MIT/LCS/TM-66

FORMAL PROPERTIES OF WELL-FORMED DATA FLOW SCHEMAS

Clement Kin Cho Leung

June 1975

TM-66

FORMAL PROPERTIES OF WELL-FORMED DATA FLOW SCHEMAS

Clement Kin Cho Leung

June 1975

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

PROJECT MAC

CAMBRIDGE MASSACHUSETTS 02139

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, α-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. α-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 comparision 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 α-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

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 & Fosseen 73]

Some of the applications of program schematology are:

<u>Studying properties of computer programs</u>

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.

<u>Comparing the expressive power of programming language features</u>

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 (wfdfs'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 wfdfs 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 wfdfs's and used it to transform free wfdfs's into maximally parallel form.

Qualitz [Qualitz 72] studied the properties of weakly productive schemas which is an extension of wfdfs's. In a wfdfs, even if all of their inputs are available, operators on alternate data flow paths (for example, the <u>then</u> and <u>else</u> 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 wfdfs'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 prlperties 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 wfdfs's, and the class of flowchart schemas, are formally defined in Chapter 2. In Section 3.1 algortihms are presented for translating a program schema in the class of wfdfs'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, α-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. α-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 α-free program schemas. In Section 3.3 we establish the equivalence in expressive power between open and complete wfdfs'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.

Chapter Two

Definitions

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, α-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 <u>variables</u> Var = $\{V_i | i>0\}$

ii. an infinite set of <u>function symbols</u> Func = $\{F_i^j | i>0, j>0\}$

iii. an infinite set of <u>predicate symbols</u> Pred = $\{P_i^j | i>0, j>0\}$

iv. a set of <u>simple statements</u> of the following form:

<u>Start statement</u>

$$\boxed{\begin{array}{c}\text{START}\\ \{X_1,\ldots,X_n\}\end{array}}$$

$X_1,\ldots,X_n \in$ Var. The X_i's are the <u>input variables</u>.

<u>Assignment statement</u>

$$\boxed{X \leftarrow E}$$

X is a variable. E is either a variable or an expression of the form $F_i^j(X_1,\ldots,X_j)$ with $F_i^j \in$ Func and $X_1,\ldots,X_j \in$ Var. X is the <u>left hand side</u> (LHS), E the <u>right hand side</u> (RHS), of the assignment. The variable(s) that occurs in E is the variable <u>accessed</u> by the assignment statement.

Predicate statement

$P_i^j \in$ Pred

$X_1,\ldots,X_j \in$ Var

$P_i^j(X_1,\ldots,X_j)$

The variables $X_1,\ldots,X_j$ are the variables accessed by the predicate statement.

Halt statement

$X_1,\ldots,X_n \in$ Var

HALT

$\{X_1,\ldots,X_n\}$

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 $e=(v_1,v_2)$ is an edge in E, then head(e)$=v_1$, tail(e)$=v_2$. A path (possibly infinite) through G is a sequence $(v_1,\ldots,v_n)$ of nodes such that for all i, $1<i\leq n$, $(v_{i-1},v_i) \in E$. An edge e is an incoming branch of a vertex v if tail(e)=v. An edge e is an outgoing branch of a vertex v if 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:

i. V_S is a set of simple statements.

ii. 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=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 <u>interpretation</u> for flowchart schemas consists of a <u>domain</u> D and a function I which assigns

to every function symbol F_j^i, a total function $f_j^i: D^i \to D$,
to every predicate symbol P_j^i, a total predicate $p_j^i: D^i \to \{\underline{T}, \underline{F}\}$.

Given an interpreted schema (S,I), an input to (S,I), $In_{(S,I)}$, is a function which assigns to every input variable of S an element of D. Formally,

$$\forall x_i \in Var,\ In_{(S,I)}(x_i) = \begin{cases} \alpha_i \in D, \text{ if } x_i \text{ is an input variable of S} \\ \text{undefined otherwise} \end{cases}$$

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

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

Let Ω be an object not in D. Ω 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 <u>value sequence</u> A for P under interpretation I with input In is an infinite sequence $A = (a_1, \ldots, a_i, \ldots)$ where $a_i=(a_{i1}, \ldots, a_{ij}, \ldots) \in (D \cup \Omega)^{\infty}$. 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 $X_1, \ldots, X_m$.

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

2. At the i-th step, $i>1$,

   i. if P_i is an assignment statement with LHS V_e and RHS E then

$$a_{ik} = \begin{cases} a_{(i-1)j} & \text{if } k=e \text{ and } E=V_j \\ f_j^n(a_{(i-1)q1},\ldots,a_{(i-1)qn}) & \text{if } k=e, \\ & E=F_j^n(V_{q1},\ldots,V_{qn}) \text{ and } I(F_j^n)=f_j^n \\ a_{(i-1)k} & \text{otherwise} \end{cases}$$

ii. if P_i is a predicate statement, then for all j

$$a_{ij} = a_{(i-1)j}$$

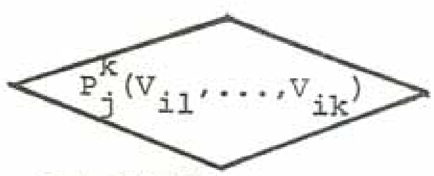
iii. if P_i is the Halt statement, then P_i is the last statement in P since the Halt statement has no outgoing branch. For all $m \geq i$,

$$a_{mj} = a_{(i-1)j}$$

An <u>execution sequence</u> 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:

$I(P_j^k)=p_j^k$



the edge (P_i,P_{i+1}) is labelled by <u>T</u> iff

$$p_j^k(a_{(i-1)i1},\ldots,a_{(i-1)ik}) = \underline{T}$$

the edge (P_i,P_{i+1}) is labelled by <u>F</u> iff

$$p_j^k(a_{(i-1)i1},\ldots,a_{(i-1)ik}) = \underline{F}$$

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 <u>terminates</u> on input In under interpretation I iff the execution sequence for (S,I,In) is finite. Otherwise S <u>diverges</u> under I and In.

The <u>output</u> of a schema S on input In under interpretation I, denoted by Out(S,I,In), is defined as follows:

i. if S terminates under I and In, $\{X_{i1},\ldots X_{in}\}$ is the set of output variables, and the execution sequence contains m statements, then

$$\mathrm{Out}(S,I,In) = (a_{m(i1)},\ldots,a_{m(in)})$$

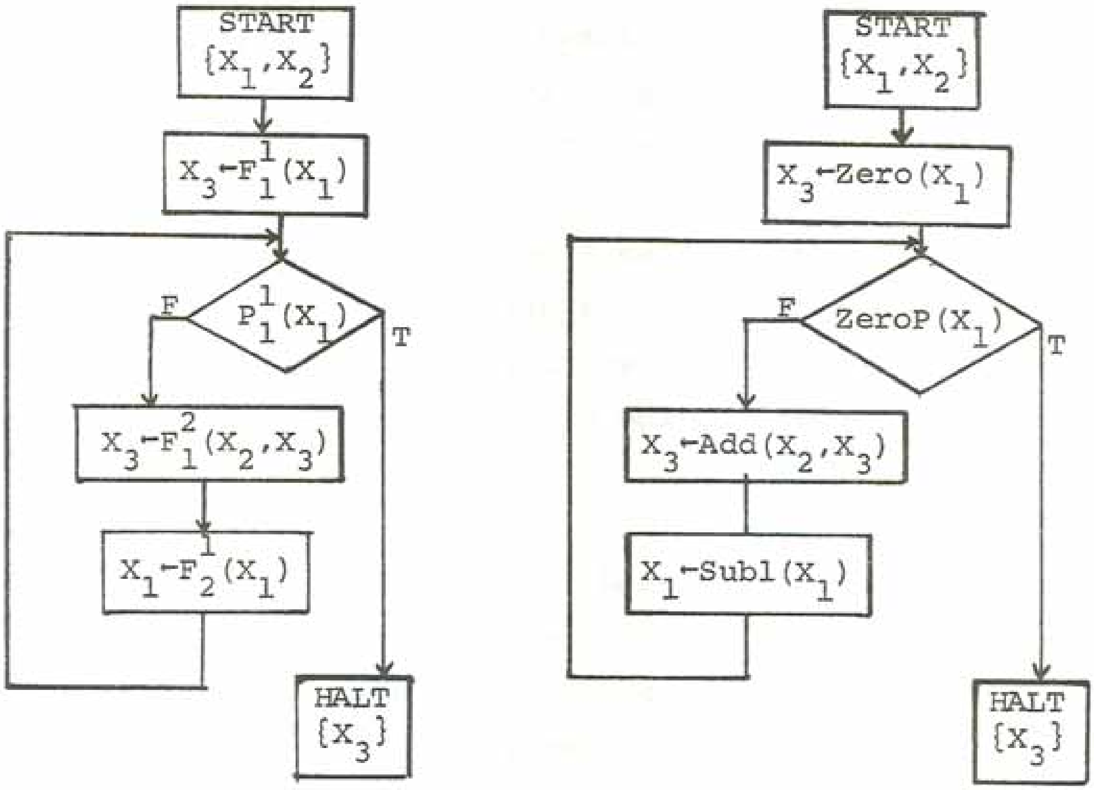
ii. if S diverges under I and In, then

$$\mathrm{Out}(S,I,In) = \Omega^n$$

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.

<u>2.2 Well-formed data flow schemas</u>

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



Interpretation I:

$D \rightarrow N$, the natural numbers

$F_1^1 \rightarrow$ Zero, the function which always returns 0

$F_2^1 \rightarrow$ Sub1, the predecessor function

$F_1^2 \rightarrow$ Add, the addition function

$P_1^1 \rightarrow$ ZeroP, the predicate which tests for 0.

Figure 2.1 An example of a flowchart schema

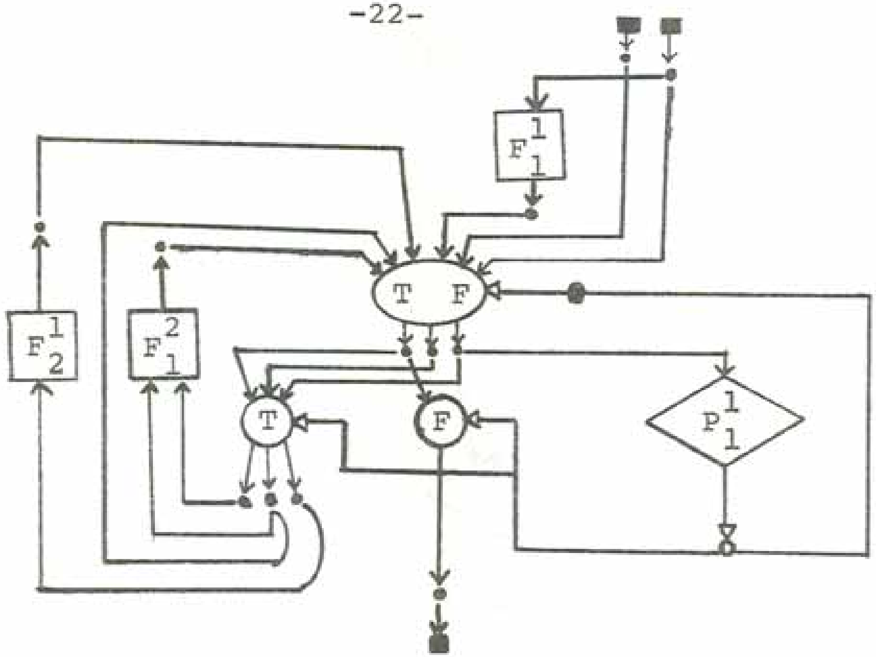


Figure 2.2 An Example of a wfdfs

To define the class of wfdfs'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 <u>bipartite directed graph</u> on V is a directed graph (V,E) with $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$. An <u>acyclic</u> 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 <u>output arc</u> of a and is an <u>input arc</u> of b.

<u>Basic Constructs</u>

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

1. <u>Link nodes</u> 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

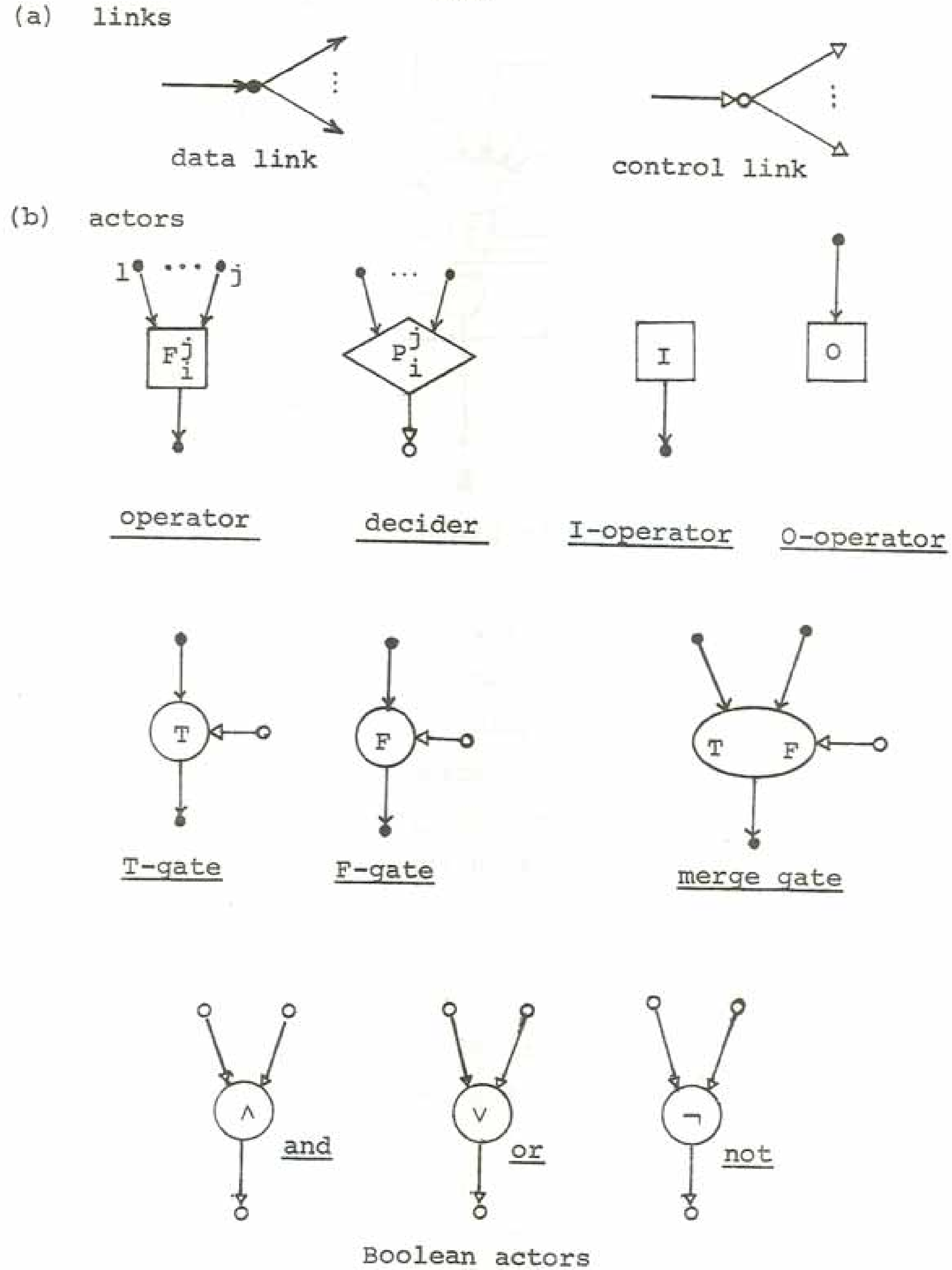


Figure 2.3 Node types for wfdfs

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)

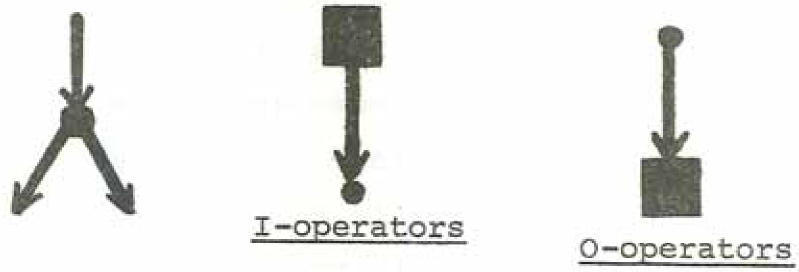
2.Actor nodes The actor nodes are the processing elements of a wfdfs. The different kinds of actor nodes are shown in Figure 2.3b. A link node is an input(output) link of an actor if the input(output) arc of the actor is an output (input) arc of the link node. The operator nodes and decider nodes are 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 connected to.

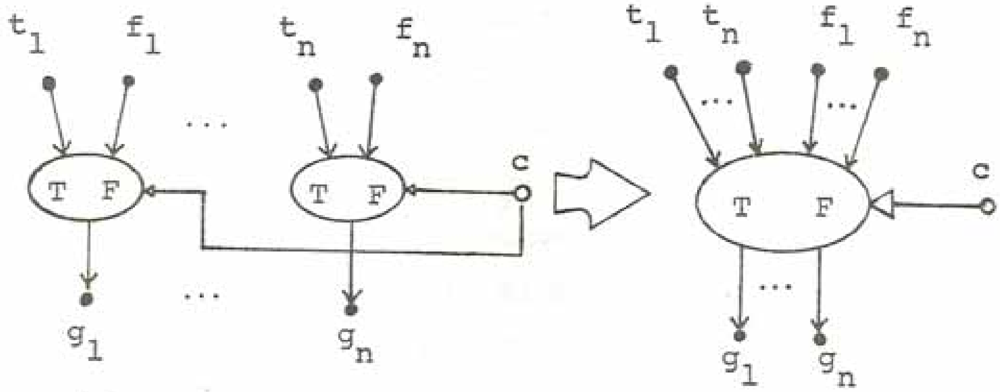
Diagramming Conventions

Before we define the composition rules for wfdfs'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

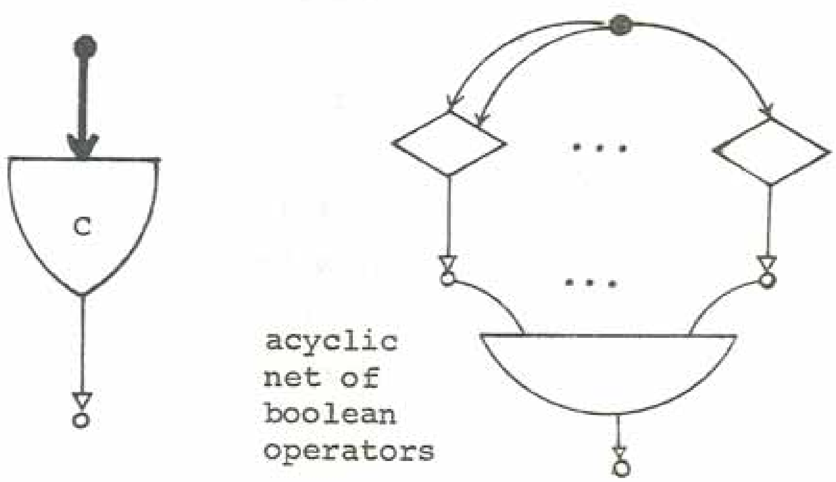


Figure 2.4 Diagramming Conventions

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 <u>multiple input-multiple output</u> 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 wfdfs is defined inductively.

A level-0 (m,n) wfdfs 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 O-operators and all the O-operators of S are leaves.

To describe the composition of wfdfs's we introduce a special operator Id (for identity operator) and define an Id-subgraph of a wfdfs.

Let Id be a function symbol, Id $\notin \{F_i^j \mid i\geq 1, j\geq 1\}$. An Id-operator is a single input, single output operator labelled by the special function symbol Id. An Id-subgraph D of a wfdfs 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 wfdfs 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 Id.

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

A set of data links D is joined to a set of I-operators L by joining data links in D to I-operators in L such that:

i. Every data link $d \in D$ is joined to at least one I-operator $\ell \in L$.

ii. For every I-operator $\ell \in L$ there is a data link $d \in D$ which is joined to ℓ.

Similarly a set of O-operators L is joined to a set of data links D by joining O-operators in L to data links in D such that:

i. Every O-operator in L is joined to a data link in D.

ii. For every data link $d \in D$ there is an O-operator $\ell \in L$ which is joined to d.

