FORMAL PROPERTIES
OF WELL-FORMED
DATA FLOW SCHEMAS

Clement Kin Cho Leung

June 1975
FORMAL PROPERTIES OF WELL-FORMED DATA FLOW SCHEMAS

Clement Kin Cho Leung

June 1975
FORMAL PROPERTIES OF WELL-FORMED DATA FLOW SCHEMAS

by

Clement Kin Cho Leung

Submitted to the Department of Electrical Engineering and Computer Science on May 16, 1975 in partial fulfillment of the requirements for the Degrees of Bachelor of Science, Master of Science, and Electrical Engineer.

ABSTRACT

This thesis presents some results in comparative schematology and some undecidability results for two models of computer programs: the class of flowchart schemas and the class of well-formed data flow schemas (wfdfs's). Algorithms are given for translating a schema in each class into an equivalent schema in the other class. The properties of freedom, $\alpha$-freedom, openness and completeness are defined and studied. For every path $P$ in a free flowchart schema $S$, there exists an interpretation under which the flow of control through $S$ is along $P$. $\alpha$-freedom is a generalization of freedom and captures the notion of freedom for wfdfs's. An open schema is one in which no basic component is redundant and a complete schema contains no subschema which, whenever enabled, does not terminate. A comparison of the expressive power of subclasses of flowchart schemas and wfdfs's possessing various combinations of these properties is made. It is shown that the class of free flowchart schemas properly contains the classes of free and $\alpha$-free wfdfs's, and that the class of open and complete flowchart schemas is equivalent in expressive power to the class of open and complete wfdfs's. Three undecidability results for open and complete program schemas are established: openness is undecidable for complete program schemas, completeness is undecidable for open program schemas, and equivalence is undecidable for open and complete program schemas.

THESIS SUPERVISOR: Jack B. Dennis
TITLE: Professor of Computer Science and Engineering
ACKNOWLEDGEMENT

I wish to thank Professor Jack B. Dennis who, as thesis supervisor, provided encouragement and valuable advice throughout the course of this research. I would also like to thank David Isaman, Ken Weng and Joseph Qualitz of the Computation Structures Group for their many helpful comments. David Isaman has also proofread drafts of this thesis. The Department of Electrical Engineering has provided me with financial support through a Teaching Assistantship throughout the course of this research. Finally, I would like to acknowledge Project MAC for the use of its facilities during the preparation of this thesis.

This work was supported in part by the National Science Foundation under research grants GJ-34671 and DCR75-04060
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>2</td>
</tr>
<tr>
<td>Acknowledgement</td>
<td>3</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>4</td>
</tr>
<tr>
<td><strong>Chapter 1: Introduction</strong></td>
<td>6</td>
</tr>
<tr>
<td>1.1 Program Schematology</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Historical Background and Related Work</td>
<td>9</td>
</tr>
<tr>
<td>1.3 Organization and Contents of Thesis</td>
<td>12</td>
</tr>
<tr>
<td><strong>Chapter 2: Definitions</strong></td>
<td>14</td>
</tr>
<tr>
<td>2.0 Introduction</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Flowchart Schemas</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Well-formed data flow schemas</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Properties of Program Schemas</td>
<td>37</td>
</tr>
<tr>
<td><strong>Chapter 3: Comparative Schematology</strong></td>
<td>53</td>
</tr>
<tr>
<td>3.0 Introduction</td>
<td>53</td>
</tr>
<tr>
<td>3.1 Translation between flowchart schemas and wfdfs's</td>
<td>53</td>
</tr>
<tr>
<td>3.1.1 Translating a flowchart schema into normal form</td>
<td>53</td>
</tr>
<tr>
<td>3.1.2 Translating nfs's into wfs's</td>
<td>59</td>
</tr>
<tr>
<td>3.1.3 Translating wfs's into wfdfs's</td>
<td>79</td>
</tr>
<tr>
<td>3.1.4 Translating wfdfs's into flowchart schemas</td>
<td>89</td>
</tr>
<tr>
<td>3.2 Freedom and α-freedom in Program Schemas</td>
<td>97</td>
</tr>
<tr>
<td>3.3 Open and Complete Program Schemas</td>
<td>108</td>
</tr>
</tbody>
</table>
Chapter 4: Decision Problems

4.0 Introduction

4.1 Undecidability of Completeness in Open wdfs's

4.2 Undecidability of Equivalence in Open and Complete wdfs's

Chapter 5: Conclusion

5.1 The Main Results

5.2 Suggestions for Further Research

Bibliography

Appendix
Chapter One
Introduction

1.1 Program Schematology

Models of computation are formulated to describe and analyse problems of interest that arise in the use of our computing facilities. Some of these models are also useful design tools. The finite state model of computation has long been used in the design and synthesis of digital systems and lexical analysers. Many of our practical parsers for programming languages are based on the pushdown automata model. Using the Turing machine model (or any other equivalent formulation of the notion of effective computability) we have conveniently expressed theoretical results which establish limits on the capabilities of our computing machinery, results which establish upper and lower bounds for resource requirements of certain types of computations and results which describe resource tradeoffs. Yet another model, the contour model, allows us to express and study algorithms implementing block structuring and name scoping in programming languages.

Currently there is much interest in using computers for automatic program generation, automatic program verification, machine independent program optimization and automatic detection of parallelism in computer programs. To study problems that arise in these areas, we need models in which properties of computer programs can be conveniently expressed. So that the techniques and results developed in studying these models will be applicable not just to individual programs,
but to all programs which exhibit the same desired properties, we would also like our models to implicitly define appropriate abstractions from computer programs. Models of computer programs are known as program schemas. In this thesis we study some properties of computer programs using a class of program schemas. In this section we shall give an introduction to program schematology.

Program schematology is the study of models of computer programs. A class of program schemas is a set of abstract programs built from a given set of basic constructs using a given set of composition rules. The set of basic constructs and composition rules determines the set of program features modelled by the class of program schemas. The set of basic constructs for a class of schemas usually consists of a set of function symbols, predicate symbols and constant symbols, a set of storage elements and a set of simple executable statements. The set of simple executable statements usually consists of the application of a function symbol to the contents of some storage elements, data dependent decisions made by applying a predicate symbol to the contents of some storage elements, assignments, sequential and parallel control flow primitives. The basic constructs are put together to form a program schema using the composition rules. By assigning functions and predicates to the function and predicate symbols, values to the constant symbols and initial values to storage elements, a program schema becomes an executable program in some programming language. We identify four major classes of program schemas:

1. Flowchart schemas. [Rutledge 64] [Paterson 70]
2. Recursive schemas. [de Bakker & Scott 69]
3. Parallel program schemas. [Karp & Miller 69] [Keller 73]
4. Data flow schemas. [Dennis & Posseen 73]

Some of the applications of program schematology are:

**Studying properties of computer programs**

One is often interested in the following questions about computer programs:

i. Does a program terminate for all possible inputs?

ii. Are two computer programs functionally equivalent?

iii. Does a program contain unreachable and useless statements?

iv. Does a parallel program exhibit a maximal degree of parallelism?

v. In a parallel program, will two simultaneously executing operators interfere with each other's activities?

In program schematology these properties are formally defined and studied. Many decision problems associated with these properties have been investigated.

**Comparing the expressive power of programming language features**

Having defined the notion of equivalence between program schemas, we can prove theorems of the following kind:

Given any schema \( a \) in class \( A \), there always exists a program schema \( b \) in class \( B \) which is equivalent to \( a \).

Such theorems establish a partial ordering on classes of program schemas. If we have different classes of program schemas with different sets of basic constructs and/or different composition rules, we can gain some insight into the relative expressive power of these classes by studying
the partial ordering.

Transformation of programs

We are often interested in applying equivalence preserving transformations to programs to make them more efficient according to some cost function. The optimization techniques developed include:

i. common subexpression elimination
ii. constant propagation
iii. dead variable analysis
iv. replacing recursion by iteration
v. increasing the degree of parallelism
vi. transforming sequential programs into parallel programs

Program schematology provides a notational and conceptual basis for expressing and analysing these techniques and transformations.

1.2 Historical Background and Related Work

In this thesis we shall formally define a class of flowchart schemas and a class of data flow schemas and study some of their properties. The earliest work on program schematology was done by Ianov as reported by Rutledge [Rutledge 64]. Ianov defined a class of flowchart schemas and studied the property of equivalence between these schemas. Paterson [Paterson 70] extended Ianov’s model and also studied the equivalence problem. Most of the earliest work in schematology dealt with the modelling of sequential ALGOL-like programs and recursive programs. Later work deals mostly with comparative schematology [Hewitt and Paterson 70] and the modelling of parallel programs. Karp and Miller introduced a
model of parallel program schemas [Karp & Miller 69] which was further studied by Slutz [Slutz 68] and Keller. [Keller 73]
For a detailed study and survey of comparative schematology and properties of flowchart schemas and recursive schemas the reader is referred to [Chandra 73]. For a study and survey of results in parallel program schemas the reader is referred to [Linderman 73].

In later chapters we shall concentrate our studies on a class of data flow schemas called the well-formed data flow schemas (wdfs's). The class of well-formed data flow schemas is first studied by Dennis and Fosseen [Dennis & Fosseen 73], and it differs from the other models in the following aspects:

1. The flow of control is completely determined by the flow of data. An operator is enabled (starts to operate) when all of its inputs are available. Thus a natural notion of parallelism is inherently embodied in the definition of the model.

2. Routing of data along different paths is performed only by constructs modelling the conditional statement and the iteration statement. The resulting schemas all have the structure of goto-less programs. This allows many theorems in the model to be proved using the techniques of induction and exhaustion.

The first data flow model was a graph model for parallel computation called program graphs. In [Rodriguez 69] conditions which guarantee deterministic hang-up free behavior in program graphs were studied.

Fosseen [Fosseen 72] defined a maximally parallel wdfs as one in which no two data links are equivalent. He also
presented an algorithm for deciding the equivalence of data links (the storage elements) in free wdfs's and used it to transform free wdfs's into maximally parallel form.

Qualitz [Qualitz 72] studied the properties of weakly productive schemas which is an extension of wdfs's. In a wdfs, even if all of their inputs are available, operators on alternate data flow paths (for example, the *then* and *else* branches of an if-then-else construct) will not be enabled until the decision governing the choice between the alternate paths is available. In a weakly productive schema, operators are activated as soon as all of their inputs are available, even though the decisions governing the choice of paths have not been made. Thus in the execution of a weakly productive schema, some temporary results may be computed and later discarded. To allow for weakly productive behavior, the execution rules and data structures Qualitz used to define computations by weakly productive schemas are considerably more complicated than those for wdfs's. Qualitz also studied the properties of determinacy, termination and productivity in his model.

Recently there has been much interest in using data flow models as a basis for the development of programming languages and concepts of computer architecture. For the design of data flow programming languages we note the work of Dennis [Dennis 74] and Kosinski [Kosinski 73]. For studies in computer architectures that execute data flow programs we note the work of Lesser [Lesser 73], Dennis and Misunas [Dennis & Misunas 74] and Rumbaugh [Rumbaugh 75].
1.3 Organization and Contents of Thesis

In this thesis we study some formal properties of a class of program schemas, the well-formed data flow schemas. Some of the distinctive features of this model have been outlined in the previous section. The class of wdfs's, and the class of flowchart schemas, are formally defined in Chapter 2. In Section 3.1 algorithms are presented for translating a program schema in the class of wdfs's or in the class of flowchart schemas into an equivalent schema in the other class.

In Chapter 2 we also define some properties of program schemas and establish some relationships between these properties. Four of these properties are of special interest to us: freedom, $\alpha$-freedom, openness and completeness. Freedom is first studied by Paterson [Paterson 70]. The class of program schemas he studied uses simple predicate statements for branching control. $\alpha$-freedom is a generalization of freedom and captures the notion of freedom for program schemas with more powerful branching controls. Openness and completeness are properties of "well-constructed" programs. Informally, an open schema is one in which no basic construct is redundant and a complete schema contains no subschema which, whenever enabled, does not terminate. In Section 3.2 we show the differences in expressive power between different classes of free and $\alpha$-free program schemas. In Section 3.3 we establish the equivalence in expressive power between open and complete wdfs's and open and complete flowchart schemas.
In Chapter 4 we prove some undecidability results for open program schemas and complete program schemas. The undecidability of equivalence in open and complete program schemas implies the non-existence of algorithms for determining equivalence between practical "well-constructed" computer programs.

In Chapter 5 the implications of the technical results in this thesis are examined and directions for further research are suggested.
2.0 Introduction

In this chapter two classes of program schemas, the class of wfdfs's and the class of flowchart schemas, are formally defined by giving the basic constructs and composition rules for each class. Flowchart schemas are models of computer programs written in an ALGOL-like programming language. The composition rules for flowchart schemas can model any control structure constructed from conditional and unconditional branching, e.g., the if-then-else conditional statement, the different kinds of iteration statements and the goto statement. In this respect the composition rules for wfdfs's are more restrictive. They can only directly model the kind of control flow permitted by the if-then-else conditional statement and the do-while iteration statement. The class of flowchart schemas is defined in Section 2.1. The class of wfdfs's is defined in Section 2.2.

Terminology for describing a computation by either a wfdfs or a flowchart schema is introduced in Section 2.3. Several properties of program schemas are then defined. Four of these properties are studied in more detail in later chapters: freedom, ω-freedom, openness and completeness. In this section we also establish some elementary relationships between program schemas possessing certain properties.
2.1 Flowchart Schemas

In this section a class of flowchart schemas is defined which models the control structures of ALGOL-like languages.

Basic Constructs
i. an infinite set of variables \( \text{Var} = \{ V_i | i > 0 \} \)
ii. an infinite set of function symbols \( \text{Func} = \{ F_i^j | i > 0, j > 0 \} \)
iii. an infinite set of predicate symbols \( \text{Pred} = \{ P_i^j | i > 0, j > 0 \} \)
iv. a set of simple statements of the following form:

\[
\begin{array}{c}
\text{START} \\
\{ X_1, \ldots, X_n \}
\end{array}
\]

\( X_1, \ldots, X_n \in \text{Var} \). The \( X_i \)'s are the input variables.

Assignment statement

\[
X \leftarrow E
\]

\( X \) is a variable. \( E \) is either a variable or an expression of the form \( F_i^j(X_1, \ldots, X_j) \) with \( F_i^j \in \text{Func} \) and \( X_1, \ldots, X_j \in \text{Var} \). \( X \) is the left hand side (LHS), \( E \) the right hand side (RHS), of the assignment. The variable(s) that occurs in \( E \) is the variable accessed by the assignment statement.
Predicate statement
\[ p_i^j \in \text{Pred} \]
\[ x_1, \ldots, x_j \in \text{Var} \]

The variables \( x_1, \ldots, x_j \) are the variables accessed by the predicate statement.

Halt statement
\[ x_1, \ldots, x_n \in \text{Var} \]

The variables \( x_1, \ldots, x_n \) are the output variables.

Composition Rules

The composition rules for flowchart schemas are defined using some concepts from graph theory.

A directed graph \( G \) is an ordered pair \( (V,E) \) where \( V \) is a set of vertices and \( E \subseteq V \times V \) is a set of edges. If \( e=(v_i, v_j) \) is an edge in \( E \), then \( \text{head}(e)=v_i \), \( \text{tail}(e)=v_j \). A path (possibly infinite) through \( G \) is a sequence \( (v_1', \ldots, v_n') \) of nodes such that for all \( i, 1 \leq i \leq n \), \( (v_{i-1}, v_i) \in E \). An edge \( e \) is an incoming branch of a vertex \( v \) if \( \text{tail}(e)=v \). An edge \( e \) is an outgoing branch of a vertex \( v \) if \( \text{head}(e)=v \). A root of \( G \) is a vertex \( v \) which has no incoming branch. A leaf of \( G \) is a vertex \( v \) which has no outgoing branch.

A flowchart schema is a finite directed graph \( S=(V_S, E_S) \) with the following properties:
1. \( V_S \) is a set of simple statements.
2. \( S \) has a unique root, which is a Start statement. This Start statement is the only Start statement in \( S \).
iii. S has a unique leaf, which is a Halt statement. This Halt
statement is the only Halt statement in S.
iv. Every statement, except the predicate statements and the
Halt statement, has exactly one outgoing branch.
v. A predicate statement has exactly two outgoing branches.
One of these branches is labelled by T, the other by F.
vi. If W is an assignment statement or a predicate statement
and X is a variable accessed by W, then either X is an
input variable, or for every path $S_1 \ldots S_n$ in S,
$S_1 = \text{Start statement in } S,$
$S_n = W,$
there is an assignment statement $S_j$, $1 < j < n$, such that
X is the LHS of $S_j$.

Intuitively ii and iii say that we have a single entry
and single exit program. vi rules out the possibility of
accessing undefined variables.

The basic constructs and composition rules define the
syntax of flowchart schemas. To relate a flowchart schema
with the family of programs it models and to study its
dynamic behavior, we have to associate interpretations with
a flowchart schema. We also have to associate inputs and
execution sequences with an interpreted schema.

An interpretation for flowchart schemas consists of a
domain $D$ and a function $I$ which assigns
to every function symbol $F^i_j$, a total function $f^i_j : D^i \rightarrow D$,
to every predicate symbol $P^i_j$, a total predicate
$P^i_j : D^i \rightarrow \{T, F\}$. 
Given an interpreted schema \((S,I)\), an input to \((S,I)\), \(\text{In}_{(S,I)}\), is a function which assigns to every input variable of \(S\) an element of \(D\). Formally,

\[
\forall x_i \in \text{var}, \text{In}_{(S,I)}(x_i) = \begin{cases} 
\alpha_i \in D, & \text{if } x_i \text{ is an input variable} \\
\text{undefined} & \text{otherwise}
\end{cases}
\]

An executable program is an interpreted schema with input, denoted by \((S,I,\text{In})\).

To describe the dynamic behavior of an executable program we need the notion of a value sequence for an arbitrary path through an interpreted schema.

Let \(\Omega\) be an object not in \(D\). \(\Omega\) denotes the undefined element. Let \(P\) be an arbitrary path (possibly infinite) through a schema \(S\), starting from the Start statement, i.e., \(P = P_1 \ldots P_n \ldots\), \(P_1\) being the Start statement, each \(P_i\) being a simple statement. The value sequence \(A\) for \(P\) under interpretation \(I\) with input \(\text{In}\) is an infinite sequence \(A = (a_1, \ldots, a_\ell, \ldots)\) where \(a_i = (a_{1i}, \ldots, a_{ij}, \ldots) \in (D \cup \Omega)^\infty\). Intuitively each \(a_i\) denotes the state of the memory at step \(i\), and \(a_{ij}\) denotes the value of the variable \(V_j\) at the \(i\)-th step. \(A\) is defined as follows:

1. \(P_1\) is the Start statement and the input variables are \(X_1, \ldots, X_m\).

\[
a_{1i} = \begin{cases} 
\text{In}(X_i) & \text{if } X_i \text{ is an input variable} \\
\Omega & \text{otherwise}
\end{cases}
\]

2. At the \(i\)-th step, \(i > 1\),

   i. if \(P_i\) is an assignment statement with LHS \(V_e\) and RHS \(E\) then
\[ a_{ik} = \begin{cases} 
 a_{(i-1)j} & \text{if } k = e \text{ and } E = v_j \\
 f_j^n(a_{(i-1)q_1}, \ldots, a_{(i-1)q_n}) & \text{if } k = e, \\
 E = f_j^n(v_{q_1}, \ldots, v_{q_n}) \text{ and } I(F_j^n) = f_j^n \\
 a_{(i-1)k} & \text{otherwise} 
\end{cases} \]

ii. if \( P_i \) is a predicate statement, then for all \( j \)
\[ a_{ij} = a_{(i-1)j} \]

iii. if \( P_i \) is the Halt statement, then \( P_i \) is the last statement in \( P \) since the Halt statement has no outgoing branch. For all \( m \geq i \),
\[ a_{mj} = a_{(i-1)j} \]

An execution sequence through \( S \) under interpretation \( I \) and input \( In \) is a path \( P \) through \( S \) such that:
1. \( P \) starts from the Start statement of \( S \).
2. \( P \) is either infinite or \( P \) terminates on the Halt statement of \( S \).
3. Let \( A \) be the value sequence for \( P \), then for all \( i \) such that \( P_i \) is a predicate statement:
\[ I(P_j^k) = p_j^k \]
\[ P_j^k(v_{il}, \ldots, v_{ik}) \]
the edge \((P_i, P_{i+1})\) is labelled by \( T \) iff
\[ P_j^k(a_{(i-1)il}, \ldots, a_{(i-1)ik}) = T \]
the edge \((P_i, P_{i+1})\) is labelled by \( E \) iff
\[ P_j^k(a_{(i-1)il}, \ldots, a_{(i-1)ik}) = E \]
Thus an execution sequence for an executable program \((S,I,In)\) is the sequence of simple statements which are executed when the function and predicate symbols are interpreted, and the input variables are initialized.

A schema \(S\) **terminates** on input \(In\) under interpretation \(I\) iff the execution sequence for \((S,I,In)\) is finite. Otherwise \(S\) **diverges** under \(I\) and \(In\).

