array}{c}
\rightarrow \\
\text{----}
\end{array} \]

Given a decider symbol set and a function symbol set nodes which may appear in a data flow schema and the graphical representation of the configuration in which the connection conventions allow them to appear are:

1. Operator Node: An operator node is a node which consists of \( n \) (\( n \geq 1 \)) data input terminals and one data output terminal, and is labeled \( f_i \) where \( f_i \in F \). If \( f_i \) is a label of more than one operator node in a data flow schema, all the nodes which are labeled \( f_i \)
\( f_i \) have the same number of data input terminals.

(2) **Decider Node:** A decider node is a node which consists of \( m \) (\( m \geq 1 \)) data input terminals and one control output terminal, and is labeled \( d_j \) where \( d_j \in D \). If \( d_j \) is a label of more than one decider in a data flow schema, all nodes which are labeled \( d_j \) have the same number of data input terminals.

(3) **True Gate Node:** A true gate node has one data input terminal, one control input terminal, and one data output terminal, and is labeled TG.

**OPERATOR NODE**

**DECIDER NODE**

**TRUE GATE**

**FALSE GATE**
(4) **False Gate Node:** A false gate node has one data input terminal, one control input terminal, one data output terminal, and is labeled FG.

(5) **Merge Node:** A merge node has two data input terminals, one control input terminal, and one data output terminal. The data input terminals of this node type are labeled 'T' and 'F' as shown below, and the node is labeled MERGE.

![Diagram of Merge Node]

(6) **And Node:** An and node has two control input terminals, one control output terminal and is labeled $\land$.

(7) **Negation Node:** A negation node has one control input terminal, one control output terminal, and is labeled $\neg$.

(8) **Data Identity Node:** A data identity node has one data input terminal, two data output terminals, and is labeled I.

(9) **Control Identity Node:** A control identity node has one control input terminal, two control output terminals, and is labeled I'.

Alternative representations for nodes (8) and (9) are shown in
FIG I(a) and FIG I(b) respectively. Whether the node is a control identity node or a data identity node will be determined by the context in which the node appears.

The following are the connection conventions for the data and control arcs within a data flow schema.

I.) Given any data arc in an arbitrary DFS the following must be true:

(a) The head of the data arc (→) is either unconnected
or is connected to a data input terminal of a DFS node.

(a)  

(b)  

FIG I

(b) The tail of the data arc is unconnected or is connected to a data output terminal of a DFS node.

II.) Given any control arc in an arbitrary DFS both the head and tail of this arc are connected to DFS node terminals such that:

(a) The head of the control arc is connected to a control input terminal.

(b) The tail of the control arc is connected to a control output terminal.

III.) Data arcs whose tails are not connected to data output terminals of nodes in a data flow schema are that schema's input arcs, and data arcs whose heads are not connected to data input terminals of nodes in a data flow schema are that schema's out-
put arcs. A data flow schema's data input arcs will be labeled $X_1, X_2, \ldots, X_i$, and its data output arcs will be labeled $Y_1, Y_2, \ldots, Y_j$.

Note that a data arc in a data flow schema may be both an input and an output arc of that schema.

Def: A data flow schema is (1) a set of data flow schemata nodes connected by data and control arcs in accordance with the data flow schema connection conventions such that every input and output terminal of every DFS node has an arc connected to it, (2) a specification of initial values on the control arcs of the nodes in the DFS, (3) a set of input arcs labeled $X_1, X_2, \ldots, X_i$, and (4) a set of output arcs labeled $Y_1, Y_2, \ldots, Y_j$.

FIG II is an example of a very simple data flow schema where an initial value of $T$ is specified to be on the control arc between $d_1$ and the true gate in $S$. If there is no specification of initial values for a control arc of a data flow schema, the arc is
assumed to be empty. This completes the definition of data flow schemata. A data flow schema represents the structure of a set of programs. An interpretation of a data flow schema is a mapping from the data flow schema to a particular program or procedure.

Execution of the represented program can be simulated by "execution" of the appropriate interpreted schema. Prior to demonstrating how one performs the "execution" of an interpreted data flow schema it is necessary to define more formally the meaning of an interpretation of a data flow schema.

Def: An interpretation $I$ of data flow schema $S$ consists of two mappings $M_1$ and $M_2$ where $M_1$ and $M_2$ are defined as follows:

(a) $M_1$ maps each $f_i \in F$, where each operator node in $S$ labeled $f_i$ has $n$ input terminals, onto a total $n$-ary function whose domain and range are the integers.

(b) $M_2$ maps each $d_i \in D$, where each decider node in $S$ labeled $d_i$ has $m$ input terminals, onto a total $m$-ary function whose domain is the integers and whose range is the set of one lettered strings $\{T, F\}^*$. The notations $f_i^I$ and $d_i^I$ will represent uninterpreted operators and deciders, and $F_i^I$ and $D_i^I$ will represent the corresponding interpreted functions under interpretation $I$.

In a data flow schema the data and control arcs represent first in first out queues of data and control information respectively. The operator nodes represent uninterpreted operators which for a given interpretation of a schema represent the functions or operators appearing in the program represented by the interpreted
data flow schema. Decider nodes, along with true gates, false gates, and merge nodes, allow data flow schemata to represent computations in which the flow of computation is data dependent. The true gates, false gates, and merge nodes use control information, such as the output of decider nodes, to control the flow of data from the input queues of these gates to their output queues.

Execution of data flow schema $S$ under interpretation $I$ is analogous to execution of the procedure represented by schema $S$ with interpretation $I$. Given a data flow schema $S$, interpretation $I$, and an initial configuration of data values on the input queues of $S$, execution of $S$ with interpretation $I$ is performed according to the following conventions:

(a) An **operator node** is eligible to fire if there is at least one data item in each of its data input queues. When an operator node $f_i$ fires, it removes data items $X_1, X_2, \ldots, X_n$ from each of its $n$ input queues and places the value $F^I_i(X_1, X_2, \ldots, X_n)$ in its output queue.

(b) A **decider node** is eligible to fire if there is at least one data item in each of its data input queues. When a decider node fires, data items $X_1, X_2, \ldots, X_m$ are removed from each of $d_1$'s input queues, and $D^I_1(X_1, X_2, \ldots, X_m)$ is placed in its control output queue.

(c) A **true gate** is eligible to fire when at least one data item is in its data input queue and one control item is in its control input queue. When a true gate fires, it removes
the data value $X_1$ from its input queue and the control value $C_1$
from its input queue and

(i) If $C_1 = T$, then $X_1$ is placed in the true gate's
output queue.

(ii) If $C_1 = F$ no further action takes place.

(d) A false gate is eligible to fire when at least one data
item is in its data input queue and one control item is in its con-
trol input queue. When a false gate fires, it removes the data
value $X_1$ from its input queue and the control value $C_1$ from its
control input queue, and

(i) If $C_1 = F$, $X_1$ is placed in the false gate's
data output queue.

(ii) If $C_1 = T$, no further action takes place.

(e) A merge node is eligible to fire either when the top
item in its control input queue equals $T$ and its data input queue
labeled $T$ contains at least one data item or when the top item in
its control input queue equals $F$ and its data input queue labeled
$F$ contains at least one data item. When the merge node fires
and the control item at the top of its control queue equals $T$,
a data item is removed from the data input queue labeled $T$ and is
placed in the data output queue, and the control item is removed from
the control input queue. When the merge node fires and the control
item at the top of its control input queue equals $F$, a data item
is removed from the data input queue labeled $F$ and is placed in the
data output queue, and the control item is removed from the control
input queue.

(f) The **and node** is eligible to fire when there is at least one control value in each of its input queues. When it does fire it performs the logical and operation on the two control values it removes from its input queues and places the result in its output queue. FIG III(a) shows the results of applying the logical and node to its inputs.

<table>
<thead>
<tr>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$C_1$</th>
<th>$C_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>

FIG III

(g) The **negation node** can fire when there is at least one control value in its control input queue. When it does fire, it
removes a control value from its input queue, applies the logical
negation operator, and places the result in its control output
queue. FIG III(b) shows the result of applying the negation node
to its input queue.

(h) The data identity node is eligible to fire when at least
one data value is present on its input queue. A data identity node
fires by removing a data value from its input queue and placing that
same value in both of its output queues.

(i) The control identity node is analogous to the data identity
node except that when it fires, it removes a control value from
its input queue and places that control value in both its output
queues.

One then simulates execution of a schema S with interpretation
I, or as we shall refer to from now on as program (S, I), by
performing the following procedure:
I.) Look for an operator or decider node in S which is eligible
to fire; if there is no such node, the execution has terminated.
Otherwise:

(a) Fire one of the operator or decider nodes which are
eligible to fire and

(b) Fire in any order merge nodes, true gates, false gates,
and identity nodes which are eligible to fire until there are no
more such nodes eligible to fire. Note that this operation is per-
sistent because one cannot make a node ineligible to fire once it
is eligible to fire.
II.) Repeat step I until execution of the data flow schema terminates (i.e. there is no operator or decider node eligible to fire).

DETERMINACY OF DATA FLOW SCHEMATA:

A data flow schema, S, can be considered as a black box with data input arcs and data output arcs (FIG IV). Upon choosing an interpretation, I, for S and a set of initial values on the data input queues, one can simulate execution of the interpreted schema to produce a set of data values on S's data output queues. FIG V(a) shows an interpreted schema prior to execution and FIG V(b) shows the schema after execution has terminated. Patil ( ) refers to the set of values on the input and output queues as arrays.

If data flow schemata are to be useful, they must be determinate or it must be decidable whether a given data flow schema is determinate. Intuitively a schema, S is determinate iff given any interpretation, I, of S execution of the interpreted schema (S; I)
with a given input array will always produce the same output array. Thus any procedure which a determinate data flow schema represents will always produce the same results if it is given the same array of input values.