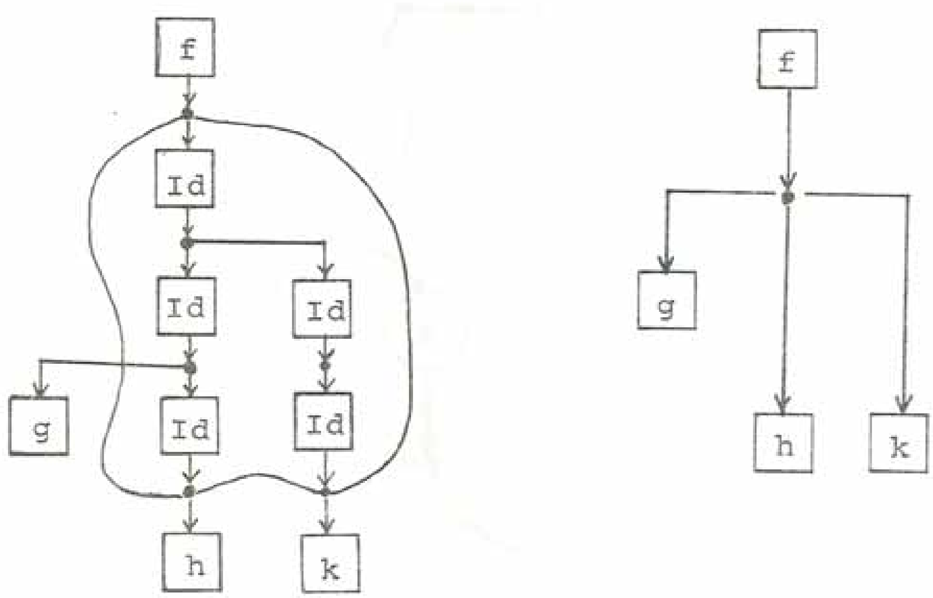


Figure 2.5 Collapsing an Id-subgraph

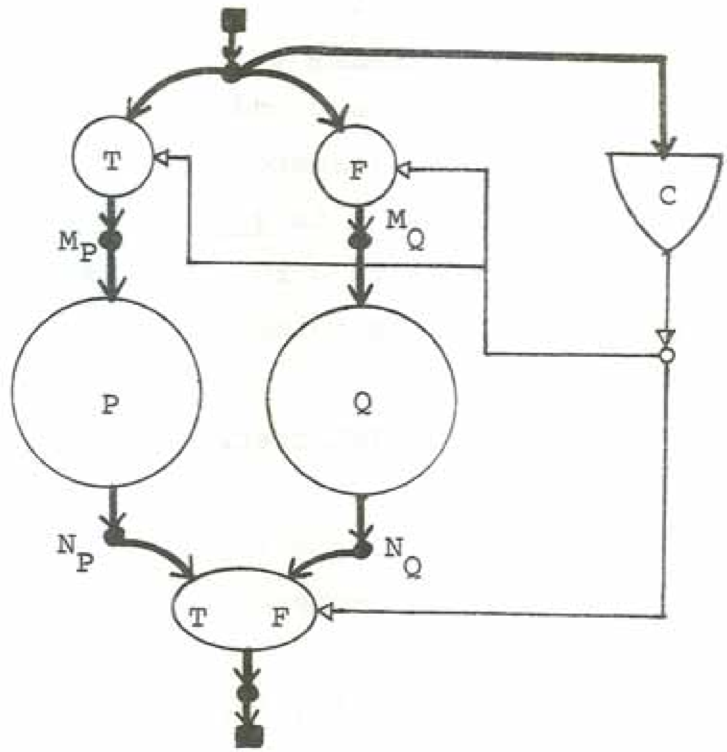


Figure 2.6 A Conditional wfdfs

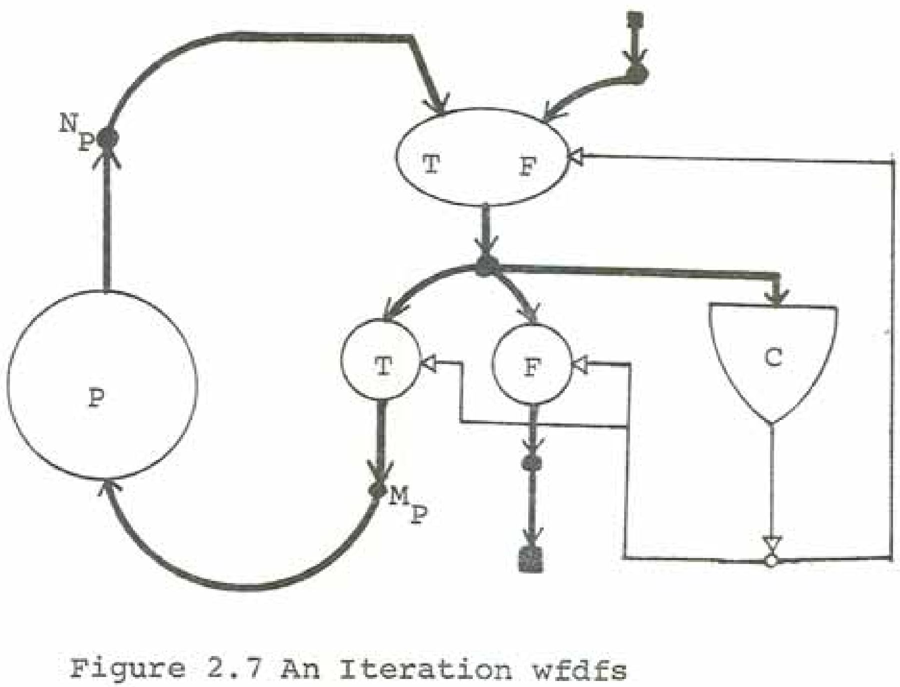


Figure 2.7 An Iteration wfdfs

A <u>level-i+1 (m_S,n_S) conditional wfdfs</u> S is constructed from two level-i wfdfs'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 <u>level-i+1 (m_T,n_T) iteration wfdfs</u> T is constructed from a level-i wfdfs 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 <u>level-i+1 (m,n) wfdfs</u> is either:

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

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

iii. a level-i+1 (m,n) iteration wfdfs, 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 $H_1,\ldots,H_k$ at I-operators $i_1,\ldots,i_j$, create a new data link d_ℓ; <u>join</u> ℓ to d_ℓ and <u>join</u> d_ℓ to $i_1,\ldots,i_j$. Perform this joining operation for every O-operator which is the input to another subschema.

b. Collapse all the Id-subgraphs formed in (a).

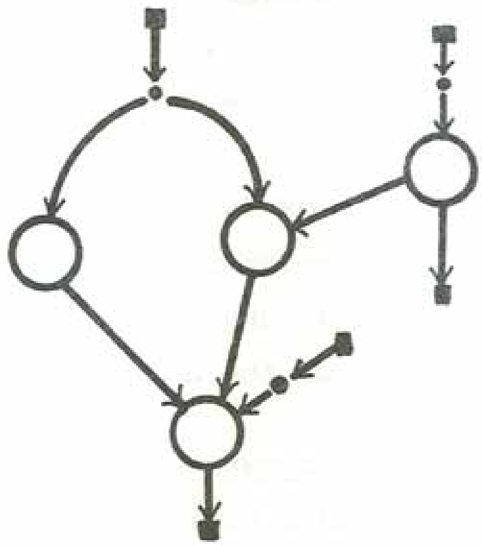


Figure 2.8 Acyclic composition of wfdfs

The acyclic composition has m I-operators and n O-operators.

A conditional wfdfs 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 wfdfs 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 wfdfs we initialize it by putting control values on certain control arcs of the wfdfs, 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 $\underline{F}$ 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.

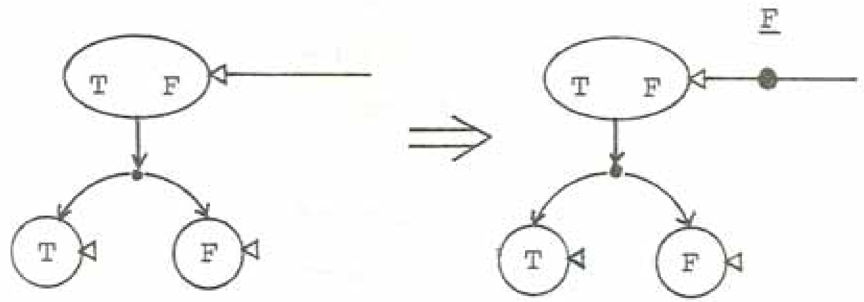


Figure 2.9 Initialization of iteration subschemas

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 $f_i^j: D^j \to D$, and to every predicate symbol P_i^j a total predicate $p_i^j: D^j \to \{\underline{T}, \underline{F}\}$. 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 {<u>T</u>,<u>F</u>,<u>null</u>} with each control arc of S.

The <u>initial configuration</u> 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 <u>data</u> tokens, <u>T</u> tokens or <u>F</u> tokens according to the associated values.

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

1.η_0 is the initial configuration of (S,I,In).

2.Each η_{i+1} is obtained from η_i by <u>the firing of some enabled node</u> of S in η_i. The <u>firing rules</u> 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 η_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

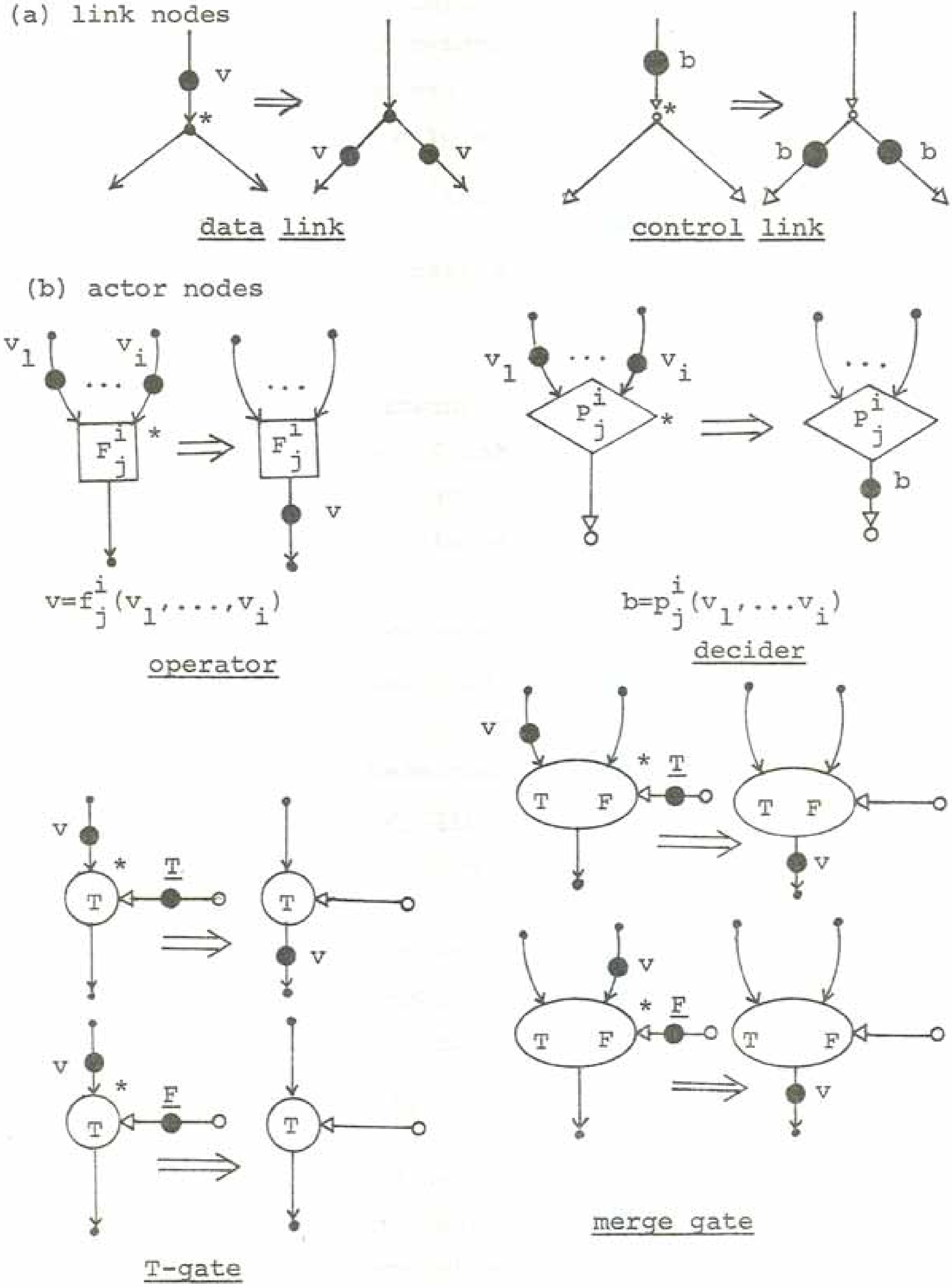


Figure 2.10 Firing rules for actors

negated. Boolean actors behave similarly to operators. I-operators and O-operators are not enabled.

3. If η is a finite sequence of n configurations, then in the terminal configuration η_{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 η is a finite configuration sequence for (S,I,In), the <u>output</u> of (S,I,In) under η, denoted by $\underline{Out}^{\eta}_{(S,I,In)}$, is a function which assigns to the input arc of every O-operator ℓ the data value associated with the input arc of ℓ in the terminal configuration of η.

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 <u>termination</u> and <u>output</u> to be meaningful we state Theorem 2.2-1 which is proved in [Fosseen 72].

<u>Theorem 2.2-1</u>: If α and β are configuration sequences for the interpreted wfdfs (S,I) under input In, then

i. α is finite if and only if β is finite.

ii. if α,β are finite, $Out^{\alpha}_{(S,I,In)} = Out^{\beta}_{(S,I,In)}$.

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 wfdfs's are determinate.

A wfdfs S under interpretation I and input In <u>terminates</u>

if and only if all configuration sequences (or any configuration sequence, by Theorem 2.2-1) are finite. Otherwise S <u>diverges</u> under I and In.

Let S be a (m,n) wfdfs. If D is the domain associated with an interpretation I for S, let Ω be an object not in D, denoting the undefined object. The <u>output</u> for (S,I) under input In, denoted by $Out_{(S,I,In)}$, is :

i. If (S,I,In) terminates, $Out_{(S,I,In)} = Out^{\alpha}_{(S,I,In)}$ for any configuration sequence α of (S,I,In).

ii. If (S,I,In) diverges, $Out_{(S,I,In)} = \underline{\Omega}$, where $\underline{\Omega}$ assigns the object Ω to the input arc of every O-operator of S.

Given a wfdfs 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 η is a configuration sequence of S, the <u>firing sequence</u> for η, denoted by $\tau=\tau_1\ldots\tau_i\ldots$, is defined as follows:

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

p^F or b^F where $p \in P$, $b \in B$

if η_i is obtained from η_{i-1} by firing p or b, and the outcome is F.

If τ_i is b^T or b^F, and the output link of b is the output control link of a decision structure $\varkappa$, we will also refer to τ_i being $\varkappa^T$ or $\varkappa^F$ 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 wfdfs's and define some properties for them.

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

Let S be a (m,n) wfdfs. 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 O-operators in S, and hence the set of output arcs of the I-operators and the set of input arcs of the O-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 Ω, an input In to (S,I) or (T,I) can be denoted by a m-tuple in D^m. An output Out for (S,I,In) or (T,I,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, 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 $F_j^i \in$ Func,

$$F_j^i \sqcap d_1 \sqcap \ldots \sqcap d_i \in D_m$$

where $\sqcap$ 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 $f_j^i : (D_m)^i \to D_m$, such that for all $d_1, \ldots, d_i \in D_m$,

$$f_j^i(d_1, \ldots, d_i) = F_j^i \sqcap d_1 \sqcap \ldots \sqcap d_i$$

(ii) to every predicate symbol $P_j^i \in$ Pred, a predicate

$$p_j^i : (D_m)^i \to \{\underline{T}, \underline{F}\}$$

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:

$1 \le i \le m, \quad In(i) = \delta_i$

Since the arity j of a function f_i^j is encoded in the function symbol F_i^j , every string generated in a computation under (S, I_f, In) can be transformed umambiguously 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 usefull 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 wfdfs), 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 wfdfs 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,

$$Out_{(S_1,I,In)} = Out_{(S_2,I,In)}$$

Let C_1 be a branching control in a program schema S. Let η be a computation sequence of S under an interpretation I. Let $\underline{C}_1$ be a decision in η made with C_1. $\underline{C}_1$ is restricted in η if the outcome of $\underline{C}_1$ is logically implied by some previously made branching control decision outcomes in η. C_1 is restricted in η if some decision by C_1 in η 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 α-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 α-freedom given below can be more easily understood in this light. In Section 3.2 some differences between free and α-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 In such that P is the execution sequence under (S,I,In).

Proof:

(if) Assume S is not free, then there is an interpretation I and an input In such that the execution sequence P of (S,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 In such that P is the execution sequence for (S,I,In) as follows:

I is a free interpretation for S. I maps each function symbol F_j^i into the function f_j^i: $D^i \to D$, where D is the associated domain of I:

$$f_j^i(d_1,\ldots,d_i) = F_j^i \square d_1 \square \ldots \square d_i, \quad d_1, \ldots, d_i \in D$$

I maps each predicate symbol P_j^i into the predicate p_j^i, $p_j^i : D^i \to \{\underline{T}, \underline{F}\}$:

$$p_j^i(s_1,\ldots,s_i) = \begin{cases} \underline{T} \text{ if } p_k \text{ is a node in P; } p_k \text{ consists of applying } P_j^i \text{ to the variables } x_{k1},\ldots, x_{ki};\ x_{k1} \text{ has value } s_1, \ldots, x_{ki} \text{ has value } s_i; \text{ and the edge } (p_k, p_{k+1}) \text{ is labelled by } \underline{T} \\ \underline{F} \text{ otherwise} \end{cases}$$

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

$1 \leq i \leq$ no. of inputs of S, $In(i) = \delta_i$

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 wfdfs 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 wfdfs being the execution sequence under some interpretation and input, we define α-freedom for program schemas.

α-freedom

A flowchart schema S is α-free if for every prefix $p_1 \dots p_n p_{n+1}$ of every execution sequence p of S, p_n being a predicate statement and (p_n, p_{n+1}) being labelled by T (F), $p_1 \dots p_n p'_{n+1}$ is also a prefix of some execution sequence of S, (p_n, p'_{n+1}) being labelled by F (T).

Let $\tau = \tau_1 \dots \tau_n$ be a prefix of the firing sequence of a configuration sequence Π of a wfdfs S, τ_n being the outcome of a branching control in S. A wfdfs S is α-free if for every such prefix τ of S, there is a prefix τ' of the firing sequence of a configuration sequence Π' of S such that:

i. For $1 \le i \le n$, τ_i and τ'_i are the firings of the same node in S.

ii. τ_n and τ'_n are the opposite outcomes of the branching control.

iii. For $1 \le k < n$, if τ_k and τ'_k are the firings of a branching control ℓ in S, τ_k and τ'_k must be identical outcomes of ℓ.

Theorem 2.3-3 A flowchart schema S is free if and only if it is α-free.

Proof:

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

(only if) It is obvious that if S is not α-free, S is not free. Q.E.D.

A redundant decision structure in a wfdfs S is a decision structure D whose decision outcome is always D^T or is always

D^F irrespective of the outcome of the input decisions.

<u>Theorem 2.3-4</u> If S is a free wfdfs which does not contain any redundant decision structure, S is α-free.

Proof:

Let $\tau_1 \ldots \tau_n$ be a prefix of a firing sequence τ of S under interpretation I and input In, τ_n being the outcome of a decision structure decision. Let $\tau_{i1}, \ldots \tau_{ik}$ be the outcomes of the input decisions to τ_n.

i. Assume $\tau_n = D^T$. By hypothesis there exists a combination of input decision outcomes $\tau'_{i1}, \ldots, \tau'_{ik}$ 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':

τ_{ij} has the same outocme as τ'_{ij}, $1 \le j \le k$.

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 $\tau_n = D^F$, 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 α-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.