The **output** of a schema \(S\) on input \(In\) under interpretation \(I\), denoted by \(Out(S,I,In)\), is defined as follows:

1. If \(S\) terminates under \(I\) and \(In\), \(\{x_1, \ldots, x_m\}\) is the set of output variables, and the execution sequence contains \(m\) statements, then
   \[
   Out(S,I,In) = (a_{m(1)}, \ldots, a_{m(n)})
   \]
2. If \(S\) diverges under \(I\) and \(In\), then
   \[
   Out(S,I,In) = \Omega^n
   \]

In Figure 2.1 we give an example of a flowchart schema and interpretation which converts the schema into a flowchart program. Some properties of flowchart schemas are defined in Section 2.3. In the remaining chapters of this thesis, we will often denote function symbols by the letters \(f, g, h\), variables by the letters \(x, y, z\), and predicate symbols by the letters \(p, q, r\). The arities of the function symbols and predicate symbols will be omitted wherever it is self-evident from the given context.

### 2.2 Well-formed data flow schemas

The class of wdfs is a graph model of parallel computation. An example of a wdfs is given is Figure 2.2. This wdfs is "equivalent" to the flowchart schema in Figure 2.1
Interpretation I:

\[ D \rightarrow \mathbb{N}, \text{the natural numbers} \]
\[ F^1_1\rightarrow \text{Zero}, \text{the function which always returns 0} \]
\[ F^2_1\rightarrow \text{Subl}, \text{the predecessor function} \]
\[ F^2_1\rightarrow \text{Add}, \text{the addition function} \]
\[ P^1_1\rightarrow \text{ZeroP}, \text{the predicate which tests for 0}. \]

Figure 2.1 An example of a flowchart schema
Figure 2.2 An Example of a wdfs

To define the class of wdfs's formally we need some additional terminology and definition from graph theory. Let $V$ be the union of two disjoint sets $V_1$ and $V_2$. A **bipartite directed graph** on $V$ is a directed graph $(V,E)$ with $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$. An **acyclic** graph is a graph which contains no path $p_1 \ldots p_n$ such that $p_1 = p_n$. If $e \in E$, $e = (a, b)$, $e$ is an **output arc** of $a$ and is an **input arc** of $b$.

**Basic Constructs**

A wdfs is a bipartite directed graph. The basic constructs of a wdfs are the two different types of nodes and the arcs.

1. **Link nodes** There are two kinds of link nodes, the data links and the control links. A link node has a unique input arc and one or several output arcs. A data link (control link) receives a data value (a control signal) on its input arc and puts a copy of this value (signal) on every
Figure 2.3 Node types for WDFS
output arc. The link nodes behave like fan-out devices in an
electrical circuit. A data link is denoted by a solid dot.
A control link is denoted by a circle. (Figure 2.3a)

2. Actor nodes. The actor nodes are the processing elements of
a wdfs. The different kinds of actor nodes are shown in
Figure 2.3b. A link node is an input(output) link of an
actor if the input(output) arc of the actor is an output
(input) arc of the link node. The operator nodes and decider
nodes are labelled by function symbols and predicate symbols
respectively. As in flowchart schemas, a function symbol is
an element of the set \( \{ F_i^j | i \geq 1, j \geq 1 \} \) and a predicate symbol is
an element of the set \( \{ P_i^j | i \geq 1, j \geq 1 \} \). If an operator is
labelled by the function symbol \( F_i^j \), the operator has \( j \) input
data links and an output data link. If a decider is labelled
by the predicate symbol \( P_i^j \), the decider has \( j \) input data
links and an output control link. An I-operator has no input
link and has one output data link. An O-operator has no
output link and has one input data link. The number and type
of input-output data links for each kind of actor are also
shown in Figure 2.3b.

3. Arcs. An arc joins an input link to an actor, or an actor
to its output link. The arcs transmit data between nodes and
are the storage elements in this model. An arc is a data
arc or a control arc depending on the link node is is con-
ected to.

Diagramming Conventions

Before we define the composition rules for wdfs's, we
introduce some additional diagramming conventions to simplify
our figures.
(a) Bundling of data links, arcs and operators

(b) Multiple input–multiple output merge gate

(c) Decision structure

---

Figure 2.4 Diagramming Conventions
1. In Figure 2.4a a broad arc represents a bundle of arcs connecting sets of links and actors. A large dot represents a set of data links. Every data link of the data link set is driven from exactly one arc of the incident bundle and each link must be the origin of at least one arc in some emanating bundle.

2. An I-operator is represented by a solid square with no input arc. An O-operator is represented by a solid square with no output arc. A set of I-operators with a set of emanating arcs is represented by a large solid square with a broad emanating arc. A set of O-operators with a set of incident arcs is represented by a large square with a broad incident arc. (Figure 2.4a)

3. A set of merge gates driven by the same control link is represented by a multiple input-multiple output merge gate as shown in Figure 2.4b. The same diagramming convention is adopted for a set of T-gates driven by the same control link and for a set of F-gates driven by the same control link.

4. A decision structure is shown in Figure 2.4c. An acyclic net of boolean actors is an acyclic bipartite directed graph on the sets of boolean operators and control links. A decision structure represents a set of deciders that provide input control values to an acyclic net of boolean operators having a single output control link.

Composition Rules

The class of wdfs is defined inductively.

A level-0 \((m,n)\) wdfs \(S\) is a bipartite directed graph whose two types of nodes are data links and operators. \(S\) has
the following properties:

i. All the roots of $S$ are I-operators and all the I-operators of $S$ are roots.

ii. All the leaves of $S$ are 0-operators and all the 0-operators of $S$ are leaves.

To describe the composition of wdfs's we introduce a special operator $\text{Id}$ (for identity operator) and define an Id-subgraph of a wdfs.

Let $\text{Id}$ be a function symbol, $\text{Id} \notin \{\text{F}_i^j \mid i \geq 1, j \geq 1\}$. An Id-operator is a single input, single output operator labelled by the special function symbol $\text{Id}$. An Id-subgraph $D$ of a wdfs $S$ is a subgraph of $S$ such that:

i. $D$ is a bipartite directed acyclic graph whose two types of nodes are data links and Id-operators.

ii. $D$ has a unique root which is a data link.

iii. All the leaves of $D$ are data links.

iv. The root of $D$ is not the output data link of an Id-operator.

v. No leaf of $D$ is the input data link of an Id-operator.

To collapse an Id-subgraph $D$ in a wdfs $S$:

i. Remove $D$ from $S$.

ii. Introduce a new data link $d$.

iii. If $a$ is an arc in $S$ connecting a node $a_1, a_1 \notin D$, to a node $a_2 \in D$, replace $a$ by an arc $b$ in the same direction, connecting $a_1$ to $d$.

An example of an Id-subgraph and the collapsing process is shown in Figure 2.5.

A data link $d$ is joined to an I-operator by adding an arc from $d$ to the I-operator and then relabelling the I-
operator by the special symbol \textit{Id}.

An \textit{O}-operator is \textit{joined} to a data link \(d\) by adding an arc from the \textit{O}-operator to \(d\) and then relabelling the \textit{O}-operator by the special symbol \textit{Id}.

A set of data links \(D\) is \textit{joined} to a set of \textit{I}-operators \(L\) by joining data links in \(D\) to \textit{I}-operators in \(L\) such that:

\begin{enumerate}
  \item Every data link \(d \in D\) is joined to at least one \textit{I}-operator \(\ell \in L\).
  \item For every \textit{I}-operator \(\ell \in L\) there is a data link \(d \in D\) which is joined to \(\ell\).
\end{enumerate}

Similarly a set of \textit{O}-operators \(L\) is \textit{joined} to a set of data links \(D\) by joining \textit{O}-operators in \(L\) to data links in \(D\) such that:

\begin{enumerate}
  \item Every \textit{O}-operator in \(L\) is joined to a data link in \(D\).
  \item For every data link \(d \in D\) there is an \textit{O}-operator \(\ell \in L\) which is joined to \(d\).
\end{enumerate}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{diagram.png}
\caption{Collapsing an \textit{Id}-subgraph}
\end{figure}
Figure 2.6 A Conditional wfdfs

Figure 2.7 An Iteration wfdfs
A \textit{level-i+1 (m_p,n_p) conditional wdfs} S is constructed from two level-i wdfs's P and Q, and a decision structure C as shown in Figure 2.6. P is connected to the data link sets M_p and N_p by:

i. Joining M_p to the set of I-operators of P.

ii. Joining the set of O-operators of P to N_p.

iii. Collapsing all Id-subgraphs formed in (i) and (ii).

Similarly Q is connected to the data link sets M_Q and N_Q.

A \textit{level-i+1 (m_p,n_p) iteration wdfs} T is constructed from a level-i wdfs P and a decision structure C as shown in Figure 2.7. P is connected to the data link sets M_p and N_p as described above.

A \textit{level-i+1 (m,n) wdfs} is either:

i. a level-i (m,n) wdfs,

ii. a level-i+1 (m,n) conditional wdfs,

iii. a level-i+1 (m,n) iteration wdfs, or

iv. an acyclic composition of subschemas of the above three types. In putting the subschemas together to form an acyclic composition, the outputs of some subschemas are used as inputs to other subschemas. The composition is performed in two steps:

a. If the output of a subschema S_j at O-operator l is to be used as input to subschemas H_1, ..., H_k at I-operators i_1, ..., i_j, create a new data link d'_l; \textit{join} l to d'_l and \textit{join} d'_l to i_1, ..., i_j. Perform this joining operation for every O-operator which is the input to another subschema.

b. Collapse all the Id-subgraphs formed in (a).
The acyclic composition has $m$ I-operators and $n$ O-operators.

A conditional workflow performs the decision represented by its decision structure and selects either the true alternative $P$ or the false alternative $Q$ for execution to provide output values. An iteration workflow uses its decision structure to test some of its input values and presents some input values to its body $P$. The output values of $P$ are tested and the cycle is repeated until some test yields a false decision, whereupon certain values are presented as the outputs of $S$. To execute a workflow we initialize it by putting control values on certain control arcs of the workflow, assign inputs to the output arcs of the I-operators, and interpret the function and predicate symbols. The actors of $S$ will then be enabled and fired according to the firing rules for each actor.
Initialization

For every iteration subschema $E$ of a wfdfs $S$, the control value $F$ is assigned to the control arc of the merge gate as shown in Figure 2.9. This enables the decision structure to receive the inputs to $E$ on the first iteration.

![Diagram of initialization of iteration subschemas](image)

Figure 2.9 Initialization of iteration subschemas

Interpretation and Input:

An interpretation for wfdfs's consists of a domain $D$ and a function $I$ which assigns to every function symbol $f_i^j$ a total function $f_i^j : D^j \rightarrow D$, and to every predicate symbol $p_i^j$ a total predicate $p_i^j : D^j \rightarrow \{T, F\}$. An interpreted schema is denoted by the pair $(S, I)$.

An input for an interpreted schema $(S, I)$ is a function $In$ which assigns to the output arc of every $I$-operator of $S$ an element from the domain $D$. An interpreted schema $(S, I)$ with input $In$ is denoted by $(S, I, In)$.

The activity of an interpreted wfdfs with input is represented by a sequence of configurations. A configuration for $(S, I)$ consists of:

1. An association of a value in domain $D$ or the symbol $null$
with each data arc of $S$.

2. An association of one of the symbols $\{T,F,\text{null}\}$ with each control arc of $S$.

The initial configuration of $(S,I,In)$ is established as follows:

1. The iteration subschemas are initialized as described.
2. An element of $D$ is associated with the output arc of every I-operator of $S$ according to $In$.

We depict a configuration of a wfdfs by drawing a solid circle on each arc having a non-null value, and writing a value-denoting symbol beside. These circles are called data tokens, $T$ tokens or $F$ tokens according to the associated values.

A configuration sequence $\eta$ for $(S,I,In)$ is a sequence of configurations $\eta_0, \eta_1, \ldots, \eta_k, \eta_{k+1}, \ldots$ where

1. $\eta_0$ is the initial configuration of $(S,I,In)$.
2. Each $\eta_{i+1}$ is obtained from $\eta_i$ by the firing of some enabled node of $S$ in $\eta_i$. The firing rules for the two types of link nodes and for four types of actors are given in Figure 2.10. Conditions under which a node is enabled are shown on the left (an enabled node is indicated by an asterisk). A necessary condition for any node to be enabled is that its output arc does not hold a token. Any node enabled in $\eta_i$ may be chosen to fire, producing the change in configuration specified in the right part of the figure. Referring to Figure 2.10, an F-gate behaves identically to a T-gate with the input control signal
(a) link nodes

\[
\begin{align*}
&v 
\Rightarrow \quad -34- \\
&b
\end{align*}
\]

data link

control link

(b) actor nodes

\[
\begin{align*}
&v_1 \quad \ldots \quad v_i 
\Rightarrow \quad -34- \\
&v = f_j^i(v_1, \ldots, v_i) \\
&b = p_j^i(v_1, \ldots, v_i)
\end{align*}
\]

operator

decider

T-gate

merge gate

Figure 2.10 Firing rules for actors
negated. Boolean actors behave similarly to operators.
I-operators and O-operators are not enabled.

3. If $\eta$ is a finite sequence of $n$ configurations, then in the
terminal configuration $\eta_{n-1}$:
   i. no node is enabled
   ii. there is a data value associated with the input arc of
every O-operator of $S$.

If $\eta$ is a finite configuration sequence for $(S,I,In)$,
the output of $(S,I,In)$ under $\eta$, denoted by $\text{Out}^\eta(S,I,In)$, is
a function which assigns to the input arc of every O-operator
$\lambda$ the data value associated with the input arc of $\lambda$ in the
terminal configuration of $\eta$.

In any configuration more than one node may be enabled
and any one of these enabled nodes may be chosen to fire to
generate the next configuration. Thus there may be several
configuration sequences associated with a given $(S,I,In)$.
For our definition of termination and output to be meaning-
ful we state Theorem 2.2-1 which is proved in [Fosseen 72].

**Theorem 2.2-1:** If $\alpha$ and $\beta$ are configuration sequences for
the interpreted wdfs $(S,I)$ under input $In$, then
i. $\alpha$ is finite if and only if $\beta$ is finite.
ii. if $\alpha, \beta$ are finite, $\text{Out}_\alpha^\alpha(S,I,In) = \text{Out}_\beta^\beta(S,I,In)$

Any schema which satisfied properties (i) and (ii) of
Theorem 2.2-1 is called a determinate schema. Theorem 2.2-1
states that all wdfs's are determinate.

A wdfs $S$ under interpretation $I$ and input $In$ terminates
if and only if all configuration sequences (or any
configuration sequence, by Theorem 2.2-1) are finite.
Otherwise \( S \) diverges under \( I \) and \( In \).

Let \( S \) be a \((m,n)\) \wdfs. If \( D \) is the domain associated
with an interpretation \( I \) for \( S \), let \( \Omega \) be an object not in \( D \),
denoting the undefined object. The output for \((S,I)\) under
input \( In \), denoted by \( Out(S,I,In) \), is :

i. If \((S,I,In)\) terminates, \( Out(S,I,In) = Out^\alpha \)
    for any configuration sequence \( \alpha \) of \((S,I,In)\).

ii. If \((S,I,In)\) diverges, \( Out(S,I,In) = \Omega \), where \( \Omega \) assigns the
    object \( \Omega \) to the input arc of every 0-operator of \( S \).

Given a \wdfs \( S \), let
\( F \) be the set of operators in \( S \),
\( P \) be the set of deciders in \( S \),
\( B \) be the set of boolean operators in \( S \),
\( C \) be the set of control links in \( S \),
\( D \) be the set of data links in \( S \),
\( G \) be the set of T-gates, F-gates and merge gates in \( S \).

If \( \eta \) is a configuration sequence of \( S \), the firing
sequence for \( \eta \), denoted by \( \tau = \tau_1 \ldots \tau_i \ldots \), is defined as
follows:

\[
\tau_i = \begin{cases} 
  f, c, d or g, & \text{where } f \in F, c \in C, d \in D, g \in G \\
  T & \text{if } \eta_i \text{ is obtained from } \eta_{i-1} \text{ by firing} \\
  p \text{ or } b^T & \text{where } p \in P, b \in B \\
  & \text{firing } p \text{ or } b, \text{ and the outcome is } T
\end{cases}
\]
\[ p^F \text{ or } b^F \text{ where } p \text{ or } b \text{ or } B \]

-37-

if \( \tau_i \) is obtained from \( \tau_{i-1} \) by firing \( p \)
or \( b \), and the outcome is \( F \).

If \( \tau_i \) is \( b^T \) or \( b^F \), and the output link of \( b \) is the
output control link of a decision structure \( \chi \), we will also
refer to \( \tau_i \) being \( \chi^T \) or \( \chi^F \) according to the outcome of \( b \).

2.3 Properties of Program Schemas

In this section we establish a set of common terminology
for flowchart schemas and wdfds's and define some properties
for them.

A program schema is either a flowchart schema or a wdfds.

Let \( S \) be a \((m,n)\) wdfds. Let \( T \) be a flowchart schema
with \( m \) input variables and \( n \) output variables. In the
remainder of this thesis we shall assume that the sets of
I-operators and Q-operators in \( S \), and hence the set of output
arcs of the I-operators and the set of input arcs of the
Q-operator, are totally ordered from 1 to \( m \) and from 1 to \( n \)
respectively. We shall also assume that the sets of input
variables and output variables of \( T \) are totally ordered from
1 to \( m \) and from 1 to \( n \) respectively. Given an interpretation
\( I \), with associated domain \( D \) and the undefined object \( \Omega \), an
input \( \text{In} \) to \((S,I)\) or \((T,I)\) can be denoted by a \( m \)-tuple in \( D^m \).
An output \( \text{Out} \) for \((S,I,\text{In})\) or \((T,I,\text{In})\) can be denoted by a
\( n \)-tuple in \((D \cup \Omega)^n \). An interpreted schema \((S,I)\) then
becomes a function \( F_{(S,I)} : D^m \to (D \cup \Omega)^n \).
Free Interpretations

In the remainder of this thesis we shall associate a special set of input symbols, \( \text{Insym} = \{ \delta_i \mid i \geq 1 \} \), with program schemas.

Let \( S \) be a \( m \)-input program schema.

Let \( D_m \) be the following recursively defined set:

(i) \( \delta_1, \ldots, \delta_m \in D_m \)

(ii) If \( d_1, \ldots, d_i \in D_m \) and \( f_{j}^i \in \text{Func} \),

\[ f_{j}^i \cdot d_1 \cdot \ldots \cdot d_i \in D_m \]

where \( \cdot \) is the string concatenation operator.

(iii) \( D_m \) contains only those strings generated by a finite number of applications of (i) and (ii).

A free interpretation \( I_f \) for \( S \) is an interpretation with associated domain \( D_m \) and assigns:

(i) to every function symbol \( f_{j}^i \), the function \( f_{j}^i : \langle D_m \rangle^i \rightarrow D_m \), such that for all \( d_1, \ldots, d_i \in D_m \),

\[ f_{j}^i (d_1, \ldots, d_i) = f_{j}^i \cdot d_1 \cdot \ldots \cdot d_i \]

(ii) to every predicate symbol \( p_{j}^i \in \text{Pred} \), a predicate

\[ p_{j}^i : \langle D_m \rangle^i \rightarrow \{ T, F \} \]

All free interpretations for a \( m \)-input program schema have the same associated domain and assign the same function to a function symbol. They differ only in their assignment of predicates to the predicate symbols.
The input In for a m-input program schema under free interpretations is uniquely determined:

\[ 1 \leq i \leq m, \quad \text{In}(i) = \delta_i \]

Since the arity \( j \) of a function \( f^j_i \) is encoded in the function symbol \( f^j_1 \), every string generated in a computation under \( (S, I_f, \text{In}) \) can be transformed unambiguously into a function application tree, showing the order in which the operators are applied to the inputs to generate the string. Behaviour of a program schema \( S \) under the set of all free interpretations is "representative" of the behaviour of \( S \) under all interpretations. This useful property of \( S \) is stated in Theorem 2.3-1.

**Theorem 2.3-1** If a program schema \( S \) generates an execution sequence or a configuration sequence under interpretation \( I \), there is a free interpretation \( I_f \) under which \( S \) generates the same sequence.

Theorem 2.3-1 implies that if we want to prove certain properties of a program schema \( S \), it is often sufficient to prove that \( S \) has these properties under the set of free interpretations. For a proof of Theorem 2.3-1 and a more general discussion on free interpretations the reader is referred to [Chandra 73]. We shall often use free interpretations in our proofs and examples.
Terminology

By a computation sequence of a program schema $S$, we mean either an execution sequence (if $S$ is a flowchart schema) or a configuration sequence (if $S$ is a wdfs), corresponding to $(S,I,In)$ for some interpretation $I$ and some input $In$. In a computation sequence a function application consists of applying an interpreted function symbol to a set of values in the domain, a predicate decision consists of applying an interpreted predicate to a set of values in the domain. The result of a function application is an element in the domain, the outcome of a predicate decision is either $T$ or $F$. Two function applications (predicate decision) are similar if each of them consists of applying the same function (the same predicate) to the same set of values. Two similar predicate decisions are consistent if they have the same outcome. A computation sequence is consistent if no two similar predicate decisions in the sequence have opposite outcomes. A branching control in a flowchart schema $S$ is any predicate statement in $S$. In a wdfs $S$ a branching control is any decision structure in $S$ or a decider in $S$ whose output control link controls a conditional subschema or an iteration subschema of $S$.