This paper will use Patil's (13) definition of a determinate system and the closure property of determinate systems to prove that all data flow schemata are determinate. First an introduction to the work of Patil is necessary to facilitate the understanding of the proof of the determinacy of data flow schemata.

One can picture an input array to a data flow schema, S, as shown in FIG VI. The first in first out queues are shown as

![Diagram](a) and (b)

FIG V

horizontal pipes of data. Input data values flow in the left hand of the pipe and out the right hand end. Given an array such as the one shown in FIG VI one can define a time slice as a vertical line which cuts each sequence of the input or output array once and only
once. Thus $T(0)$ is a time slice for the array shown in FIG VI.

![Diagram of array]

FIG VI

Each time slice, $T$, of an array denotes a prefix of that array.

A prefix of an array is that portion of the array which appears to the right of the time slice, $T$. Thus the prefix defined by time slice $T(i)$ in FIG VI is shown in FIG VII. An array $A'$ is a prefix of array $A$ if and only if there exists a time slice $T$ of $A$ which denotes the array $A'$. The prefix of $A$ denoted by time slice $T$ will be referred to as $A^T$. Patil introduces the relation $\leq$ which
holds between \( A^{T_1} \) and \( A^{T_j} \) iff \( A^{T_1} \) is a prefix of \( A^{T_j} \). A sequence of time slices \( T(0), T(1), \ldots, T(k) \) of array \( A \) where:

(a) \( A^{T(0)} = \) the null array

(b) \( A^{T(k)} = A \).

(c) \( T(i) \) precedes \( T(j) \) \( \forall i \forall j \) such that \( i \leq j \leq k \).

defines a history of array \( A \). A time slice \( T(i) \) precedes time slice \( T(j) \) if \( A^{T(i)} \leq A^{T(j)} \). Notice that for any given input array there are a number of possible histories for that array. It is a property of a determinate system that the output array of the system will be the same for a given input array irregardless of the history of the input array.

Def: A system is determinate iff it satisfies both a consistency and dependency requirement.

Given that for a system \( S \) the interpretation \( I \) maps the set of input arrays into a set of output arrays \( (I: X \rightarrow Y) \) the two requirements for determinacy of system \( S \) are defined as follows:

Def: \( I \) satisfies the consistency requirement iff for any \( X_1 \in X \) and \( X_2 \in X \) such that \( X_1 \leq X_2 \), then \( I(X_1) \leq I(X_2) \).

Def: \( I \) satisfies the dependency requirement iff for \( \sigma^\omega = T(0), T(1), \ldots, T(n) \), a sequence of time slices of the history of \( T(n) \) the output \( Y \), and for any slice \( T(j) \) in the sequence, let \( T(i) \) be the earliest \( T \) such that \( Y = Y \) and let \( T(i-1) \) be the immediate predecessor of \( T(i) \). Then \( Y^{T(i)} \leq I(X^{T(i-1)}) \).

The consistency requirement guarantees that the system is function-
al since \( x_1 = x_2 \Rightarrow x_1 \leq x_2 \) and \( x_2 \leq x_1 \Rightarrow I(x_1) \leq I(x_2) \)
and \( I(x_2) \leq I(x_1) \Rightarrow I(x_1) = I(x_2) \). The dependency requirement insures that a system will not produce the output array corresponding to a given input array until the input array has been absorbed by the system. A determinate system, \( S \), can neither predict a future input nor can it act instantaneously. Patil proves the following theorem about the interconnection of determinate systems.

**Theorem 2-1:** Any interconnection of a finite number of determinate systems is also a determinate system. /

It is now necessary to distinguish between the concept of a determinate system as defined by Patil and the concept of a determinate data flow schema.

**Def:** A **determinate data flow schema** is a data flow schema \( S \) such that given any interpretation \( I \) of \( S \) the corresponding interpreted schema, \( (S, I) \), is a determinate system. /

**Lemma 2-1:** Given a data flow schema \( S \) and an interpretation \( I \) of \( S \) there exists a set of single noded schemata \( S_1, S_2, \ldots, S_n \) a set of corresponding interpretations \( I_1, I_2, \ldots, I_n \) and an interconnection of the interpreted schemata \( (S_1, I_1) \) such that the resulting interpreted DFS is equivalent to \( (S, I) \) /

The proof of this lemma is quite clear, for given the DFS \( S \) with interpretation \( I \) consider the data flow schema \( S \) as an inter-
connection of single noded schemata $S_1$, $S_2$, ..., $S_n$ where $S_1$, $S_2$, ..., $S_n$ are the individual nodes of $S$. If $S_i$ is either a true gate, false gate, merge node or a boolean operator, the corresponding $I_i$ is the empty relation. If $S_i$ is an operator node labeled $f_j$, then $I_i$ is the relation $\left\{ (f_j, F^I_j) \right\}$, and if $S_i$ is a decider node labeled $d_j$, then $I_i$ is $\left\{ (d_j, D^I_j) \right\}$.

To prove the determinacy of an arbitrary data flow schema it is only necessary to show that any single noded schema under any interpretation forms a determinate system.

Lemma 2-2: Given any single noded schema $S$ and any interpretation $I$ of that schema, the interpreted schema $(S, I)$ is a determinate system. /

To prove Lemma 2-2 it is necessary to show that any single noded schema with interpretation $I$ satisfies the consistency and dependency requirement. It is really only necessary to outline the proof for an operator node and a merge gate node because the proof for the decider node follows that of the operator node as does the proof of determinacy of the other control gates follow that of the merge gate node.

I.) Given an operator node $f_i$ with interpretation $\left\{ (f_i, F^I_i) \right\}$ and with inputs $X_1$, $X_2$, ..., $X_n$ one has the interpreted data flow schema shown in FIG IX(a). FIG IX(b) shows an arbitrary input array with a corresponding output array for the operator $f_i$ with the interpretation $\left\{ (f_i, F^I_i) \right\}$. The nature of the transition rule of the operator node insures that the consistency require-
ment is satisfied for any interpretation of an operator node. In addition since the operator node is not eligible to fire until it receives a volley of input values on its input queues, the operator node schema satisfies the dependency requirement under all interpretations. The decider node's operation is the same as the operator node except that the range of the interpreted decider is limited to the strings T and F. Thus any interpreted schema consisting of one decider is a determinate system.

II.) In FIG IX(a) there is a diagram of a schema consisting of

![Diagram](image)

\[
\begin{array}{ccc}
X_{31} & X_{21} & X_{11} \\
X_{32} & X_{22} & X_{12} \\
\vdots & \vdots & \vdots \\
X_{3n} & X_{2n} & X_{1n}
\end{array}
\]

\[\ldots F_1^I(X_2), F_1^I(X_1)\]

(a) (b)

FIG IX

one merge gate node, and FIG IX(b) shows an example of an input array to the merge gate with the corresponding output array.

The action of the merge gate is similar to the action of the operator node with a fixed interpretation with one exception. This exception is that the action of the gate is data dependent (i.e. the data value is removed from \(X_1\) or \(X_2\) depending on the value at the
top of the control queue $X_3$). Despite this difference the transformation rule of the merge gate node guarantees that this node will meet the consistency requirement. The same reasoning used to demonstrate the fulfillment of the dependency requirement by the operator node applies to the merge gate's fulfillment of the dependency requirement. The argument remains the same even if there are initial control values on the control input queue.

**Theorem 2-2:** All data flow schemata are determinate. 

From the definition of determinate schemata it is known that a DFS $S$ is determinate iff given $S$ and any interpretation $I$ of $S$ the interpreted data flow schema $(S, I)$ is a determinate system. Given an arbitrary interpreted DFS $(S, I)$ Lemma 2-1 allows one to treat
(S, I) as an interconnection of interpreted single noded schemata 
\[ \left\{ (S_1, I_1), (S_2, I_2), \ldots, (S_n, I_n) \right\} \]. By Lemma 2-2 each of these single noded schemata is a determinate system, and by Theorem 2-1 the interconnection of these determinate systems to form (S, I) guarantees that (S, I) is a determinate system.
CHAPTER III

WELL FORMED DATA FLOW SCHEMATA

Since our main purpose is to develop determinate schemata in maximum parallel form and to provide insight into the equivalence problem for data flow schemata, a more limited class of data flow schemata is defined. This more limited class of schemata will be called well formed data flow schemata (WF-DFS). It will then be shown that well formed data flow schemata can represent Algol like programs which contain conditional transfer statements and assignment statements through the use of Ashcroft and Manna's (1) algorithm for transforming a flowchart program into a program which contains only conditional statements and loop statements.

31
Well-formed data flow schemata are defined recursively as follows:

Def: A data flow schema $S$ is a \textit{well-formed data flow schema} iff:

(a) $S$ is a data flow schema which contains only operator nodes, or

(b) $S$ is a \textit{conditional data flow schema} such that the data flow schemata $S_1$ and $S_2$ which form the body of the conditional schema are well formed data flow schemata, or

(c) $S$ is a \textit{loop data flow schema} such that the schema $S_1$ which is the body of the loop schema is a well-formed data flow schema, or

(d) $S$ is an interconnection of well-formed data flow schemata. 

It now remains to define the meaning of loop and conditional data flow schemata. First we provide some insight into the reasons why the loop and conditional schema were chosen as a method of representing certain classes of computation. In general an Algol Program $P$ with input variables $X_1, X_2, \ldots, X_m$ and output variables $Y_1, Y_2, \ldots, Y_n$ is represented by data flow schema $S$ if there is an interpretation $I$ of $S$ such that the interpreted
schema \((S, I)\) with input queues \(X_1, X_2, \ldots, X_m\) and output queues \(Y_1, Y_2, \ldots, Y_n\) is equivalent to program \(P\) in the sense that if one feeds the same values of input variables to both \(P\) and \((S, I)\) both the Algol program and the interpreted data flow schema will always return the same values for the output variables. The structure of the conditional data flow schema is such that it can represent Algol conditional statements of the form IF boolean expression \(\text{THEN } P_1 \text{ ELSE } P_2;\) where \(P_1\) and \(P_2\) are Algol program blocks which can be represented by well formed data flow schemata. This leads one to the following definition:

Def: A conditional data flow schema consists of a set of true gates, a set of false gates, a set of merge nodes, a set of deciders, a set of boolean operators, and two data flow schemata \(S_1\) and \(S_2\) which form the body of the conditional data flow schema. These gates, boolean operators, deciders, and the two schemata \(S_1\) and \(S_2\) must be connected as shown in FIG X. The input queues to \(S_1\) and \(S_2\) are each connected via true and false gates respectively to a subset of the input queues to the conditional data flow schema.