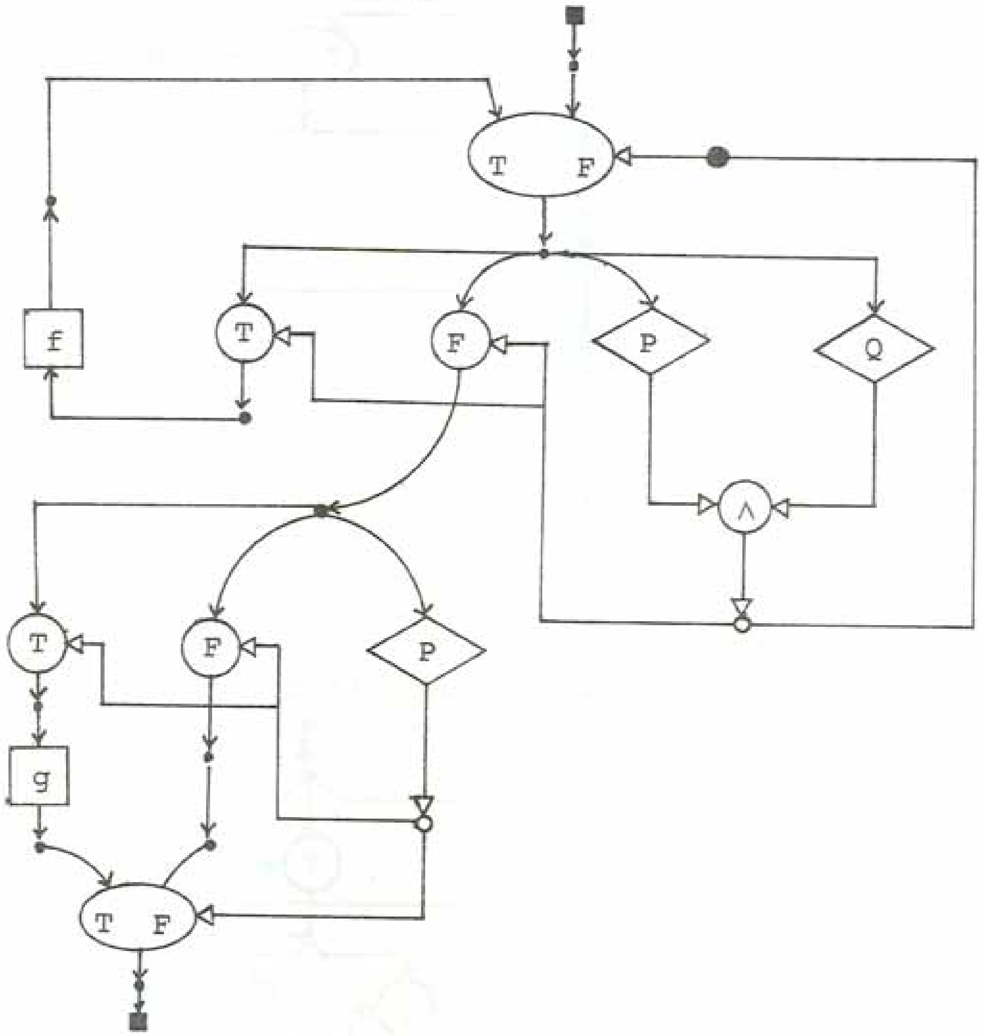


Figure 2.11 A wfdfs which is α-free but not free

Openness

A program schema S is <u>open</u> 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 <u>complete</u> if every finite prefix

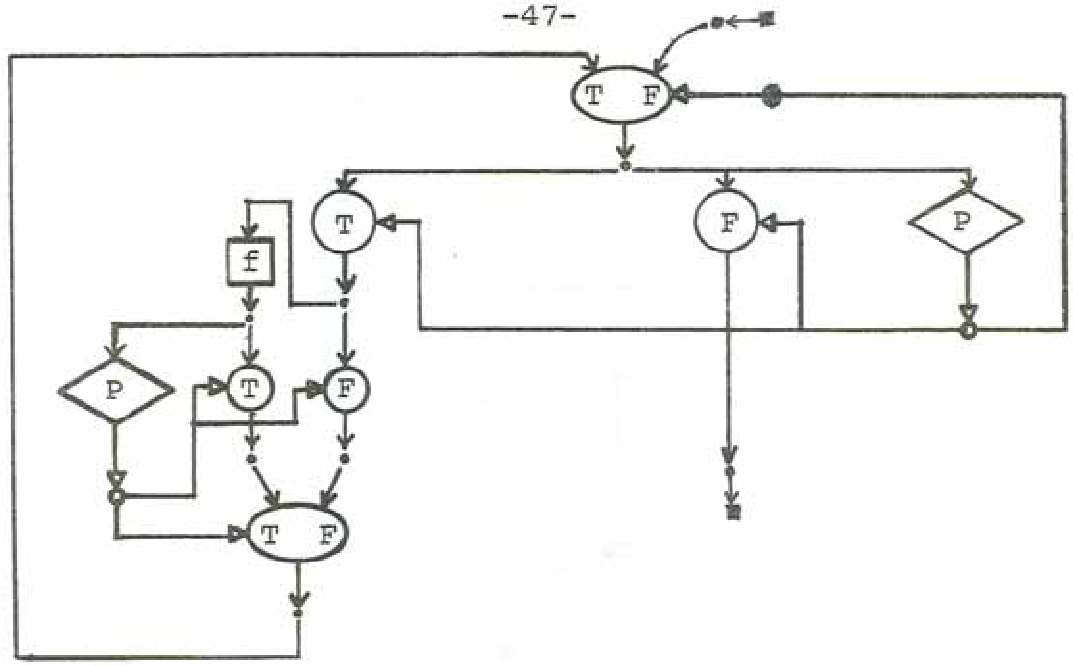


Figure 2.12 A wfdfs which is open but not complete

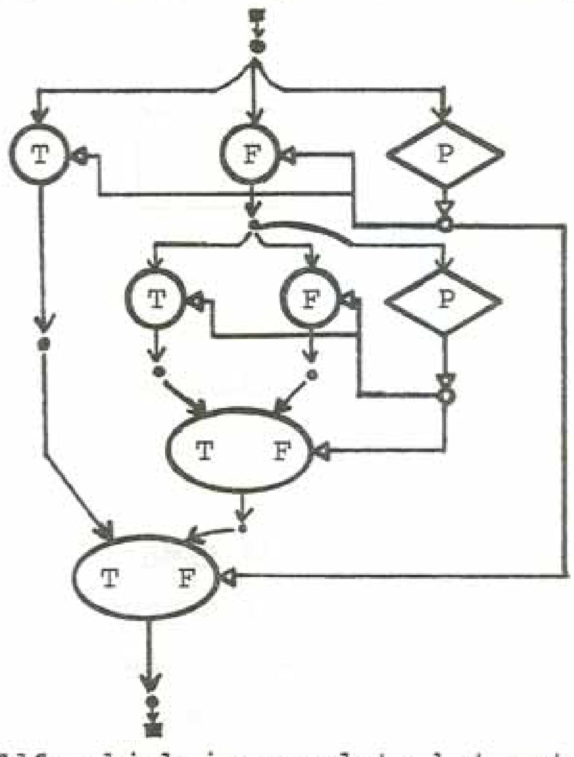


Figure 2.13 A wfdfs which is complete but not open

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, α-freedom, openness and completeness, Theorem 2.3-5 and 2.3-6 should be obvious.

<u>Theorem 2.3-5</u> Let S_1, S_2 be flowchart schemas.

i. S_1 is free if and only if S_1 is α-free.

ii. If S_1 is free, S_1 is open. [1]

iii. If S_1 is free and contains no predicate free loop, S_1 is open and complete.

<u>Theorem 2.3-6</u> Let S_1, S_2 be wfdfs's.

i. If S_1 is free and contains no redundant decision structure, S_1 is α-free.

ii. If S_1 is α-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.

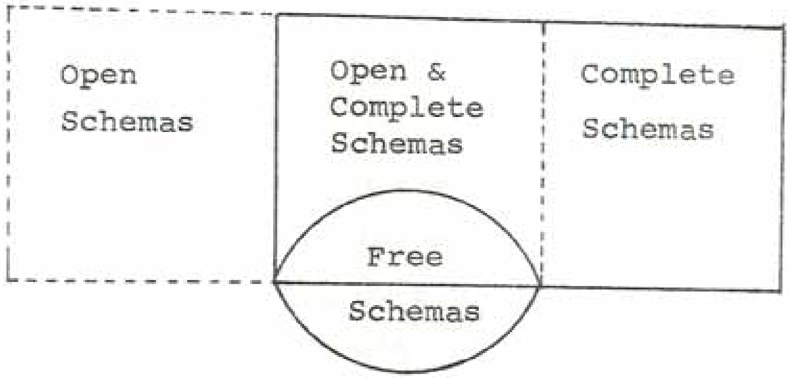


Figure 2.14 Relationship between free, open and complete program schemas.

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 ξ of a computation sequence η of S_1 that cannot be extended to a finite computation sequence of S_1. Let I be the free interpretation under which η is a computation sequence for S_1. Let A be the set of decision outcomes contained in ξ, under I.

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

$\cup$ { $p^k(d_1,\ldots,d_k)=\underline{F}$ | $p^k \in$ Pred,
$d_1, \ldots, d_k \in D$, the domain associated with I,
$p^k(d_1,\ldots,d_k)$ is a decision made in ξ and has outcome $\underline{F}$ under I }

A is finite since ξ 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 η_h be the computation sequence of S_2 under h. Let $A_h \subseteq A$ be the subset of A which is contained in η_h. Since A is finite, A_h is finite and hence there is a finite prefix ξ_h of η_h which contains A_h. ξ_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 <u>T</u>, or

(iii) P always has outcome <u>F</u>.

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 ever 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 mere 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 wfdfs's. In Section 3.1 we establish the equivalence in expressive power between flowchart schemas and wfdfs's. Algorithms are presented for translating a flowchart schema into an equivalent wfdfs and for trnaslating a wfdfs into an equivalent flowchart schema. In Section 3.2 we prove that the class of free flowchart schemas properly contains the class of free wfdfs's and the class of α-free wfdfs's. In Section 3.3 classes of open and complete program schemas are studied.

3.1 Translation between flowchart schemas and wfdfs's

There are three steps involved in translating a flowchart schema into an equivalent wfdfs:

1. Translate a flowchart schema S into an equivalent <u>normal form flowchart schema</u> (abbreviated <u>nfs</u>) T. [Engeler 71]
2. Translate a nfs T into a <u>well-formed flowchart schema</u> (abbreviated <u>wfs</u>) W. [Ashcroft & Manna 71]
3. Translate a well-formed flowchart schema W into a wfdfs Z.

3.1.1 Translating a flowchart schema into normal form

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

A nf-block is defined recursively as follows:

1. A <u>basic nf-block</u> is any acyclic, tree-like, single entrance

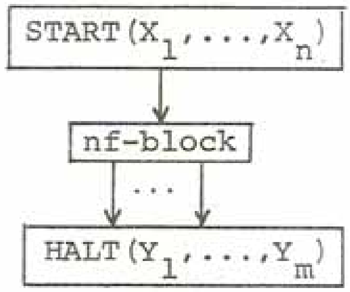


Figure 3.1 Skeleton of a nfs

subchema 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 subchema derived from a finite number of steps from basic nf-blocks using rules 1, 2 and 3.

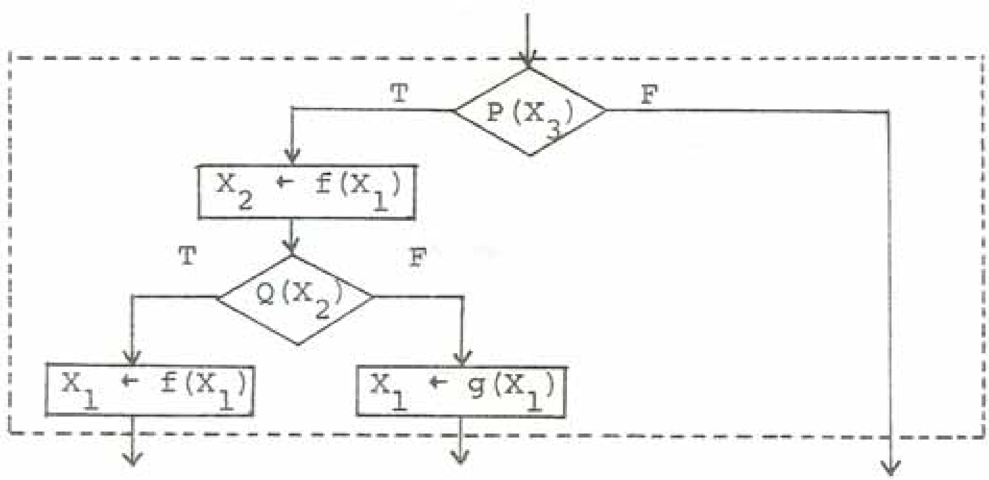


Figure 3.2 A basic nf-block

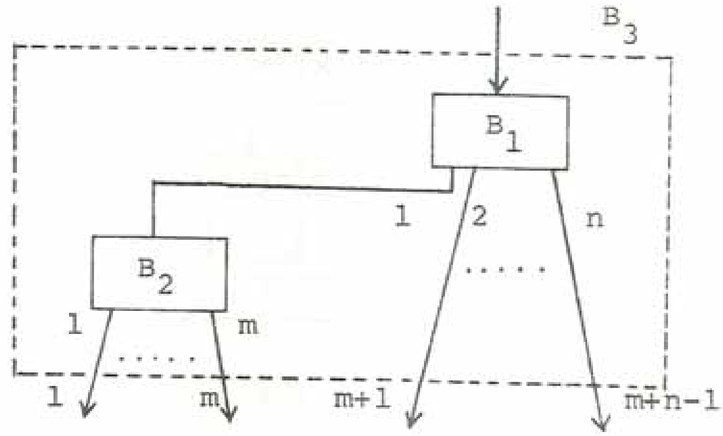


Figure 3.3 Blocks composition

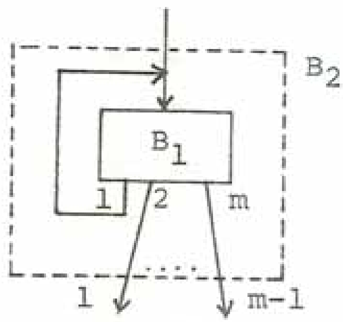


Figure 3.4 Loop formation

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.

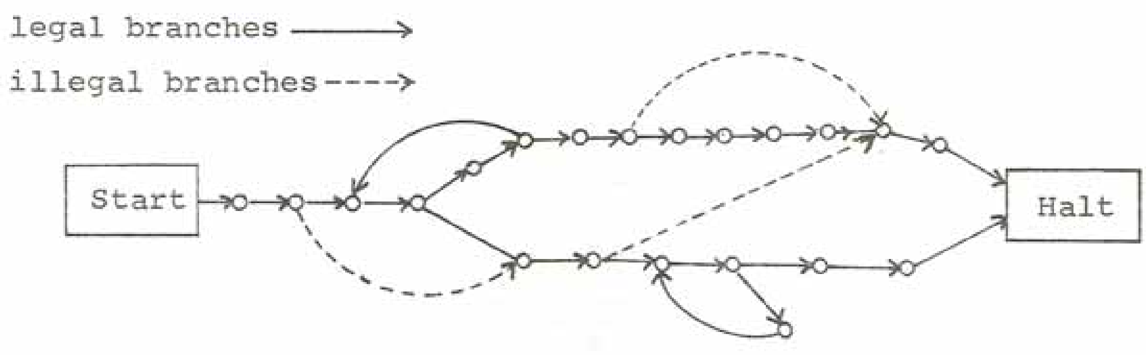


Figure 3.5 Structure of a normal form flowchart schema

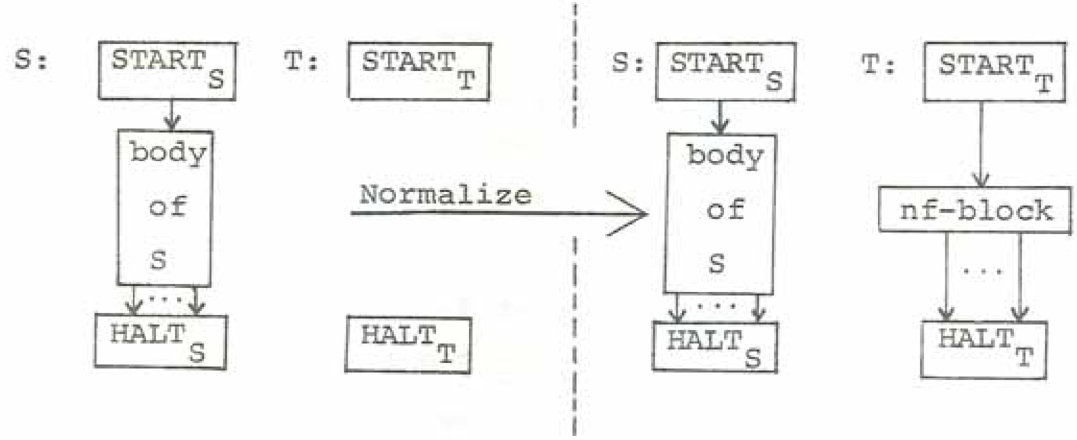


Figure 3.6 Constructing a nfs using Normalize

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, $\underline{START}_T$ and $\underline{HALT}_T$, which are copies of the START statement ($\underline{START}_S$) and the HALT statement ($\underline{HALT}_S$) 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, $p = p_0 \cdots p_i \cdots p_n$, such that

p_0 is the START statement.

p_i is S_1, p_n is S_2

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

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 Snode

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 Snode is a <u>Predicate</u> statement then

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

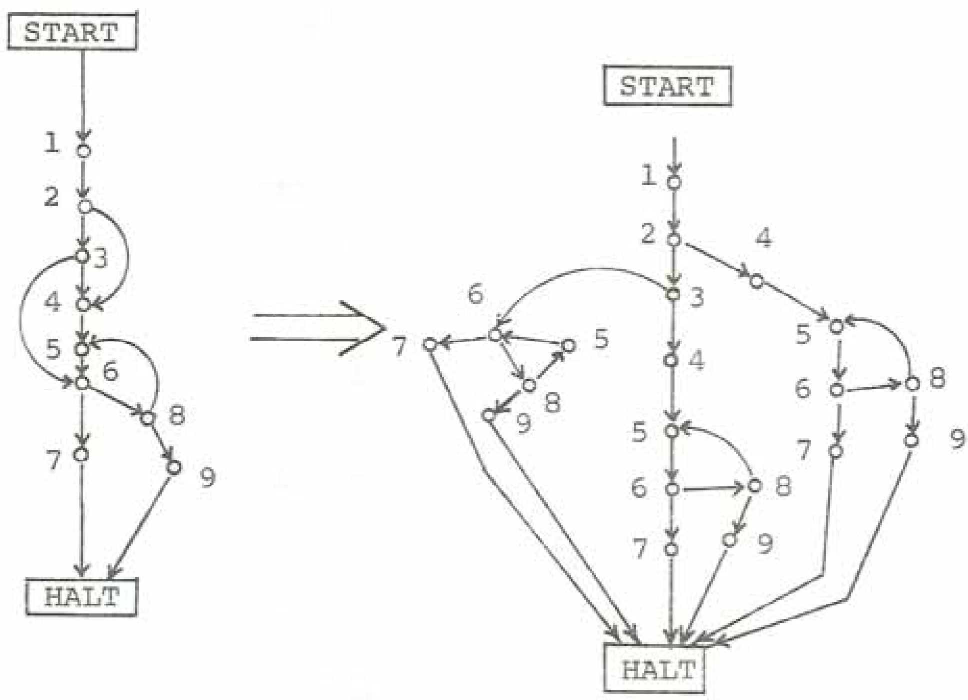


Figure 3.7 Normalization of flowchart schemas

3.1.2 Translating nfs's into wfs's

The class of wfs's, as the class of wfdfs's, is a model for "gotoless" computer programs. A wfs is a flowchart schema whose body is a <u>well-formed block</u> (abbreviated wf-block). A wf-block is a single-entry, single-exit subchema, defined recursively as follows:

1. The <u>empty schema</u>, denoted by ∅, is a wf-block.
2. An <u>assignment statement</u> is a wf-block.
3. <u>Linear Concatenation</u>: 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. <u>Conditional Composition</u>: 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. <u>Iteration</u>: 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).