Convergence

A program schema $S$ is convergent if $(S,I,In)$ terminates for all interpretations $I$ and all inputs $In$. 
Divergence

A program schema $S$ is divergent if for all interpretations $I$ and all inputs $In$, $(S,I,In)$ diverges.

Equivalence

Two $m$-input $n$-output program schemas $S_1$ and $S_2$ are equivalent if for all interpretations $I$ and all inputs $In$,

$$\text{Out}(S_1,I,In) = \text{Out}(S_2,I,In)$$

Let $C_1$ be a branching control in a program schema $S$. Let $\eta$ be a computation sequence of $S$ under an interpretation $I$. Let $C_1$ be a decision in $\eta$ made with $C_1$. $C_1$ is restricted in $\eta$ if the outcome of $C_1$ is logically implied by some previously made branching control decision outcomes in $\eta$. $C_1$ is restricted in $\eta$ if some decision by $C_1$ in $\eta$ is restricted. $C_1$ is restricted in $S$ if there is some computation sequence of $S$ in which $C_1$ is restricted. A free program schema is one in which no branching control is restricted, if all branching controls are single deciders. An $\alpha$-free program schema is one in which no branching control is restricted, if branching controls can be decision structures. The formal definitions of freedom and $\alpha$-freedom given below can be more easily understood in this light. In Section 3.2 some differences between free and $\alpha$-free program schemas are delineated.

Freedom

A program schema $S$ is free if there are no similar predicate applications in any computation sequence of $S$. 
The following theorem states an important property of free program schemas. This property can be used as an alternative definition of freedom.

**Theorem 2.3-2:** A flowchart schema $S$ is free if and only if for every path $P$ through $S$ from the Start statement to the Halt statement, there is an interpretation $I$ and an input $I_{in}$ such that $P$ is the execution sequence under $(S, I, I_{in})$.

**Proof:**

(if) Assume $S$ is not free, then there is an interpretation $I$ and an input $I_{in}$ such that the execution sequence $P$ of $(S, I, I_{in})$ contains two similar predicate decisions. Let $p_1 \ldots p_i \ldots p_n$ be a prefix of $P$, where $p_1$ is the Start statement and $p_n$ is the first predicate decision similar to a previously made decision $p_i$. If the edge $(p_i, p_{i+1})$ is labelled by $T(F)$, then the path $p_1 \ldots p_i p_{i+1} \ldots p_n p_{n+1}$ with the edge $(p_n, p_{n+1})$ labelled by $F(T)$ cannot be the prefix of any execution sequence. Hence not all paths through $S$ can be an execution sequence for $S$.

(only if) Assume $S$ is free, then for all paths $P=p_1 \ldots p_n$ through $S$, we can construct an interpretation $I$ and input $I_{in}$ such that $P$ is the execution sequence for $(S, I, I_{in})$ as follows:
I is a free interpretation for S. I maps each function symbol $f^i_j$ into the function $f^i_j: D^i \to D$, where D is the associated domain of I:

$$f^i_j(d_1, \ldots, d_i) = f^i_j(d_1 \ldots d_i), \quad d_1, \ldots, d_i \in D$$

I maps each predicate symbol $p^i_j$ into the predicate $p^i_j$, $p^i_j: D^i \to \{T, F\}$:

$$p^i_j(s_1, \ldots, s_i) = \begin{cases} 
T & \text{if } p_k \text{ is a node in } P; p_k \text{ consists of applying } p^i_j \text{ to the variables } x_{kl}, \ldots, x_{ki}; x_{kl} \text{ has value } s_1, \ldots, x_{ki} \text{ has value } s_i; \text{ and the edge } (p_k, p_{k+1}) \text{ is labelled by } T \\
F & \text{otherwise}
\end{cases}$$

Since I is a free interpretation for S, the input $In$ for S under I is given by:

$$\text{lsisno. of inputs of } S, \quad In(i) = \delta_i$$

From the freedom of S it follows that I is self-consistent and hence is an interpretation for S.

Q.E.D.

In a flowchart schema, branching controls are predicate statements. In a wdfs a branching control may contain a set of deciders connected to a set of boolean operators. To capture the notion of every path through a wdfs being the execution sequence under some interpretation and input, we define $\alpha$-freedom for program schemas.
A flowchart schema \( S \) is \( \alpha \)-free if for every prefix \( P_1 \ldots P_n P'_{n+1} \) of every execution sequence \( p \) of \( S \), \( P_n \) being a predicate statement and \((P_n, P'_{n+1})\) being labelled by \( T \) (\( F \)), \( P_1 \ldots P_n P'_{n+1} \) is also a prefix of some execution sequence of \( S \), \((P_n, P'_{n+1})\) being labelled by \( F \) (\( T \)).

Let \( \tau = \tau_1 \ldots \tau_n \) be a prefix of the firing sequence of a configuration sequence \( \eta \) of a wdfs \( S \), \( \tau_n \) being the outcome of a branching control in \( S \). A wdfs \( S \) is \( \alpha \)-free if for every such prefix \( \tau \) of \( S \), there is a prefix \( \tau' \) of the firing sequence of a configuration sequence \( \eta' \) of \( S \) such that:

i. For \( 1 \leq i \leq n \), \( \tau_i \) and \( \tau'_i \) are the firings of the same node in \( S \).

ii. \( \tau_n \) and \( \tau'_n \) are the opposite outcomes of the branching control.

iii. For \( 1 \leq k < n \), if \( \tau_k \) and \( \tau'_k \) are the firings of a branching control \( \lambda \) in \( S \), \( \tau_k \) and \( \tau'_k \) must be identical outcomes of \( \lambda \).

Theorem 2.3-3 A flowchart schema \( S \) is free if and only if it is \( \alpha \)-free.

Proof:

(if) It is obvious that if \( S \) is not free, \( S \) is not \( \alpha \)-free.

(only if) It is obvious that if \( S \) is not \( \alpha \)-free, \( S \) is not free.

Q.E.D.

A redundant decision structure in a wdfs \( S \) is a decision structure \( D \) whose decision outcome is always \( D^T \) or is always
Theorem 2.3-4 If $S$ is a free wfdfs which does not contain any redundant decision structure, $S$ is $\alpha$-free.

Proof:

Let $\tau_1 \ldots \tau_n$ be a prefix of a firing sequence $\tau$ of $S$ under interpretation $I$ and input $In$, $\tau_n$ being the outcome of a decision structure decision. Let $\tau_{il}' \ldots \tau_{ik}'$ be the outcomes of the input decisions to $\tau_n$.

i. Assume $\tau_n = \tau_T$. By hypothesis there exists a combination of input decision outcomes $\tau_{il}' \ldots \tau_{ik}'$ such that the decision structure decision under these inputs has outcome $D^F$. Modify $I$ to $I'$ such that $I$ is identical to $I'$ except that under $I'$:

- $\tau_{ij}$ has the same outcome as $\tau_{ij}'$, $1 \leq j \leq k$.

Under interpretation $I'$ and input $In$, $\tau_1 \ldots \tau_{n-1} D^F$ is the prefix of a firing sequence.

The consistency of $I'$ follows from the freedom of $S$.

ii. Assume $\tau_n = \tau_T$, then we may proceed as in (i) to construct $I'$.

Q.E.D.

The converse of Theorem 2.3-4 does not hold. Figure 2.11 gives an example of a wfdfs which is $\alpha$-free but not free. This difference between flowchart schemas and wfdfs's is due to the use of decision structures as branching controls in wfdfs's.
Openness

A program schema $S$ is open if given any branching control $B$ in $S$, there is a computation $C_1$ of $S$ in which a decision by $B$ has outcome $B^T$, and there is a computation $C_2$ of $S$ in which a decision by $B$ has outcome $B^F$.

Completeness

A program schema $S$ is complete if every finite prefix
Figure 2.12 A wdfds which is open but not complete

Figure 2.13 A wdfds which is complete but not open
of a computation of $S$ can be extended to a finite computation of $S$.

In an open schema every elementary statement or actor participates in some computation. If a program schema $S$ is complete, at any point in a computation of $S$ there is a set of conditions which if satisfied will cause $S$ to terminate. An example of a wfdfs which is open but not complete is given in Figure 2.12. A wfdfs which is complete but not open is shown in Figure 2.13.

From the definition of freedom, $\alpha$-freedom, openness and completeness, Theorem 2.3-5 and 2.3-6 should be obvious.

**Theorem 2.3-5** Let $S_1$, $S_2$ be flowchart schemas.

i. $S_1$ is free if and only if $S_1$ is $\alpha$-free.

ii. If $S_1$ is free, $S_1$ is open.

iii. If $S_1$ is free and contains no predicate free loop, $S_1$ is open and complete. [1]

**Theorem 2.3-6** Let $S_1$, $S_2$ be wfdfs's.

i. If $S_1$ is free and contains no redundant decision structure, $S_1$ is $\alpha$-free.

ii. If $S_1$ is $\alpha$-free, $S_1$ is open and complete.

A summary of the relationships between free program

[1] A predicate free loop is a loop in a flowchart schema which does not contain any predicate statement.
Figure 2.14 Relationship between free, open and complete program schemas.

schemas, open program schemas and complete program schemas is presented in Figure 2.14.

Theorem 2.3-7 Let $S_1$, $S_2$ be equivalent program schemas. $S_1$ is complete if and only if $S_2$ is complete.

Proof:

Assume that $S_1$ is not complete. Then there is a finite prefix $\xi$ of a computation sequence $\eta$ of $S_1$ that cannot be extended to a finite computation sequence of $S_1$. Let $I$ be the free interpretation under which $\eta$ is a computation sequence for $S_1$. Let $A$ be the set of decision outcomes contained in $\xi$, under $I$.

$$A = \{ p^k(d_1, \ldots, d_k) = T \mid p^k \in \text{Pred}, \quad d_1, \ldots, d_k \in D, \text{ the domain associated with } I, \quad p^k(d_1, \ldots, d_k) \text{ is a decision made in } \xi \text{ and has outcome } T \text{ under } I \}$$
\[ \bigcup \{ p^k(d_1, \ldots, d_k) = F \mid p^k \in \text{Pred}, \]
\[ d_1, \ldots, d_k \in D, \text{ the domain associated with } I, \]
\[ p^k(d_1, \ldots, d_k) \text{ is a decision made in } \xi \text{ and has outcome } F \]
\[ \text{under } I \} \]

A is finite since \( \xi \) is finite.

Consider the set \( H \) of all free interpretations consistent with \( I \). Let \( h \) be a free interpretation, \( h \in H \). \( S_1 \) diverges under \( h \). Since \( S_1, S_2 \) are equivalent, \( S_2 \) also diverges under \( h \). Let \( \eta_h \) be the computation sequence of \( S_2 \) under \( h \). Let \( A_h \subseteq A \) be the subset of \( A \) which is contained in \( \eta_h \). Since \( A \) is finite, \( A_h \) is finite and hence there is a finite prefix \( \xi_h \) of \( \eta_h \) which contains \( A_h \). \( \xi_h \) is then the finite computation sequence prefix of \( S_2 \) that cannot be extended to any finite computation sequence. \( S_2 \) hence is not complete.

Similarly we can prove that if \( S_2 \) is not complete, \( S_1 \) cannot be complete. Thus \( S_1 \) is complete if and only if \( S_2 \) is complete.

Q.E.D.

**Theorem 2.3-8** Every complete program schema is equivalent to an open and complete program schema.

**Proof:**

Let \( S \) be a complete flowchart schema. If \( S \) is not open, there is a predicate statement \( P \) in \( S \) such that
(i) P is never executed, or
(ii) P always has outcome T, or
(iii) P always has outcome F.

In each of these cases S can be modified to an equivalent flowchart schema S' which does not contain P, as follows:

(i) Mark a set of statements of S:
   (a) Mark P.
   (b) If every successor of statement \( S_1 \) is marked, mark \( S_1 \).
   (c) If every predecessor of statement \( S_1 \) is marked, mark \( S_1 \).

Remove all the marked statements from S.

(ii) Mark a set of statements of S:
   (a) Mark the F-successor of P.
   (b) Same as (i)(b).
   (c) Same as (i) (c).

Remove all the marked statements and P from S.
Connect every predecessor of P to the T-successor of P.

(iii) Mark a set of statements in S:
   (a) Mark the T-successor of P.
   (b) Same as (i)(b).
   (c) Same as (i)(c).

Remove all the marked statements and P from S.
Connect every predecessor of P to the F-successor of P.
If $S$ is a wfdfs that is not open, there is at least one decision structure $C$ in $S$ such that
(i) $C$ is never enabled, or
(ii) $C$ has outcome $T$ whenever it is enabled, or
(iii) $C$ has outcome $F$ whenever it is enabled.

For case (i) and (iii), $S$ can be modified to an equivalent wfdfs $S'$ by removing $C$ and some subschemas in $S$ using procedures similar to those described. For case (ii), some complications arise if $C$ controls an iteration subschema of $S$. Since $S$ is complete, this is impossible. Hence for all cases, $C$ can be removed from $S$ to obtain an equivalent wfdfs.

Given a complete program schema $S$, there is thus an open and complete schema $S'$ which is equivalent to it. Note that this proof does not imply that there is a procedure which transforms any complete schema into an equivalent open and complete schema. It merely demonstrates the existence of such an equivalent schema.

Q.E.D.
Chapter Three

Comparative Schematology

3.0 Introduction

In this chapter we compare the expressive power of subclasses of flowchart schemas and wdfds's. In Section 3.1 we establish the equivalence in expressive power between flowchart schemas and wdfds's. Algorithms are presented for translating a flowchart schema into an equivalent wdfds and for translating a wdfds into an equivalent flowchart schema. In Section 3.2 we prove that the class of free flowchart schemas properly contains the class of free wdfds's and the class of $\alpha$-free wdfds's. In Section 3.3 classes of open and complete program schemas are studied.

3.1 Translation between flowchart schemas and wdfds's

There are three steps involved in translating a flowchart schema into an equivalent wdfds:
1. Translate a flowchart schema S into an equivalent normal form flowchart schema (abbreviated nfs) T. [Engeler 71]
2. Translate a nfs T into a well-formed flowchart schema (abbreviated wfs) W. [Ashcroft & Manna 71]
3. Translate a well-formed flowchart schema W into a wdfds Z.

3.1.1 Translating a flowchart schema into normal form

A flowchart schema is a nfs if its body is a normal-form block (abbreviated nf-block) as shown in Figure 3.1.

A nf-block is defined recursively as follows:
1. A basic nf-block is any acyclic, tree-like, single entrance
subschema of a flowchart schema. An example of a basic nf-block is shown in Figure 3.2.

2. Composition of nf-blocks

If $B_1$, $B_2$ are nf-blocks, $B_3$ in Figure 3.3 is a nf-block formed by composition from $B_1$ and $B_2$.

3. Loop formation

If $B_1$ is a nf-block, $B_2$ in Figure 3.4 is a nf-block formed by loop formation from $B_1$.

4. A nf-block is a subschema derived from a finite number of steps from basic nf-blocks using rules 1, 2 and 3.
Informally a normal form flowchart schema is a flowchart schema with no forward jumps. The graph structure of a normal form flowchart schema is shown in Figure 3.5.

legal branches

illegal branches

Figure 3.5 Structure of a normal form flowchart schema
Given a flowchart schema $S$, an equivalent nfs $T$ can be constructed from $S$ using Algorithm 3.1 Normalize. Nodes in $T$ are copies of nodes in $S$. Initially $T$ contains 2 nodes, $START_T$ and $HALT_T$, which are copies of the START statement ($START_S$) and the HALT statement ($HALT_S$) in $S$, respectively. To construct $T$, Normalize is applied to $START_S$ and $START_T$. The translation is illustrated in Figure 3.6. If we label every node $s$ in $S$ by a unique label, and label every copy of $s$ in $T$ by the same label, then $S$ and $T$ will have the same set of labelled paths. It follows that $T$, as constructed, is equivalent to $S$.

Definition: In a flowchart schema $S$, a node $S_1$ is an ancestor of a node $S_2$ iff there is a path $p$ in $S$, $p = p_0 \ldots p_1 \ldots p_n$, such that

- $p_0$ is the START statement.
- $p_i$ is $S_1$.
- $p_n$ is $S_2$.

for all $j$, $0 < j < n$, $p_j \neq S_2$. 

Figure 3.6 Constructing a nfs using Normalize
The following functions are used in Algorithm 3.1:
Snode is a node in S, Tnode is a node in T. Start_{S}, Start_{T} are the unique Start statements and Halt_{S}, Halt_{T} are the unique Halt statements in S and T, respectively.

Copy(Snode) - makes a copy of Snodes
Successor(Snode), T-Successor(Snode), F-Successor(Snode) - gets the unique successor, the T-branch successor and the F-branch successor of Snode, respectively.
Add-branch(Tnode1, Tnode2), Add-T-branch(Tnode1, Tnode2), Add-F-branch(Tnode1, Tnode2) - adds the unlabelled edge (Tnode1, Tnode2), the edge (Tnode1, Tnode2) labelled by T, or the edge (Tnode1, Tnode2) labelled by F, to T, respectively.

Algorithm 3.1 Normalize(Snode, Tnode):

If Snode is a Start statement or an Assignment statement, then
Step 1: x ← Successor(Snode)
Step 2: If x is Halt_{S}, Add-branch(Tnode, Halt_{T}) and return.
Step 3: If Tnode has an ancestor node z in T and z is a copy of Snode, Add-branch(Tnode, z) and return.
Step 4: Otherwise
   y ← Copy(Snode);
   Add-branch(Tnode, y);
   Normalize(x,y); return;
If Snod is a Predicate statement then

Step 1: Perform steps 1, 2, 3 and 4 above with the functions Successor and Add-branch replaced by T-Successor and Add-T-branch respectively.

Step 2: Perform steps 1, 2, 3 and 4 above with the functions Successor and Add-branch replaced by F-Successor and Add-F-branch respectively.

End of Algorithm 3.1

The construction terminates because every path through S either terminates on Halt_S or loops on itself. An example of how the algorithm works is shown in Figure 3.

Figure 3.7 Normalization of flowchart schemas
3.1.2 Translating nfs's into wfs's

The class of wfs's, as the class of wdfs's, is a model for "gotoless" computer programs. A wfs is a flowchart schema whose body is a well-formed block (abbreviated wf-block). A wf-block is a single-entry, single-exit subschema, defined recursively as follows:
1. The empty schema, denoted by $\emptyset$, is a wf-block.
2. An assignment statement is a wf-block.
3. Linear Concatenation: If $B_1$, $B_2$ are wf-blocks, a linear concatenation of $B_1$ and $B_2$, as shown in Figure 3.8, is a wf-block.
4. Conditional Composition: If $B_1$, $B_2$ are wf-blocks, $B$ formed from $B_1$, $B_2$ and a branching control (defined below), as shown in Figure 3.9, is a wf-block.
5. Iteration: If $B_1$ is a wf-block, $B$ formed from $B_1$ and a branching control as shown in Figure 3.10 is a wf-block.
6. A wf-block is any subschema formed from the basic blocks in (1) and (2) in a finite number of steps using (3), (4) and (5).

![Figure 3.8 Linear Concatenation](image1)

![Figure 3.9 Conditional Composition](image2)

![Figure 3.10 Iteration in wfs](image3)
As in wdfs's, branching in a wfs is controlled by predicate decisions or by the result of the evaluation of a boolean expression whose basic constituents are predicate decisions. Let \( P = \{ P^i_j(X_1, \ldots, X_i) \mid P_j \in \text{Pred}, X_k \in \text{Var}, 1 \leq k \leq i \} \) be the set of simple predicate statements. Syntactically a branching control \( <BC> \) in a wfs is an expression generated by the following BNF:

\[
<BC> ::= P \mid P \land <BC> \mid P \lor <BC> \mid \neg <BC>
\]

Semantically, given an interpretation \( I \), the outcome of a branching control decision by a branching control \( <BC> \) is obtained by:

(i) evaluating the constituent predicate decisions \( (P) \) under \( I \).

(ii) applying the boolean function specified by \( <BC> \) to the predicate decision outcomes.

Value sequences, execution sequences and properties of wfs's are defined as in Section 2.1 and Section 2.3, with the straightforward generalization of substituting branching controls for simple predicates in these definitions. Properties of wfs's are not studied in this thesis, but we note in passing that an \( \alpha \)-free wfs need not be free, for the same reason that an \( \alpha \)-free wdfs need not be free.

Our next goal is to translate nfs's into wfs's. The translation algorithm, Algorithm 3.2, is based on an algorithm due to Ashcroft and Manna [Ashcroft & Manna, 71]. Algorithm 3.2 describes precisely how to apply the ideas
embodied in Ashcroft and Manna's algorithm to translate nfs's into wfs's. The major distinction between nf-blocks and wf-blocks (bodies of nfs's and wfs's respectively) is the number of exits they may have. A nf-block has a single entry point and may have more than one exit. A wf-block is a single-entry, single-exit block. Algorithm 3.2, when applied to a nf-block B with m exits, generates:

(i) a wf-block G that simulates the "loops" in B.
(ii) for the i-th exit, 1≤i≤m, 2 blocks αᵢ and βᵢ. αᵢ is a sequence of assignment statements. βᵢ is a sequence of assignment statements $\hat{\theta}_i$ followed by a branching control $\theta_i$.