In the data flow schema in FIG X under interpretation \(I\) if the control value at the top of the control arc or queue which is connected to the deciders of \(S\) equals \(T\), then \((S_1, I)\) is executed, and its output values are placed in the output queues \(Y_1, Y_2, \ldots, Y_n\); otherwise \((S_2, I)\) is executed, and its output values are placed in output queues \(Y_1, Y_2, \ldots, Y_n\).

The structure of the loop data flow schema is such that it
acyclic schema of deciders & boolean operators
can represent Algol loop statements of the form \( \text{DO } P_1 \text{ WHILE Boolean expression; where } P_1 \text{ is an Algol program block which can be represented by a well formed data flow schema.} \)

Def: A loop data flow schema is a set of true gates, false gates, merge nodes, deciders, boolean operators, initial values in control arcs, and a data flow schema \( S_1 \) such that the gates, deciders, and the data flow schema \( S_1 \) are connected as shown in FIG XI, and the specification of initial control values must be as shown in this same figure. The loop data flow schema will cause data flow schema \( S_1 \) to be executed and to return its output values to its own input queues as long as the control value in the control queue at \( A \) is \( T \). When the control value at \( A \) is \( F \), the output from the repeated application of \( S_1 \) is placed in output queues \( Y_1, Y_2, \ldots, Y_m \).

Def: A decider schema is an acyclic interconnection of deciders and boolean operators such that the resulting schema has an arbitrary number of data input queues and one control output queue. / An example of an Algol conditional statement represented by a conditional data flow schema with interpretation I is shown in FIG XII(a) with its corresponding Algol conditional statement in FIG XII(b). Likewise FIG XIII(a) shows an interpreted loop data flow schema, and an equivalent Algol program consisting of one DO WHILE statement is shown in FIG XIII(b). One should note that a conditional data flow schema or a loop data flow schema is not necessarily a well formed data flow schema because even though a
acyclic schema of deciders and boolean operators
Interpretation I

\[ f_1 := F_1(Z_1, Z_2) \]
\[ f_2 := F_2(Z_1) \]
\[ d_1 := D_1(Z_1) \]

(a)

IF \( D_1(X_2) \) THEN \( Y_1 := F_1(X_1, X_2) \) ELSE \( Y_1 := F_2(X_2) \);  

(b)

FIG XII

37
Interpretation I

\[ f_1 := F_1(Z_1, Z_2) \]
\[ d_1 := D_1(Z_1) \]

(a)

DO \[ X_2 := F_1(X_1, X_2) \] WHILE \[ D_1(X_2) \];

(b)

FIG XIII

38
schema may have the structure of a conditional or loop data flow schema, the subschemata may very well not be well formed data flow schemata.

FLOWCHART SCHEMATICA AND WF-DFS:

Well formed data flow schemata form quite a powerful class of schemata, for it can be shown that the WF-DFS can model a wide range of algorithms. Ashcroft and Manna ( ) present a class of programs which they call flowchart programs, which can be represented by flowchart schemata as introduced by Luckham and Park ( ). Flowchart programs can, as Ashcroft and Manna state, be considered as flowchart schemata if the tests and operators are uninterpreted. A flowchart schema consists of four types of nodes which are connected by arcs which indicate flow of control within the schema. The four types of nodes and their corresponding configurations are:

(a) **Assignment Node**

\[ L_i := f_j(L_{k_1}, L_{k_2}, \ldots, L_{k_n}) \]

(b) **Test Node**

\[ d_i(L_i) \]
(c) **Start Node**

```
START (L)
```

(continued)

(d) **Halt Node**

```
HALT (L)
```

where $L_1, L_2, \ldots, L_m$ are memory locations in the schema, $f_i$ is an uninterpreted operator which operates on its domain locations and places the result in its range location, and $d_j$ is an uninterpreted unary operator whose outcome is either $T$ or $F$. Each flowchart schema begins with a unique start node and ends with a unique halt node. One can think of the start node as placing a token on its output arc. Both the assignment node and test node are eligible to fire or execute when a token is present on their input arc. When the assignment node is finished placing its result in location $L_i$, it places a token on its output arc. If the firing of a test node results in $d_j(L_j)$ being true, then a token is placed on the arc labeled $T$; however if $d_j(L_j)$ is false, a token is placed in the arc labeled $F$. When a token is present on the halt node's input arc, the schema is finished executing and the vector $L$ of memory locations contains the results of the interpreted flowchart schema.

Since both the assignment and test nodes preserve the number
of tokens in the flow chart schema and since the start node places only one token on its output arc, there is always one and only one locus of activity in a flowchart schema, and therefore parallel processing cannot be represented by such a schema.

Ashcroft and Manna introduce a class of programs which they refer to as while programs. While programs are Algol like programs which contain only the following three types of statements:

(a) Assignment statements of the form

\[ L_{k+1} := F_i(L_{k_1}, L_{k_2}, \ldots, L_{k_n}); \]

(b) Conditional statements of the form

\[ \text{IF } D_i(L_j) \text{ THEN } P_1; \text{ ELSE } P_2 ; \]

(c) Do while statements of the form

\[ \text{DO } P_1 \text{ WHILE } D_i(L_j); \]

where \( P_1 \) and \( P_2 \) are blocks of while programs. The major difference between while programs and flowchart programs is that in the flowchart program one is allowed to use an explicit GO TO statement.

Ashcroft and Manna present two algorithms which transform arbitrary flowchart programs into equivalent while programs. This paper will describe one of these algorithms and will show that this algorithm can be applied to the transformation of flowchart schemata into equivalent while schemata and that the while schemata can be transformed into equivalent WF-DFS.

Ashcroft and Manna require that their flowchart schema be in normal form as defined by Cooper and Engeler. Cooper (3)
shows that any flowchart schema can be transformed into an equivalent flowchart schema in normal form without altering the semantics of the flowchart schema. A flowchart schema is in normal form if it is of the form:

![Diagram of normal form flowchart schema]

where a block is defined recursively as follows:

I.) Any tree like, loop free, one entrance piece of flowchart schema is a block.

II.) If

![Diagram of recursive block definition]

and

![Diagram of recursive block definition]
are blocks, then

is also a block.
This is construction by composition.

III.) If

\[ B' \]

is a block, so is

\[ B' \]

This is construction by looping.

FIG XIV shows the form of the transformation Cooper uses to transform an arbitrary flowchart schema into normal form.

For ease of notation in representing the transformation from the normal form flowchart schema to the while program schema Ashcroft and Manna treat \( l_1, l_2, \ldots, l_n \) as a vector of memory locations and thus represent \( l_{k_{j+1}} := f_1(l_{k_1}, l_{k_2}, \ldots, l_{k_j}) \) by \( \bar{x} := f_1(\bar{x}) \). The same
notation will be used here while keeping in mind that each $f_1$
has a one location range.

Ashcroft and Manna describe the transformation from flow-chart schema to while schema by associating with each block a
while schema, $\sigma_B(\overline{x}, \overline{y})$, and a set of ordered pairs consisting
of an exit condition and an exit term $<\psi_1(\overline{x}, \overline{y}), \psi_1(\overline{x})>$,
$<\psi_2(\overline{x}, \overline{y}), \psi_2(\overline{x})>$, ..., $<\psi_n(\overline{x}, \overline{y}), \psi_n(\overline{x})>$ such that
each ordered pair is associated with an exit arc of the block and
the set of exit conditions is complete and mutually exclusive. The schema
shown in FIG XV(a) is then equivalent to the flowchart schema with

![Diagram]

FIG XIV

45
associated while schema and exit conditions shown in FIG XV(b).
\[ \alpha_B(\overline{x}, \overline{y}) \]

IF \( \phi_1(\overline{x}, \overline{y}) \) THEN \( \gamma_1(\overline{x}) \) ELSE
\[
[ \text{IF } \phi_2(\overline{x}, \overline{y}) \text{ THEN } \gamma_2(\overline{x}) \text{ ELSE }
\]
\[
\vdots
\]
\[
[ \text{IF } \phi_m(\overline{x}, \overline{y}) \text{ THEN } \gamma_m(\overline{x}) \text{ ELSE } ] ] \ldots ] ;
\]

(a)

(b)

FIG XV

46
The only difference between the flowchart program and the flowchart schema discussed here is that in the flowchart schema the operators and deciders are uninterpreted operators and deciders. The vector \( \bar{y} \) in FIG XV is a set of additional memory locations used to store information needed to determine future exit conditions. Ashcroft and Mann prove that one cannot transform arbitrary flowchart schemata into equivalent while schemata without the use of additional storage to store either past values of program variables or the values of certain boolean variables.

The while schema and the set of exit conditions and terms associated with a block are constructed as follows:

I.) If \( B \) is a tree-like, loop free, one entrance flowchart schema and \( \alpha_B(\bar{x}) \) is null, \( \Theta_1(\bar{x}) \) is the condition which will cause the block given a token on its input arc to execute so that a token is placed on output arc 1, and \( \gamma_1(\bar{x}) \) is the resulting transformation of \( \bar{x} \) where \( \bar{x} \) is the block's input vector. Therefore \( \gamma_1(\bar{x}) \) is the functional composition of operations on \( \bar{x} \) under condition \( \Theta_1(\bar{x}) \). For example FIG XVI shows the exit terms and conditions associated with a tree-like, loop free, one entrance flowchart schema.