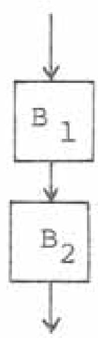


Figure 3.8
Linear
Concatenation

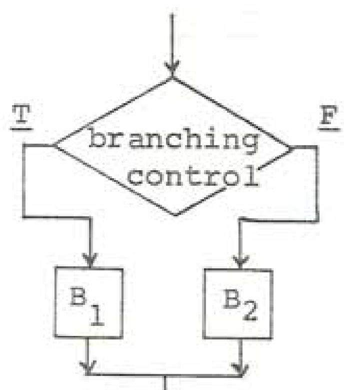


Figure 3.9
Conditional
Composition

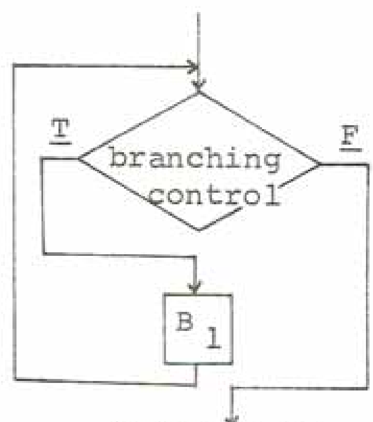


Figure 3.10
Iteration in
wfs

As in wfdfs'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 $P = \{P_j^i(X_1,\ldots,X_i) \mid P_j^i \in \mathrm{Pred},\ X_k \in \mathrm{Var},\ 1 \le k \le i\}$ be the set of simple predicate statements. Syntactically a <u>branching control</u> <BC> in a wfs is an expression generated by the following BNF:

<BC> ::= P | P ∧ <BC> | P ∨ <BC> | ¬ <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 α-free wfs need not be free, for the same reason that an α-free wfdfs need not be free.

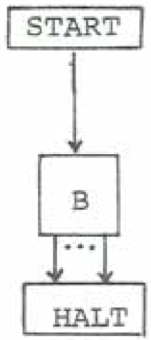
Our next goal is to translate nfs's into wfs's. The translation algorithm, Algorithm 3.2, is based on an algorithm due to Ashcroft and Manna [Ashcroft & Manna, 71]. Algorithm 3.2 describes precisely how to apply the ideas embodied in Ashcroft and Manna's algorithm to translate nfs's into wfs's. The major 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, $1 \le i \le m$, 2 blocks α_i and β_i. α_i is a sequence of assignment statements. β_i is a sequence of assignment statements Φ_i followed by a branching control θ_i.

The blocks G, α_i's and β_i'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 β_i'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 α_i. If we furthermore impose the requirement that none of the β_i's modified the variable values accessed by the α_i's and the other β_k's, $k \neq i$, then B can be simulated by a wf-block constructed from G, α_i's and β_i's as shown in Figure 3.11. Algorithm 3.2 constructs G, α_i's and β_i'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, α_i's and β_i's, and of Algorithm 3.2, we need soem additional terminology.



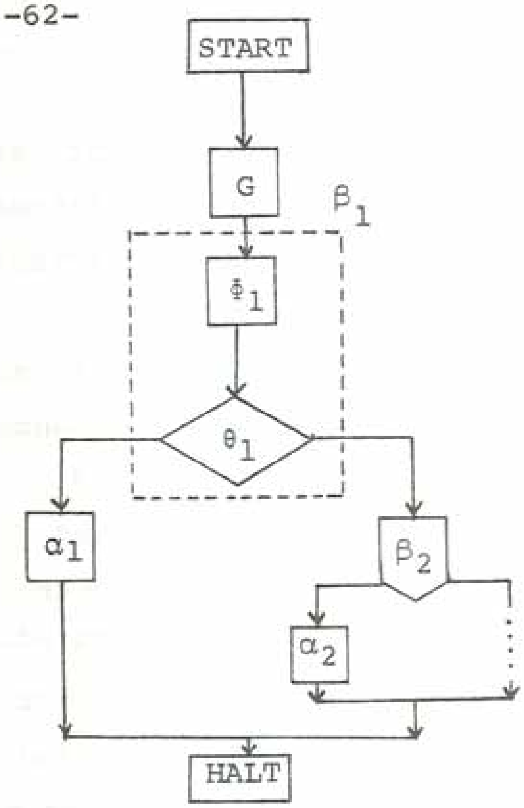


Figure 3.11 Blocks

Simulation of a nfs

Let I be an interpretation with associated domain D, and undefined object Ω. A memory state σ_I is a total function σ_I: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 $\subseteq$ Var. Let σ_I, σ_I' be memory states under interpretation I. Let β be a block consisting of a sequence of assignment statements Φ followed by a branching control θ.

V_B, V_Q denote the set of variables used in B and Q, respectively.

L_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_B, 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 σ_I, B exits at its i-th exit with memory state σ'_I.

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

$\beta[\sigma_G] = \underline{T}(\underline{F})$ denotes that the branching control θ, evaluated in $\Phi[\sigma_G]$, has outcome $\underline{T}(\underline{F})$.

$\sigma_I =V= \sigma'_I$ 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. α_i, β_i, $1 \le i \le m$, are 2m blocks. Each α_i is a sequence of assignment statements. Each β_i is a sequence of assignment statements Φ_i followed by a branching control θ_i. Let σ_B, σ'_B and σ_G be memory states. G, α_i's and β_i's simulate B if

(i) Termination: If $\sigma_B =_{V_T}= \sigma_G$, B terminates on σ_B iff G terminates on σ_G.

(ii) Determining the exit taken by B using G and β_i's

Let $\sigma_B =_{V_T}= \sigma_G$.

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

$$\beta_k[G[\sigma_G]] = \underline{F}, \text{ for } 1 \le k < i, \quad \text{and} \quad \beta_i[G[\sigma_G]] = \underline{T}$$

(iii) Simulating the computation of B using G and α_i's

If $\sigma_B =_{V_T}= \sigma_G$, and $B[\sigma_B] = (\sigma'_B, i)$

then $\quad \sigma'_B =_{V_T}= \alpha_i[G[\sigma_G]]$

(iv) Non-interference between the α_i's and the β_i's

This condition states that the subcomputations by the β_i's do not affect the variables used by the other β_k's, $k \ne i$, and the α_i's.

For $1 \le i,k \le m$, $k \ne i$, $\quad L_{\Phi_i} \cap R_{\Phi_k} = \{\}$, the empty set.

For $1 \le i,j \le m$, $\quad L_{\Phi_i} \cap R_{\alpha_j} = \{\}$, the empty set .

Lemma 3.1 Let B be a nf-block, G be a wf-block, α_i's, β_i's be blocks, as given in the above definition. If G, α_i's and β_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 σ_B and σ_G which are equivalent with respect to V_T respectively. If $B[\sigma_B] = (\sigma_B', i)$, then $\beta_1[G[\sigma_G]]$, ..., $\beta_{i-1}[G[\sigma_G]]$ all have outcome $\underline{F}$, and $\beta_i[G[\sigma_G]]$ has outcome $\underline{T}$. α_i is then executed in T. Due to the non-interference condition, if σ_G' is the state under which α_i terminates,

$$\sigma_G' =_{V_T}= \alpha_i[G[\sigma_G]] =_{V_T}= \sigma_B'$$

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

Q.E.D.

<u>Algorithm 3.2</u> 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, α_i^j's and β_i^j's which simulate B_j.

(ii) for B, a set of blocks G, α_i's and β_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 <u>Generate</u>, applied to subblocks of B, constructs blocks which simulate these subblocks. The induction basis is established by showing that <u>Generate</u> correctly constructs blocks to simulate basic nf-blocks. */

/* Functions used in <u>Generate</u>:

Gensym() - <u>Gensym</u> is a variable symbol generator. Each time <u>Gensym</u> is activated, it returns a variable symbol X which is not used in T and has not been returned in any previous activation of <u>Gensym</u>.

Transform(P) - P is a block which consists of a sequence of assignment statements Φ followed by a branching control θ. From P, Transform constructs a block Q which can be executed to determine the outcome of θ without modifying the value of any variable used in any other block. Q consists of a sequence of assignment statements Φ' followed by a branching control θ' such that:

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

(ii) For all memory states σ, σ' such that $\sigma =_{V_T}= \sigma'$,

$$Q[\sigma] = \underline{T} \text{ iff } P[\sigma'] = \underline{T} \qquad */$$

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 $1 \le i \le m$, 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, α_i's and β_i's

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

α_i: α_i is the linear concatenation of all the assignment statements on the path P_i.

β_i: β_i is used to determine whether B exits at its i-th exit when entered with memory state σ, and is a concatenation of a sequence of assignment statements Φ_i with a branching control θ_i. θ_i is a conjunction of the conditions which cause B to take the i-th exit. Φ_i is a sequence of assignment statements that uses new variables to compute the values tested by θ_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.

Φ_i is the concatenation of all the F'_j constructed above.

θ_i is a branching control of the form "$C_1 \wedge \ldots \wedge C_n$", where for $1 \le k \le n$,

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

Simulation of B by G, α_i's and β_i's

It is straightforward to verify that G, α_i's and β_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

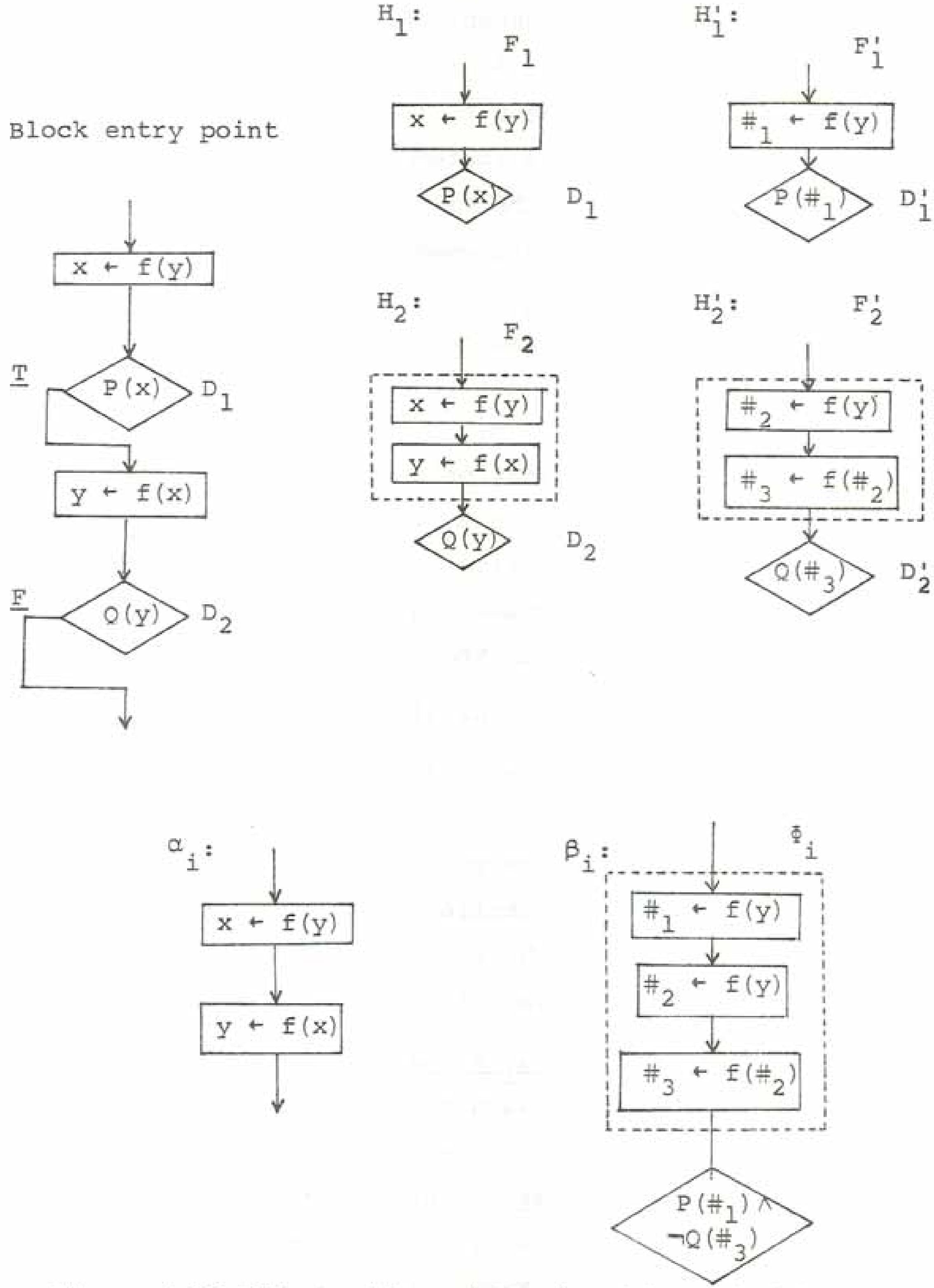


Figure 3.12 Illustration of Construction 3.2-1

<u>Construction 3.2-2</u> 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.

<u>Construction 3.2-2-1</u> 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 <u>Generate</u> to B_1 to construct blocks G_1, α_i^1's and β_i^1's, $1 \le i \le m$, (Figure 3.3) to simulate B_1.

Apply <u>Generate</u> to B_2 to construct blocks G_2, α_i^1's and β_i^1's, $1 \le i \le n$, (Figure 3.3) to simulate B_2. From Construction 3.2-1, G_2 is the null wf-block $\emptyset$.

<u>Construction of G, α_i's and β_i's</u>

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, α_i^1's and β_i^1's simulate B_1. G_2, α_i^2's and β_i^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 $B[\sigma_B] = (\sigma_B', i)$, and $1 \le i \le n$, then

(i) $\sigma_B' =_{V_T} \alpha_i^2[\alpha_1^1[G[\sigma_G]]]$

(ii) $\beta_1^1[G[\sigma_G]] = \underline{T}$,

$\beta^2_i[\alpha^1_1[G[\sigma_G]]] = \underline{T}$

(iii) For $1 \le k < i$, $\beta^2_k[\alpha^1_1[G[\sigma_G]]] = \underline{F}$

(2) Let $\sigma_B =_{V_T}= \sigma_G$.

If $B[\sigma_B] = (\sigma'_B, i)$, and $n+1 \le i \le m+n-1$, then

(i) $\sigma'_B =_{V_T}= \alpha^1_{i-(n-1)}[G[\sigma_G]]$

(ii) $\beta^1_{i-(n-1)}[G[\sigma_G]] = \underline{T}$

(iii) For $n \le k < i$ $\beta^1_{k-(n-1)}[G[\sigma_G]] = \underline{F}$

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

α_i: <u>For $1 \le i \le n$</u>, 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.

<u>α_i is the concatenation of α^1_1 and α^2_i.</u> (Observation 1(i))

<u>For $n+1 \le i \le m+n-1$</u>, taking the i-th exit of B implies taking the $(i-(n-1))$-th exit of B_1, without entering B_2.

<u>α_i is $\alpha^1_{i-(n-1)}$</u> (Observation 2(i))

Thus α_{n+1} is α^1_2, α_{n+2} is α^1_3, and so on.

β_i: <u>For $1 \le i \le n$</u>, 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. β_i is constructed as follows (Observation 1(ii)):

(i) Apply <u>Transform</u> to β^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 α_1^1 and Φ_i^2 to form a block H_i.
Concatenate H_i and θ_i^2 to form a block K_i.
Apply Transform to K_i to construct a block L_i 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.

Φ_i is the concatenation of F with F_i.
θ_i is the conjunction of D and D_i.
β_i is the concatenation of Φ_i and θ_i.

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

β_i is $\beta^1_{i-(n-1)}$. (Observation 2(ii))

Simulating B with G, α_i's and β_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 β_i's

From the observations made above and the construction of β_i's, it should be straightforward to see that for $1 \le i \le m+n-1$, if $\sigma_B = V_T = \sigma_G$, then $B[\sigma_B] = (\sigma_B', i)$ iff $\beta_i[G[\sigma_G]] = \underline{T}$ and for $1 \le k \le m+n-1$, $k \ne i$, $\beta_k[G[\sigma_G]] = \underline{F}$.

<u>Simulating the computation of B by G and α_i's</u>

From the observations made above and the construction of α_i's, it should be obvious that for $1 \le i \le m+n-1$, if $\sigma_B =_{V_T} \sigma_G$ and $B[\sigma_B] = (\sigma_B', i)$,

$$\sigma_B' =_{V_T} \alpha_i[G[\sigma_G]]$$

<u>Non-interference between the α_i's and the β_i's</u>

Every one of the Φ_i's are, or have been, constructed using <u>Transform</u>. The non-interference condition is thus trivially satisfied.

<u>End of Construction 3.2-2-1</u>

<u>Construction 3.2-2-2</u> 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 <u>Generate</u> to B_1 to construct G_1, α_i^1's and β_i^1's to simulate B_1.

Apply <u>Generate</u> to B_2 to construct G_2, α_i^2's and β_i^2's to simulate B_2.

<u>Construction of G, α_i's and β_i's</u>

G: Since B_2 is not a basic block, G_2 is non-null. G is constructed from G_1, α_1^1 and G_2. If in B, B_2 is entered, α_1^1 and G_2 in G is also executed. α_1^1 and G_2 may modify some of the variables accessed by the β_i^1's. This may cause the evaluation of the β_i^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 α_1^1 and G_2, new variables are introduced to store the values of the variables that may be modified by α_1^1 and G_2. These variable values are stored at the time G_1 terminates. The accompanying modifications to the β_i^1's are described below. G is constructed as follows:

Let $Y_1, \ldots, Y_\varkappa$ be the set of variables appearing on the LHS of the assignment statements in α_1^1 and G_2. Let $Z_1, \ldots, Z_\varkappa$ be $\varkappa$ new variable symbols generated by $\varkappa$ activations of <u>Gensym</u>. <u>Savestate</u> is a sequence of $\varkappa$ assignment statements (Figure 3.13), $Z_k \leftarrow Y_k$, $1 \le k \le \varkappa$. <u>Savestate</u> 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, α_1^1, β_1^1 and <u>Savestate</u> as shown in Figure 3.13.