The blocks G, αᵢ's and βᵢ's are constructed so that they can simulate any computation by B. Informally if B is executed, the exit taken by B can be determined by executing G and then the βᵢ's. If B is entered and exits at its i-th exit, the computation performed by B can be simulated by executing G and then αᵢ. If we furthermore impose the requirement that none of the βᵢ's modified the variable values accessed by the αᵢ's and the other βᵦ's, k≠i, then B can be simulated by a wf-block constructed from G, αᵢ's and βᵢ's as shown in Figure 3.11. Algorithm 3.2 constructs G, αᵢ's and βᵢ's for a given nf-block B so that the wfs W and the nfs T in Figure 3.11 are indeed equivalent.

To give a precise description of the relationship between B, G, αᵢ's and βᵢ's, and of Algorithm 3.2, we need some additional terminology.
Let $I$ be an interpretation with associated domain $D$, and undefined object $\Omega$. A memory state $\sigma_I$ is a total function $\sigma_I: \text{Var} \to D \cup \{\Omega\}$. Memory states are modified by executing assignment statements and are not modified by evaluating branching controls. Any block, a nf-block or a wf-block, can be looked at as a mapping, under an interpretation $I$, between memory states.

Let $B$ be a nf-block, and $G$ a wf-block. Let $V$ be a set of variables, $V \subseteq \text{Var}$. Let $\sigma_I, \sigma'_I$ be memory states under interpretation $I$. Let $\beta$ be a block consisting of a sequence of assignment statements $\delta$ followed by a branching control $\theta$.

$V_B, V_Q$ denote the set of variables used in $B$ and $Q$, respectively.
\[ B, L_Q \] denote the set of variables on the left hand side of assignment statements in \( B \) and \( Q \), respectively. These are the variables which may be updated by executing \( B \) and \( Q \).

\[ R_Q, \] denote the set of variables appearing on the right hand side of assignment statements and in branching controls in \( B \) and \( Q \), respectively. These are the variables accessed in \( B \) and \( Q \) and their values determine the control flow through \( B \) and \( Q \) when \( B \) and \( Q \) are executed.

\[ B[\sigma_I] = (\sigma'_{I}, i) \] denotes that if \( B \) is entered under interpretation \( I \) and memory state \( \sigma_I \), \( B \) exits at its \( i \)-th exit with memory state \( \sigma'_I \).

\[ G[\sigma_I] \] denotes the memory state \( \sigma'_I \) such that if \( G \) is entered under interpretation \( I \) and memory state \( \sigma_I \), \( G \) exits with memory state \( \sigma'_I \).

\[ \beta[\sigma_G] = T(E) \] denotes that the branching control \( \theta \), evaluated in \( \xi[\sigma_G] \), has outcome \( T(E) \).

\[ \sigma_I = \sigma'_I \] denotes that \( \sigma_I(v) = \sigma'_I(v) \) for all \( v \in V \).

In the remainder of this subsection our definitions and lemmas hold for arbitrarily chosen interpretations. To simplify the presentation, all references to interpretations are omitted.

**Definition** Let \( B \) be a nf-block with \( m \) exits in a nfs \( T \). Let \( G \) be a wf-block. \( \alpha_i, \beta_i, 1 \leq i \leq m \), are \( 2m \) blocks. Each \( \alpha_i \) is a sequence of assignment statements. Each \( \beta_i \) is a sequence of assignment statements \( \xi_i \) followed by a branching
control $\theta_i$. Let $\sigma_B$, $\sigma'_B$ and $\sigma_G$ be memory states. $G$, $\alpha_i$'s and $\beta_i$'s simulate $B$ if

(i) **Termination** If $\sigma_B = v_T = \sigma_G$, $B$ terminates on $\sigma_B$ iff $G$ terminates on $\sigma_G$.

(ii) **Determining the exit taken by $B$ using $G$ and $\beta_i$'s**

Let $\sigma_B = v_T = \sigma_G$.

$$B[\sigma_B] = (\sigma'_B, i) \text{ iff }$$

$$\beta_k[G[\sigma_G]] = E, \text{ for } 1 \leq k < i, \text{ and } \beta_i[G[\sigma_G]] = T$$

(iii) **Simulating the computation of $B$ using $G$ and $\alpha_i$'s**

If $\sigma_B = v_T = \sigma_G'$ and $B[\sigma_B] = (\sigma'_B, i)$

then $\sigma'_B = v_T = \alpha_i[G[\sigma_G]]$

(iv) **Non-interference between the $\alpha_i$'s and the $\beta_i$'s**

This condition states that the subcomputations by the $\beta_i$'s do not affect the variables used by the other $\beta_k$'s, $k \neq i$, and the $\alpha_i$'s.

For $1 \leq i, k \leq m$, $k \neq i$,

$$L_{\alpha_i} \cap R_{\beta_k} = \{\}, \text{ the empty set.}$$

For $1 \leq i, j \leq m$,

$$L_{\alpha_i} \cap R_{\alpha_j} = \{\}, \text{ the empty set.}$$

**Lemma 3.1** Let $B$ be a nf-block, $G$ be a wf-block, $\alpha_i$'s, $\beta_i$'s be blocks, as given in the above definition. If $G$, $\alpha_i$'s and $\beta_i$'s simulate $B$, then the nfs $T$ and the wfs $W$ in Figure 3.11 are equivalent.

**Proof:**

$W$ and $T$ are equivalent because:
(i) $B$ terminates iff $G$ terminates. Hence $W$ terminates iff $T$ terminates.

(ii) $W$ and $T$ have identical input statements. Hence $B$ and $G$ are entered under memory states $\sigma_B$ and $\sigma_G$ which are equivalent with respect to $V_T$ respectively. If $B[\sigma_B] = (\sigma'_B, i)$, then $\beta_1[G[\sigma_G]], \ldots, \beta_{i-1}[G[\sigma_G]]$ all have outcome $E$, and $\beta_i[G[\sigma_G]]$ has outcome $T$. $\alpha_i$ is then executed in $T$. Due to the non-interference condition, if $\sigma'_G$ is the state under which $\alpha_i$ terminates,

$$\sigma'_G \models_{V_T} \alpha_i[G[\sigma_G]] \models_{V_T} \sigma'_B$$

$W$ and $T$ have identical output statements, hence produce identical outputs.

Q.E.D.
Algorithm 3.2 Generate(B)

/* Comments on Algorithm 3.2:

Given a nf-block B in a nfs T, the algorithm generates:

(i) for each subblock $B_j$ of B, a set of blocks $G_j$,
    $\alpha^j_i$'s and $\beta^j_i$'s which simulate $B_j$.

(ii) for B, a set of blocks G, $\alpha^i$'s and $\beta^i$'s which simulate B. These blocks are constructed from the blocks
    generated in (i). The construction techniques applied depend on the block type (a basic block, a block
    formed by loop formation or a block formed by composition) of B.

Algorithm 3.2 is presented as 3 constructions, each
applicable to the specific block type identified for B.
Formal and informal arguments are included in each
construction to explain why the wf-blocks generated
simulate B. In these arguments we implicitly invoke the
hypothesis (an induction hypothesis) that Generate,
applied to subblocks of B, constructs blocks which
simulate these subblocks. The induction basis is
established by showing that Generate correctly constructs
blocks to simulate basic nf-blocks. */

/* Functions used in Generate:

Gensym() - Gensym is a variable symbol generator. Each time
Gensym is activated, it returns a variable
symbol X which is not used in T and has not been
returned in any previous activation of Gensym.
Transform(P) - P is a block which consists of a sequence of assignment statements $\theta$ followed by a branching control $\theta$. From P, Transform constructs a block Q which can be executed to determine the outcome of $\theta$ without modifying the value of any variable used in any other block. Q consists of a sequence of assignment statements $\theta'$ followed by a branching control $\theta'$ such that:

(i) $L_{\theta'}$ is a set of variable symbols which appear only in $\theta'$. They are generated by calls to Gensym.

(ii) For all memory states $\sigma, \sigma'$ such that $\sigma = v_T = \sigma'$,

$$Q[\sigma] = T \iff P[\sigma'] = T$$

Construction 3.2-1 Generate blocks which simulate a basic nf-block B.

B is a basic block with m-exits. A basic block contains no subblocks and has a tree structure. There is a unique path leading from the block entry point of B to an exit of B. For $1 \leq i \leq m$, let $P_i$ be the path which leads from the block entry point of B to the i-th exit of B.

Construction of $G, \alpha_i$'s and $\beta_i$'s

$G$: G is the null wf-block $\emptyset$. $\emptyset$ always terminates and for all $\sigma$, $\emptyset[\sigma] = \sigma$.

$\alpha_i$: $\alpha_i$ is the linear concatenation of all the assignment statements on the path $P_i$.

$\beta_i$: $\beta_i$ is used to determine whether B exits at its i-th exit when entered with memory state $\sigma$, and is a concatenation
of a sequence of assignment statements $\delta_i$ with a branching control $\theta_i$. $\theta_i$ is a conjunction of the conditions which cause B to take the $i$-th exit. $\delta_i$ is a sequence of assignment statements that uses new variables to compute the values tested by $\theta_i$. If $P_i$ contains $n$ predicate statements, then for every such predicate statement $D_j$

Let $F_j$ be the concatenation of all the assignment statements lying on $P_i$ between the block entry point of B and $D_j$.

Let $H_j$ be the concatenation of $F_j$ and $D_j$.

Apply Transform to $H_j$ to construct a block $H'_j$ which is the concatenation of a sequence of assignment statements $F_j'$ and a branching control $D'_j$. $H_j$ and $H'_j$ are related as described in the specification of Transform.

$\delta_i$ is the concatenation of all the $F_j'$ constructed above.

$\theta_i$ is a branching control of the form "$C_1 \land ... \land C_n$", where for $1 \leq k \leq n$,

$$C_k = \begin{cases} D'_j & \text{if on the path } P_i, \text{ the edge emanating from } D_j \text{ is labelled by } T. \\ D_j & \text{if on the path } P_i, \text{ the edge emanating from } D_j \text{ is labelled by } E. \\ -D'_j & \text{if on the path } P_i, \text{ the edge emanating from } D_j \text{ is labelled by } E. \end{cases}$$

Simulation of $B$ by $G$, $\alpha_i$'s and $\beta_i$'s

It is straightforward to verify that $G$, $\alpha_i$'s and $\beta_i$'s satisfy the 4 conditions for simulating $B$, using the specification of Transform and the construction steps detailed above. Here we give an example of using Construction 3.2-1 in Figure 3.12.

End of Construction 3.2-1
Figure 3.12 Illustration of Construction 3.2-1
Construction 3.2-2 Generate blocks to simulate a nf-block $B$ formed by composition from nf-blocks $B_1$ and $B_2$.

$B$, formed by composition from $B_1$ and $B_2$, has the structure shown in Figure 3.3. We divide this case into 2 subcases:

(i) $B_2$ is a basic block.
(ii) $B_2$ is not a basic block.

Construction 3.2-2-1 Generate blocks to simulate a nf-block formed by composition from nf-blocks $B_1$ and $B_2$, where $B_2$ is a basic block.

Apply Generate to $B_1$ to construct blocks $G_1$, $a_1^1$'s and $b_1^1$'s, $1 \leq i \leq m$, (Figure 3.3) to simulate $B_1$.

Apply Generate to $B_2$ to construct blocks $G_2$, $a_1^1$'s and $b_1^1$'s, $1 \leq i \leq n$, (Figure 3.3) to simulate $B_2$. From Construction 3.2-1, $G_2$ is the null wf-block $\emptyset$.

Construction of $G$, $a_1^1$'s and $b_1^1$'s

$G$: $G$ is the concatenation of $G_1$ with $G_2$, which is simply $G_1$, since $G_2$ is $\emptyset$.

From the facts:

(i) $G$ is $G_1$, $G_2$ is $\emptyset$.
(ii) $G_1$, $a_1^1$'s and $b_1^1$'s simulate $B_1$. $G_2$, $a_1^2$'s and $b_1^2$'s simulate $B_2$.

and the definition of simulation, the structure of $B$, we can deduce the following observations:

1. Let $\sigma_B = V_T = \sigma_G$.
   If $B[\sigma_B] = (\sigma', i)$, and $1 \leq i \leq n$, then
   (i) $\sigma'_B = V_T = a_1^2[\sigma_1^1[G[\sigma_G]]]$
   (ii) $b_1^1[G[\sigma_G]] = T$,
\( \beta_i^2 [\alpha_i^1 [G[\sigma_G]]] = T \)

(iii) For \( 1 \leq k < i \),
\( \beta_k^2 [\alpha_k^1 [G[\sigma_G]]] = F \)

(2) Let \( \sigma_B = \nu_T = \sigma_G \).

If \( B[\sigma_B'] = (\sigma_B', i) \), and \( n+1 \leq i \leq m+n-1 \), then

(i) \( \sigma_B' = \nu_T = \alpha_{i-(n-1)}^1 [G[\sigma_G]] \)

(ii) \( \beta_i^{1} [G[\sigma_G]] = T \)

(iii) For \( n \leq k < i \),
\( \beta_k^{1} [G[\sigma_G]] = F \)

These observations serve both as motivation and justification for the following constructions:

\( \alpha_i^1 \): For \( 1 \leq i \leq n \), taking
the \( i \)-th exit of \( B \) implies taking the first exit of \( B_1 \), entering \( B_2 \), and then taking the \( i \)-th exit of \( B_2 \).

\( \alpha_i^1 \) is the concatenation of \( \alpha_1^1 \) and \( \alpha_i^{2} \) (Observation 1(i))

For \( n+1 \leq i \leq m+n-1 \), taking the \( i \)-th exit of \( B \) implies
taking the \( (i-(n-1)) \)-th exit of \( B_1 \), without entering \( B_2 \).

\( \alpha_i^{1} \) is \( \alpha_i^{1-(n-1)} \) (Observation 2(i))

Thus \( \alpha_{n+1}^1 \) is \( \alpha_2^1 \), \( \alpha_{n+2}^1 \) is \( \alpha_3^1 \), and so on.

\( \beta_i^1 \): For \( 1 \leq i \leq n \), determining whether the \( i \)-th exit of \( B \) is taken requires determining whether the first exit of \( B_1 \) is taken, and then determining whether the \( i \)-th exit of \( B_2 \) is taken. \( \beta_i^1 \) is constructed as follows (Observation 1(ii)):

(i) Apply Transform to \( \beta_1^{1} \) to construct a block \( L \) which
is the concatenation of a sequence of assignment statements \( F \) with a branching control \( D \).

Evaluating \( D \) enables us to determine whether \( B_1 \)
exits at its first exit, using a set of variables not used in other blocks.

(ii) Concatenate the two sequences of assignment statements $a_1^1$ and $a_1^2$ to form a block $H_1$.
Concate $H_1$ and $a_1^2$ to form a block $K_1$.
Apply Transform to $K_1$ to construct a block $L_1$ which is the concatenation of a sequence of assignment statements $F_i$ with a branching control $D_i$.
Evaluating $D_i$ enables us to determine whether $B_2$ exits at its $i$-th exit, again using a set of new variables.

$\delta_i$ is the concatenation of $F$ with $F_i$.
$\theta_i$ is the conjunction of $D$ and $D_i$.
$\beta_i$ is the concatenation of $\delta_i$ and $\theta_i$.

For $n+1 \leq i \leq m+n-1$, determining whether $B$ has taken its $i$-th exit only requires determining whether $B_1$ has taken its $(i-(n-1))$-th exit.

$\beta_i$ is $\beta_1^{i-(n-1)}$. (Observation 2(ii))

Simulating $B$ with $G$, $a_1$'s and $\beta_i$'s:

Termination $B$ terminates iff $B_1$ terminates. $B_1$ terminates iff $G_1$ terminates. $G$ is $G_1$. Hence $B$ terminates iff $G$ terminates.

Determining the exit taken by $B$ using $G$ and $\beta_i$'s

From the observations made above and the construction of $\beta_i$'s, it should be straightforward to see that for $1 \leq i \leq m+n-1$, if $\sigma_B = V_T = \sigma_G$, then $B[\sigma_B] = (\sigma', i)$ iff $\beta_i[G[\sigma_G]] = T$ and for $1 \leq k \leq m+n-1$, $k \neq i$, $\beta_k[G[\sigma_G]] = F$. 
Simulating the computation of $B$ by $G$ and $\alpha_i$'s

From the observations made above and the construction of $\alpha_i$'s, it should be obvious that for $1 \leq i \leq m+n-1$, if $\sigma_B = \nu_T = \sigma_G$ and $B[\sigma_B] = (\sigma'_B, i)$,

$$\sigma'_B = \nu_T = \alpha_i [G[\sigma_G]]$$

Non-interference between the $\alpha_i$'s and the $\beta_i$'s

Every one of the $\beta_i$'s are, or have been, constructed using Transform. The non-interference condition is thus trivially satisfied.

End of Construction 3.2-2-1

Construction 3.2-2-2 Generate blocks to simulate a nf-block $B$ formed by composition from $B_1$ and $B_2$, where $B_2$ is not a basic block.

Apply Generate to $B_1$ to construct $G_1$, $\alpha_1^1$'s and $\beta_1^1$'s to simulate $B_1$.

Apply Generate to $B_2$ to construct $G_2$, $\alpha_1^2$'s and $\beta_1^2$'s to simulate $B_2$.

Construction of $G$, $\alpha_i$'s and $\beta_i$'s

$G$: Since $B_2$ is not a basic block, $G_2$ is non-null. $G$ is constructed from $G_1$, $\alpha_1^1$ and $G_2$. If in $B$, $B_2$ is entered, $\alpha_1^1$ and $G_2$ in $G$ is also executed. $\alpha_1^1$ and $G_2$ may modify some of the variables accessed by the $\beta_1^1$'s. This may cause the evaluation of the $\beta_1^1$'s to have outcomes which do not correctly determine the exit taken by $B_1$, and in turn the exit taken by $B$. To account for the side-effects produced by executing $\alpha_1^1$ and $G_2$, new variables are intro-
duced to store the values of the variables that may be modified by $a_1^1$ and $G_2$. These variable values are stored at the time $G_1$ terminates. The accompanying modifications to the $\beta_1^1$'s are described below. $G$ is constructed as follows:

Let $Y_1$, ..., $Y_n$ be the set of variables appearing on the LHS of the assignment statements in $a_1^1$ and $G_2$. Let $Z_1$, ..., $Z_n$ be $n$ new variable symbols generated by $n$ activations of Gensym. **Savestate** is a sequence of $n$ assignment statements (Figure 3.13), $Z_k \leftarrow Y_k$, 1$\leq k \leq n$. **Savestate** saves the variable values which may be tested at the termination of $G_2$ to determine if the flow of control leaves $B_1$ at other than its first exit. $G$ is the wf-block constructed from $G_1$, $G_2$, $a_1^1$, $\beta_1^1$ and **Savestate** as shown in Figure 3.13.

![Flowchart](Image)

**Figure 3.13 G for Construction 3.2-2-2**
From the fact that $G_1$, $\alpha_1^1$'s and $\beta_1^1$'s correctly simulate $B_1$ and $G_2$, $\alpha_1^2$'s and $\beta_1^2$'s correctly simulate $B_2$, together with the structure of $B$ and the structure of $G$, we can deduce the following observations:

(1) Let $\sigma_B = V_T = \sigma_G$.
   If $B[\sigma_B] = (\sigma_B', i)$ and $1 \leq i \leq n$, then
   (i) $G[\sigma_G] = V_T = G_2[\alpha_1^1[G_1[\sigma_G]]]$
       $\sigma_B' = V_T = \alpha_1^2[G_2[\alpha_1^1[G_1[\sigma_G]]]] = V_T = \alpha_1^2[G[\sigma_G]]$
   (ii) $\beta_1^1[G_1[\sigma_G]] = T$ and $\beta_1^2[G[\sigma_G]] = T$
   (iii) For $1 \leq k < i$, $\beta_k^2[G[\sigma_G]] = F$

(2) Let $\sigma_B = V_T = \sigma_G$.
   If $B[\sigma_B] = (\sigma_B', i)$ and $n + 1 \leq i \leq m + n - 1$, then
   (i) $G[\sigma_G] = V_T = G_1[\sigma_G]$
       $\sigma_B' = V_T = \alpha_1^{i-(n-1)}[G[\sigma_G]]$
   (ii) $\beta_1^{i-(n-1)}[G[\sigma_G]] = T$
   (iii) For $n \leq k < i$, $\beta_k^{i-(n-1)}[G[\sigma_G]] = F$

These observations serve both as motivation and justification for the following constructions:

$\alpha_1$: For $1 \leq i \leq n$, the $i$-th exit of $B$ is the $i$-th exit of $B_2$.

$\alpha_1$ is $\alpha_1^1$. (Observation 1(i))

For $n + 1 \leq i \leq m + n - 1$, the $i$-th exit of $B$ is the $(i - (n - 1))$-th exit of $B_1$.