II.) When a block is constructed by composition, one can consider two separate cases.

(a) In the first case \( B(\bar{x}) \) is constructed from \( B' \) and \( B'' \) where \( B' \) and \( B'' \) are of the form shown in FIG XVII(a) and \( B \) is
FIG XVI

formed by composition as shown in FIG XVII(b). The new exit terms
and conditions are also shown in FIG XVII(b).

(b) If $\alpha_B^n(x) \neq \text{null but equals } \alpha_B^n(x, y')$, then one
constructs $\alpha_B(x, y)$ and the exit conditions and terms of $B$ as
follows. Let $\circ$ by the operator which adjoins two vectors of memory
locations. If $\bar{x}_0$ is a subsector of $\bar{x}$ which along with $\bar{y}'$ determines
the exit conditions from $B'(\bar{x}, \bar{y}')$, then let $\bar{y} = \bar{y}' \circ \bar{y}'' \circ \bar{y}_0$ and the
newly formed block with exit conditions and terms is shown in FIG XXIX.

III.) If the block is formed by looping, then the resulting
$\alpha_B(x, y)$ and exit conditions and terms are calculated as follows.
If $B'$ is of the form as shown in FIG XIX(a), then $B$ is of the form
\[ \alpha_{B'}(\bar{x}, \bar{y}') \]

\[ \alpha_{B''}(\bar{x}) = \text{null} \]

(a)

\[ < \phi_1(\bar{x}, \bar{y}'), \gamma_1(\bar{x}) > \]
\[ < \phi_n(\bar{x}, \bar{y}'), \gamma_n(\bar{x}) > \]
\[ < \psi_1(\bar{x}), \delta_1(\bar{x}) > \]
\[ < \psi_m(\bar{x}), \delta_m(\bar{x}) > \]

(b)

\[ < \phi_1(\bar{x}, \bar{y}'), \psi_1(\gamma_1(\bar{x})), \delta_1(\gamma_1(\bar{x})) > \]
\[ < \phi_n(\bar{x}, \bar{y}'), \psi_n(\gamma_n(\bar{x})), \delta_n(\gamma_n(\bar{x})) > \]

FIG XVII

49
where $\alpha_B(x, y)$ is

$$
\begin{cases}
\alpha_B((x, y'), i), \\
\bar{y}_0 := x_0; \\
\text{if } \varphi_1(x, y') \text{ then } [x := \nu_1(x); \\
\alpha_{B''}(x, y'')] 
\end{cases}
$$

FIG XIX

50
\[ B'(x) \]

\[ \beta(x, y) \]

\[ < \phi_1(x, y), \gamma_1(x) > \]

\[ < \phi_n(x, y), \gamma_n(x) > \]

\[ \ldots \]

(a)

\[ B(x) \]

\[ \overline{B(x)} \]

\[ B'(x) \]

\[ 1 \quad 2 \quad \ldots \]

\[ < \phi_2(x, y), \gamma_2(x) > \]

\[ < \phi_n(x, y), \gamma_n(x) > \]

where \( \alpha_B(x, y) \) is \( \alpha_B(x, y) \);

while \( \phi_1(x, y) \) do

\[ \overline{x} := \gamma_1(x); \alpha_B(x, y) \]

(b)

FIG XIX

51
shown in FIG XIX(b) where \( \varphi_2, \varphi_3, \ldots, \varphi_m \) are complete and mutually exclusive and \( \varphi_j(x, y') \) implies \( \varphi_j(x, y), \forall j \), \( 2 \leq j \leq m \).

Thus any flowchart schema can be transformed into an equivalent while schema. Notice that the result of this transformation is an uninterpreted while schema which contains assignment statements and conditional and looping statements which contain only uninterpreted operators, the boolean logical and operator, and the boolean negation operator.

It is quite straightforward to show that any while schema produced by the transformation outlined by Ashcroft and Manna can be transformed into an equivalent well formed data flow schema. Since it was assumed that all operators appearing in the flowchart schema were of the form \( L_k_{j+1} := f_1(L_{k1}, L_{k2}, \ldots, L_{kj}) \), and the transformation from flowchart schema to while schema introduced only the boolean operations \( \wedge \) and \( \neg \), while schemata consist only of the following types of statements:

(a) \( L_{k_{j+1}} := f_1(L_{k_1}, L_{k_2}, \ldots, L_{kj}) \);

(b) \( \text{IF } B_1 \text{ THEN } P_1 \text{ ELSE } P_2; \)

(c) \( \text{DO } P_1 \text{ WHILE } B_1; \)

where \( B_1 \) is a boolean expression of uninterpreted boolean operators and the boolean operators \( \wedge \) and \( \neg \). \( P_1 \) and \( P_2 \) are while schemata blocks. The assignment statement (a) is represented by a j-ary operator node labeled \( r_1 \). The conditional statement (b) is rep-
represented by a conditional data flow schema whose subschemata $S_1$ and $S_2$ represent while schemata blocks $P_1$ and $P_2$, and the loop statement (c) is represented by a loop data flow schema whose subschema $S_1$ represents while schema block $P_1$.

A formal method for constructing a WF-DFS which is equivalent to a while schema is accomplished recursively as follows. Given a while schema $W_0$ for each memory location in $W_0$ construct a WF-DFS with a data arc for each of these locations, and label each data arc with its corresponding memory location name (FIG XX). Call this WF-DFS $S_0$. For the remainder of this algorithm a WF-DFS will will include the labels of the data arcs. Now consider the recursive

\[
\begin{array}{ccccccc}
  & L_1 & & L_2 & & L_3 & \cdots & L_n \\
\downarrow & & & & & & & \downarrow \end{array}
\]

FIG XX

algorithm CONSTRUCT $(S, W)$ where $S$ is a WF-DFS and $W$ is a while schema. When one calls the algorithm CONSTRUCT $(S_0, W_0)$, it returns a WF-DFS which is equivalent to while schema $W_0$. CONSTRUCT is defined as follows:

I.) If the first statement of $W$ is an assignment statement of the form $L_{k_{j+1}} := f_i(L_{k_1}, L_{k_2}, \ldots, L_{k_j})$, locate data arcs labeled $L_{k_1}, L_{k_2}, \ldots, L_{k_j}$ in $S$, add $f_i$ to $S$ using as inputs to $f_i$ the arcs with the labels corresponding to its input locations. Locate the
data arc in S labeled $L_{k+j+1}$, erase its label, and label the data output arc of $f_{1}L_{k+j+1}$. Call the resulting WF-DFS $S'$. Remove the first statement of W and call the resulting while schema $W'$. RETURN CONSTRUCT $(S', W')$.

II.) If the first statement of $W$ is a conditional statement with subschemata $W_1$ and $W_2$, locate all labeled data arcs in S and connect them through true gates and false gates as shown in FIG XXI. Label the data arcs emanating from the true gates and false gates with the corresponding labels of their data input arcs.

![Diagram](image)

FIG XXI

Call the resulting data flow schema $S_1$. Consider the output arcs of the true gates as WF-DFS $S_2$ and append CONSTRUCT $(S_2, W_1)$ to $S_1$. In a similar manner consider the output arcs of the false gates as WF-DFS $S_3$ and append CONSTRUCT $(S_3, W_2)$ to $S_1$. The DFS
constructed by appending data flow schemata \textsc{construct} \((S_2, W_1)\)
and \textsc{construct} \((S_3, W_2)\) will be called \(S_4\) and will be of the form
shown in FIG XXII. Connect identically labeled outputs of the DFS
resulting from the applications of \textsc{construct} to merge gates and
label the data arcs emanating from these merge gates with the label
of their data input arcs. Erase all previous labels and call
this WF-DFS \(S_5\). Remove the first statement, the conditional state-
ment, from \(W\) to form \(W_3\). RETURN \textsc{construct} \((S_5, W_3)\).

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure}
\caption{FIG XXII}
\end{figure}

II.) If the first statement of \(W\) is a loop statement with sub-
schema \(W_1\), locate all labeled arcs in \(S\), and connect them to
merge gates as shown in FIG XXIII. Connect the outputs of the
merge gates to true gates and false gates and label the data arcs
as shown in FIG XXIII. Call the DFS resulting from adding these
gates to $S_1$. Consider the data output arcs of the true gates

FIG XXIII

as WF-DFS $S_2$ and append $\text{CONSTRUCT } (S_2, w_1)$ to $S_1$ as shown in

56
FIG XXIII. Connect the data output arcs of CONSTRUCT \((S_2, W_1)\)
to the merge gates in \(S_1\) as indicated by the dotted lines in
FIG XXIII. The resulting DFS is \(S_3\). Erase all data arc labels in
\(S_3\) except those labels of the data output arcs of the false gates.
Remove the first statement, the loop statement, from \(W\) to form
\(W_2\) and RETURN CONSTRUCT \((S_3, W_2)\).

IV.) If \(W\) is the empty schema (i.e. contains no statements)
RETURN \(S\).