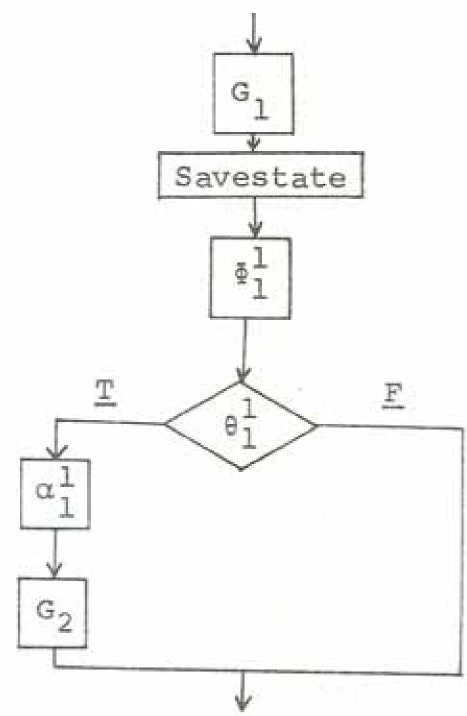


Figure 3.13 G for Construction 3.2-2-2

From the fact that G_1, α_i^1's and β_i^1's correctly simulate B_1 and G_2, α_i^2's and β_i^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 $\underline{1 \le i \le n}$, then
(i) $G[\sigma_G] =V_T= G_2[\alpha_1^1[G_1[\sigma_G]]]$

$\sigma_B' =V_T= \alpha_i^2[G_2[\alpha_1^1[G_1[\sigma_G]]]] =V_T= \alpha_i^2[G[\sigma_G]]$

(ii) $\beta_1^1[G_1[\sigma_G]]=\underline{T}$ and $\beta_i^2[G[\sigma_G]] = \underline{T}$
(iii) For $1 \le k < i$, $\beta_k^2[G[\sigma_G]] = \underline{F}$

(2) Let $\sigma_B =V_T= \sigma_G$.
If $B[\sigma_B] = (\sigma_B', i)$ and $\underline{n+1 \le i \le m+n-1}$, then
(i) $G[\sigma_G] =V_T= G_1[\sigma_G]$

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

(ii) $\beta_{i-(n-1)}^1[G[\sigma_G]] = \underline{T}$
(iii) For $n \le k < i$, $\beta_{k-(n-1)}^1[G[\sigma_G]] = \underline{F}$

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

α_i: For $1 \le i \le n$, the i-th exit of B is the i-th exit of B_2.
$\underline{\alpha_i \text{ is } \alpha_i^2.}$ (Observation 1(i))
For $n+1 \le i \le m+n-1$, the i-th exit of B is the (i-(n-1))-th exit of B_1.
$\underline{\alpha_i \text{ is } \alpha_{i-(n-1)}^1.}$ (Observation 2(i))

β_i: For $1 \le i \le n$, β_i 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 β_1^1 when G_1 terminates. The second event can be determined by evaluating β_i^2 when G_2 terminates. (Observation 1(ii)) Since β_i is evaluated when G terminates, and α_1^1 and G_2 may have side-effects, β_1^1 has to be modified before it can be used to construct β_i. For $1 \le k \le \varkappa$, Y_k is a variable which may be modified by α_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 β_1^1 by an access to Z_k. $\beta_1^{1}{}'$ is a concatenation of $\Phi_1^{1}{}'$ and $\theta_1^{1}{}'$. β_i^2, unmodified, is a concatenation of Φ_i^2 and θ_i^2. Φ_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 Φ_i^2. θ_i, the branching control which uses the values computed by Φ_i to determine whether B takes its i-th exit, is the conjunction of $\theta_1^{1}{}'$ and θ_i^2. <u>β_i is the concatenation of Φ_i and θ_i.</u>

<u>For $n+1 \le i \le m+n-1$</u>, the i-th exit of B is the i-(n-1)-th exit of B_1. $\beta_{i-(n-1)}^1$ can be evaluated when G_1 terminates to determine whether B_1 takes its i-th exit. Since β_i is evaluated when G terminates, and α_1^1, G_2 may produce side effects, $\beta_{i-(n-1)}^1$ must be modified to use the values saved by <u>Savestate</u>. Construct $\beta_{i-(n-1)}^{1}{}'$ by replacing accesses to Y_k in $\beta_{i-(n-1)}^1$ by accesses to Z_k. <u>β_i is $\beta_{i-(n-1)}^{1}{}'$.</u>

<u>Simulation of B using G, α_i's and β_i's</u>

From the above observations and the way G, α_i's and β_i's are constructed, it is again straightforward to show that G, α_i's and β_i's satisfy the 4 conditions for simulating B. The details are omitted.

End of Construction 3.2-2-2

End of Construction 3.2-2

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 B_1. B "loops" until B_1 exits at other than its first exit at some iteration. Apply Generate recursively to construct blocks G_1, α_i^1's and β_i^1's to simulate B_1.

Construction of G, α_i's and β_i's

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

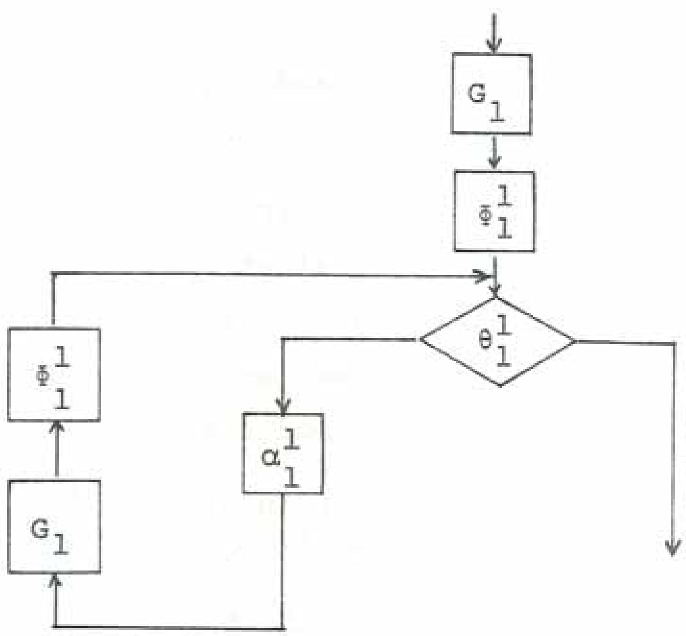


Figure 3.14 G for Construction 3.2-3

Lemma 3.2 Let $\sigma_B =_{V_T}= \sigma_G$. B, if entered with σ_B, terminates iff G, if entered with σ_G, terminates. If B terminates,

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

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

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

Proof:

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

Lemma 3.2 motivates and justifies the following constructions:

α_i: For $1 \le i \le m-1$, α_i is α_{i+1}^1.

β_i: For $1 \le i \le m-1$, β_i is β_{i+1}^1.

Simulation of B by G, α_i's and β_i's

Using Lemma 3.2, it is straightforward to show that G, α_i's and β_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 IN_B.

For each variable symbol X,

$X \in IN_B$ 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 OUT_B.

For each variable symbol X,

$X \in OUT_B$ iff X appears on the LHS of some assignment statement in B.

<u>An environment for translating B</u> is a set of labelled data links D in a wfdfs Z such that

(i) Every variable in 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 IN_B.

A wfs W is translated into an equivalent wfdfs 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 wfdfs 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 IN_B.

(ii) Labelling the output data links of Z_B by variable symbols from OUT_B, and removing from E any label not in the set 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 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.

<u>Algorithm 3.3</u> WtoZ-Translate (B,E) - Given a wfs or a wf-block B, modify E to E' which contains a subschema equivalent to B.

1. <u>B is a wfs</u>.

(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 (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.

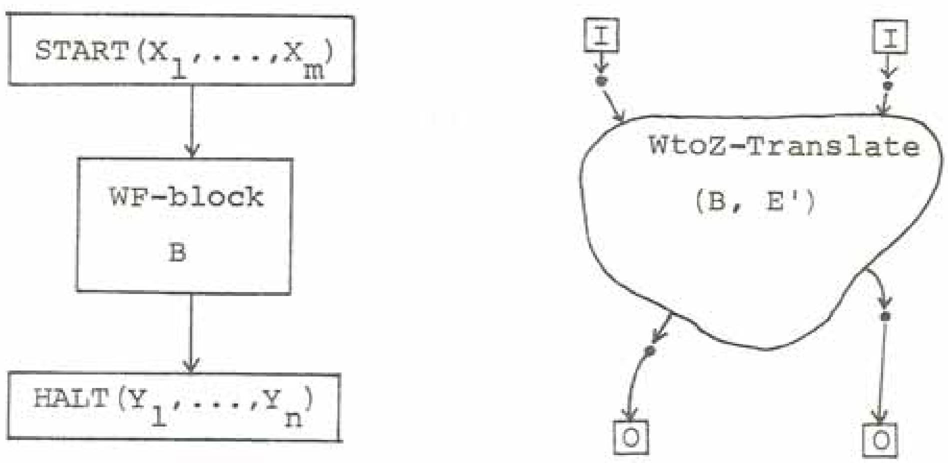


Figure 3.15 Algorithm 3.3 applied to a wfs

2. <u>B is a concatenation of an assignment statement and a wf-block B_1.</u>

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_j^i to the variables $X_1, \ldots, X_i$, add an operator Op labelled by F_j^i to E (Figure 3.16). The k-th input data link to Op, for $1 \le k \le i$, 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'.

WtoZ-Translate(B_1 , E')

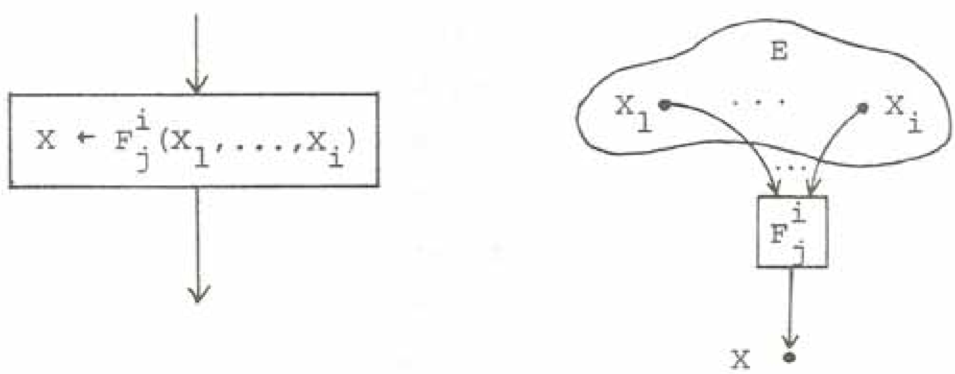


Figure 3.16 Algorithm 3.3 applied to an Assignment statement

3. B is a concatenation of B' and B_1. B' is formed from wf-blocks B_T, 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.

$I_{B_T} = IN_{B_T} \cup (OUT_{B_F} - OUT_{B_T})$

$I_{B_F} = IN_{B_F} \cup (OUT_{B_T} - OUT_{B_F})$

V_L = the set of variables accessed by the branching control BC

Let X be a variable symbol in $OUT_{B_T} \cup OUT_{B_F}$ 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)

WtoZ-Translate(B_T , D_T)

WtoZ-Translate(B_F , D_F)

(iv) After (iii), the data link sets E_T and E_F will be

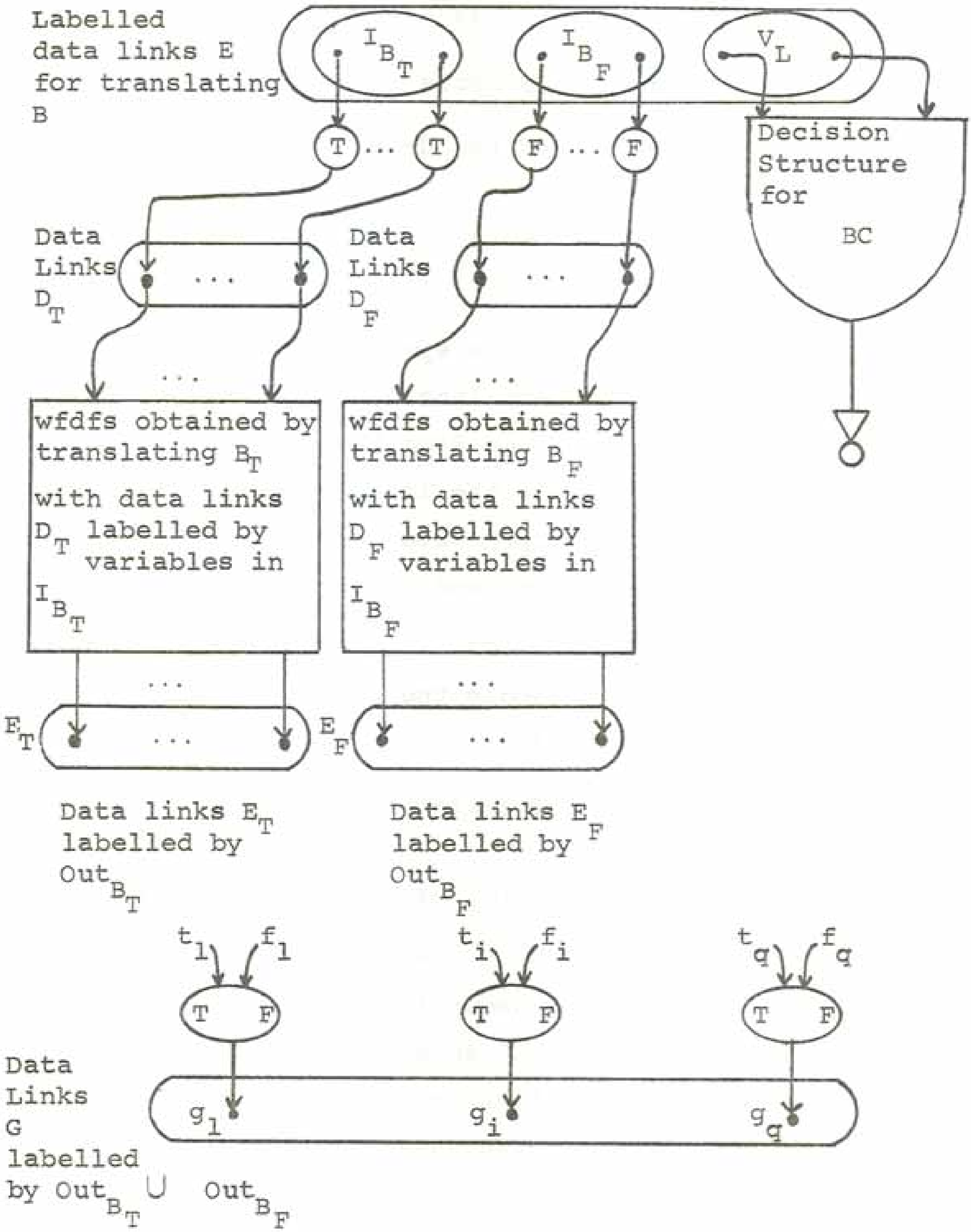


Figure 3.17 Algorithm 3.3 applied to a block formed from conditional composition

labelled by variable symbols in OUT_{B_T} and 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 $OUT_{B_T} \cup OUT_{B_F}$.

Let y_i be a variable symbol in $OUT_{B_T} \cup OUT_{B_F}$. 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).

WtoZ-Translate(B_1 , E')

4. <u>B is a concatenation of B' and B_1. B' is formed from wf-block B_L and the branching control BC by iteration.</u>

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

$LBL = IN_{B_L} \cup OUT_{B_L} \cup V_L$

Let X be a variable symbol in 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 OUT_{B_L}. Label h with the labels of a from OUT_{B_L}.

(iv) WtoZ-Translate(B_L , F)

(v) After (iv) the data link set G is labelled by variable symbols from OUT_{B_L}. For every t_i, let d_i be the output link of the merge node one of whose input arc is t_i. If d_i is labelled by y, $y \in OUT_{B_L}$, t_i is from the data link in G labelled by y. If d_i is labelled by y, $y \notin OUT_{B_L}$, t_i is from the data link in F labelled by y.

(vi) Every label $y \in OUT_{B_L}$ is removed from the data link set E. Let E' denote the set of labelled data links in $E \cup H$.

WtoZ-Translate(B_1 , E')

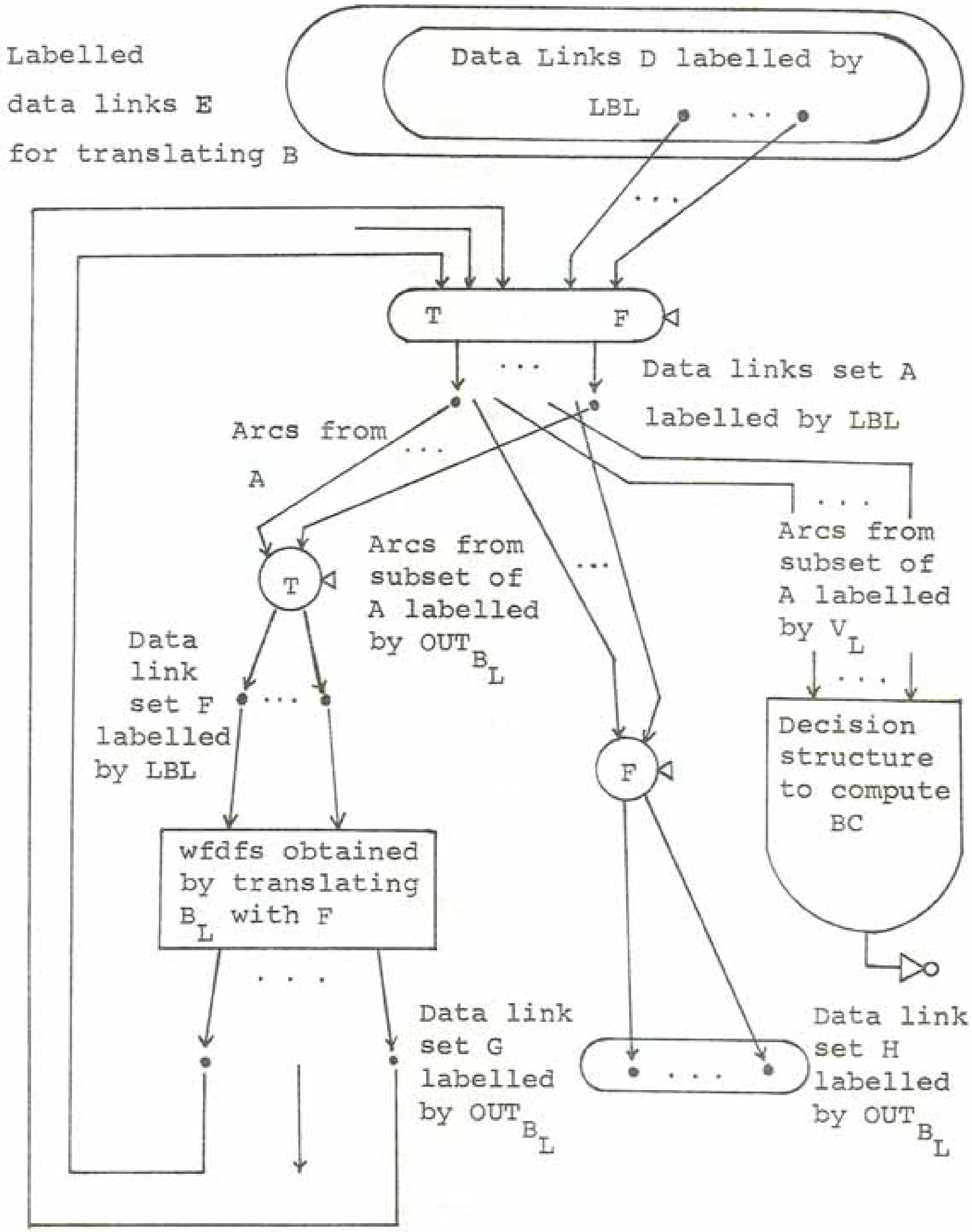


Figure 3.18 Algorithm 3.3 applied to a block
formed from iteration

5. In each of the 4 cases considered above, B_1 may be the null wf-block. Algorithm 3.3, when applied to the null WF-block, returns without modifying E. In the translation process many nodes in the resulting wfdfs may be introduced but are not connected to the O-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 <u>input data link</u> 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) <u>T is an operator.</u> Label the output data link of T by a new variable symbol X.
   (ii) <u>T is a conditional subschema.</u> A data link d in $D_T \cup$

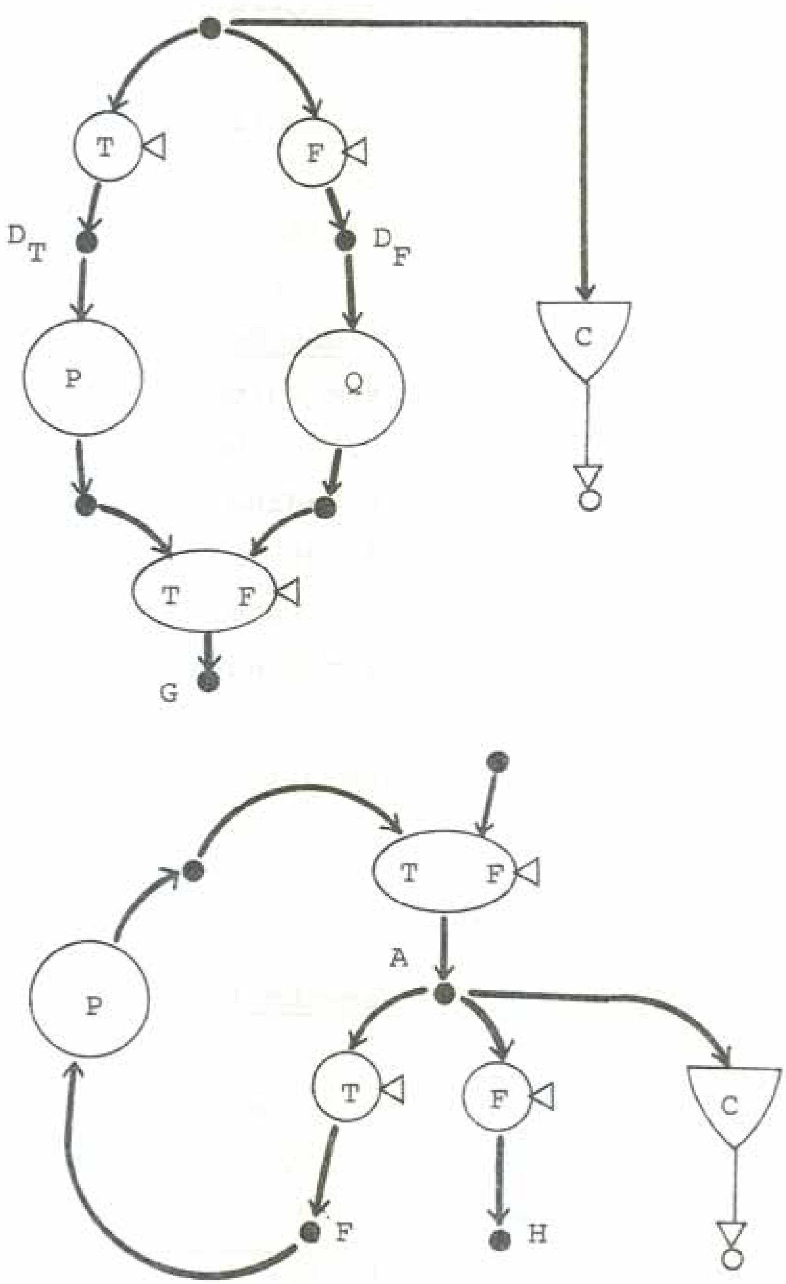


Figure 3.19 Structure of wfdfs's to be labelled

D_F (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 ∪ 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 wfdfs Z

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

$$\boxed{\text{START}(X_1, \ldots, X_m)}$$

(ii) For an operator whose input data links are labelled by $X_{i1}, \ldots, X_{ik}$, whose output data link is labelled by X_j, and whose label is F_ℓ^k, the statement generated is

$$\boxed{X_j \leftarrow F_\ell^k(X_{i1},\ldots,X_{ik})}$$

(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 $X_1,\ldots,X_i$, $X_k \in \mathrm{Var}$, $1 \le k \le i$.