$\alpha_1$ is $\alpha_1^{i-(n-1)}$. (Observation 2(i))

$\beta_1$: For $1 \leq i \leq n$, $\beta_1$ determines whether $B_1$ takes its $1$-st exit and whether $B_2$ takes its $i$-th exit. The first event can
be determined by evaluating $\beta_1^1$ when $G_1$ terminates. The second event can be determined by evaluating $\beta_2^2$ when $G_2$ terminates. (Observation 1(ii)) Since $\beta_1^1$ is evaluated when $G$ terminates, and $\alpha_1^1$, and $G_2$ may have side-effects, $\beta_1^1$ has to be modified before it can be used to construct $\beta_1^1$. For $1 \leq k \leq \kappa$, $y_k$ is a variable which may be modified by $\alpha_1^1$ or $G_2$, and $z_k$ saves the value of $y_k$ at the termination of $G_1$. Construct $\beta_1^1$ by replacing every access to $y_k$ (an occurrence of $y_k$ on the RHS of an assignment statement or an occurrence of $y_k$ in a branching control) in $\beta_1^1$ by an access to $z_k$. $\beta_1^1$ is a concatenation of $\phi_1^1$ and $\theta_1^1$. $\beta_2^2$, unmodified, is a concatenation of $\phi_2^1$ and $\theta_1^2$. $\phi_i$, the sequence of assignment statements which computes the values used to determine whether $B$ takes its $i$-th exit, is the concatenation of $\phi_1^1$ and $\phi_1^2$. $\theta_i$, the branching control which uses the values computed by $\phi_i$ to determine whether $B$ takes its $i$-th exit, is the conjunction of $\theta_1^1$ and $\theta_1^2$. $\beta_i^1$ is the concatenation of $\phi_i$ and $\theta_i$.

For $n+1 \leq i \leq m+n-1$, the $i$-th exit of $B$ is the $i-(n-1)$-th exit of $B_1$. $\beta_i^1-(n-1)$ can be evaluated when $G_1$ terminates to determine whether $B_1$ takes its $i$-th exit. Since $\beta_i^1$ is evaluated when $G$ terminates, and $\alpha_1^1$, $G_2$ may produce side effects, $\beta_i^1-(n-1)$ must be modified to use the values saved by $\text{SaveState}$. Construct $\beta_i^1-(n-1)'$ by replacing accesses to $y_k$ in $\beta_i^1-(n-1)$ by accesses to $z_k$. $\beta_i^1$ is $\beta_i^1-(n-1)'$.

Simulation of $B$ using $G$, $\alpha_i's$ and $\beta_i's$

From the above observations and the way $G$, $\alpha_i's$ and $\beta_i's$ are constructed, it is again straightforward to show that $G$, $\alpha_i's$ and $\beta_i's$ satisfy the 4 conditions for simulating $B$. The
Construction 3.2-3 Generate blocks to simulate a nf-block B formed by loop formation.

A nf-block B formed by loop formation has the structure shown in Figure 3.4. The body of B is B₁. B "loops" until B₁ exits at other than its first exit at some iteration. Apply Generate recursively to construct blocks G₁, α₁₁'s and β₁₁'s to simulate B₁.

Construction of G, α₁₁'s and β₁₁'s

G: G is the wf-block shown in Figure 3.14.

Figure 3.14 G for Construction 3.2-3
Lemma 3.2 Let $\sigma_B = \nu = \sigma_G$. $B$, if entered with $\sigma_B$, terminates iff $G$, if entered with $\sigma_G$, terminates. If $B$ terminates,

$$B[\sigma_B] = (\sigma_B', i) \text{ iff }$$

for $1 \leq k \leq i$, $\beta_k^1[G[\sigma_G]] = \mathbb{F}$, and $\beta_{i+1}^1[G[\sigma_G]] = T$.

If $B[\sigma_B] = (\sigma_B', i)$, then $\nu = \alpha_{i+1}^1[G[\sigma_G]]$.

Proof:

The proof is a simple induction proof which is included in the Appendix.

Lemma 3.2 motivates and justifies the following constructions:

$\alpha_i$: For $1 \leq i \leq m-1$, $\alpha_i$ is $\alpha_{i+1}^1$.

$\beta_i$: For $1 \leq i \leq m-1$, $\beta_i$ is $\beta_{i+1}^1$.

Simulation of $B$ by $G$, $\alpha_i$'s and $\beta_i$'s

Using Lemma 3.2, it is straightforward to show that $G$, $\alpha_i$'s and $\beta_i$'s satisfy the 4 conditions for simulating $B$. The details are omitted.

End of Construction 3.2-3

End of Algorithm 3.2
3.1.3 Translating wfs's into wfdfs's

Wfs's and wfdfs's are formed using the same set of recursive rules of construction: composition, conditional composition and iteration. In wfdfs's data flow is not separated from control flow and there is no data path into or out of iteration subschemas and conditional subschemas except at the input and output link nodes. In wfs's the data paths across the wf-blocks defined by conditionals and iterations are created by storing and accessing data via variables. To translate wfs's into wfdfs's we have to identify the set of input data paths and the set of output data paths for any wf-block in a wfs.

Given a wf-block B:

The set of input variables of B is denoted by $\text{IN}_B$.

For each variable symbol $X$, $X \in \text{IN}_B$ iff $X$ is accessed by some statement $P_T$ in $B$, such that there is a path leading from the entry point of $B$ to $P_T$, and $X$ does not appear on the LHS of any assignment statement which is an ancestor of $P_T$.

The set of output variables of B is denoted by $\text{OUT}_B$.

For each variable symbol $X$, $X \in \text{OUT}_B$ iff $X$ appears on the LHS of some assignment statement in $B$. 
An environment for translating B is a set of labelled data links D in a wdfs Z such that

(i) Every variable in $\text{IN}_B$ is the label of a data link d in D.
(ii) Every data link in D is labelled by at least one variable in $\text{IN}_B$.

A wfs W is translated into an equivalent wdfs Z by applying Algorithm 3.3 to W and the empty environment which contains no node. Algorithm 3.3 is a recursive procedure which treats a wfs W as a special case and in general, when applied to a wf-block B in W and an environment E, modifies E by:

(i) Adding to E a wdfs $Z_B$ which performs the same computation as B. The input data links of $Z_B$ are the data links in E labelled by variable symbols from $\text{IN}_B$.
(ii) Labelling the output data links of $Z_B$ by variable symbols from $\text{OUT}_B$, and removing from E any label not in the set $\text{OUT}_B$.

If W is a wfs obtained by applying Algorithm 3.1 and Algorithm 3.2 to translate a flowchart schema S into a wfs, there will always be data links labelled by $\text{IN}_B$ in E when a block B in W is translated by applying Algorithm 3.3. This is because the composition rules for flowchart schemas eliminates the possibility of accessing undefined variables.
Algorithm 3.3  

WtoZ-Translate \((B, E')\) — Given a wfs or a \(\text{wfs-block} B\), modify \(E\) to \(E'\) which contains a subschema equivalent to \(B\).

1. \(B\) is a wfs.
   
   (i) For every input variable \(X\) of \(B\), add an I-operator \(J\) and an output data link \(d\) for \(J\) to \(E\). Label \(d\) by \(X\). Let the environment resulting from (i) be denoted by \(E'\).
   
   (ii) WtoZ-Translate \((\text{Body of } E, E')\)
   
   Let the environment resulting from (ii) be denoted by \(E''\).
   
   (iii) For every output variable \(Y\) of \(B\), join the data link in \(E''\) labelled by \(Y\) to an O-operator.

The translation is depicted in Figure 3.15.

\[
\text{START}(X_1, \ldots, X_m) \\
\text{WF-block } B \\
\text{HALT}(Y_1, \ldots, Y_n) \\
\]

\[
\text{WtoZ-Translate} \quad (B, E') \\
\]

\[
\text{I} \quad \text{I} \\
\text{O} \quad \text{O} \\
\]

Figure 3.15 Algorithm 3.3 applied to a wfs

2. \(B\) is a concatenation of an assignment statement and a \(\text{wfs-block } B_1\).
   
   Let the assignment statement be \(X \leftarrow H\).
   
   (i) If \(H\) is a variable symbol \(Y\), add the label \(X\) to the data
link d labelled by Y. Remove the label X, if any, from the data links in E.

If H consists of applying the function symbol $F^i_j$ to the variables $X_1, \ldots, X_i$, add an operator $Op$ labelled by $F^i_j$ to E (Figure 3.16). The $k$-th input data link to $Op$, for $1 \leq k \leq i$, is the data link in E labelled by $X_k$. Label the output data link of $Op$ by X and remove the label X, if any, from the data links in E.

(ii) Let the environment resulting from (i) be denoted by $E'$. 

\[
\text{WtoZ-Translate}(B_1, E')
\]

\[
X \xrightarrow{F^i_j(X_1, \ldots, X_i)} E
\]

Figure 3.16 Algorithm 3.3 applied to an Assignment statement

3. $B$ is a concatenation of $B'$ and $B_1$. $B'$ is formed from wf-blocks $B_1'$, $B_F'$ and branching control BC by conditional composition.

$B'$ has the structure shown in Figure 3.9, with BC as its branching control, $B_T'$ substituted for $B_1$ and $B_F'$ substituted for $B_2$. 
The translation is illustrated in Figure 3.17.

\[
\begin{align*}
I_{B_T} &= I_{B_T} \cup (\text{OUT}_{B_F} - \text{OUT}_{B_T}) \\
I_{B_F} &= I_{B_F} \cup (\text{OUT}_{B_T} - \text{OUT}_{B_F}) \\
V_L &= \text{the set of variables accessed by the branching control BC}
\end{align*}
\]

Let \( X \) be a variable symbol in \( \text{OUT}_{B_T} \cup \text{OUT}_{B_F} \) which does not label any data link in \( E \). Choose a data link in \( E \) arbitrarily and add the label \( X \) to the data link. This does not affect the result of the translation if \( B \) is a wf-block in a wfs \( W \) obtained by applying Algorithm 3.1 and Algorithm 3.2 to translate a flowchart schema \( S \). Due to the restrictions in the composition rules for flowchart schemas, if \( X \) is accessed in \( B_1 \), \( X \) will be the LHS of an assignment statement in \( B_T \) and of an assignment statement in \( B_F \) or \( X \) will be the LHS of an assignment statement in \( B_1 \) which is always executed before \( X \) is accessed.

The translation steps are:

(i) A decision structure is constructed using deciders and boolean operators to compute \( BC \).

(ii) Every data link \( d \) in \( D_T \) is the output data link of a T-gate whose input data link \( g \) is labelled by some variable symbol in \( I_{B_T} \). Label \( d \) with those labels of \( g \) from \( I_{B_T} \). Similarly label all the data links in \( D_F \).

(iii) \[
\text{WtoZ-Translate}(B_T, D_T) \\
\text{WtoZ-Translate}(B_F, D_F)
\]

(iv) After (iii), the data link sets \( E_T \) and \( E_F \) will be
Figure 3.17 Algorithm 3.3 applied to a block formed from conditional composition
labelled by variable symbols in \( \text{OUT}_{B_T} \) and \( \text{OUT}_{B_F} \) respectively. Some of the data links in \( D_T \) and \( D_F \) may still be labelled.

The set of labels in \( D_T \cup E_T \) or \( D_F \cup E_F \) is exactly the set \( \text{OUT}_{B_T} \cup \text{OUT}_{B_F} \).

Let \( y_i \) be a variable symbol in \( \text{OUT}_{B_T} \cup \text{OUT}_{B_F} \). The arc \( t_i \) is from the data link labelled by \( y_i \) in \( D_T \cup E_T \) and the arc \( f_i \) is from the corresponding data link in \( D_F \cup E_F \). Remove the label \( y_i \) from \( D_T \cup E_T \) and from \( D_F \cup E_F \). Label the output data link of the merge gate whose input arcs are \( t_i \) and \( f_i \) by \( y_i \). Remove the label \( y_i \) from \( E \).

(v) After (iv), let \( E' \) denote the set of labelled data links in \( E \cup G \) (Figure 3.17).

\( \text{WtoZ-Translate}(B_1, E') \)

4. \( B \) is a concatenation of \( B' \) and \( B_1 \). \( B' \) is formed from \( W^I \)-block \( B_L \) and the branching control \( BC \) by iteration.

\( B' \) has the structure shown in Figure 3.10, with \( BC \) as its branching control and \( B_L \) substituted for \( B_1 \).

The translation is illustrated in Figure 3.18.

\( V_L \) = the set of variables accessed by the branching control \( BC \)

\( LBL = \text{IN}_{B_L} \cup \text{OUT}_{B_L} \cup V_L \)

Let \( X \) be a variable symbol in \( \text{OUT}_{B_L} \) which does not label any data link in \( E \). Choose a data link in \( E \) arbitrarily and
add the label X to the data link. The translation steps are:

(i) A decision structure is constructed using deciders and boolean operators to compute BC.

(ii) Every data link a in A is the output data link of a merge gate which has an input data link labelled by some variable symbol in LBL. Label a with the labels of this input data link of the merge gate.

(iii) After (ii), every data link f in F is the output data link of a T-gate whose input data link a is labelled by some labels in LBL. Label f with the labels of a. Every data link h in H is the output data link of a F-gate whose input data link a is labelled by some labels in OUTBL. Label h with the labels of a from OUTBL.

(iv) \text{WtoZ-Translate}(BL, F)

(v) After (iv) the data link set G is labelled by variable symbols from OUTBL. For every ti, let di be the output link of the merge node one of whose input arc is ti. If di is labelled by y, y ∈ OUTBL, ti is from the data link in G labelled by y.
If di is labelled by y, y ∉ OUTBL, ti is from the data link in F labelled by y.

(vi) Every label y ∈ OUTBL is removed from the data link set E. Let E' denote the set of labelled data links in E U H.

\text{WtoZ-Translate}(BL, E')
Figure 3.18 Algorithm 3.3 applied to a block formed from iteration
5. In each of the 4 cases considered above, $B_1$ may be the null \( w_5 \)-block. Algorithm 3.3, when applied to the null \( W_5 \)-block, returns without modifying \( E \). In the translation process many nodes in the resulting \( wfdfs \)s may be introduced but are not connected to the \( 0 \)-operators by any directed path. To obtain the \( wfdfs \) \( Z \) equivalent to \( W \), all such nodes are removed.

End of Algorithm 3.3

Algorithm 3.3 is the last step in the translation from flowchart schemas into \( wfdfs \)s. We can now state Theorem 3.1.

**Theorem 3.1** Every \( m \)-\( n \) flowchart schema is equivalent to a \( m \)-\( n \) \( wfdfs \). Furthermore there is an algorithm to translate a \( m \)-\( n \) flowchart schema into an equivalent \( m \)-\( n \) \( wfdfs \).

We note that Algorithm 3.1, by splitting nodes, increases the number of nodes and hence the 'size' of a flowchart schema. In translating a normal form flowchart schema to a \( wfs \), some look-ahead computation is required in the \( wfs \) to simulate the control flow of the \( nfs \). In this sense, then, the equivalent \( wfdfs \) is 'less efficient' than the given flowchart schema.
3.1.4 Translating wfdfs's into flowchart schemas

The translation from a wfdfs $S$ to a flowchart schema involves labelling the data links in $S$, generating blocks of statements for subschemas of $S$ and then sequencing these blocks, producing a total ordering on the blocks which is consistent with the partial ordering imposed on these blocks by the data dependence relationship between them. Since algorithms for deriving total orderings from partial orderings are well known, we will only describe the labelling procedure, the generation of blocks for different types of subschemas, and give an example.

Labelling data links in a wfdfs

Let $Z$ be a $m$-input $n$-output wfdfs. The data links in $Z$ are labelled by applying the following recursive procedure to $Z$ after labelling the output data links of the $m$ I-operators of $Z$ by the variable symbols $X_1, \ldots, X_m$.

Label($S$) - $S$ is a subschema of a wfdfs $Z$. An input data link of $S$ is a data link in $S$ which is not the output data link of any subschema in $Z$. All the input data links of $S$ are labelled.

1. Pick a subschema $T$ of $S$ (which is either an operator, a conditional subschema or an iteration subschema) all of whose input links have been labelled. Label $T$ as follows:

(i) $T$ is an operator. Label the output data link of $T$ by a new variable symbol $X$.

(ii) $T$ is a conditional subschema. A data link $d$ in $D_T$ U
Figure 3.19 Structure of wdfs's to be labelled
-91-

D_r (Figure 3.19) is the output data link of a T-gate or a F-gate. Label d with the variable symbol which labels the input link of the corresponding T or F-gate. Label every data link g in G (Figure 3.19) with a new variable symbol.

Label(P) (Figure 3.19)
Label(Q) (Figure 3.19)

(iii) T is an iteration subschema. A data link a in A (Figure 3.19) is the output data link of a merge gate. Label a by the label which labels the F input link of the corresponding merge gate. A data link d in F U H (Figure 3.19) is the output link of a T-gate or a F-gate. Label d by the variable symbol which labels the input data link of the corresponding T or F-gate.

Label(P) (Figure 3.19)

End of Label

Generating blocks of statements for subschemas in a wdfs Z

(i) With the output data links of the m I-operators of Z labelled by X_1, ..., X_m, the first statement generated is

   \text{START}(X_1, ..., X_m)

(ii) For an operator whose input data links are labelled by X_{1l}, ..., X_{kx}, whose output data link is labelled by X_j, and whose label is F_{k}, the statement generated is
(iii) **Conditional subschema**

Let $C$ be the decision structure in the conditional subschema.

Let the input data link to $C$ be labelled by the variable symbols $x_1, \ldots, x_i$, $x_k \in \text{Var}$, $1 \leq k \leq i$.

Let the deciders in $C$ be labelled by $p_1, \ldots, p_l$.

Let the input data links to $p_k$, $1 \leq k \leq l$, be labelled by $a_1^k, \ldots, a_k^k$, $a_1^k, \ldots, a_k^k \in \{x_1, \ldots, x_i\}$.

Let the predicate statement whose predicate symbol is $p_k'$ and which accesses the variable symbols $a_1^k, \ldots, a_k^k$ be denoted by $\text{P}_k$.

A predicate statement network is generated to simulate the decision structure as shown in Figure 3.20.

The block generated for the conditional subschema is shown in Figure 3.21. $B_T$ and $B_F$ are the blocks generated for the corresponding subschemas of the conditional subschema. $A_T$ and $A_F$ are sequences of assignment statements. If the output link of the i-th output merge gate of the conditional subschema is labelled by $x_0'$, the T input link of the merge gate is labelled by $x_t$ and the F input link of the merge gate is labelled by $x_f'$, then the i-th statement in $A_T$ is $x_0 + x_t$ and the i-th statement in $A_F$ is $x_0 + x_f$. 
Figure 3.20 Simulating decision structure with predicate statement network
(iv) **Iteration subschema.** A predicate statement network is generated to simulate the decision structure in the iteration subschema as shown in Figure 3.20.

The block generated for the iteration subschema is shown in Figure 3.22. B is the block generated for the subschema in the iteration subschema. A is a sequence of assignment statements. If \( x_a \) in A (Figure 3.19) is the output link of the \( i \)-th merge gate and the T input link of the merge gate is labelled by \( x_b \), the \( i \)-th statement in A is \( x_a + x_b \).

(v) With the \( n \) input data links of the \( O \)-operators of \( Z \) labelled by \( Y_1, \ldots, Y_n \), the last statement generated is \( \text{HALT}(Y_1, \ldots, Y_n) \).
Figure 3.23 An example of generating blocks for wdfs's

(b) Blocks generated with data dependence relationship indicated by dotted directed edges
Example: The example wfdfs and the intermediate steps in the translation process are shown in Figure 3.23. An equivalent flowchart schema is formed by sequencing the blocks in Figure 3.23b, using a total ordering which is consistent with the data dependence relationship among these blocks.

The flowchart schema resulting from the translation contains many assignment statements whose purpose is to update the output variables of loops and conditionals properly. In many cases these assignment statements can be removed by labelling the appropriate nodes with the same variable symbol. This translation algorithm from wfdfs's into flowchart schemas, unlike its inverse, preserves freedom and openness. Completeness is preserved by translation in either direction. We now state Theorem 3.2 and Theorem 3.3.

**Theorem 3.2** The class of wfdfs's and the class of flowchart schemas are equivalent, i.e. for every schema in one class, there is an equivalent schema in the other class.

**Theorem 3.3** The equivalence between wfdfs's and flowchart schemas is effective, i.e., there is a procedure to translate any wfdfs into an equivalent flowchart schema and there is a procedure to translate any flowchart schema into an equivalent wfdfs.
3.2 Freedom and $\alpha$-freedom in Program Schemas

Using purely syntactic rules a set of computation sequences $E$ for a program schema $S$ can be derived from the control structure of $S$. Given an interpretation $I$, any computation sequence $e \in E$ consistent with $I$ can be invoked to process the inputs. However, the control structure of an arbitrary program schema may allow computation sequences which are not consistent with any interpretation. For an $\alpha$-free program schema (and for most free schemas) every computation sequence derivable from the control structure is consistent with some interpretation. By studying and comparing different classes of $\alpha$-free or free program schemas we can gain some insight into the expressive power of the control structures upon which these classes of schemas are based.

In this section we prove two theorems which demonstrate that the class of free $\text{wfdfs}'s$ and the class of $\alpha$-free $\text{wfdfs}'s$ are both proper subclasses of free flowchart schemas. These two results pinpoint some differences between computer programs which use the goto statement and those which do not.

Theorem 3.4 The free flowchart schema $T$ in Figure 3.24 is not equivalent to any free $\text{wfdfs}$. 