No mention was made in any of the above steps of the treatment
of boolean expressions when transforming from while schemata to
WF-DFS. When constructing DFS from loop statements and conditional
statements the procedure is the same. The transformation from
flowchart schema to equivalent while schema introduces only the
boolean operators \(\land\) and \(\neg\). Therefore when a loop or con-
ditional statement is encountered in steps II or III and that
statement contains a boolean expression \(B\), construct an equivalent
decider schema from deciders, logical and nodes, and negation
nodes. Connect the control output arc of this decider schema
to the merge gates, true gates, and false gates of the DFS being
formed. Also include in the definition of the loop DFS an
initial \(F\) token on the control input arcs of the merge gates. For
example a while schema boolean expression and its equivalent
decider schema is shown in FIG XXIV. This completes the descrip-
tion of the transformation from an arbitrary flowchart schema to an
equivalent WF-DFS.

\[ \neg (d_1(x_1) \land (d_{15}(x_4) \land d_4(x_1, x_2))) \]
CHAPTER IV

WELL FORMED DATA FLOW SCHEMATA AND THE EQUIVALENCE PROBLEM

Paterson (12) has introduced a class of schemata which he calls program schemata. A program schema consists of a finite series of program schemata statements of the following two types:

(a) \( q: L_{k+1} := f_u(L_{k_1}, L_{k_2}, \ldots, L_{k_j}) \);

(b) \( r: d_u(L_j) \ s, t \)

where \( q, r, s, \) and \( t \) are instruction labels, the \( L \)'s are memory location or variable names, \( f_u \) is an uninterpreted \( j \)-ary operator and \( d_u \) is an uninterpreted test function. Statement (a) is the traditional assignment statement, and the second statement is a conditional transfer statement where control is transferred to
the statement labeled s if $d_u(L_j)$ is true and is transferred to
the statement labeled t if $d_u(L_j)$ is false. Otherwise execution
of the statements in a program schema is serial.

Since any program schema as defined by Paterson is equi-
valent to a flowchart schema and since one can transform any flow-
chart schema into normal form by using the method outlined by
Cooper and since one can use the algorithm described in the pre-
vious chapter to transform an arbitrary flowchart schema in
normal form into an equivalent WF-DFS, one can transform an
arbitrary program schema into an equivalent WF-DFS. This result
will be used to show that the undecidability results which apply
to Paterson's program schemata also apply to WF-DFS.

A decision problem enquires as to the truth or falsity of
each of a whole class of statements. A problem is recursively
undecidable if there is no algorithm to apply to the solution of
that problem such that it will give the correct answer in every
case. Thus a decision problem is recursively decidable if there
exists an algorithm which when given an arbitrary member of the
class of statements of the problem can tell whether that member
is true or false, and that answer is always correct. A decision
problem is partially decidable if there exists an algorithm which
when given any member of the decision class which is true, will answer
true, but upon being given any member of the decision class which
is false will fail to terminate.

Let us now formally define the meaning of Paterson program
schemata.

Def: A **program schema** \( S \) consists of the following:

(a) \( F = \{ f_1, f_2, \ldots, f_m \} \) a set of function symbols.

(b) \( D = \{ d_1, d_2, \ldots, d_n \} \) a set of decider symbols.

(c) \( L = \{ L_1, L_2, \ldots, L_q \} \) a set of memory locations.

(d) A finite sequence of statements of the type defined previously such that the last statement is either a conditional transfer statement or a statement with the right or left terminal address as a statement label.

(e) An initial address.

(f) A right and a left terminal address.

Another definition for the program schema includes the following as part of the definition:

(g) \( R = \{ L_{k_1}, L_{k_2}, \ldots, L_{k_j} \} \) a set of output memory locations.

Whether (g) is part of the definition of program schemata will be made clear by the context in which it appears.

Def: Two program schemata \( S_1 \) and \( S_2 \) are **R-equivalent** iff given any interpretation \( I \) and any initial values in the memory locations of \( S_1 \) and \( S_2 \), either (1) both \( S_1 \) and \( S_2 \) diverge or (2) both converge with the same values in the memory locations of \( R \).

\( S_1 \equiv_R S_2 \) will be used to denote the relation defined by R-equivalence.

Def: Paterson defines two program schemata \( S_1 \) and \( S_2 \) to be **strongly equivalent** iff for all initial values in the set of locations \( L \) and for all interpretations \( I \) either neither inter-
preted program schema converges or they both converge and \( \text{Val}(S^I_1) = \text{Val}(S^I_2) \). \\

\( \text{Val}(S^I) = 0 \) if \( S^I \) halts on the left terminal address, and \( \text{Val}(S^I) = 1 \) if \( S^I \) halts on the right terminal address; otherwise \( S^I \) is said to diverge.

It can be shown that given program schemata \( S_1 \) and \( S_2 \) one can construct program schemata \( S'_1 \) and \( S'_2 \) such that \( S_1 = S_2 \) iff \( S'_1 \overset{R}{=} S'_2 \), where \( = \) is used to denote the relation of strong equivalence.

To prove this assume that the union of the function symbol sets of \( S_1 \) and \( S_2 \) is \( f_1, f_2, \ldots, f_m \) and the union of the memory location sets of \( S_1 \) and \( S_2 \) is \( L_1, L_2, \ldots, L_q \). Then add the following statements to both program schemata \( S_1 \) and \( S_2 \):

(a) \( e_0 : L_{q+1} := f_{m+1}(L_{q+1}) \)

(b) \( e_1 : L_{q+1} := f_{m+2}(L_{q+1}) \)

In these statements \( e_0 \) is the left terminal address of program schemata \( S'_1 \) and \( S'_2 \), and \( e_1 \) is the right terminal address of these program schemata. Call these new program schemata, \( S'_1 \) and \( S'_2 \) and define \( R \) for both these schemata to be \( L_{q+1} \). These new program schemata satisfy the condition that \( S_1 = S_2 \) iff \( S'_1 \overset{R}{=} S'_2 \).

The proof of the following lemma follows immediately from these results.

**Lemma 4-1:** The strong equivalence problem for program schemata is recursively decidable if the \( R \)-equivalence problem for program schemata is recursively decidable. \\

Paterson proves the following theorem.
Theorem 4-1: The strong equivalence problem for program schemata is recursively undecidable. /

From Lemma 4-1 and this theorem one can derive the following corollary:

Corollary 4-1-1: The R-equivalence problem for program schemata is recursively undecidable. /

Def: An interpreted well formed data flow schema, \((S, I)\), with an array of input values converges iff when one simulates execution of \((S, I)\) with that array of input values, the simulation terminates (i.e. there are no more decider or operator nodes eligible to fire). /

Def: Two well formed data flow schemata, \(S_1\) and \(S_2\), are R-equivalent iff given any interpretation, \(I\), of \(S_1\) and \(S_2\) and the same array of values on \(S_1\)'s and \(S_2\)'s input arcs either (1) both \(S_1\) and \(S_2\) do not converge, or (2) both \(S_1\) and \(S_2\) converge with the same array of values in their output arcs. /

The following decidability theorem about WF-DFS then follows from Theorem 4-1 and its corollary.

Theorem 4-2: The R-equivalence problem for WF-DFS is recursively undecidable. /

Assume that the R-equivalence problem for WF-DFS is recursively decidable. Then given any two program schemata \(S_1\) and \(S_2\) use Cooper's and Aschcroft and Manna's transformations to transform \(S_1\) and \(S_2\) into equivalent while schemata \(S'_1\) and \(S'_2\). Then transform \(S'_1\) and \(S'_2\) into equivalent WF-DFS \(S''_1\) and \(S''_2\). These transformations can be performed with any program schema. Thus the R-equivalence problem for program schemata is recursively decidable because
\( S_1^R = S_2^R \) iff \( S_1'^R = S_2'^R \). This is a contradiction of Corollary 4-1-1.

In a WF-DFS consider the output arcs of operator nodes as primary data arcs. Thus in FIG XXV the data arcs labeled with a "p" are primary data arcs.

Def: An empty path in a well formed data flow schema \( S \) is a path of data arcs such that none of the nodes in the path are operator nodes. /

One can easily visualize the nature of the execution of a WF-DFS
by constructing data dependence (dadep) graphs of the WF-DFS.
The construction of a dadep graph is accomplished as follows:
I.) If a WF-DFS S contains only operator and identity nodes,
construct the dadep graph D as follows:

For each primary data arc in S and input arc to S draw a
cell in D. Label each of the cells which represent S's data input
arcs with the same label as the label of S's corresponding data
input arc. For each operator node, with function symbol label \( f_1 \),
in S draw the same operator node in D, and construct an arc from
\( f_1 \) in D to the cell in D which represents \( f_1 \)'s data output arc
in S. For each data input arc of \( f_1 \) in S follow a path back from
this data input arc until the first primary data arc is encountered.
In dadep graph D construct an arc from the cell which represents
this primary data arc to the corresponding input of \( f_1 \). FIG
XXVI shows a WF-DFS which contains only operator nodes and its

![Diagram](image-url)
corresponding dadep graph.

II.) If a WF-DFS $S$ is a conditional data flow schema with subschemata $S_1$ and $S_2$, construct the dadep graph for this DFS as follows:

First construct cells which represent the data input arcs of the conditional data flow schema. Pick a set of outcomes of the deciders and record these in the dadep graph by drawing an arc from the cells which represent the data input arcs to the deciders to the diamonds which contain a record of the outcome of the deciders they represent. Then depending upon whether the outcome of the decider schema of the conditional data flow schema is $T$ or $F$ construct the dadep graph of $S_1$ or $S_2$ treating the cells already drawn as input cells to the newly constructed dadep graph. FIG XXVII is an example of a conditional data flow schema with its associated dadep graphs.

III.) If a WF-DFS is a loop data flow schema with subschema $S_1$, construct its corresponding dadep graphs as follows:

Add on a dadep graph of $S_1$ for each outcome of the decider schema of the loop DFS which is $T$, and record the outcome of the decider schema as was done for the conditional DFS. A loop DFS and its corresponding dadep graphs are shown in FIG XXIX.

IV.) If a WF-DFS is none of the above, decompose $S$ into subschemata of the above three types, construct the dadep graphs of these subschemata, and then connect them together as the subschemata of $S$ are connected together.
V.) While performing steps I-IV record the sequence of control values absorbed by the merge gates of each loop and conditional subschema during the construction of dadep graph D. For each loop subschema this sequence includes the initial F's on the control arcs which are connected to the merge nodes of the loop subschema and does not include the F's remaining on these same control arcs after the execution of S has terminated. Thus there will be associated with the merge nodes of a subschema a string which contains T's and F's. Once steps I-IV have been completed, locate in the resulting dadep graph, D, all cells which are not input cells to any operator nodes in D. Call these cells output cells of D. For each output arc of S perform the following algorithm:

(a) Assume the output arc is labeled $Y_i$, and let $p_j$ point to this arc. Go to step (b).