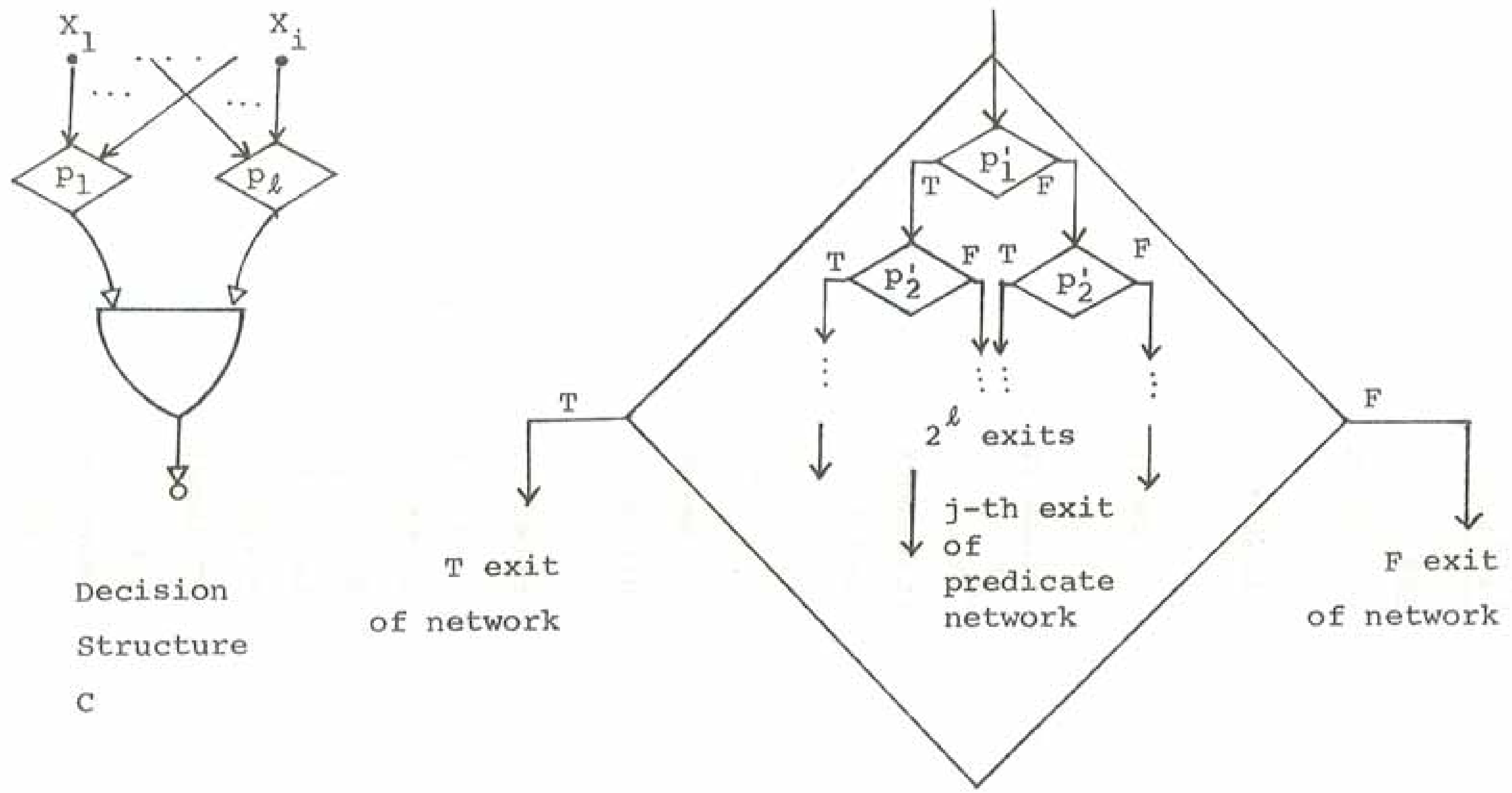
Let the deciders in C be labelled by $p_1,\ldots,p_\ell$, $p_k \in \mathrm{Pred}$, $1 \le k \le \ell$.

Let the input data links to p_k, $1 \le k \le \ell$, be labelled by $\alpha_1^k,\ldots,\alpha_{\nu_k}^k$, $\alpha_1^k,\ldots,\alpha_{\nu_k}^k \in \{X_1,\ldots,X_i\}$.

Let the predicate statement whose predicate symbol is p_k, and which accesses the variable symbols $\alpha_1^k,\ldots,$ $\alpha_{\nu_k}^k$ be denoted by $\langle p_k' \rangle$

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_o, 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 $\boxed{x_o \leftarrow x_t}$ and the i-th statement in A_F is $\boxed{x_o \leftarrow x_f}$.



j-th exit of predicate network is connected to

- T exit if the predicate outcomes along path leading to exit j implies outcome of decision structure is <u>T</u>.
- F exit if the predicate outcomes along path leading to exit j implies outcome of decision structure is <u>F</u>.

Figure 3.20 Simulating decision structure with predicate statement network

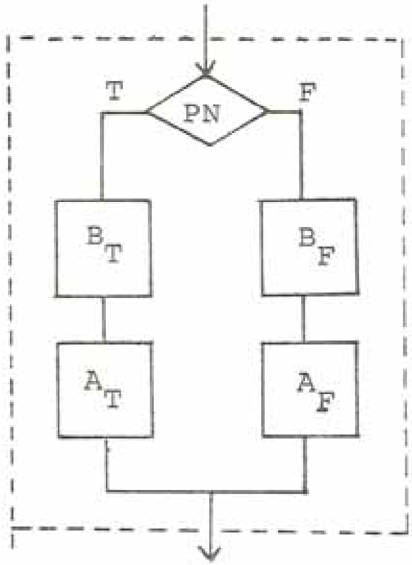


Figure 3.21 Block generated for conditional subschema

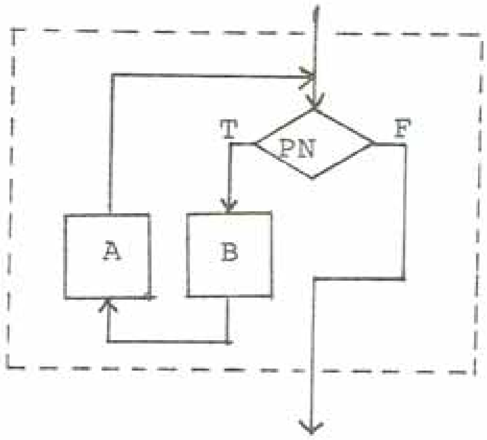


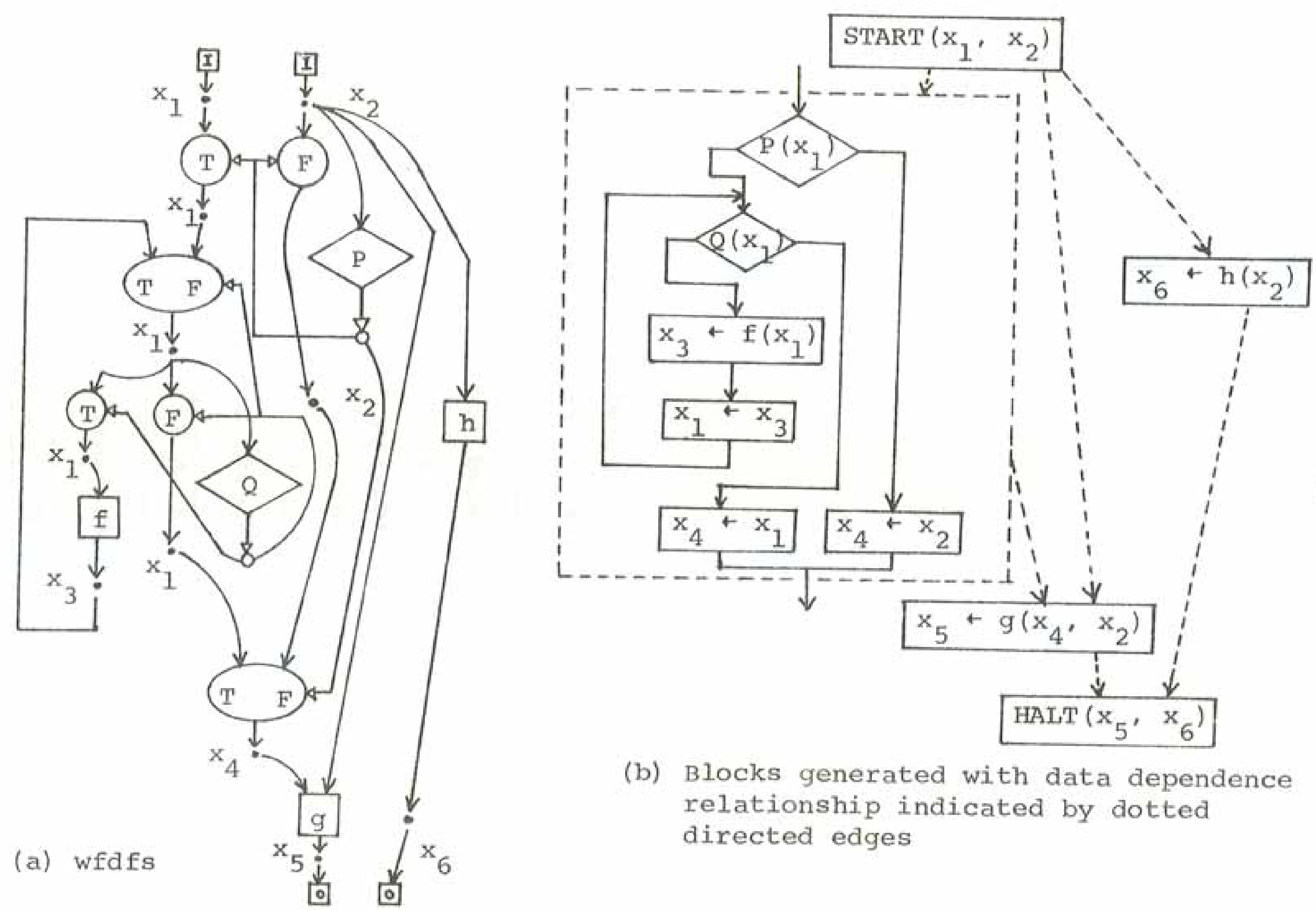
Figure 3.22 Block generated for iteration subschema

(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 $\boxed{x_a \leftarrow x_b}$.

(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

$$\boxed{\mathrm{HALT}(Y_1,\ldots,Y_n)} \quad .$$



(a) wfdfs

(b) Blocks generated with data dependence relationship indicated by dotted directed edges

Figure 3.23 An example of generating blocks for wfdfs's

<u>Example</u>: 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.

<u>Theorem 3.2</u> 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.

<u>Theorem 3.3</u> 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 α-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 α-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 α-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 wfdfs's and the class of α-free 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 wfdfs.

We note that T is equivalent to the α-free wfdfs shown in Figure 2.11. Theorem 3.4 thus implies that the class of free wfdfs's is a proper subclass of α-free wfdfs's. We prove Theorem 3.4 by proving a set of lemmas.

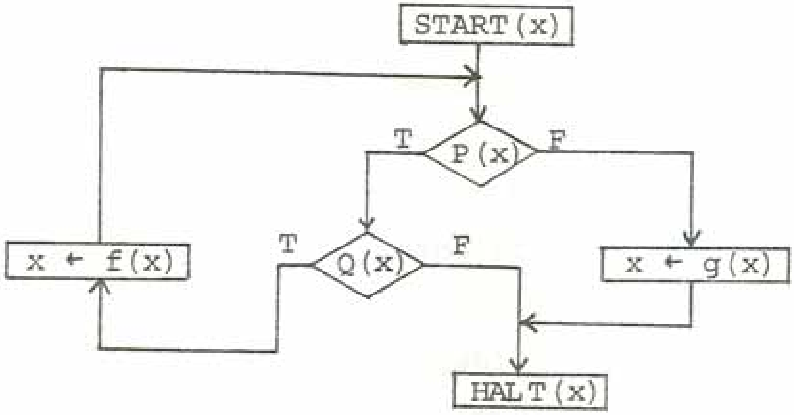


Figure 3.24 A free flowchart schema which is not equivalent to any free wfdfs

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 L.

Proof:

If W does not contain any iteration subschema, the set of output it can produce under all free interpretation 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 α-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 F, 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 $f^{p_I}(a_1)$ has not been tested by P when L is entered under I. Let q_I be the least integer such that $f^{q_I}(a_1)$ 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'$\in$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.

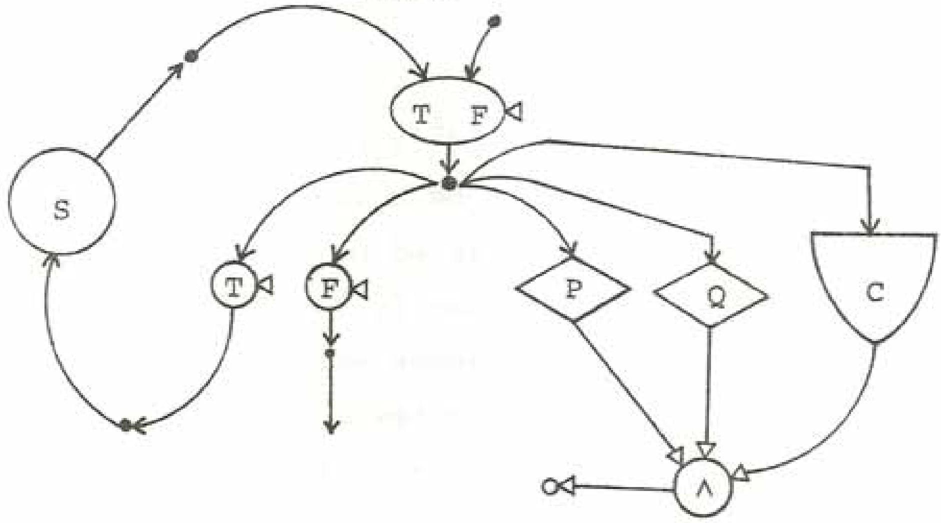


Figure 3.25 L for Lemma 3.4-4

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':

$P(f^p(x)) = \underline{F}$

$Q(y) = \underline{T}$

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

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

$P(f^P(a_1)) = \underline{T}$

$Q(y) = \underline{F}$

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 α-free wfdfs.

We prove Theorem 3.5 by proving a set of lemmas.

Consider the following regular expressions:

$$E_P = (bg^* af^*)^* a_1$$

$$E_Q = (bg^* af^*)^* af^* a_1$$

$$E = E_P \cup E_Q$$

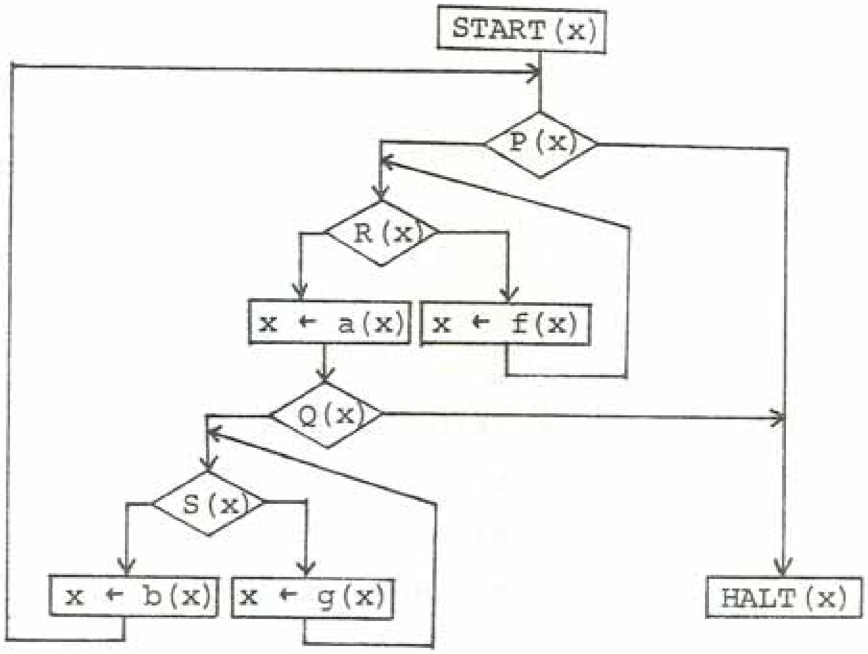


Figure 3.26 A free flowchart schema that is not equivalent to any α-free wfdfs

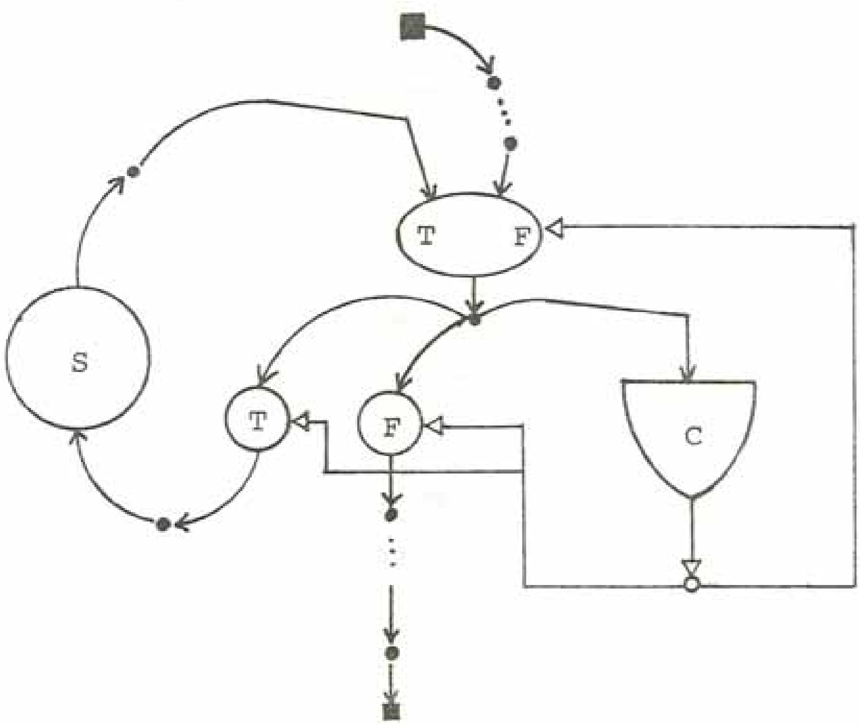


Figure 3.27 Schematic representation of data path in W

$E = \{ e \mid$ e is the output of T under some free interpretation of I $\}$

$E_P = \{ e \mid$ e is tested by the predicate symbol P in a computation sequence of T under some free interpretation I $\}$

$E_Q = \{ e \mid$ e is tested by the predicate symbol Q in a computation sequence under some free interpretation I $\}$

Let W be an α-free wfdfs equivalent to T. We may again assume that W is open and complete.