We note that $T$ is equivalent to the $\alpha$-free $\text{wfdfs}$ shown in Figure 2.11. Theorem 3.4 thus implies that the class of free $\text{wfdfs}'s$ is a proper subclass of $\alpha$-free $\text{wfdfs}'s$. We prove Theorem 3.4 by proving a set of lemmas.
Figure 3.24 A flowchart schema which is not equivalent to any free wfdfs

Assume there is a free wfdfs $W$ equivalent to $T$. $W$ must be complete since $T$ is complete. (Theorem. 2.3-7)

By Theorem 2.3-8, we may also assume that $W$ is open and complete.

**Lemma 3.4-1:** $W$ must contain an iteration subschema $I$.

**Proof:**

If $W$ does not contain any iteration subschema, the set of output it can produce under all free interpretations is finite. Under the set of all free interpretations the set of outputs $T$ can produce is infinite. Hence $W$ and $T$ cannot be equivalent if $W$ contains no iteration subschema.

Q.E.D.

**Lemma 3.4-2:** During the execution of $W$ under a free interpretation, after making a predicate decision whose predicate symbol is $P$ or $Q$, whose input is the string $f_i(a)$ and whose outcome is $F$, no iteration subschema in $W$
can be entered.

**Proof:**

Since $W$ is free and complete, $W$ is also $\alpha$-free. If an iteration subschema is entered after such a decision is made, a free interpretation can be chosen under which $P(f^i(a_1))$ or $Q(f^i(a_1))$ (depending on the predicate symbol of the decider involved) has outcome $E$, and under which the iteration subschema entered is never exited. Under such an interpretation $T$ converges but $W$ diverges, and $W$ cannot be equivalent to $T$.

Q.E.D.

From the openness of $W$, the iteration subschema $L$ of Lemma 1 must first be entered under some interpretation $I$. When $L$ is first entered, only a finite set of values, $V$, would have been tested by the predicate symbols $P$ and $Q$ under $I$. Let $M$ be the set of free interpretations which agree with $I$ on $V$. Then under every free interpretation in $M$, $L$ is entered. Let $p_I$ be the least integer such that $f^P(a_1)$ has not been tested by $P$ when $L$ is entered under $I$. Let $q_I$ be the least integer such that $f^Q(a_1)$ has not been tested by $Q$ when $L$ is entered under $I$. For any free interpretation $m$ in $M$, $p_m=p_I$ and $q_m=q_I$. Denote $p_I$ by $p$ and $q_I$ by $q$.

**Lemma 3.4-3** After some number of iterations of $L$, the decision structure that controls $L$ tests $P(f^P(a_1))$. After some number of iterations, the decision structure also tests $Q(f^Q(a_1))$. 
Proof:

By definition \( P(f^P(a_1)) \) was not tested before \( L \) is entered. Let \( J \) in \( M \) be a free interpretation under which \( P(f^P(a_1)) \) is never tested by the decision structure which controls \( L \). We can modify \( J \) to \( J' \), \( J' \in \mathcal{M} \), such that under \( J' \) \( L \) is entered but never exited, the decision structure which controls \( L \) never tests \( P(f^P(a_1)) \), and \( P(f^P(a_1)) \) has outcome \( F \). Since \( P(f^P(a_1)) \) is never tested by the decision structure which controls \( L \), \( L \) will diverge under \( J' \). \( T \) converges under \( J' \). Hence \( W \) and \( T \) cannot be equivalent. \( P(f^P(a_1)) \) must thus be tested by the decision structure which controls \( L \) after \( L \) is entered. Similarly \( Q(f^Q(a_1)) \) must be tested by this decision structure.

Q.E.D.

Lemma 3.4-4: \( L \) must be of the form shown in Figure 3.25.

Proof:

The relevant features of this particular form are:

(i) \( P, Q \) label deciders which are components of the decision structure that controls \( L \).

(ii) If the outcome of either of these 2 deciders if \( F \), the loop \( L \) is exited.

(i) follows directly from Lemma 3.4-3.

(ii) follows directly from Lemma 3.4-2.

Q.E.D.
Lemma 3.4-5 T and W are not equivalent.

Proof:

For an interpretation J in M, either
P(f^P(a_1)) is tested before Q(f^Q(a_1)) is tested, or
Q(f^Q(a_1)) is tested before P(f^P(a_1)) is tested, or
P(f^P(a_1)) is tested when Q(f^Q(a_1)) is tested.

Suppose that under J P(f^P(a_1)) is tested before
Q(f^Q(a_1)) is tested. The other two cases can be treated
analogously.

When f^P(a_1) is the input to the decider P, the
decider Q must be testing some value y. (y may be f^P(a_1))

Modify J to J', such that under J':

P(f^P(x)) = F
Q(y) = T

Otherwise every decision under J' has the same outcome
as it does under J.
Modify $J$ to $J''$, such that under $J''$:

$$P(f^P(a_1)) = T$$

$$Q(y) = F$$

Otherwise every decision under $J''$ has the same outcome as it does under $J$.

Under $J'$ and $J''$, $L$ would terminate when the input to the decider labelled by $P$ is $f^P(a_1)$ and the input to the decider labelled by $Q$ is $y$. Since $W$ is free, under either $J'$ or $J''$, the decisions $P(f^P(a_1))$ and $Q(y)$ will not be made again after $L$ has terminated. Thus under $J'$ and $J''$, $W$ has identical outputs. Under $J'$ and $J''$, $T$ gives different outputs. Hence $W$ and $T$ are not equivalent.

Q.E.D.

Proof of Theorem 3.4:

Lemma 3.4-1 through Lemma 3.4-5.

Q.E.D.

Theorem 3.5 The free flowchart schema $T$ in Figure 3.26 is not equivalent to any $a$-free wfdfs.

We prove Theorem 3.5 by proving a set of lemmas.

Consider the following regular expressions:

$$E_P = (bg \ast af \ast)^{*}a_1$$

$$E_Q = (bg \ast af \ast)^{*} a_1$$

$$E = E_P \cup E_Q$$
Figure 3.26 A free flowchart schema that is not equivalent to any c-free wdfs

Figure 3.27 Schematic representation of data path in W
\[ E = \{ e \mid e \text{ is the output of } T \text{ under some free interpretation of } I \} \]
\[ E_P = \{ e \mid e \text{ is tested by the predicate symbol } P \text{ in a computation sequence of } T \text{ under some free interpretation } I \} \]
\[ E_Q = \{ e \mid e \text{ is tested by the predicate symbol } Q \text{ in a computation sequence under some free interpretation } I \} \]

Let \( W \) be an \( \alpha \)-free wdfs equivalent to \( T \). We may again assume that \( W \) is open and complete.

Lemma 3.5-1 During the execution of \( W \) under a free interpretation, after making a predicate decision with a decider whose label is \( F(Q) \), whose input is in \( E_P \) (or \( E_Q \)) and whose outcome is \( F \), no iteration subschema of \( W \) can be enabled.

Proof:

Similar to the proof of Lemma 3.4-2.

Q.E.D.

Lemma 3.5-2 \( W \) contains an iteration subschema \( L \). Let \( R_1, R_2, R_3 \) and \( R_4 \) denote the regular sets generated by the regular expressions \((bg^*af^*)*, (g^*af^*b^*)*, (af^*bg^*)^* \) and \((f^*bg^*a^*)^* \) respectively. For at least one of the \( R_i \)'s, \( 1 \leq i \leq 4 \), the following must hold:

If \( s, s \in R_1 \), is an input data value to \( L \), then for every \( y, y \in R_1 \), there is a free interpretation under which \( y \sqcup s \) (\( y \) concatenated with \( s \)) is an output data
value of L. (Figure 3.27)

Proof:

Tracing back along data paths from the output link z of W, there must exist, along one of the data paths, an iteration subschema as described. Otherwise the set of outputs at z cannot be the entire set E.

Q.E.D.

Since Lemma 3.5-2 must hold for only one of the $R_i$'s, in the following lemmas we shall assume that Lemma 3.5-2 holds for $R_1$, the regular set generated by $(bg^*af^*)^*$. The other cases can be treated similarly.

Lemma 3.5-3 S shown in Figure 3.27 must have a data path of the form shown in Figure 3.28.

Proof:

(i) Only such a path can generate $(bg^*af^*)^*$.

(ii) W is $\alpha$-free, and open.

Let J be a free interpretation under which S is entered.

Let V be the finite set of predicate decisions that have been made when S is entered under J.

Let M be the set of free interpretations consistent with J on V.

Let $J' \in M$ be a free interpretation under which S is entered, and after S is entered, the first data value $y$ that appears on the output link of the operator labelled by "a" has not been tested by P previously in the computation sequence under
Input to S

EDP = empty data path, a path which does not contain any operator node

Figure 3.28 Schematic representation of subschema $S$ in Figure 3.27
Let \( P(y) = \overline{F} \) under \( J' \).

By Lemma 3.5-1, the iteration subschema which generates \( q^* \) cannot be entered. Under \( J' \), since by Lemma 3.5-1 no previous test made by deciders labelled by \( P \) or \( Q \) on data value \( y' \in E_p \cup E_q \) can have outcome \( F \). For \( W \) to be equivalent to \( T \), the operator labelled by "b" must be bypassed under \( J' \).

The data path thus must be as shown in Figure 3.28.

Q.E.D.

Lemma 3.5-4 W cannot be \( \alpha \)-free

Proof: We have shown that the subschema \( S \) in Figure 3.27 must have a data path as shown in Figure 3.28. Under \( J' \), by Lemma 3.5-1, the iteration subschema \( L \) containing \( S \) must be exited after the predicate decision \( P(y) \) is made, with outcome \( F \). In the computation sequence of \( W \) under \( J' \), the decision made by the decision structure controlling \( L \), following the decision \( P(y) = \overline{F} \) in the computation sequence, must have outcome \( F \). \( W \) thus cannot be \( \alpha \)-free.

Q.E.D.

Proof of Theorem 3.5:

Lemma 3.5-1 through Lemma 3.5-4.

Q.E.D.
3.3 Open and Complete Program Schemas

Theorem 3.6 Every open and complete flowchart schema is equivalent to an open and complete wfdfs.

Proof: By applying Algorithms 3.1, 3.2 and 3.3, we can translate any flowchart schema into an equivalent wfdfs. Let flowchart schema $S$ be translated into wfdfs $S'$. By Theorem 2.3-7, $S'$ is complete if (and only if) $S$ is complete. By Theorem 2.3-8, if $S'$ is complete, $S'$ is equivalent to an open and complete wfdfs $S''$. Thus every open and complete flowchart schema is equivalent to an open and complete wfdfs.

Q.E.D.

Theorem 3.7 Every open and complete wfdfs is equivalent to an open and complete flowchart schema.

Proof: The translation from a wfdfs into a flowchart schema preserves openness and completeness.

Q.E.D.

Theorem 3.8 The class of open and complete flowchart schemas is equivalent to the class of open and complete wfdfs's.

Proof: Theorem 3.6 and Theorem 3.7.

Q.E.D.
other words, Z is open iff \( S_1 \) and \( S_2 \) are not equivalent. By constructing Z we have reduced the undecidable equivalence problem for open and complete wfdfs's to the openness problem for complete wfdfs's. The openness problem for complete wfdfs's is thus also undecidable.

Q.E.D.
Chapter Four

Decision Problems

4.0 Introduction

Decision problems in program schemas have been studied extensively in the literature [Paterson 72] [Garland & Luckham 73]. The equivalence problems for several subclasses of wdfs's have been studied by Qualitz [Qualitz 75]. The equivalence problem for data links in free wdfs's has been studied by Dennis and Fosseen [Dennis & Fosseen 73]. In this chapter we present three undecidability results for open and complete wdfs's:

1. Completeness is undecidable in open wdfs's.
2. Openness is undecidable in complete wdfs's.
3. Equivalence is undecidable in open and complete wdfs's.

These results also hold for open and complete flowchart schemas. The relationship between these results and other decidability or undecidability results in program schematology is discussed in Chapter 5.

4.1 Undecidability of Completeness in Open wdfs's

The Post Correspondence Problem (PCP) can be reduced to the problem of deciding completeness in open wdfs's. For a detailed discussion of the PCP the reader is referred to Post's original paper [Post 46]. The version of PCP we use can be stated as follows:

A PCP $C$ is an ordered pair, $C = (A, B)$, where
A = \{A_1, \ldots, A_n\} \quad 1 \leq i \leq n, \quad A_i, B_i \in (0U1)^* - \{\lambda\}

B = \{B_1, \ldots, B_n\}

and for \(1 \leq i \leq n\),

\[
A_i = a_{i1} \cdots a_{im} \\
B_i = b_{i1} \cdots b_{im}
\]

\(a_{ij}, b_{ij} \in \{0, 1\}\)

A PCP C has a solution iff there exists a sequence of subscripts \(i_1, \ldots, i_m\) such that:

(i) for \(1 \leq k \leq m\), \(1 \leq i_k \leq n\)

(ii) \(A_{i_1} \cdots A_{i_m} = B_{i_1} \cdots B_{i_m}\)

It is well-known that the set of correspondence problems which have solutions is not recursive.

To reduce the Post Correspondence Problem to the completeness problem we will firstly show how to construct a wfdfe \(S_{PCP}\) that "simulates" a PCP. Two unary function symbols \(f_0\) and \(f_1\) are used to denote the symbols 0 and 1. Let I be a free interpretation for \(S_{PCP}\) with associated domain \(D\). Under I, the input data values to \(S_{PCP}\) are elements of \(\text{Insym} = \{\delta_i | 1 \leq i\}\) and the data values are strings in \(D\) (Section 2.3). There is then a natural correspondence between \((f_0 \cup f_1)^* \square \text{Insym}\) and strings over \((0 \cup 1)\). This correspondence can be expressed as an isomorphism \(\varphi\),

\[
\varphi: (f_0 \cup f_1)^* \square \text{Insym} \rightarrow (0 \cup 1)^*
\]

\(\varphi(f_0') = 0, \quad \varphi(f_1') = 1; \quad i \neq 1, \quad \varphi(\delta_i) = \lambda\)

\(\varphi(f \square d) = \varphi(f) \square \varphi(d), \quad f \in \{f_0', f_1\}, \quad d \in (f_0 \cup f_1)^* \square \text{Insym}\)
The wdfs $S_{PCP}$ simulates a PCP $C = (A, B)$ if for a fixed yet arbitrarily chosen input, with input value $\delta_i$ under a free interpretation, there are two $O$-operators of $S_{PCP}$, $Out_1$ and $Out_2$, such that:

Under any free interpretation, the outputs at $Out_1$ and $Out_2$ are $s_1$ and $s_2$, $s_1, s_2 \in (f_0 \cup f_1)^* \delta_i$, iff there exists integers $i_1', ..., i_m'$,

$$\varphi(s_1) = A_{i_1'} \square ... \square A_{i_m'}$$

$$\varphi(s_2) = B_{i_1'} \square ... \square B_{i_m'}$$

**Construction 4.1** Given a PCP $C$, construct a wdfs $S_{PCP}$ which simulates $C$.

Let $C = (A, B)$ be a PCP given as above.

(i) For each $A_i$ and $B_i$, $1 \leq i \leq n$, construct a wdfs which simulates it. Concatenation of $A_i$, or $B_i$, to a string $d$ is simulated by applying the corresponding wdfs to input string $d$ under a free interpretation. For each $A_i = a_{i1} \square ... \square a_{in}$, construct a $(1, 1)$-wdfs $H_i$ by cascading $\alpha_i$ functional operators labelled by $f_{i1}, ..., f_{in}$ as shown in Figure 4.1. For $1 \leq j \leq \alpha_i$,

$$f_{ij} = \begin{cases} 
    f_0 & \text{if } a_{ij} = 0 \\
    f_1 & \text{if } a_{ij} = 1 
\end{cases}$$

Similarly construct a wdfs $K_i$ for each $B_i$.

(ii) From the $H_i$'s and $K_i$'s construct a wdfs $S_C$ which provides $n$ alternative data paths between its input
and output data links. Choosing among one of these data paths corresponds to picking one of the pairs \((A_i, B_i)\), \(1 \leq i \leq n\). \(S_C\) is constructed inductively as follows:

\(S_{n-1}\) is a **conditional subschema**. One branch of \(S_{n-1}\) contains \(H_n\) and \(K_n\). The other branch contains \(H_{n-1}\) and \(K_{n-1}\). \(S_{n-1}\) is shown in Figure 4.2.

For \(1 \leq j \leq n-2\), \(S_j\) is constructed from \(S_{j+1}\), \(H_j\) and \(K_j\) as shown in Figure 4.3. Each such \(S_j\) is a conditional schema containing \(S_{j+1}\) in its T-branch and \(H_j, K_j\) in its F-branch. \(S_C\) is \(S_1\).

\(S_C\) has 3 input data links labelled by \(x_1, y_1, z_1\) and 2 output links labelled by \(p_1\) and \(q_1\). Under a free interpretation, the input values at \(x_1, y_1, z_1\) are \(\delta_1, \delta_2\) and \(\delta_3\) respectively. Consider the following sets of decision outcomes:

For \(1 \leq k \leq n-1\),

\[
E_k = \{ \text{P}(f^k_{\delta_3}) = F \} \cup \{ \text{P}(f^j_{\delta_3}) = T \mid 1 \leq j < k \}
\]

\[
E_n = \{ \text{P}(f^j_{\delta_3}) = T \mid 1 \leq j \leq n \}
\]

If \(I\) is a free interpretation consistent with \(E_j\), the output of \(S_C\) at \(p_1\) and \(q_1\) under \(I\) are \(F_{j\delta_3}^1\) and \(F_{j\delta_3}^2\).
such that

\[ \varphi(F_jH_1) = A_j \quad \text{and} \quad \varphi(F_jH_2) = B_j \]

Figure 4.2 \( S_{n-1} \) for Construction 4.1(ii)

Figure 4.3 \( S_j \) for Construction 4.1(ii)
(iii) Construct an iteration subschema \(L\) which contains \(S_C\) as its body. Executing the body \(S_C\) of \(L\) then corresponds to picking one of the pairs \((A_i, B_i)\), for some \(i, 1 \leq i \leq n\). Iterating the iteration schema \(L\) \(m\) times then corresponds to picking \(m\) integers \(i_1, \ldots, i_m\) to form the strings \(A_{i_1} \cdots A_{i_m}\) and \(B_{i_1} \cdots B_{i_m}\).

Since a proposed solution to a PCP must have used at least one of the pairs \((A_i, B_i)\), i.e. \(m \geq 1\), the \(S_{PCP}\) simulating \(C\) is constructed by concatenating a copy of \(S_C\) with \(L\), as shown in Figure 4.4.

Figure 4.4 \(S_{PCP}\) for simulating the PCP \(C\)
$S_{PCP}$ has two input data links, labelled by $x$ and $z$, and two output data links, labelled by $p$ and $q$. Let the inputs to $x$ and $z$ under free interpretations be $\delta_1$ and $\delta_2$ respectively. Under a free interpretation $I$:

Let $m$ be the smallest integer $i$, $i \geq 1$, such that

$$P(g^i_1 \delta_2) = F$$

$m$ is the number of times $S_C$ is executed under $I$ and corresponds to the total number of pairs from $\{(A_i, B_i) \mid 1 \leq i \leq n\}$ used in constructing a proposed solution to the PCP $C$. If $m$ does not exist under $I$, $S_{PCP}$ diverges under $I$.

Let $i_j$, $1 \leq j$, be the largest integer $k$, $k \in \{1, \ldots, n\}$ such that

$$\text{for all } l, 1 \leq l < k, P(f^l \Pi g^{j-1} \Pi \delta_2) = T$$

On the $j$-th execution of $S_C$ in $S_{PCP}$ under $I$, the data path through $S_C$ is determined by $i_j$. $i_j$ corresponds to the $j$-th pair from $\{(A_i, B_i) \mid 1 \leq i \leq n\}$ picked in constructing a proposed solution to the PCP $C$.

If $m$ exists under $I$, the outputs of $S_{PCP}$ under $I$ at the data links labelled by $p$ and $q$ (Figure 4.4) are $F_{p \Pi \delta_1}$ and $F_{q \Pi \delta_1}$ such that

$$\varphi(F_{p \Pi \delta_1}) = A_{i_1} \Box \ldots \Box A_{i_m}$$

$$\varphi(F_{q \Pi \delta_1}) = B_{i_1} \Box \ldots \Box B_{i_m}$$

with $m$, $i_1$, $\ldots$, $i_m$ determined as above.

From the above discussion it should also be obvious how to construct a free interpretation $I$, given a proposed solution $\left( A_{i_1} \Box \ldots \Box A_{i_m}, B_{i_1} \Box \ldots \Box B_{i_m} \right)$, such that:
Under I, the outputs of \( S_{PCP} \) at output data links \( p \) and \( q \) are \( F_p^\delta \) and \( F_q^\delta \) respectively, and

\[
\begin{align*}
\varphi(F_p^\delta) &= A_1 \sqcup \ldots \sqcup A_i \\
\varphi(F_q^\delta) &= B_1 \sqcup \ldots \sqcup B_i 
\end{align*}
\]

\( S_{PCP} \) thus simulates the PCP C.