(b) In S follow a path of data arcs back from the data arc to which $p_j$ points until one encounters a merge node or a data arc which corresponds to an output cell of D. Go to (c).

(c) If one encounters a merge node, M, and if the last control value in the sequence of control values associated with the merge nodes of M's subschema is T, remove T from this sequence, set $p_j$ to point to the data arc which is connected to M's data input terminal labeled T, and go to step (b). If one encounters a merge node, M, and if the last control value in the sequence of control values associated with the merge nodes of the subschema
containing M is F, remove F from this sequence, set p_j to point to the data arc which is connected to M's data input terminal labeled F, and go to step (b). If one encounters a data arc which corresponds to an output cell of D, label that output cell Y_i and halt. /

Note that in performing this algorithm one reinitializes the sequences of control values for each data output arc of S.

It is important to note that not all dadep graphs correspond to execution sequences of a WF-DFS. In a given dadep graph such as shown in FIG XXIX the same decider may have operated on the same input in two different instances, where in one instance it has an outcome T and in the other an outcome of F. Such a dadep graph is said to be self-inconsistent and does not correspond to an execution sequence because no interpretation of d is exists such that both d(x) = T and d(x) = F.

Def: Two dadep graphs D_1 and D_2 are functionally equivalent iff:

(a) The set of D_1's output cell labels equals the set of D_2's output cell labels, and

(b) Given in D_1 and D_2 any pair of output cells labeled Y_i, the dadep graph of D_1 and the dadep graph of D_2 which define a cascade composition of functions for these cells define the same cascade composition of functions. /

Def: A dadep graph D is self-consistent if there is no pair of deciders in D with the same label such that the dadep graph defining the input to one is functionally equivalent to that de-
fining the input to the other, and the outcome of the decider in the one case is T and in the other case is F. /

Otherwise it is self-consistent.

Def: Two dadep graphs $D_1$ and $D_2$ are mutually consistent if there is no pair of deciders one of which is in $D_1$, and one of which is in $D_2$, and which have the same label such that the dadep graph of the input to one is functionally equivalent to the dadep graph of the input to the other, the outcome of one of the decider is T, and the outcome of the other is F. /

---

FIG XXIX

70
Paterson defines a program schema to be free if every path through the program schema represents a possible execution sequence. A WF-DFS \( S \) is said to be free if every dadep graph of the schema is self-consistent (i.e. does not contain any conflicting decisions). Thus for every dadep graph \( D \) of a free WF-DFS \( S \) there is an interpretation \( I \) of \( S \) such that \( D \) represents an execution sequence of \((S, I)\). Paterson has defined what it means for two program schemata to be strongly equivalent, and it was previously shown that the strong equivalence problem for program schemata is decidable if the \( R \)-equivalence problem for WF-DFS is decidable. Paterson defines two program schemata \( P_1 \) and \( P_2 \) to be weakly equivalent iff, for all interpretations \( \text{Val}(P_1^I) = \text{Val}(P_2^I) \) when \( P_1^I \) and \( P_2^I \) both converge. We now define the meaning of weak \( R \)-equivalence of WF-DFS, and this definition is analogous to Paterson's definition of weak equivalence.

Def: Two WF-DFS \( S_1 \) and \( S_2 \) are weakly \( R \)-equivalent iff given any interpretation \( I \) of \( S_1 \) and \( S_2 \) and the same array of values on \( S_1 \)'s and \( S_2 \)'s input arcs such that \((S_1, I)\) and \((S_2, I)\) converge, then \((S_1, I)\) and \((S_2, I)\) converge with the same array of values in their output arcs. 

The following lemma can be proven:

Lemma 4-2: Two WF-DFS \( S_1 \) and \( S_2 \) are weakly \( R \)-equivalent iff given any dadep graph \( D_1 \) of \( S_1 \) and any dadep graph \( D_2 \) of \( S_2 \) such that \( D_1 \) and \( D_2 \) are each self-consistent and \( D_1 \) and \( D_2 \) are mutually consistent, then \( D_1 \) and \( D_2 \) are functionally equivalent.
I.) First it is shown that R-equivalence of WF-DFS $S_1$ and $S_2$ guarantees functional equivalence of dadep graphs which are mutually consistent and are each self-consistent. Given $S_1$ and $S_2$ choose as initial data values on the data input arcs to $S_1$ and $S_2$ the strings which are the respective labels of these data input arcs. Choose an interpretation, $I$, of the operator nodes of $S_1$ and $S_2$ such that each node $f_i$ forms a string which represents the functional composition of its input strings with the string "$f_i". Thus the output value of $f_i$ is string representing the composition of functions which compute that value. Given any two dadep graphs $D_1$ and $D_2$ of $S_1$ and $S_2$ such that these two dadep graphs are mutually consistent and are each self-consistent, choose an interpretation of the deciders of $S_1$ and $S_2$ such that the outcome of the deciders is the same as that in the dadep graphs $D_1$ and $D_2$. Then since $S_1$ and $S_2$ are weakly R-equivalent, the strings which represent the functional composition of $S_1$'s and $S_2$'s input values will be equal; therefore $D_1$ and $D_2$ must be functionally equivalent.

II.) To prove that weak R-equivalence is a necessary condition for functional equivalence of dadep graphs which are mutually consistent and are each self-consistent, given an interpretation $I$ of $S_1$ and $S_2$ such that $(S_1, I)$ and $(S_2, I)$ converge, construct dadep graphs in which the deciders have the same outcomes as they would have if $(S_1, I)$ and $(S_2, I)$ were being executed. Since the two dadep graphs are mutually consistent and are each self-consistent, they must be
functionally equivalent. Thus the output values of \((S_1, I)\) and 
\((S_2, I)\) must be the same for any interpretation \(I\).

Using Lemma 4-2 one can treat the problem of weak \(R\)-equivalence of 
WF-DFS as a problem of whether two WF-DFS always define pairs of 
mutually consistent and self-consistent dadep graphs which are 
functionally equivalent.

**Def:** A \(p\)-subschema of a WF-DFS \(S\) where \(p\) is a primary data arc 
in \(S\) consists of (1) all those nodes \(N\) in \(S\) such that node \(n \in N\) 
iff there is a directed path from \(n\) to \(p\), (2) all the data and control 
arc which lie on the directed paths from nodes in \(N\) to \(p\), and 
(3) all data input arcs to \(S\) which are connected to nodes in \(N\).

The data output arc of a \(p\)-subschema is \(p\).

**Def:** Two primary data arcs \(p_1\) and \(p_2\) in a WF-DFS \(S\) are said 
to be equivalent if the \(p_1\)-subschema of \(S\) is \(R\)-equivalent to 
the \(p_2\)-subschema of \(S\).

**Def:** Two primary data arcs \(p_1\) and \(p_2\) in a WF-DFS \(S\) are said 
to be weakly equivalent if the \(p_1\)-subschema of \(S\) is weakly \(R\)- 
equivalent to the \(p_2\)-subschema of \(S\).

From Paterson's results and from Theorem 4-2 one can conclude the 
following theorem.

**Theorem 4-3:** It is recursively undecidable whether two primary 
data arcs in an arbitrary WF-DFS are equivalent.

The proof is by contradiction. Assume that it is recursively 
decidable whether two primary data arcs are equivalent. Then 
given two arbitrary WF-DFS \(S_1\) and \(S_2\) as shown in FIG XXX(a) one
FIG XXX

74
can transform them into a WF-DFS as shown in FIG XXX(b). It only remains to apply the appropriate algorithm to determine whether $Y_1$ and $Y_1'$, $Y_2$ and $Y_2'$, ..., $Y_n$ and $Y_n'$ are equivalent. If any pair of data arcs is not a pair of equivalent data arcs, then $S_1$ and $S_2$ are not R-equivalent; otherwise, $S_1$ and $S_2$ are equivalent.

Since it is not recursively decidable whether two WF-DFS are R-equivalent, then this is a contradiction. Therefore it is recursively undecidable whether two primary data arcs of a WF-DFS are equivalent.
CHAPTER V

FREE MAXIMALLY PARALLEL WELL-FORMED DATA FLOW SCHEMATA

In this chapter it will be shown that the problem of deciding whether two data arcs in a free well formed data flow schema are equivalent is decidable. Then maximally parallel WF-DFS will be discussed, and it will be shown that a free well formed data flow schema can be transformed into maximum parallel form.

First consider the following definitions:

Def: a free well formed data flow schema $S$ is a WF-DFS such that each dadep graph of $S$ is self consistent.

Thus in a free well formed data flow schema for every dadep graph
D of that schema there exists an interpretation I of that schema such that D represents an execution sequence of the data flow schema under interpretation I.

Just as in dealing with other types of schemata, it is necessary to use the concept of an execution sequence. However, the form of the data flow schema allows one to have more than one operator or decider node with the same function or decider symbol. It therefore becomes necessary to label operator nodes and decider nodes with distinct labels. The operator nodes will be
labeled \( a_i \) where \( i \) denotes a unique instance of an operator mode; likewise \( b_j \) will denote a unique instance of a decider mode. Therefore \( a_i \) and \( a_j \) will be different operator modes iff \( i \neq j \), but \( a_i \) and \( a_j \) may well have the same function symbol label even though \( i \neq j \). The \( a \)'s and \( b \)'s will be referred to respectively as operator mode labels and decider mode labels.

Def: An execution sequence of a WF-DFS \( S \) is a string \( Q \) of operator mode labels and decider mode labels with superscripts denoting the outcome of the deciders such that there exists an interpretation \( I \) of \( S \) which results in a simulation of the execution of procedure \( (S, I) \) where the interpreted operators and deciders are fired in the same order as the corresponding operator mode and decider mode labels appear in \( Q \), the outcomes of the fired deciders correspond to the outcomes of their corresponding decider mode labels in \( Q \), and after the decider mode or operator mode corresponding to the last mode label in \( Q \) has fired, \( (S, I) \) halts (i.e. there are no more decider or operator modes eligible to fire).