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

<u>Proof</u>:

Similar to the proof of Lemma 3.4-2.

Q.E.D.

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

If s, $s \in R_i$, is an input data value to L, then for every y, $y \in R_i$, there is a free interpretation under which $y \sqcap s$ (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 α-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

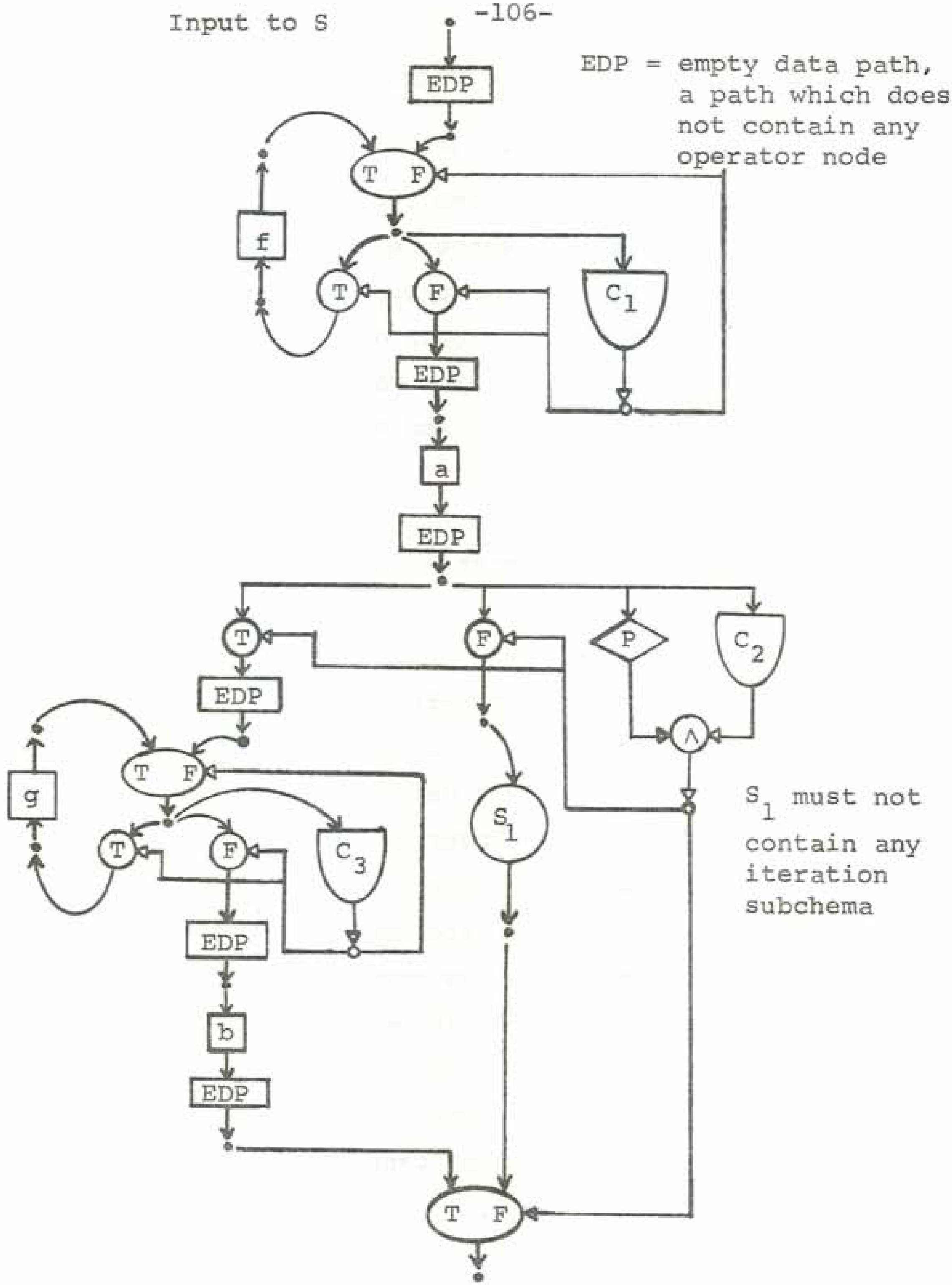


Figure 3.28 Schematic representation of subschema S in Figure 3.27

J'.

Let P(y)=F under J'.

By Lemma 3.5-1, the iteration subschema which generates g^* 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 α-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)=F in the computation sequence, must have outcome F. W thus cannot be α-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 wfdfs's have been studied by Qualitz [Qualitz 75]. The equivalence problem for data links in free wfdfs's has been studied by Dennis and Fosseen [Dennis & Fosseen 73]. In this chapter we present three undecidability results for open and complete wfdfs's:

1.Completeness is undecidable in open wfdfs's.

2.Openness is undecidable in complete wfdfs's.

3.Equivalence is undecidable in open and complete wfdfs'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 wfdfs's

The Post Correspondence Problem (PCP) can be reduced to the problem of deciding completeness in open wfdfs'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

$A = \{A_1, \ldots, A_n\}$

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

$1 \le i \le n$, A_i, $B_i \in (0 \cup 1)^* - \{\lambda\}$
where λ is the empty string

and for $1 \le i \le n$,

$A_i = a_{i1} \sqcap \ldots \sqcap a_{i\alpha_i}$

$B_i = b_{i1} \sqcap \ldots \sqcap b_{i\beta_i}$

$a_{ij}, b_{ij} \in \{0, 1\}$

A PCP C has a <u>solution</u> iff there exists a sequence of subscripts $i_1, \ldots, i_m$, such that:

(i) for $1 \le k \le m$, $1 \le i_k \le n$

(ii) $A_{i_1} \sqcap \ldots \sqcap A_{i_m} = B_{i_1} \sqcap \ldots \sqcap B_{i_m}$

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

To reduce the Post Correspondence Problem to the completeness problem we will firstly show how to construct a wfdfs 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 <u>free interpretation</u> for S_{PCP} with associated domain D. Under I, the input data values to S_{PCP} are elements of Insym = $\{\delta_i | 1 \le i\}$ and the data values are strings in D (Section 2.3). There is then a natural correspondence between $(f_0 \cup f_1)^* \sqcap$Insym and strings over $(0 \cup 1)$. This correspondence can be expressed as an isomorphism φ,

$$\varphi : (f_0 \cup f_1)^* \sqcap \text{Insym} \rightarrow (0 \cup 1)^*$$

$$\varphi(f_0) = 0, \qquad \varphi(f_1) = 1; \quad i \ge 1, \quad \varphi(\delta_i) = \lambda$$

$$\varphi(f \sqcap d) = \varphi(f) \sqcap \varphi(d), \quad f \in \{f_0, f_1\}, \; d \in \{f_0 \cup f_1\}^* \sqcap \text{Insym}$$

The wfdfs S_{PCP} simulates a PCP $C = (A, B)$ if for a fixed yet arbitrarily chosen input, with input value δ_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 $i_1, \ldots, i_m$,

$$\varphi(s_1) = A_{i_1} ⌑ \ldots ⌑ A_{i_m}$$

$$\varphi(s_2) = B_{i_1} ⌑ \ldots ⌑ B_{i_m}$$

Construction 4.1 Given a PCP C, construct a wfdfs S_{PCP} which simulates C.

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

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

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

Similarly construct a wfdfs K_i for each B_i.

(ii) From the H_i's and K_i's construct a wfdfs S_C which provides n alternative data paths between its input

Figure 4.1 H_i for Construction 4.1 (i)

and output data links. Choosing among one of these data paths corresponds to picking one of the pairs (A_i, B_i), $1 \le i \le n$. S_C is constructed inductively as follows:

S_{n-1} is a <u>conditional subschema</u>. 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 $1 \le j \le n-2$, 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 <u>T-branch</u> and H_j, K_j in its <u>F-branch</u>.
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 and z_1 are δ_1, δ_2 and δ_3 respectively. Consider the following sets of decision outcomes:

For $1 \le k \le n-1$,

$$E_k = \{ P(f^k \square \delta_3) = \underline{F} \} \cup \{ P(f^j \square \delta_3) = \underline{T} \mid 1 \le j < k \}$$

$$E_n = \{ P(f^j \square \delta_3) = \underline{T} \mid 1 \le j \le n \}$$

If I is a free interpretation consistent with E_j, the output of S_C at p_1 and q_1 under I are $F_j \square \delta_1$ and $F'_j \square \delta_2$, such that

$$\varphi(F_j \sqcap \delta_1) = A_j \text{ and } \varphi(F'_j \sqcap \delta_2) = B_j$$

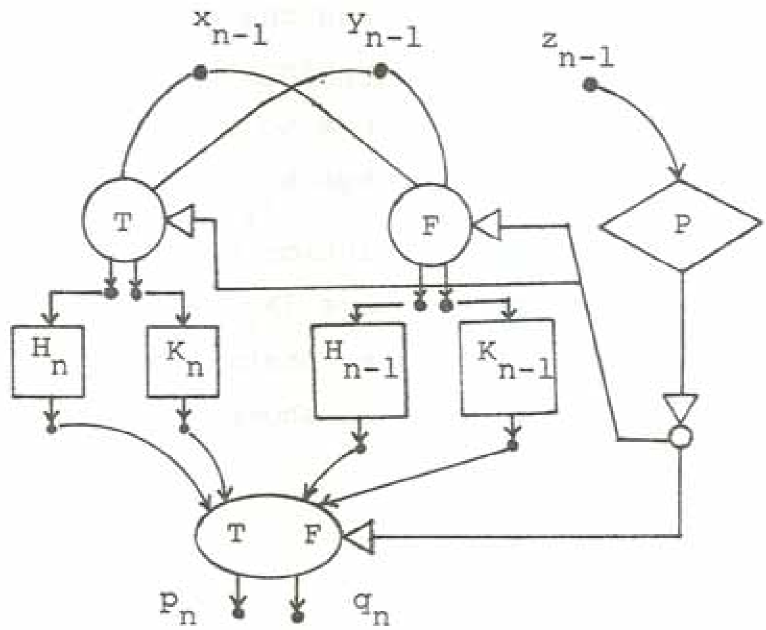


Figure 4.2 S_{n-1} for Construction 4.1(ii)

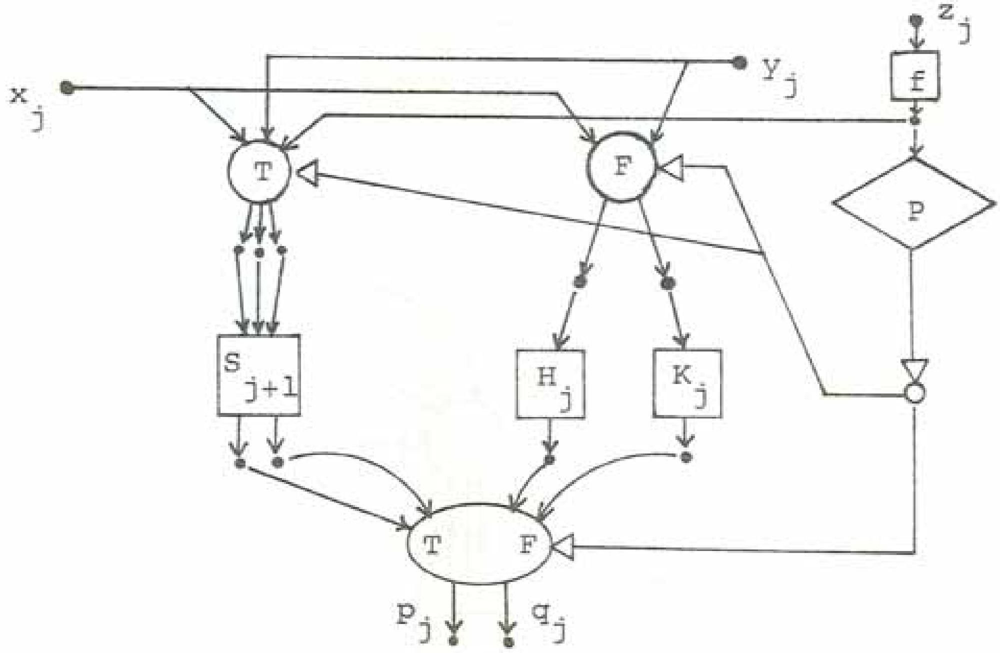


Figure 4.3 S_j for Construction 4.1(ii)

(iii) Construct an <u>iteration subschema</u> L which contains S_C as its body. <u>Executing the body</u> S_C of L then corresponds to picking one of the pairs (A_i, B_i), for some i, $1 \le i \le n$. <u>Iterating</u> the iteration schema L m times then corresponds to picking m integers $i_1, \ldots, i_m$ to form the strings $A_{i_1} \square \ldots \square A_{i_m}$ and $B_{i_1} \square \ldots \square B_{i_m}$. Since a proposed solution to a PCP must have used <u>at least</u> one of the pairs (A_i, B_i), i.e. $m \ge 1$, the wfdfs S_{PCP} simulating C is constructed by concatenating a copy of S_C with L, as shown in Figure 4.4.

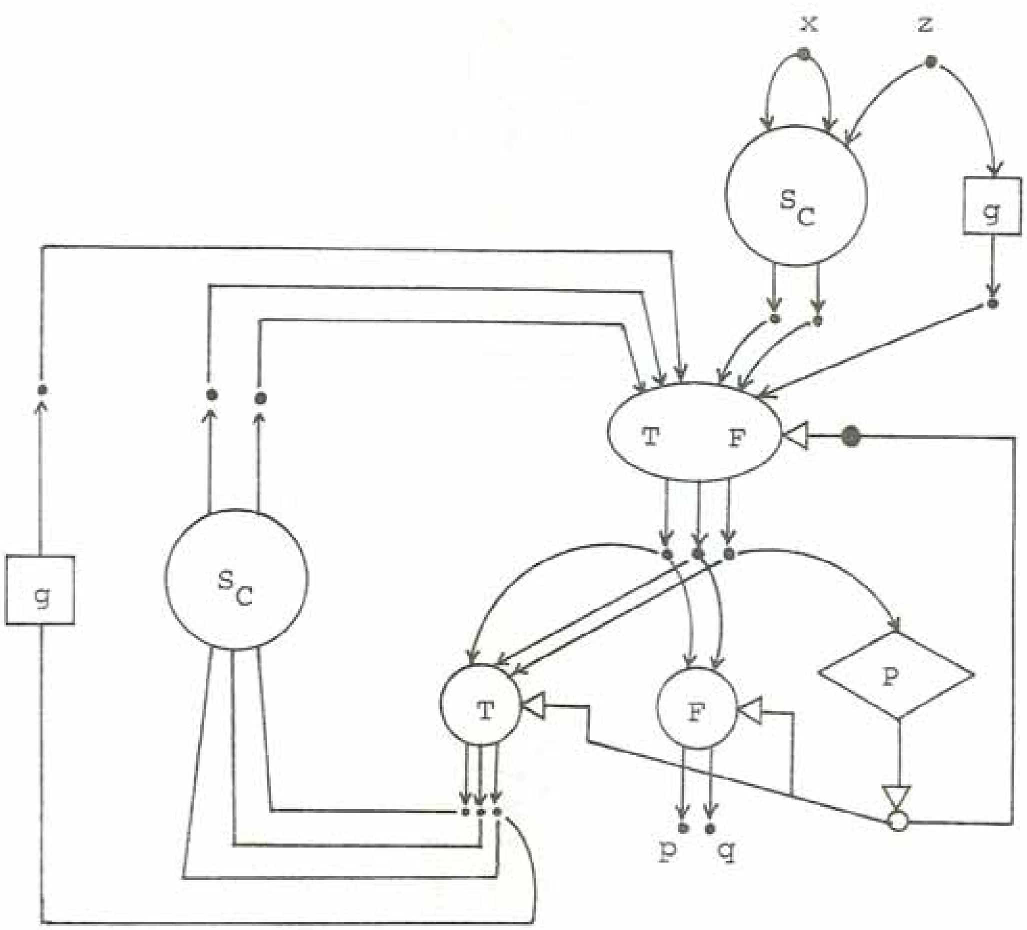


Figure 4.4 S_{PCP} for simulating the PCP C

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 δ_1 and δ_2 respectively. Under a free interpretation I:

Let $\underline{m}$ be the $\underline{\text{smallest}}$ integer i, i≥1, such that

$$P(g^i \sqcap \delta_2) = \underline{F} \quad .$$

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

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

$$\text{for all } \ell,\ 1 \le \ell \le k,\ P(f^\ell \sqcap g^{j-1} \sqcap \delta_2) = \underline{T} \quad .$$

On the j-th execution of S_C in S_{PCP} under I, the data path through S_C is determined by $\underline{i_j}$. $\underline{i_j}$ corresponds to the j-th pair from $\{(A_i, B_i) | 1 \le i \le n\}$ 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 $F_p \sqcap \delta_1$ and $F_q \sqcap \delta_1$ such that

$$\varphi(F_p \sqcap \delta_1) = A_{i_1} \sqcap \ldots \sqcap A_{i_m}$$
$$\varphi(F_q \sqcap \delta_1) = B_{i_1} \sqcap \ldots \sqcap B_{i_m}$$

with m, i_1, ..., 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 $(A_{i_1} \sqcap \ldots \sqcap A_{i_m},\ B_{i_1} \sqcap \ldots \sqcap B_{i_m})$, such that:

Under I, the outputs of S_{PCP} at output data links p and q are $F_p \amalg \delta_1$ and $F_q \amalg \delta_1$ respectively, and

$$\varphi(F_p \amalg \delta_1) = A_{i_1} \amalg \ldots \amalg A_{i_m}$$
$$\varphi(F_q \amalg \delta_1) = B_{i_1} \amalg \ldots \amalg B_{i_m}$$

S_{PCP} thus simulates the PCP C.