End of Construction 4.1

Using \( S_{PCP} \) we can construct an open wdfs \( S \) as shown in Figure 4.5. \( S \) contains two copies of \( S_{PCP} \), \( S_{PCP}^1 \) and \( S_{PCP}^2 \). Under a free interpretation \( I \), \( S_{PCP}^1 \) proposes a solution \( (F^A_1 \sqcup F^B_1, F^A_2 \sqcup F^B_2) \) to the PCP C. On successive iterations of the subschema L containing \( S_{PCP}^2 \), the outputs of \( S_{PCP}^2 \) are of the form \( F^A_i \sqcup F^A_i \) and \( F^B_i \sqcup F^B_i \), for \( i = 1 \). If the proposed solution is indeed a solution to the PCP C, \( F^A_1 \sqcup F^A_1 \) and \( F^B_1 \sqcup F^B_1 \), and all pairs of strings \( F^A_i \sqcup F^A_i \) and \( F^B_i \sqcup F^B_i \), are pairs of identical strings. If the proposed solution is not a solution, \( F^A_1 \sqcup F^A_1 \) and \( F^B_1 \sqcup F^B_1 \), for \( 0 \leq i \), are always different strings. Due to the pair of tests performed by deciders labelled by \( Q \), the iteration subschema containing \( S_{PCP}^2 \) does not terminate whenever entered, if the solution proposed by \( S_{PCP}^1 \) under \( I \) is indeed a solution to C. These observations are formalized in Lemma 4.1-2.

Lemma 4.1-1 S in Figure 4.5 is open.

Proof:

It should be obvious from the internal structure and
Figure 4.5 Open wfdfs
S which is complete iff
the PCP C has a solution
mode of operation of $S_{PCP}^1$ that for every decider $d$ in $S_{PCP}^1$ there is an interpretation under which $S_{PCP}^1$ terminates and under which a decision by $d$ has outcome $T(F)$. Under any interpretation $I$, the input data value which determines the outcomes of decisions made by deciders in $S_{PCP}^2$ remains unchanged on successive activations of $S_{PCP}^2$. This input data value is the same value which determines the outcomes of decisions made by deciders in $S_{PCP}^1$. $S_{PCP}^1$ and $S_{PCP}^2$ are identical in structure. Hence for every decider in $S_{PCP}^2$ there is also an interpretation under which that decider has outcome $T(F)$.

Let $L$ be the iteration subschema containing $S_{PCP}^2$. Let $q_1$ be the decider which controls $L$. $q_1$ is labelled by the predicate symbol $Q$. The first test made with $q_1$ in any computation is always the first decision made with predicate symbol $Q$ in that computation, and can have outcome $T$, or $F$.

Let $q_2$ be the decider which controls the conditional schema cascaded with $S_{PCP}^2$ in $L$. $q_2$ is also labelled by $Q$. On successive iterations of $L$, the values tested by $q_2$ are from the set $\{F^A(F^A)^{-1}E_1 \mid 0 \leq i \}$. None of these values has been tested previously by a decider labelled by $Q$ before it is tested by $q_2$. Thus every one of these tests can have outcome $T$, or $F$.

$S$ is thus open.

Q.E.D.
Lemma 4.1-2: S in Figure 4.5 is complete iff the PCP C simulated by $S_{PCP}^1$ and $S_{PCP}^2$ in S has no solution.

Proof:

Let L be the iteration subschema in S containing $S_{PCP}^2$.
Let $q_1$ be the decider, labelled by the predicate symbol Q, which controls L.
Let $q_2$ be the decider, labelled by the predicate symbol Q, which controls the conditional subschema cascaded with $S_{PCP}^2$ in L.

(only if)

Consider the set of decisions made in each iteration of L. Let $d_1$ be the data value tested by $q_1$. The test Q($d_1$) must have outcome T if the body of L is to be entered. Let the two outputs of $S_{PCP}^2$ be $o_1$ and $o_2$. After $S_{PCP}^2$ terminates, $o_1$ is tested by $q_2$. If the test by $q_2$, Q($o_1$), has outcome T, the next test made by $q_1$ is Q($o_2$). If the test by $q_2$, Q($o_1$) has outcome F, the next test made by $q_1$ is again Q($d_1$), which must have outcome T. Thus in the latter case the iteration subschema L will always be re-entered. If, furthermore, $o_1$ is always equal to $o_2$, the test Q($o_2$) always has the same outcome as Q($o_1$). Hence if $o_1$ is always equal to $o_2$, L diverges if it is entered, independent of the outcome of decisions made with $q_2$.

Assume that C has a solution, then there exist positive integers $v, w_1, \ldots, w_v$, $1 \leq w_1, \ldots, w_v \leq n$, such that

$$A_{\square}^{w_1} \ldots \square A_{\square}^{w_v} = B_{\square}^{w_1} \ldots \square B_{\square}^{w_v}$$
Let \( F^A \Delta_1 \) be such that \( \phi(F^A \Delta_1) = A_{w_1} \ldots A_{w_v} \).
Let \( F^B \Delta_1 \) be such that \( \phi(F^B \Delta_1) = B_{w_1} \ldots B_{w_v} \).
\( F^A \) and \( F^B \) are identical strings over \( \{ f_0, f_1 \} \).

For \( 1 \leq i \leq v \), define a finite set of predicate decision outcomes \( W_i \):
\[
W_i = \{ P(f^j g^{i-j-1} \Delta_2) = T \mid 1 \leq j < w_i \} \cup \{ P(f^{w_i} g^{i-1} \Delta_2) = F \}
\]

Define the set \( G \) of predicate decision outcomes:
\[
G = \{ P(g^j \Delta_2) = T \mid 0 \leq j \leq v \} \cup \{ P(g^{v+1} \Delta_2) = F \}
\]

Under any free interpretation the inputs to \( S \) are \( \delta_1 \) and \( \delta_2 \). Let \( I \) be a free interpretation consistent with \( V \).

\( G \cup \bigcup_{i=1}^v W_i \). From our discussion on the internal structure and mode of operation of \( S_{PCP} \), it should be obvious that under \( I \) \( S_{PCP}^1 \) converges with outputs \( F^A \Delta_1 \) and \( F^B \Delta_1 \). Under \( I \), the outputs of \( S_{PCP}^2 \) are of the form
\[
F^A (F^A)^i \Delta_1 \quad \text{and} \quad F^B (F^A)^i \Delta_1
\]
for \( i \geq 0 \). Since \( F^A \Delta_1 \) and \( F^B \Delta_1 \) are identical strings, the outputs of \( S_{PCP}^2 \) are always identical. Hence if \( L \) is entered under \( I \), \( L \) diverges. The finite prefix of the computation sequence of \( S \) under \( I \), containg the computations by \( S_{PCP}^1 \) and the decision \( \phi(F^{A \Delta_1}) = T \) by \( q_1 \), cannot be extended to to any finite computation sequence. \( S \) is thus not complete.

Hence if \( S \) is complete, \( C \) has no solution.
Under any free interpretation \( \mathcal{I} \), subschemas \( S^1_{\text{PCP}} \) and \( S^2_{\text{PCP}} \) always converge and diverge together. From Construction 4.1, \( S^1_{\text{PCP}} \) and \( S^2_{\text{PCP}} \) diverge iff \( P(q^i \cdot \delta_2) = \mathbb{F} \) for all \( i \), under \( \mathcal{I} \). Thus any finite prefix of a computation sequence of \( S \) can be extended to a computation sequence in which every activation of \( S^1_{\text{PCP}} \) and \( S^2_{\text{PCP}} \) terminates.

Let \( \eta \) be a finite prefix of a computation sequence of \( S \). We can always extend (maybe trivially) \( \eta \) to a finite prefix \( \eta' \) such that every activation of \( S^1_{\text{PCP}} \) and \( S^2_{\text{PCP}} \) terminates in \( \eta' \). Extend \( \eta' \) to \( \eta'' \) such that the first decision \( D \) by \( q_2 \) in \( \eta'' \), but not in \( \eta' \), has outcome \( T \) and the first decision by \( q_1 \) in \( \eta'' \) occurring after \( D \) has outcome \( F \). The two input data values to the decisions by \( q_1 \) and \( q_2 \) are outputs of \( S^2_{\text{PCP}} \), of the form \( F^A (F^A)^i \cdot \delta_1 \) and \( F^B (F^A)^i \cdot \delta_1 \) for some fixed \( i \). Since \( C \) has no solution, no two outputs of \( S^2_{\text{PCP}} \) of this form can be identical. Hence we can assign these two opposite decision outcomes to the two decisions without introducing inconsistency. Under these assignment of outcomes \( L \) terminates after the decision by \( q_1 \). \( \eta'' \) can then be extended to a finite computation sequence. \( S \) is complete since \( \eta \) can be any finite prefix of any computation sequence of \( S \).

Q.E.D.

**Theorem 4.1** Completeness is undecidable for open wdfs.

**Proof:** Construction 4.1, Lemma 4.1-1 and Lemma 4.1-2.
4.2 Undecidability of Equivalence in Open and Complete wfdfs's

The undecidability of equivalence in open and complete wfdfs's is established by reducing the undecidable Hilbert's Tenth Problem to the equivalence problem. Hilbert's Tenth Problem is an undecidability result on polynomials with integer coefficients over the domain of integers. The version of Hilbert's Tenth Problem we use is on polynomials with non-negative integer coefficients over the domain of natural numbers and can be stated as follows:

Let $P(\bar{x})$ and $Q(\bar{x})$ be 2 polynomials of $r$ variables with non-negative integer coefficients such that for all $\bar{x}_o \in \mathbb{N}^r$, where $\mathbb{N}$ is the set of natural numbers augmented with infinity (denoted by $\infty$), $P(\bar{x}_o) \geq Q(\bar{x}_o)$. There is no effective procedure which, given any two such polynomials $P(\bar{x})$ and $Q(\bar{x})$, determines whether there exists $\bar{x}_o \in \mathbb{N}^r$ such that $P(\bar{x}_o) = Q(\bar{x}_o)$.

In the remainder of this section a polynomial is taken to be a polynomial with non-negative integer coefficients over $\mathbb{N}$. For every such polynomial $P(\bar{x})$, $\bar{x} = (x_1, \ldots, x_r)$, and $\bar{x}_o \in \mathbb{N}^r$, $P(\bar{x}_o) \in \mathbb{N}$. We shall define the notion of a wfdfs simulating a polynomial $P(\bar{x})$ and describe the reduction constructions.

To describe the simulation of polynomials by wfdfs's we first of all establish a correspondence between elements of $\mathbb{N}^r$ and the set of free interpretations for wfdfs's with at least $r$ inputs.
Let $S$ be a wdfs with $r+m$ inputs, $m \geq 0$. Let $f$ be a unary function symbol. Let $\beta$ be a unary predicate symbol. Under a free interpretation $I$ the inputs to $S$ at input data links labelled $d_1, \ldots, d_r, \ldots, d_m$ are $\delta_1, \ldots, \delta_r, \ldots, \delta_m$, respectively.

Given a free interpretation $I$ for $S$, define, for $1 \leq j \leq r$,

\[ i_j = \begin{cases} \text{the least natural number for which } \beta(f^{i_j} x_j) = T \text{ under } I, \\ \infty, \text{ if no such natural number exists.} \end{cases} \]

Let $\bar{x}_I = (i_1, \ldots, i_r), \bar{x}_I \in \mathbb{N}^r$. $\bar{x}_I$ is the $r$-tuple derived from $I$.

Conversely, given an $r$-tuple $\bar{x}_o = (i_1, \ldots, i_r), \bar{x}_o \in \mathbb{N}^r$, a free interpretation $I$ for $S$ derived from $\bar{x}_o$ is any free interpretation consistent with the set of predicate decisions:

For $1 \leq j \leq r$,

\[ \beta(f^{i_j} x_j) = \begin{cases} T, & \text{for } 0 \leq k \leq i_j, \\ F, & \text{for } k > i_j. \end{cases} \]

We note that this correspondence between $r$-tuples and free interpretations covers the cases where some of the $i_j$'s are infinite.

Let $P(\bar{x})$ be a polynomial with $r$ variables. Let $S_p$ be a $(r+1, l)$-wdfs. Under a free interpretation the inputs to $S_p$ are $\delta_1, \ldots, \delta_{r+1}$. In the following definition we use the additional convention:

If $P(\bar{x}_o) = \infty$, and the output of $S$ under $I$ is $f^{i_{r+1}} x_{r+1}$, then $S$ diverges under $I$. 

S_p simulates P(x) if
(i) For any free interpretation I, the output of S_p under I is \( f^P(x) \equiv \prod_{i=1}^{r} x_i^{\alpha_i} \), where \( x_i \) is the r-tuple of natural numbers derived from I.
(ii) For every r-tuple \( x_0 \in N^r \), let I be a free interpretation derived from \( x_0 \). The output of S_p under I is \( f^P(x) \equiv \prod_{i=1}^{r} x_i^{\alpha_i} \).

There may be several wdfs's which simulate a given polynomial p(x). To obtain a procedure for constructing a wdfs S_p to simulate a polynomial P(x) it is sufficient to give procedures for:
(i) Given a wdfs Q which simulates \( Q(x) = \prod_{i=1}^{r} x_i^{\alpha_i}, \alpha_i \geq 0 \), and a variable \( x_j \), construct a wdfs S_p which simulates the polynomial \( P(x) = Q(x) \times x_j \).
(ii) Given 2 wdfs's which simulate \( Q(x) \) and \( R(x) \), construct a wdfs S_p which simulates the polynomial \( P(x) = Q(x) + R(x) \).

We can represent a polynomial P(x) as a sum of terms, each term being a product of factors and each factor being a variable. (In this representation a term with integer coefficient n, n>1, is represented by a sum of n terms, each with coefficient 1. A variable raised to the n-th power, n>1, is represented by a product of the variable multiplied by itself n times.) Figure 4.6 shows a wdfs which simulates the polynomial P(x) = x_j. Starting
from a wfdfs of this type, a wfdfs which simulates a term can be constructed by applying (i) repeatedly. Having obtained the wfdfs's which simulate the terms, (ii) can be applied repeatedly to obtain a wfdfs which simulates the sum of the terms.

Construction 4.2

(i) Given a wfdfs $S_Q$ which simulates $Q(\bar{x}) = \Pi_{i=1}^{r} x_i^{a_i}$, $a_i \geq 0$, and a variable $x_j$, construct a wfdfs $S_p$ which simulates $P(\bar{x}) = Q(\bar{x}) \cdot x_j$.

Let $B_Q$ be the index set of the variables which are factors of $Q(\bar{x})$, $B_Q = \{ i | a_i > 0 \}$. $D_Q$ is the set of input data links to $S_Q$. $D_Q = \{ d_i | i \in B_Q \}$. Under free interpretations, the input at data link $d_i$ is $\delta_i$. $d_j$ is the input data link associated with $x_j$. $d_j$ may be an element of $D_Q$. $S_Q$ is represented in Figure 4.7 (a). $S_p$ is an iteration wfdfs, "controlled" by $x_j$, and contains $S_Q$ as its body. $S_p$ is shown in Figure 4.7 (b). $S_p$ iterates $x_j$ times, and on each iteration concatenates $Q(\bar{x})$ function.
symbols (all of them f's) to the input data value.
(ii) Given 2 wdf's $S_Q$ and $S_R$ which simulates $Q(\bar{x})$ and $R(\bar{x})$, construct a wdf $S_P$ which simulates $P(\bar{x}) = Q(\bar{x}) + R(\bar{x})$.

Let $D_Q$ and $D_R$ be the subsets of input data links which correspond to variables appearing in $Q(\bar{x})$ and $R(\bar{x})$. $D_Q$ and $D_R$ need not be disjoint and either set may contain $d_{r+1}$.

$S_Q$ and $S_R$ are shown in Figure 4.8(a). $S_P$ is a concatenation of $S_Q$ and $S_R$ as shown in Figure 4.8(b).

Figure 4.7 $S_P$ for simulating $P(\bar{x}) = Q(\bar{x}) \cdot x_j$

Figure 4.8 $S_P$ for simulating $P(\bar{x}) = Q(\bar{x}) + R(\bar{x})$
In the process of computing $P(\bar{x}_I)$ under a free interpretation $I$, $S_p$ generates all intermediate values $f^j_{x_1+1}, 1 \leq j \leq P(\bar{x}_I)$. In reducing Hilbert's Tenth Problem to the equivalence problem, we have to test every such intermediate value by the predicate $\beta$ and determine whether $\beta(f^j_{x_1+1})=T$ for all $1 \leq j \leq P(\bar{x}_I)$. To do this we modify $S_p$ to $S'_p$ which has two additional input data links $w_1', w_2'$ and one additional output data link $w'_1$. If under $I$, $P(f^j_{x_1+1})=T$ for $1 \leq j \leq P(\bar{x}_I)$, the output at $w_1'$ is the input at $w'_1$. Otherwise the output at $w_1'$ is the input at $w_2'$. The modification is achieved by:

(i) Every operator which lies on a data path from the input data link $x_{r+1}$ to the output data link of $S_p$ is labelled by $f$. Conditional wfs's are added which test the outputs of these operators and route the data values from $w_1$ and $w_2$ accordingly. The modification is shown in Figure 4.9 (a).

(ii) Two input links and two output links are added to every subschema of $S_p$ to transmit the data values from $w_1$ and $w_2$. $S_p$ has two additional output links $w'_1$ and $w'_2$. We are only interested in $w'_1$. Delete $w'_2$ and all data paths which lead only to $w'_2$ and not to any decider or operator. The modification to $S_p$ is shown in Figure 4.9 (b).

We are now ready for the main construction.
Construction 4.3

Let $P(\overline{x})$ and $Q(\overline{x})$ be two polynomials of $r$ variables such that for all $\overline{x}_0 \in \mathbb{N}^r$, $P(\overline{x}_0) \geq Q(\overline{x}_0)$. Construct two wfdfs's $S_1$ and $S_2$ such that $S_1$ and $S_2$ are equivalent iff for all $\overline{x}_0 \in \mathbb{N}^r$, $P(\overline{x}_0) > Q(\overline{x}_0)$.

Construction of $S_1$ from $P(\overline{x})$ construct $S_p$ which simulates $P(\overline{x})$ using Construction 4.2. $S_1$ is constructed from $S_p$ as shown in Figure 4.10.

Behaviour of $S_1$ under free interpretations Under a free interpretation $I$, the inputs to $S_1$ are $\delta_1, \ldots, \delta_r, \delta_{r+1}$ and $\delta_{r+2}$. Let $\overline{x}_I$ be the $r$-tuple derived from $I$. If under $I$:

(i) $\beta(\delta_{r+2}) = T$. The output of $S_1$ is $\delta_{r+1}$.
(ii) $\beta(\delta_{r+2}) = F$ and $\beta(\delta_{r+2}) = F$. The output of $S_1$ is $\delta_{r+1}$.
(iii) $\beta(\delta_{r+2}) = F$ and $\beta(\delta_{r+2}) = T$, $P(\overline{x}_I) = \infty$. $S_1$ diverges since $S_p$ diverges.
Figure 4.10 $S_1$ in Construction 4.3
(iv) $\beta(\delta_{r+2}) = T$ and $\beta(\delta_{\bar{x}_I}) = T$, $P(\bar{x}_I) = \infty$. In this case $S_p$ is activated and converges with output $P(\bar{x}_I) \cdot \frac{f(\delta_{r+1})}{E}$. The output of $S_1$ is $f(\delta_{r+1})$.

Construction of $S_2$. Construct $S_p$ from $P(\bar{x})$ using Construction 4.2. Modify $S_p$ to $S'_p$ by adding conditional wdfs's, two input data links and one output data link to $S_p$ as described above (Figure 4.9). Construct $S_Q$ from $Q(\bar{x})$ using Construction 4.3. $S_2$ is constructed from $S'_p$ and $S_Q$ as shown in Figure 4.11. We shall describe the structure of $S_2$ in more detail as we explain the behaviour of $S_2$ under free interpretations.

Behaviour of $S_2$ under free interpretations. Under a free interpretation the inputs to $S_2$ are $\delta_1, \ldots, \delta_r, \delta_{r+1}$ and $\delta_{r+2}$. Let $I$ be a given free interpretation and $\bar{x}_I$ the $r$-tuple derived from $I$. If under $I$ (Figure 4.11):

(i) $\beta(\delta_{r+2}) = T$

The T-branch of the outermost conditional wdfs in $S_2$.

The output of $S_2$ is $\delta_{r+1}$.

(ii) $\beta(\delta_{r+2}) = E$ and $\beta(\delta_{\bar{x}_I}) = E$

The F-branch of the conditional wdfs which is contained in the F-branch of the outermost conditional wdfs is taken. The output of $S_2$ is $\delta_{r+1}$.

(iii) $\beta(\delta_{r+2}) = E$, $\beta(\delta_{\bar{x}_I}) = T$, and $P(\bar{x}_I) = \infty$

The T-branch of the conditional wdfs which contains $S_p$
Figure 4.11 $S_2$ for
Construction 4.3
is taken. \( S'_p \) is activated and diverges. \( S_2 \) diverges.