Fig XXXI is an example of a WF-DFS with corresponding execution sequences.

In order to deal with the equivalence problem of primary data arcs in a free WF-DFS we first make the following definition and prove the following lemma.

Def: An empty path in a WF-DFS is a directed path of data arcs which contains no operator modes.

Lemma 5-1: A p-subschema of a WF-DFS will define daedep graphs.
such that the function symbol label of the operator node whose output arc is connected to the dadep graph's cell which corresponds to p-subschema's data arc p is always \( f_1 \) iff given any operator node \( a_j \) such that there is an empty path from \( a_j \) to \( p \), then \( a_j \) is labeled with the function symbol label \( f_1 \).

Suppose there is an empty path from an operator node with function symbol label \( f_j \) where \( i \neq j \) to \( p \). Choose the outcome of the deciders in this WF-DFS such that this operator node with function symbol label \( f_j \) appears in a dadep graph of the p-subschema such that its output arc is connected to the cell which represents data arc \( p \). If all the empty paths to data arc \( p \) emanate from operator modes with function symbol label \( f_1 \), then each dadep graph of the p-subschema will contain an operator node which is labeled with function symbol label \( f_1 \) and whose output arc is connected to the cell which corresponds to data arc \( p \).

We introduce the following transformation to show that any free WF-DFS may be transformed into an equivalent (R-equivalent) free WF-DFS such that the equivalence of primary data arcs in the transformed schema is decidable.

**Transformation T:**

Perform the following transformation for each data output arc of a true gate or false gate of a WF-DFS's loop subschema until there are no more output arcs eligible to participate in the transformation.

I.) Locate all operator nodes in \( S \) from which there is an empty
path to p. If these operator nodes do not all have the same function symbol label, p is no longer eligible to participate in the transformation. Otherwise proceed to step II.

II.) Let A be the set of instances of operator nodes from which there is an empty path to data arc p. Consider the data arcs, true gates, false gates, merge nodes, and data identity nodes on paths from operator nodes in A to p as part of a subschema $S_1$ with data input arcs the data output arcs of operator nodes in A (FIG XXXIII). The data output arcs of $S_1$ are data output arcs of nodes in $S_1$ which are not on a path from $S_1$'s input arcs to p. All data output arcs of $S_1$ except p are data output arcs of data identity nodes. If the data identity node is in a loop subschema as shown in FIG XXXIII(a), include in $S_1$ the true or false gate...
FIG XXXIII

as shown in FIG XXXIII(b). Convert $S_1$ to $S_2$ as will be shown, and connect the operator nodes to $S_2$ as shown in FIG XXXIV.

FIG XXXIV

81
In transforming from $S_1$ to $S_2$ one merely propagates the fan-in to the operator modes through $S_1$ as follows:

(a) If the mode is an identity mode, perform the following transformation:

(b) If the mode is a true or false gate, perform the following transformation:

(c) If the mode is a merge mode of a loop subschema, perform the following transformation where the fan-in will be propagated back through the body of the loop subschema to merge mode's data.
input arc labeled T.

(d) If the mode is a merge mode of a conditional subschema, propagate the fan-in down to the other input to the merge mode and perform the following transformation:

The resulting schema $S_2$ will appear as in FIG XXXIV.
We now make the following definition so that we can prove that Transformation T halts.

Def: An acyclic path in a well formed data flow schema S is a path, D, of data arcs such that D does not pass through the T data input terminal of a merge node which forms a loop subschema. 

Lemma 5-2: Transformation T halts.

We show that given any loop subschema, L, in a well formed data flow schema, S, Transformation T applied to this loop subschema will halt. Consider the set of operator nodes, A, such that there is an empty path from each node in A to an output node of L. Call the subschema of nodes and data arcs on empty paths from operator nodes in A to L's output arcs $S_1$. Call the nodes and data arcs on acyclic paths from S's input arcs to $S_1$'s input arcs $S_2$ (FIG XXV). When Transformation T is performed on an output arc of L, at least one operator node is removed from $S_1$. After each performance of Transformation T redefine subschemata $S_1$ and $S_2$. The nature of Transformation T allows $S_1$ to change only as shown by the dotted line in FIG XXV. Thus the number of nodes in $S_1$ will decrease until Transformation T halts.

A well formed data flow schema which is the result of Transformation T is a transformed well formed data flow schema (TWF-DFS).

We prove Theorem 5-1 by an induction on the path depth where path depth is defined below. First we make some preliminary definitions.

Associated with each loop subschema, $L_1$, of a well formed data flow schema S is the set of merge nodes, $M_1$, whose data input arcs
FIG XXXV

are the data input arcs of $L_1$. $M_1$ is a merge node set. Thus if there are a loop subschemata in $S$, there will be a merge node sets, $M_1, M_2, \ldots, M_n$ associated with $S$. Given a data arc $d_1$ in $S$, $d_1$'s set of merge node sets is $M_1 = \{M_{k1}, M_{k2}, \ldots, M_{kt}\}$ where $M_{kj} \in M_1$ iff there is a directed path of data arcs from one of $S$'s input arcs through a merge node in $M_{kj}$ to $d_1$.

Def: Given a data arc $d_1$ in $S$, $d_1$'s path depth is the cardinality of its set of merge node sets. /
Thus if \( d_1 \)'s set of merge node sets is \( M^{d_1} = \{ M_{k_1}, M_{k_2}, \ldots, M_{k_t} \} \), then \( d_1 \)'s path depth is \( t \).

Def: The path depth of a well formed data flow schema, \( S \), is the maximum of the path depths of \( S \)'s output arcs.

Def: The path depth of a set, \( D \), of data arcs is the maximum of the path depths of the data arcs in \( D \).

Def: A productive conditional subschema \( C \) within a WF-DFS is a conditional subschema of \( S \) such that given any data output arc \( p \) of \( C \) there exists dadep graphs \( D_1 \) and \( D_2 \) of the \( p \)-subschema such that the outcomes of the deciders which are not in \( C \)'s decider schema are mutually consistent, and \( D_1 \) and \( D_2 \) are not functionally equivalent.

Theorem 5-1: The equivalence problem for primary data output arcs \( p_1 \) and \( p_2 \) of a well formed data flow schema which is a subschema of a free TWF-DFS \( S \) is decidable.

One must consider two cases. In the first case \( p_1 \) and \( p_2 \) are not contained in a loop subschema.

I.) For the basis of the induction assume that the path depth of \( S \) is less than or equal to one. If the path depth of \( S \) is zero, then the equivalence of \( p_1 \) and \( p_2 \) is decidable because the number of dadep graphs generated by \( p_1 \)-subschema and \( p_2 \)-subschema is finite. If there is a loop subschema \( L \) in the \( p_1 \)-subschema, then the equivalence problem of \( p_1 \)-subschema's data input arcs is decidable. If the \( p_1 \)-subschema contains a loop subschema, \( L \), and if \( p_1 \)-subschema \( \rightarrow^R p_2 \)-subschema, then \( p_2 \)-subschema must also contain \( L \).
If $p_1$-subschema contains $L$ and $p_2$-subschema does not contain $L$, an interpretation exists such that $p_2$-subschema halts and $p_1$-subschema does not halt, and therefore $p_1$-subschema $\not\equiv p_2$-subschema. Now we show that $p_1$ and $p_2$ are equivalent only if given a path from an output arc $p_i$ of $L$ to $p_1$ there is another path from some data arc $p_j$ to $p_2$ such that $p_1$ and $p_j$ are equivalent. This can be seen from FIG XXXVI, for as one follows the path from $p_1$ back to $p_1$, there must be a corresponding path back from $p_2$ to $L$ such that the data arcs along these paths are pairwise equivalent if $p_1$ and $p_2$ are to be equivalent. We want to show that in FIG XXXVII a necessary condition for $p_1$ and $p_2$ to be equivalent is for $p_1$ and $p_j'$ to be equivalent where $p_1$ and $p_j'$ are data output arcs of $L$. In FIG XXXVI one need only show that $S_1$ is equivalent to the empty DFS (i.e. one which only contains a data arc) in order to show that $p_j$ is equivalent to a data output arc of $L$. If $S_1$ contains a productive conditional subschema, then $p_1$ and $p_j$ are not equivalent. If $S_1$ contains a conditional subschema which is not productive, then that conditional subschema can be transformed as shown in FIG XXXIX. $S_1$ cannot contain any loop subschema. Thus if $p_1$ and $p_j$ are equivalent, $S_1$ is equivalent to a DFS which contains only operator nodes. Observe the operator node whose data output arc is $p_j$. If this operator node is labeled $f_k$, all dadep graphs of $p_1$ must end with an operator with the function symbol label $f_k$. Thus $p_1$ and $p_j$ cannot be equivalent because $S$ is a free TNF-DFS and by Lemma 5-1 there must exist two dadep graphs of $p_1$-subschema which end with two different operator nodes. Therefore $S_1$ is
FIG XXXVI

FIG XXXVII

88
is equivalent to

FIG XXXIX

89
is equivalent to the empty DFS, and $p_j$ is equivalent to a data output arc $p_j'$ of $L$. By the same argument one can show that $p_1$ and $p_2$, which are data output arcs of a loop subschema $L$ of a free TWF-DFS $S$ with body $L_1$, are equivalent only if that portion of the $p_1$-subschema which lies within $L$ has data input arcs which are equivalent to the data input arcs to that portion of the $p_2$-subschema which lies within $L_1$.