(iv) \( \beta(\delta_{r+2}) = F, \beta(f_{j} \delta_{r+2}) = T, \) and \( P(\bar{x}_i) < \infty \)

As in Case (iii), the T-branch of the conditional wdfs containing \( S'_p \) and \( S'_q \) is taken. \( S'_p \) and \( S'_q \) are both activated. Since \( P(\bar{x}_i) < \infty \) and \( S'_p \) simulates \( P(\bar{x}_i \bar{x}_j) \), \( S'_p \) terminates. For all \( \bar{x}_o \), \( P(\bar{x}_o) \geq q(\bar{x}_o) \). Hence \( Q(\bar{x}_i) < \infty \) and \( S'_q \) also terminates. Furthering processing of the outputs of \( S'_p \) and \( S'_q \) depends on the other decision outcomes under \( I \). There are three subcases:

Case 1 \( \exists j, 1 \leq j \leq P(\bar{x}_i), \beta(f_{j} \delta_{r+2}) = F \)

From the structure of \( S'_p \), the outputs of \( S'_p \) are \( P(\bar{x}_i) \)

\( f_{j} \delta_{r+1} \) and \( f_{j} \delta_{r+2} \) respectively. Under \( I \) \( \beta(f_{j} \delta_{r+2}) \) has outcome \( T \). After \( S'_p \) terminates, the output of \( S'_p \) which is \( f_{j} \delta_{r+2} \) is tested by a decider labelled by \( \beta \). This test has outcome \( T \). The T-branch of the conditional wdfs which is controlled by the decider is taken. The output of this conditional wdfs, hence of \( S'_2 \), is \( P(\bar{x}_i) + 1 \)

\( f_{j} \delta_{r+1} \).

Case 2 \( \forall j, 1 \leq j \leq P(\bar{x}_i) + 1, \beta(f_{j} \delta_{r+2}) = T \)

The outputs of \( S'_p \) in this case are \( f_{j} \delta_{r+1} \) and \( \delta_{r+2} \). Under \( I \), \( \beta(\delta_{r+2}) \) has outcome \( F \). After \( S'_p \) terminates, the output of \( S'_p \) which is \( \delta_{r+2} \) is tested by a decider labelled by \( \beta \). This decision has outcome \( F \). The F-branch of the conditional wdfs controlled by this decider is taken. This F-branch contains a
conditional wdfs which is controlled by another
decider labelled by $\beta$. This second decider tests the
data value $P(x_{\bar{I}}) + 1$ and has outcome $T$. The output
of the conditional wdfs controlled by this second
decider, and hence of $S_2$, is $P(x_{\bar{I}}) + 1$.

Case 3 $\forall j, 1 \leq j \leq P(x_{\bar{I}}), \beta(f_j \delta_{r+1}) = T$,

$$P(x_{\bar{I}}) + 1 \qquad \beta(f \delta_{r+1}) = F$$

In this case the outputs of $S'_P$ are $f \delta_{r+1}$ and $\delta_{r+2}$. After $S'_P$ converges, these two outputs of
$S'_P$ are tested by deciders labelled by $\beta$. Both of these
tests have outcome $F$. Iteration wdfs $L$ is activated
with input data value $Q(x_{\bar{I}}) + 2$. The output of

$$f \delta_{r+1}$$

$S_2$ is the output of $L$. Let $\tilde{j}$ be the least integer, $\tilde{j} \geq 0$, such that under $I$,

$$Q(x_{\bar{I}}) + 2 + \tilde{j}$$

$\beta(f \delta_{r+1}) = F$.

The output of $S_2$ under $I$ is

$$f \delta_{r+1}$$

If no such $\tilde{j}$ exists, $L$ diverges and hence $S_2$ diverges.

By comparing the behaviour of $S_1$ and $S_2$ under free
interpretations, it follows that $S_1$ and $S_2$ have the same
output under cases (i), (ii), (iii), (iv).1 and (iv).2. The
behaviour of $S_2$ under case (iv).3 requires further analysis.
Assume $P(\bar{x}_o) > Q(\bar{x}_o)$ for all $\bar{x}_o \in N^r$, then
\[ P(\bar{x}_I) + 1 \geq Q(\bar{x}_I) + 2. \]

Under Case (iv).3, $V_j, 1 \leq j \leq P(\bar{x}_I), \beta(f^{H_r+1}) = \mathbb{T}$
\[ P(\bar{x}_I) + 1 \beta(f^{H_5+1}) = \mathbb{E}. \]
\[ \tilde{j} = P(\bar{x}_I) - Q(\bar{x}_I) - 1 \] by definition, and
\[ Q(\bar{x}_I) + 2 + (P(\bar{x}_I) - Q(\bar{x}_I) - 1) \]
the output of $L = \frac{P(\bar{x}_I) + 1}{H_{6+1}}$
\[ = \frac{P(\bar{x}_I) + 1}{H_{5+1}}. \]

and $S_1, S_2$ have the same output.

Thus if $P(\bar{x}_o) > Q(\bar{x}_o)$ for all $\bar{x}_o \in N^r$, $S_1$ and $S_2$ are equivalent.

Assume that for some $\bar{x}_o \in N^r$, $P(\bar{x}_o) = Q(\bar{x}_o)$. Let $I$ be a free interpretation which is derived from $\bar{x}_o$ and is consistent with the set of decision outcomes under Case (iv).3.

\[ \bar{x}_I = \bar{x}_o \] by definition of $I$
\[ P(\bar{x}_I) = Q(\bar{x}_I) \]
\[ Q(\bar{x}_I) + 2 > P(\bar{x}_I) + 1 \]
The output of $S_1$ under $I$ is $\frac{P(\bar{x}_I) + 1}{H_{6+1}}$.
The output of $S_2$ under $I$ is $\frac{Q(\bar{x}_I) + 2 + \tilde{j}}{H_{5+1}}$.

$S_1, S_2$ are not equivalent under $I$.

Thus if $P(\bar{x}_o) = Q(\bar{x}_o)$ for some $\bar{x}_o \in N^r$, $S_1$ and $S_2$ are not
Theorem 4.2 The equivalence problem for open and complete wfdfs's is undecidable.

Proof:

Let \( T \) be the set of pairs of wfdfs's:

\[
T = \{ (S_1, S_2) \mid S_1, S_2 \text{ are constructed using Construction 4.3, given two polynomials } P(\overline{x}), Q(\overline{x}) \text{ such that for all } \overline{x}_0 \in \mathbb{N}^\mathbb{R}, P(\overline{x}_0) \geq Q(\overline{x}_0) \}.
\]

Hilbert's Tenth Problem can be reduced to the problem of deciding equivalence for all the pairs of wfdfs's in \( T \). For every such pair \((S_1, S_2)\), it is straightforward to show:

(i) \( S_1 \) is open and complete.
(ii) \( S_2 \) is complete.

When \( S_2 \) is constructed for \( P(\overline{x}) = Q(\overline{x}) + 1 \), however, \( S_2 \) is not open. The first decision made when \( L \) is activated in \( S_2 \) under a free interpretation \( I \) is

\[
\beta(f_{\overline{x}_1} + \overline{\delta}_{n+1}) = Q(\overline{x}_1) + 2.
\]

This decision always follows the decision

\[
\beta(f_{\overline{x}_1} + \overline{\delta}_{n+1}) = Q(\overline{x}_1) + 1.
\]

If \( L \) is activated under \( I \), this latter decision has outcome \( F \). Since \( P(\overline{x}) = Q(\overline{x}) + 1 \), the former decision always has the same outcome as the latter. This means that the decider which controls \( L \), when activated, always has outcome \( F \). \( S_2 \) is thus not open.
Figure 4.12 Modifying $S_2$ to $S'_2$ in Theorem 4.2
To prove the undecidability of equivalence in open
and complete wdfs's, we have to modify $S_2$ to an equivalent
open and complete wdfs $S'_2$. The modification consists of
replacing $S_Q$ in $S_2$ by a conditional wdfs controlled by the
decision $\beta(\delta_{r+1})$ (Figure 4.12). If this decision has
outcome $\mathbb{F}$, the output of the conditional wdfs is the output
of $S_Q$, which is $Q(x_I)$. If the decision has outcome $\mathbb{T}$,
the output is $\delta_{r+1}$.

On analysing the behaviour of $S'_2$, we can readily see
that under cases (i), (ii) and (iii), as in Construction 4.3,
$S_2$ and $S'_2$ are equivalent, and hence $S_1$ and $S'_2$ are equiva-

Under Case (iv), if $\beta(\delta_{r+1}) = \mathbb{F}$ under $I$, $S_Q$ is activated
and $S_2$, $S'_2$ exhibit identical behaviour. $S'_2$ and $S_1$ (just as
$S_2$ and $S_1$) are not equivalent iff $\exists \overline{x}_o$ such that $P(\overline{x}_o) = Q(\overline{x}_o)$.

Under Case (iv), if $\beta(\delta_{r+1}) = \mathbb{T}$ under $I$, $S_Q$ is bypassed
and the input data to L is $\delta_{r+1}$. When L is activated, the
set of decision outcomes already fixed include:

for $1 \leq j \leq p(\overline{x}_I)$, $\beta(f^j \overline{\delta}_{r+1}) = \mathbb{T}$,
$F(\overline{x}_I) + 1$
$\beta(f^j \overline{\delta}_{r+1}) = \mathbb{F}$
$\beta(\delta_{r+1}) = \mathbb{T}$

since the input to L is $\delta_{r+1}$, the body of L is entered at
least once. As in Construction 4.3, the output of L is
$\tilde{f}_j \overline{\delta}_{r+1}$, where $\tilde{j}$ is the least integer, $\tilde{j} \geq 0$, such that
\[ \beta(\tilde{F}^j + 1, \tilde{t}^{r+1}) \in I. \] The set of decision outcomes listed above implies that the output of \( L \) is \( P(\overline{x}) + 1 \), which is also the output of \( S_1 \) under Case (iv). \( S_1 \) and \( S'_2 \) are thus equivalent in this case.

From these considerations we conclude that \( S_1 \) and \( S'_2 \) are equivalent iff \( P(\overline{x}_0) > Q(\overline{x}_0) \) for all \( \overline{x}_0 \). \( S_1 \) and \( S'_2 \) are both open and complete. We have reduced Hilbert's Tenth Problem to the equivalence problem for open and complete wdfs's. The equivalence problem for open and complete wdfs's is thus undecidable.

Q.E.D.

**Corollary 4.3-1** Openness is undecidable in complete wdfs's.

**Proof:**

In Theorem 4.3 the equivalence problem for the pairs of open and complete wdfs's \( \{ (S_1, S'_2) \} \) is undecidable. For every such pair we can construct a wdfs \( Z \) which activates \( S_1 \) and \( S'_2 \), makes a decision with the output of \( S'_2 \), and then another decision with the output of \( S_2' \). If \( S_1 \) and \( S'_2 \) are equivalent and these two decisions are made with deciders labelled by the same predicate symbol, they always have the same outcome. \( Z \) is shown in Figure 4.13. \( a \) is a predicate symbol not used in \( S_1 \) nor \( S'_2 \). \( Z \) is complete as \( S_1 \) and \( S'_2 \) both are. There is no decider in \( S_1 \) or \( S'_2 \) which always has outcome \( T \) (or \( F \)). Thus \( Z \) is open iff the decision made on the output of \( S_2' \) need not always have the same outcome as the decision made on the output of \( S_1 \). In
Chapter Five

Conclusion

5.1 The Main Results

Modelling Computer Programs

We have studied 2 models of computer programs: the class of flowchart schemas and the class of wfdfs's. The class of flowchart schemas is a model of ALGOL-like programs, and includes primitives for assignment, data-dependent decision and transfer of control via goto's. The class of wfdfs's does not directly model any conventional programming language, and has been constructed to provide a basis for studying data flow computations. Due to the composition rules and execution rules for this class, every wfdfs exhibits a high degree of parallelism and a modular syntactic structure. We have also studied some interesting properties of these schemas.

Comparative Schematology

We have presented a complete set of algorithms for equivalence-preserving translation between flowchart schemas and wfdfs's, and have thus shown that the class of wfdfs's is equivalent in expressive power to the class of flowchart schemas. We have also compared the expressive power between subclasses of flowchart schemas and wfdfs's. It is well-known, from studies in structured programming [Knuth & Floyd 71], that the class of free flowchart schemas properly contains the class of free wfdfs's. Our result showing that the class of α-free wfdfs's is also properly contained in the class of free flowchart schemas is an extension of that result. Due to the properties of open and complete program
other words, \( Z \) is open iff \( S_1 \) and \( S_2' \) are not equivalent. By constructing \( Z \) we have reduced the undecidable equivalent problem for open and complete wfdfs's to the openness problem for complete wfdfs's. The openness problem for complete wfdfs's is thus also undecidable.

Q.E.D.

Figure 4.13 Complete wfdfs \( Z \) for Corollary 4.3-1
schemas, we have also established the equivalence in expressive power between open and complete flowchart schemas and open and complete wfdfs's by an existence proof.

**Decision Problems**

We have established three new undecidability results for the two classes of program schemas we have studied:

(i) Completeness is undecidable for open program schemas.
(ii) Openness is undecidable for complete program schemas.
(iii) Equivalence is undecidable for open and complete program schemas.

Since the property of completeness is closely related to the property of divergence, and it is well-known that divergence is undecidable for program schemas, the first result is not at all surprising. It merely indicates that whether every component in a program is reachable or not bears no relationship to whether the program is complete or not. The converse is not true. Non-openness can be introduced into a non-complete program schema $S$ by concatenating components with the divergent subschemas in $S$. By restricting our attention to complete program schemas, we have eliminated a possible cause for non-openness. (ii) says that even with this restriction, openness is still undecidable. (iii) has been established by reducing Hilbert's Tenth Problem to the equivalence problem for open and complete program schemas. We feel that except for certain operating system modules, every computer program should be open and complete. (iii) says that equivalence is still undecidable even when we restrict ourselves to computer programs with these desirable properties.
5.2 Suggestions for Further Research
Modelling Computer Programs and Comparative Schematology

Two features of programming languages have received very little or no attention in this thesis: parallelism and data structures. Keller [Keller 73] has defined some measures of parallelism for a model of parallel programs and has studied transformations that introduce more parallelism into parallel programs. He has shown that in certain classes of parallel program schemas, locally maximal parallelism implies globally maximal parallelism. An immediate extension of this thesis research is to define and study measures of parallelism in wdfs's and to compare parallel programming using a data flow formalism with parallel programming using other primitives, e.g. the fork and join constructs, semaphores or the parbegin, parend constructs.

In this thesis we have interpreted program schemas as programs on unstructured data. The domains of our interpretations are sets of elementary objects and the functions and predicates operating on these objects treat them as indivisible elements with no internal structure. In other works on schematology, structured data objects and operations on them have been incorporated into program schemas. The data structures studied include stacks with push and pop operations, queues with enqueue and dequeue operations, and arrays with indexing operations. The expressive power of several classes of program schemas incorporating different kinds of data structures has been compared in [Constable & Gries 71]. It would be of great interest to incorporate data structures, e.g. the list structures of LISP, with
primitives `cons`, `car`, `cdr`, `set`, `rplaca` and `rplacd`, into a model of parallel programs and use this extended model to study such problems as:

(i) the degree of parallelism achievable with different representations of data structures, and

(ii) the interaction between parallelism, determinacy and data structure modifications.

These issues are currently being investigated by Isaman [Isaman 75].

**Decision Problems**

The equivalence problem for free flowchart schemas, posed by Paterson [Paterson 70], is still an open problem. This problem is closely related to some open problems in automata theory. We have shown that the classes of free and $\alpha$-free wfdfs's are proper subclasses of free flowchart schemas. The equivalence problems for free and $\alpha$-free wfdfs's are also open. The undecidability of any one of these latter problems implies the undecidability of the equivalence problem for free flowchart schemas. On the other hand, the modular structure of wfdfs's makes it easier to establish decidability results for them than for flowchart schemas. Qualitz [Qualitz 75] has already shown that equivalence in certain interesting subclasses of free wfdfs's is decidable.

Studying the decision problems in free and $\alpha$-free wfdfs's is thus an approach to the solution of the equivalence problem in free program schemas.


[19] Paterson, M.S. Equivalence problems in a model of computation. MIT AI Laboratory TM No. 1,1970


Proof of Lemma 3.2 (Section 3.1.2)

Let $B$ be a nf-block formed by loop formation from a nf-block $B_1$, as shown in Figure A.1.

Generate blocks $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s to simulate $B_1$. Let $G'$ be the subschema constructed from $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s as shown in Figure A.1. $G'$ is equivalent to $G$ in Figure 3.14, but $G'$ is not a wf-block.

**Theorem A.1** Let $\sigma_B$ and $\sigma_G$ be 2 memory states which are equivalent wrt $V_T$. $B$, if entered with $\sigma_B$, reaches the point $Q$ (Figure A.1) $n$ times and $B[\sigma_B]=(\sigma_B', i)$ iff $G'$, if entered with $\sigma_G$, reaches the point $R$ (Figure A.1) $n$ times, and

for $1 \leq k \leq i$, $\beta^1_k[G'[\sigma_G]] = T$, $\beta^1_{i+1}[G'[\sigma_G]] = F$,

and $\sigma_B' = V_T = \alpha_1^1[G'[\sigma_G]]$

---

**Figure A.1** $B$ and $G'$ for Theorem A.1
Proof of Theorem A.1:

By induction on $n$, the number of times $Q$ and $R$ are reached.

**Basis:** $n=1$

Let $B$ be entered with $\sigma_B'$, reach the point $Q$ only once and exit at its $i$-th exit with $\sigma_B'$.

From the block structure of $B$, and the facts that $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s simulate $B_1$:

(i) $B_1[\sigma_B'] = (\sigma_B', i+1)$

(ii) $\sigma_B' = \nu = \alpha_i^1[G_1[\sigma_G]]$

(iii) For $1 \leq k \leq i$, $\beta_i^1[G_k[\sigma_G]] = F$, $\beta_i^1[G_k[G_{k+1}[\sigma_G]] = T$.

From the branching control outcomes in (iii), the structure of $G'$, and the non-interference condition for $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s:

(iv) $R$ in $G'$ is reached only once, and $G'[\sigma_G] = F = G_1[\sigma_G]$.

(v) for $1 \leq k \leq i$, $\beta_i^1[G_k'[\sigma_G]] = F$, $\beta_i^1[G_k'[G_k'[\sigma_G]] = T$, and

$$\sigma_B' = \nu = \alpha_i^1[G_i'[\sigma_G]]$$

Conversely, if the point $R$ is reached only once in $G'$, then there exists $i, 1 \leq i \leq m$, such that:

for $1 \leq k \leq i$, $\beta_k^1[G_k'[\sigma_G]] = F$, $\beta_i^1[G_i'[\sigma_G]] = T$, and

$$G'[\sigma_G] = F = G_1[\sigma_G]$$

Since $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s simulate $B_1$,

$$B_1[\sigma_B] = (\sigma_B', i+1)$$

$$\sigma_B' = \nu = \alpha_i^1[G_1[\sigma_G]] = \nu = \alpha_{i+1}^1[G_1'[\sigma_G]]$$

From the structure of $B$, the point $Q$ in $B$ is reached only once, and $B[\sigma_B] = (\sigma_B', i)$.
Induction Hypothesis: Theorem A.1 holds for $n=\ell$.

Induction Step: $n=\ell+1$

Let $B$ be entered with state $\sigma_B$ and exits at its $i$-th exit with state $\sigma'_B$ after reaching the point $Q$ $\ell+1$ times. There is a state $\sigma''_B$ such that

(i) $B_1[\sigma_B] = (\sigma''_B, 1)$

(ii) $B$ is reentered with state $\sigma''_B$ and reaches the point $Q$ \ell more times.

(iii) $B[\sigma_B] = B[\sigma''_B]$ 

Since $G_1$, $\alpha_i^1$'s and $\beta_i^1$'s simulate $B_1$,

\[
\sigma''_B \quad = \quad \alpha_1^1[G_1[\sigma_G]]
\]

\[
\beta_1^1[G_1[\sigma_G]] = T
\]

From the block structure of $G'$, if $G'$ is entered with $\sigma_G$, $G'$ is reentered with $\sigma''_G$ after reaching the point $R$ once, and

(i) $\sigma''_G = \alpha_1^1[G_1[\sigma_G]] = V_T = \sigma''_B$

(ii) $G'[\sigma_G] = G'[\sigma''_G]$

(iii) $G'$ exits after reaching the point $R \ell$ more times.

Similarly, if $G'$ is entered with $\sigma_G$ and reaches the point $R \ell+1$ times, there exist states $\sigma''_B$ and $\sigma''_G$ such that

(i) $G'$, when entered with $\sigma''_G$, reaches the point $R \ell$ more times. $B$, when entered with state $\sigma''_B$, reaches the point $Q \ell$ more times.

(ii) $G'[\sigma_G] = G'[\sigma''_G], \quad B[\sigma_B] = B[\sigma''_B]$

It then follows from the induction hypothesis that $B$ and $G'$ satisfy the statement of the theorem for $n=\ell+1$

Q.E.D.
Proof of Lemma 3.2:

Since $G$ in Figure 3.14 is equivalent to $G'$ in Figure A.1, Theorem A.1 also holds for $B$ and $G$. Hence if $\sigma'_B = \nu = \sigma'_G$, then:

(i) $B$ terminates on $\sigma'_B$ iff $G$ terminates on $\sigma'_G$.

(ii) If $B$ terminates and $B[\sigma'_B] = (\sigma'_B, i)$, then $G$ terminates and for $1 \leq k \leq i$, $\beta^l_1[\sigma'_G] = F$, $\beta^l_{i+1}[\sigma'_G] = T$

$\sigma'_B = \nu = \alpha^l_{i+1}[G[\sigma'_G]]$

Q.E.D.