Thus the equivalence of data input arcs to $L$ partition $L$'s data input arcs into equivalence classes. If one considers the body $L_1$ of $L$ as a free TWF-DFS, this partition of the data input arcs of $L_1$ results in a partition of $L_1$'s output arcs and thus another partition of $L_1$'s data input arcs. In [FIG XXXIX]:

```
Input Partition: 
\{ (X_1, X_2), (X_3, X_4, X_5), (X_6) \}

Partition After 1st Pass:
\{ (X_1), (X_2), (X_3, X_4), (X_5), (X_6) \}

Partition After 2nd Pass:
\{ (X_1), (X_2), (X_3, X_4, X_5), (X_6) \}

Final Output Partition:
\{ (X_1), (X_2), (X_3, X_4), (X_5), (X_6) \}
```

[FIG XXXIX]
if $X_1$ and $X_2$ are equivalent data input arcs to $L$, the first pass through $L$ results in $X'_1$ and $X'_2$ not being equivalent. If after each pass through the loop one performs an intersection of the old partition with the new partition (FIG XXXIX) of data input arcs, one will eventually have an input partition which results in an identical output partition. This final partition gives the equivalence classes of the data input arcs and data output arcs of the body of the loop subschema $L$. This in turn determines the equivalence classes of equivalent data output arcs of $L$. One performs an intersection of partitions $P_1$ and $P_2$ so that $X_1$ and $X_2$ are in the same equivalence class of $P_1$ intersect $P_2$ iff $X_1$ and $X_2$ are in the same equivalence class in both $P_1$ and $P_2$. Once the output data arcs of $L$ have been partitioned into equivalence classes of equivalent data arcs one can decide whether $p_1$ and $p_2$ are equivalent by considering them as data output arcs of a subschema which does not contain $L$ and which has the data output arcs of $L$ as data input arcs.

II.) The inductive step is to show that given a free TNF-DFS $S$ with data output arc $p_1$ such that the $p_1$-subschema has path depth $m+1$, one can solve the equivalence problem for data arcs $p_1$ and $p_2$ if the equivalence problem for two data arcs which define subschemata with path depth less than or equal to $m$ is decidable. Consider data arc $p_1$ in FIG XI where $L$ is the last loop subschema on a path from the input arcs of $S$ to $p_1$ such that the path goes
through \( m+1 \) loop subschemata. There could be a number of such loop subschemata, but the freedom of \( S \) guarantees that we can treat each loop independently. If \( p_1 \)-subschema contains \( L \), by the same reasoning used in the basis of this proof the \( p_2 \)-subschema must also contain \( L \) and the output arcs of \( L \) which serve as data input arcs to that portion of the \( p_1 \)-subschema and \( p_2 \)-subschema which lie below \( L \) must be equivalent if \( p_1 \) and \( p_2 \) are to be equivalent. Since the path depth of the TWF-DFS which defines the input arcs to \( L \) is less than or equal to \( m \), the inputs to \( L \) can
be partitioned into equivalence classes of equivalent arcs. Since
the body of $L$, $L_1$, has path depth less than or equal to $m$, one
can given a partition of input arcs to $L$ compute a corresponding
partition of output arcs. As previously the result of continuing
to compute an output partition from the input partition and per-
forming an intersection of these two partitions to form a new input
partition will terminate in the sense that the old input partition
will produce an identical new input partition of data input arcs.
If there is another loop subschema in this TWF-DFS on a path of
path depth $m+1$, perform the same decomposition into problems
involving portions of the free TWF-DFS with path depth less than
or equal to $m$.

For the case where $p_1$ and $p_2$ are contained in a loop subschema
the proof is by an induction on the number of loops which contain
$p_1$ and $p_2$.

I.) Assume that $p_1$ and $p_2$ are contained in only one loop sub-
schema $L$ where the body of the loop is $L_1$. Then by the procedure
outlined in the first part of this proof partition the inputs to
$L_1$ into equivalence classes of equivalent data arcs. Knowing this
partition one can determined whether $p_1$ and $p_2$ are equivalent by
considering them as part of a free TWF-DFS with the input arcs
to the body of the loop subschema $L$ as inputs to this subschema.

II.) Assume that $p_1$ and $p_2$ are contained in $m+1$ loops. Then by
the inductive assumption one can determine the equivalence classes
of the inputs to the $m+1$'th loop. As in the first part of this
proof solving the equivalence problem of the inputs to a loop sub-
schema is sufficient to determine the equivalence of data arcs within loop subschemata.

Given a WF-DFS $S'$ which is a subschema of a WF-DFS $S$ and which has sets of equivalent primary data output arcs it is possible to transform $S'$ into an equivalent WF-DFS subschema $S''$ such that $S''$ has no two primary data output arcs which are equivalent. FIG XLI shows the proof, for the transformation shown does not destroy the well formedness of $S'$ and yet it eliminates pairs of equivalent primary data output arcs of $S'$.

![Diagram](image-url)
Def: An irredundant WF-DFS is a WF-DFS $S$ such that no two distinct primary data arcs which are data output arcs of a WF-DFS subschema of $S$ are equivalent.

MAXIMALLY PARALLEL DATA FLOW SCHEMATA:

It is now possible to define what it means for a WF-DFS to be in maximum parallel form and to show that any free WF-DFS can be transformed into an equivalent free irredundant WF-DFS in maximum parallel form.

Def: An instance of an operator mode $a_j$ conflicts with an instance of an operator mode $a_i$ if there is a path from $a_i$ to $a_j$ which does not pass through another operator or decider mode.

Def: An instance of a decider mode $b_j$ conflicts with an instance of an operator mode $a_i$ if there is a path from $a_i$ to $b_j$ which does not pass through another operator or decider mode.

In the following definition of maximum parallel form $X$ will denote a string of operator and decider mode labels with associated outcomes. String $X_1$ of length $m$ is a prefix of string $X_2$ of length $n$ where $m \geq n$ iff the string formed by taking the first $m$ symbols of $X_2$ is equal to $X_1$.

Def: A WF-DFS $S$ is in maximum parallel form if:

(a) $X_a a_j$ is a prefix of an execution sequence of $S$, and $a_j$ does not conflict with $a_i$ implies that $X_{a_j}$ is a prefix to an execution sequence of $S$.

(b) $X_{a_i} b_j^T$ is a prefix of an execution sequence of $S$ and $b_j$ does not conflict with $a_i$ implies that $X_{b_j^T}$ is a prefix of
an execution sequence of $S$.

(c) $X_{a_1}^F b_j$ is a prefix of an execution sequence of $S$ and $b_j$ does not conflict with $a_1$ implies that $X_b^F j$ is a prefix of an execution sequence of $S$.

The following theorem can now be proven.

Theorem 5-2: An free well formed data flow schema $S$ can be transformed into an equivalent free irredundant WF-DFS $S'$ such that $S'$ is in maximum parallel form.

By Theorem 5-1 one can transform $S$ into an equivalent well formed data flow schema, $S'$, such that the equivalence of two primary data arcs which are output arcs of a well formed data flow schema in $S$ is decidable. Thus it is possible to classify primary data arcs in $S'$ into equivalence classes of equivalent data arcs. Given this partition into equivalence classes it is then possible to transform $S'$ by repeated applications of the transformation shown in FIG XII into an equivalent irredundant free well formed data flow schema $S''$. It now remains to show that $S'$ is in maximum parallel form. The construction of any WF-DFS assures that every WF-DFS will satisfy conditions (a), (b), and (c) of the definition of maximum parallel form. Well formed data flow schemata are persistent so that the only way $X_{a_1} a_j$ could be a prefix of an execution sequence of a well formed data flow schema $S$ and $X_{a_j}$ could not be a prefix of an execution sequence of $S$ is for there to be a path from $a_1$ to $a_j$ which does not pass through any operator or decider nodes. That is, if $X_{a_1} a_j$
is a prefix of an execution sequence of $S$ and $X_{a_j}$ is not a prefix of an execution sequence of $S$, then $a_j$ conflicts with $a_i$. A similar argument shows that conditions (b) and (c) hold for any WF-DFS.

**CONCLUSION**

The class of data flow schemata is defined to provide insight into two closely related problems. The first problem is that of representing schemata in maximum parallel form, and the second is the equivalence problem for schemata. This thesis discussed well formed data flow schemata in relation to these two problems. There are many questions concerning well formed data flow schemata which remain unanswered. Whether the equivalence problem of free well formed data flow schemata is decidable is at present unknown. It might be possible to formulate a definition of maximally parallel data flow schemata such that a maximally parallel free well formed data flow schema is in a canonic form. Likewise, if the equivalence problem of free well formed data flow schemata is decidable, then a stronger definition of maximum parallel form could be defined such that any free WF-DFS could be transformed into maximum parallel form.

Another area of future research is the investigation of the relationship between well formed data flow schemata, which only operate on unstructured data, and schemata which allow the use of structured data. Thus it would be possible to represent determinate parallel execution of programs which use structured
Another area of possible research interest is the analysis of equivalence preserving transformations of data flow schemata to provide for program simplification. A canonical form for free well formed data flow schemata might very well not be the simplest representation of these schemata, and therefore the series of transformations to transform a free WF-DFS into a canonical form may be quite different from the series of transformations to transform a data flow schema into a simpler form. Investigation of simplifying transformations of data flow schemata would perhaps provide insight into program structure and methods to simplify this structure.

It is felt that data flow schemata provide a good basis for research dealing with determinate schemata which allow parallel execution of schema operators. Data flow schemata are determinate schemata which are in maximum parallel form as is defined in this chapter. In this thesis a subset of data flow schemata, well formed data flow schemata, were defined, and the equivalence problem of data arcs within a well formed data flow schema was shown to be undecidable. It was then shown that a free WF-DFS can be transformed by Transformation T into a free TWF-DFS in which the equivalence of primary data arcs is decidable.
Bibliography

1. Ashcroft E., and Mann Z., *The Translation of 'Go To' Programs to 'While' Programs*, Stanford AI Memo 138, Stanford University, Palo Alto, California