End of Construction 4.1

Using S_{PCP} we can construct an open wfdfs S as shown in Figure 4.5. S contains two copies of S_{PCP}, S^1_{PCP} and S^2_{PCP} . Under a free interpretation I, S^1_{PCP} proposes a solution $(F^A \amalg \delta_1, F^B \amalg \delta_1)$ to the PCP C. On successive iterations of the subschema L containing S^2_{PCP}, the outputs of S^2_{PCP} are of the form $F^A \amalg (F^A)^i \amalg \delta_1$ and $F^B \amalg (F^A)^i \amalg \delta_1$, for $1 \le i$. If the proposed solution is indeed a solution to the PCP C, $F^A \amalg \delta_1$ and $F^B \amalg \delta_1$, and all pairs of strings $F^A (F^A)^i \amalg \delta_1$ and $F^B (F^A)^i \amalg \delta_1$, are pairs of identical strings. If the proposed solution is not a solution, $F^A (F^A)^i \amalg \delta_1$ and $F^B (F^A)^i \amalg \delta_1$, for $0 \le i$, are always different strings. Due to the pair of tests performed by deciders labelled by Q, the iteration subschema containing S^2_{PCP} does not terminate whenever entered, if the solution proposed by S^1_{PCP} 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

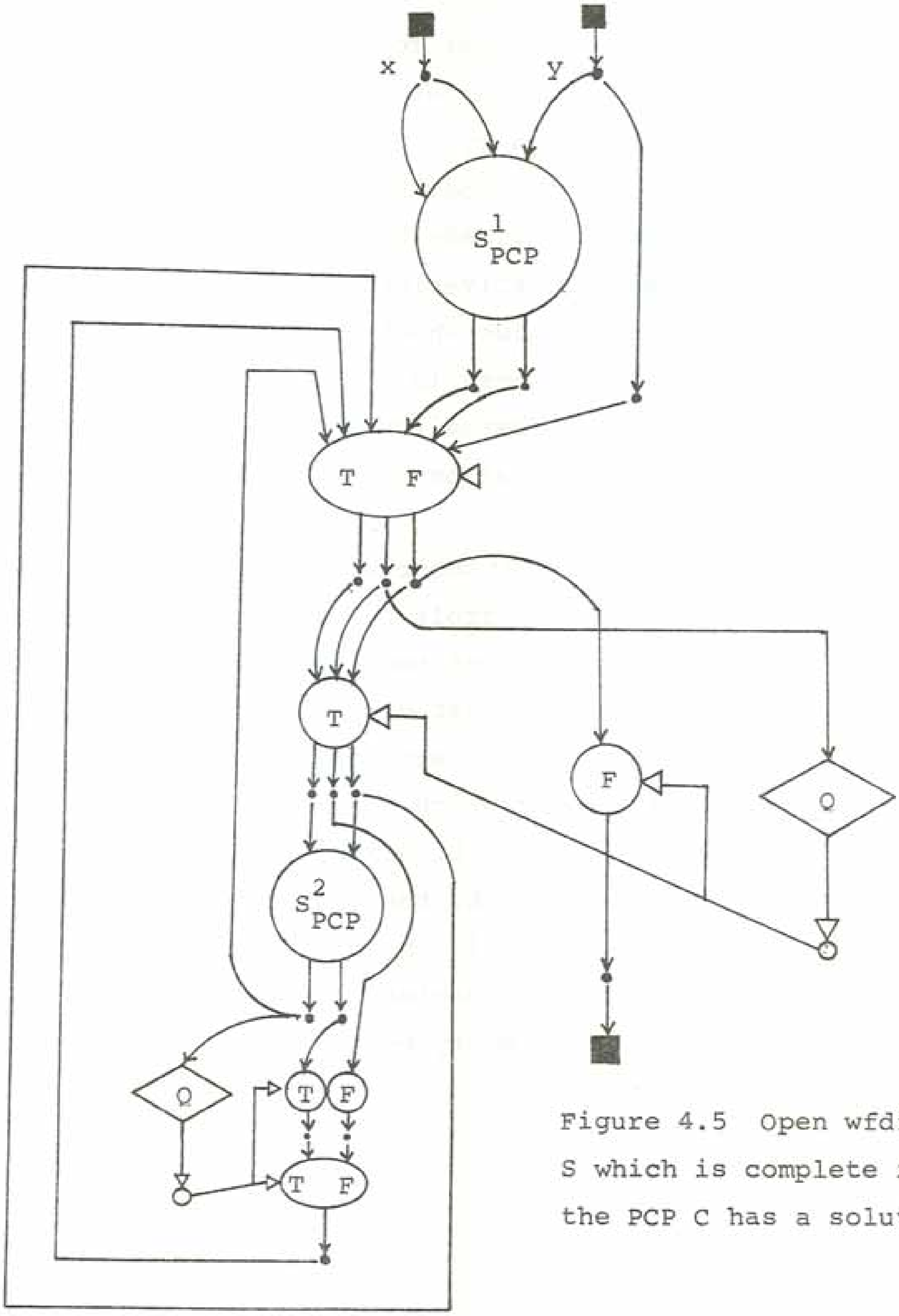


Figure 4.5 Open wfdfs S which is complete iff the PCP C has a solution

mode of operation of S_{PCP} that for every decider d in S^1_{PCP} there is an interpretation under which S^1_{PCP} terminates and under which a decision by d has outcome $\underline{T}$($\underline{F}$). Under any interpretation I, the input data value which <u>determines</u> the outcomes of decisions made by deciders in S^2_{PCP} remains unchanged on successive activations of S^2_{PCP}. This input data value is the <u>same</u> value which <u>determines</u> the outcomes of decisions made by deciders in S^1_{PCP}. S^1_{PCP} and S^2_{PCP} are identical in structure. Hence for every decider in S^2_{PCP} there is also an interpretation under which that decider has outcome $\underline{T}$($\underline{F}$).

Let L be the iteration subschema containing S^2_{PCP}. 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 $\underline{T}$, or $\underline{F}$.

Let q_2 be the decider which controls the conditional schema cascaded with S^2_{PCP} in L. q_2 is also labelled by Q. On successive iterations of L, the values tested by q_2 are from the set $\{F^A(F^A)^i \boxtimes \delta_1 | 0 \le i\}$. 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 $\underline{T}$, or $\underline{F}$.

S is thus open.

Q.E.D.

<u>Lemma 4.1-2</u> S in Figure 4.5 is complete iff the PCP C simulated by S^1_{PCP} and S^2_{PCP} in S has no solution.

<u>Proof</u>:

Let L be the iteration subschema in S containing S^2_{PCP}.

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^2_{PCP} in L.

(only if)

Consider the set of decisions made in <u>each</u> iteration of L. Let d_1 be the data value tested by q_1. The test $Q(d_1)$ must have outcome $\underline{T}$ if the body of L is to be entered. Let the two outputs of S^2_{PCP} be o_1 and o_2. After S^2_{PCP} terminates, o_1 is tested by q_2. If the test by q_2, $Q(o_1)$, has outcome $\underline{T}$, the next test made by q_1 is $Q(o_2)$. If the test by q_2, $Q(o_1)$ has outcome $\underline{F}$, the next test made by q_1 is again $Q(d_1)$, which must have outcome $\underline{T}$. 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 decisons made with q_2.

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

$$A_{w_1} \sqcap \ldots \sqcap A_{w_v} = B_{w_1} \sqcap \ldots \sqcap B_{w_v}$$

Let $F^A \sqcap \delta_1$ be such that $\varphi(F^A \sqcap \delta_1) = A_{w_1} \sqcap \ldots \sqcap A_{w_v}$

Let $F^B \sqcap \delta_1$ be such that $\varphi(F^B \sqcap \delta_1) = B_{w_1} \sqcap \ldots \sqcap B_{w_v}$

F^A and F^B are identical strings over $\{f_0, f_1\}$.

For $1 \le i \le v$, define a finite set of predicate decision outcomes W_i:

$$W_i = \{ P(f^j \sqcap g^{i-1} \sqcap \delta_2) = \underline{T} \mid 1 \le j < w_i \} \cup \{ P(f^{w_i} \sqcap g^{i-1} \sqcap \delta_2) = \underline{F} \}$$

Define the set G of predicate decision outcomes:

$$G = \{ P(g^j \sqcap \delta_2) = \underline{T} \mid 0 \le j \le v \} \cup \{ P(g^{v+1} \sqcap \delta_2) = \underline{F} \}$$

Under any free interpretation the inputs to S are δ_1 and δ_2. Let I be a free interpretation consistent with $G \cup \bigcup_{i=1}^{v} W_i$. From our discussion on the internal structure and mode of operation of S_{PCP}, it should be obvious that under I S^1_{PCP} converges with outputs $F^A \sqcap \delta_1$ and $F^B \sqcap \delta_1$. Under I, the outputs of S^2_{PCP} are of the form

$$F^A (F^A)^i \sqcap \delta_1 \quad \text{and}$$

$$F^B (F^A)^i \sqcap \delta_1$$

for $i \ge 0$. Since $F^A \sqcap \delta_1$ and $F^B \sqcap \delta_1$ are identical strings, the outputs of S^2_{PCP} 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^1_{PCP} and the decision $Q(F^A \sqcap \delta_1) = \underline{T}$ 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.

(if)

Under any free interpretation I, subschemas S^1_{PCP} and S^2_{PCP} always converge and diverge together. From Construction 4.1, S^1_{PCP} and S^2_{PCP} diverge iff $P(g^i \boxtimes \delta_2)=\underline{F}$ for all i, under 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_{PCP} and S^2_{PCP} terminates.

Let η be a finite prefix of a computation sequence of S. We can always extend (maybe trivially) η to a finite prefix η' such that every activation of S^1_{PCP} and S^2_{PCP} terminates in η'. Extend η' to η'' such that the first decision D by q_2 in η'', but not in η', has outcome $\underline{T}$ and the first decision by q_1 in η'' occurring after D has outcome $\underline{F}$. The two input data values to the decisions by q_1 and q_2 are outputs of S^2_{PCP}, of the form $F^A(F^A)^i \boxtimes \delta_1$ and $F^B(F^A)^i \boxtimes \delta_1$ for some fixed i. Since C has no solution, no two outputs of S^2_{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. η'' can then be extended to a finite computation sequence. S is complete since η can be any finite prefix of any computation sequence of S.

Q.E.D.

Theorem 4.1 Completeness is undecidable for open wfdfs.

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 $\bar{x}_o \in N^r$, where N is the set of natural numbers augmented with infinity (denoted by ∞), $P(\bar{x}_o) \geq Q(\bar{x}_o)$. There is no effective procedure which, given any two such polynomials $P(\bar{x})$ and $Q(\bar{x})$, determines whether there exists $\bar{x}_o \in N^r$ 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 N. For every such polynomial $P(\bar{x})$, $\bar{x} = (x_1,\ldots,x_r)$, and $\bar{x}_o \in N^r$, $P(\bar{x}_o) \in N$. 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 N^r and the set of free interpretations for wfdfs's with at least r inputs.

Let S be a wfdfs with r+m inputs, m≥0. Let f be a unary function symbol. Let β be a unary predicate symbol. <u>Under a free interpretation I the inputs to S at input data links labelled</u> $d_1,\ldots,d_r,\ldots,d_m$ <u>are</u> $\delta_1,\ldots,\delta_r,\ldots,\delta_m$, <u>respectively.</u>

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

$$i_j = \begin{cases} \text{the } \underline{\text{least}} \text{ natural number for which } \beta(f^{i_j}\boxtimes\delta_j)=\underline{F} \text{ under I,} \\ \infty, \text{ if no such natural number exists.} \end{cases}$$

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

Conversely, given an r-tuple $\bar{x}_o = (i_1, \ldots, i_r)$, $\bar{x}_o \in N^r$, <u>a free interpretation I for S derived from</u> $\bar{x}_o$ is any free interpretation consistent with the set of predicate decisions: For $1\le j\le r$,

$$\text{for } 0\le k\le i_j,\quad \beta(f^k\boxtimes\delta_j) = \underline{T}$$
$$\beta(f^{i_j}\boxtimes\delta_j) = \underline{F}$$

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, 1)-wfdfs. 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 $f^{P(\bar{x}_o)}\boxtimes\delta_{r+1}$, then S diverges under I.

S_P simulates $P(\bar{x})$ if

(i) For any free interpretation I, the output of S_P under I is $f^{P(\bar{x}_I)} \boxtimes \delta_{r+1}$, where $\bar{x}_I$ is the r-tuple of natural numbers derived from I.

(ii) For every r-tuple $\bar{x}_o \in N^r$, let I be a free interpretation derived from $\bar{x}_o$. The output of S_P under I is $f^{P(\bar{x}_I)} \boxtimes \delta_{r+1}$.

There may be several wfdfs's which simulate a given polynomial $p(\bar{x})$. To obtain a procedure for constructing a wfdfs S_P to simulate a polynomial $P(\bar{x})$ it is sufficient to give procedures for:

(i) Given a wfdfs S_Q which simulates $Q(\bar{x}) = \prod_{i=1}^{r} x_i^{\alpha_i}$, $\alpha_i \geq 0$, and a variable x_j, construct a wfdfs S_P which simulates the polynomial $P(\bar{x}_o) = Q(\bar{x}) * x_j$.

(ii) Given 2 wfdfs's which simulate $Q(\bar{x})$ and $R(\bar{x})$, construct a wfdfs S_P which simulates the polynomial $P(\bar{x}) = Q(\bar{x}) + R(\bar{x})$.

We can represent a polynomial $P(\bar{x})$ 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 wfdfs which simulates the polynomial $P(\bar{x}) = x_j$. Starting

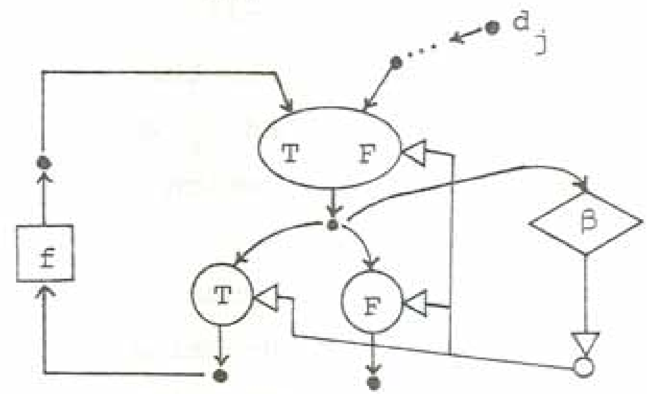


Figure 4.6 S_P for simulating $P(\bar{x}) = x_j$

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.

<u>Construction 4.2</u>

(i) Given a wfdfs S_Q which simulates $Q(\bar{x}) = \prod_{i=1}^{r} x_i^{\alpha_i}$, $\alpha_i \geq 0$, and a variable x_j, construct a wfdfs S_P which simulates $P(\bar{x}) = Q(\bar{x}) * x_j$.

Let B_Q be the index set of the variables which are factors of $Q(\bar{x})$, $B_Q = \{i | \alpha_i > 0\}$. 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 δ_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 wfdfs's S_Q and S_R which simulates $Q(\bar{x})$ and $R(\bar{x})$, construct a wfdfs 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).

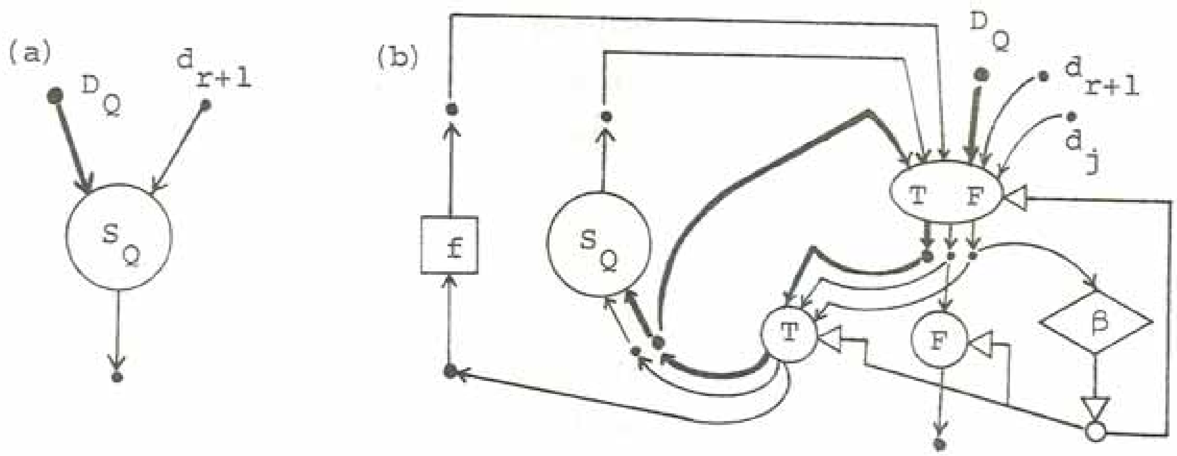


Figure 4.7 S_P for simulating $P(\bar{x})=Q(\bar{x})*x_j$

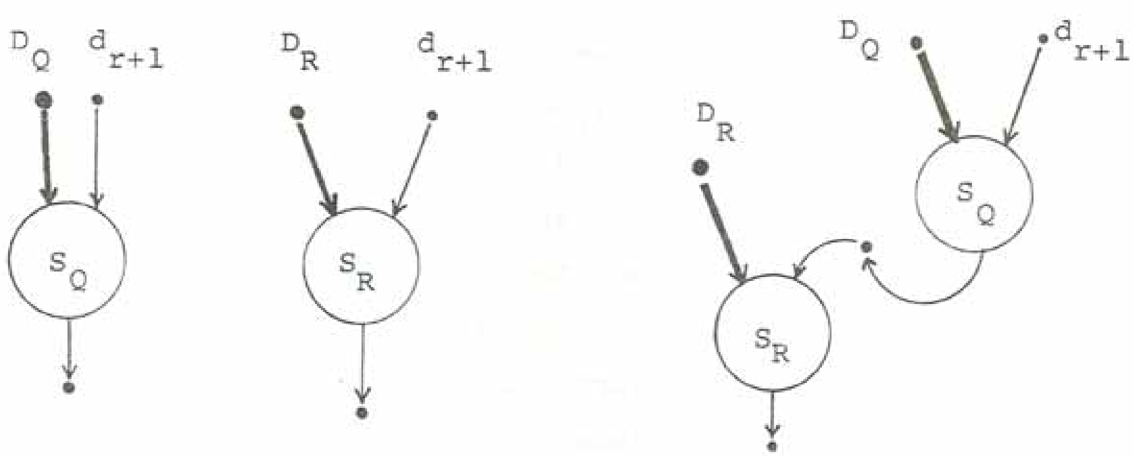


Figure 4.8 S_P for simulating $P(\bar{x})=Q(\bar{x})+R(\bar{x})$

In the process of computing $P(\bar{x}_I)$ under a free interpretation I, S_P generates all intermediate values $f^j \square \delta_{r+1}$, $1 \le j \le P(\bar{x}_I)$. In reducing Hilbert's Tenth Problem to the equivalence problem, we have to test every such intermediate value by the predicate β and determine whether $\beta(f^j \square \delta_{r+1}) = \underline{T}$ for all $1 \le j \le P(\bar{x}_I)$. 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, $P(f^j \square \delta_{r+1}) = \underline{T}$ for $1 \le j \le P(\bar{x}_I)$, 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 d_{r+1} to the output data link of S_P is labelled by f. Conditional wfdfs'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.

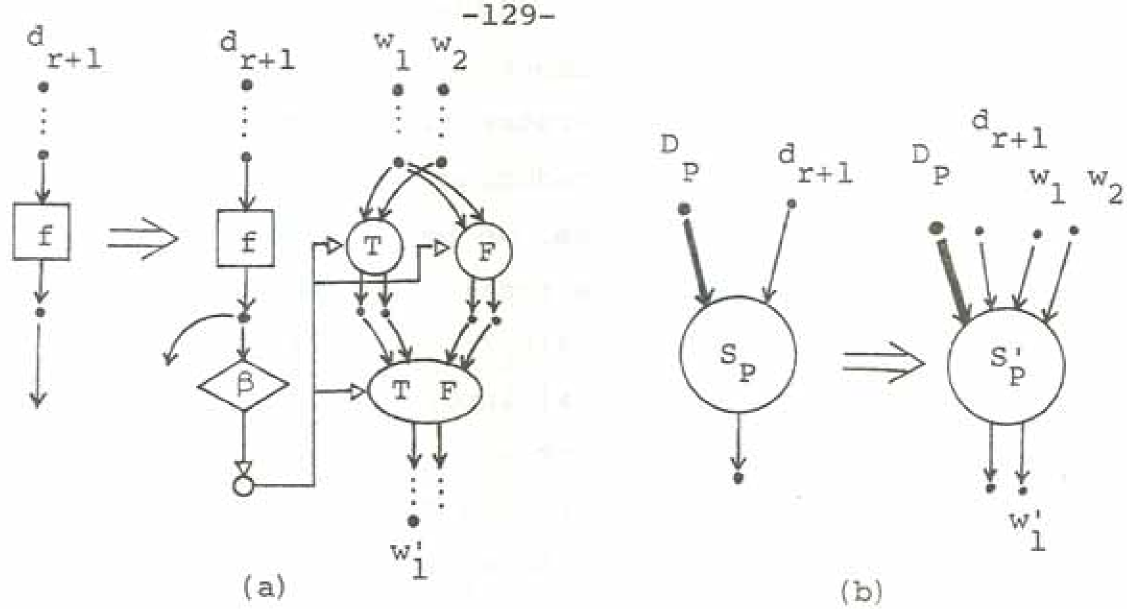


Figure 4.9 Modifying S_P to S'_P

Construction 4.3

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

Construction of S_1 From $P(\bar{x})$ construct S_P which simulates $P(\bar{x})$ 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 $\delta_1, \ldots, \delta_r$, δ_{r+1}, and δ_{r+2}. Let $\bar{x}_I$ be the r-tuple derived from I. If under I:

(i) $\beta(\delta_{r+2})=\underline{T}$. The output of S_1 is δ_{r+1}.

(ii) $\beta(\delta_{r+2})=\underline{F}$ and $\beta(f\Box\delta_{r+2})=\underline{F}$. The output of S_1 is δ_{r+1}.

(iii) $\beta(\delta_{r+2})=\underline{F}$ and $\beta(f\Box\delta_{r+2})=\underline{T}$, $P(\bar{x}_I)=\infty$. S_1 diverges since S_P diverges.

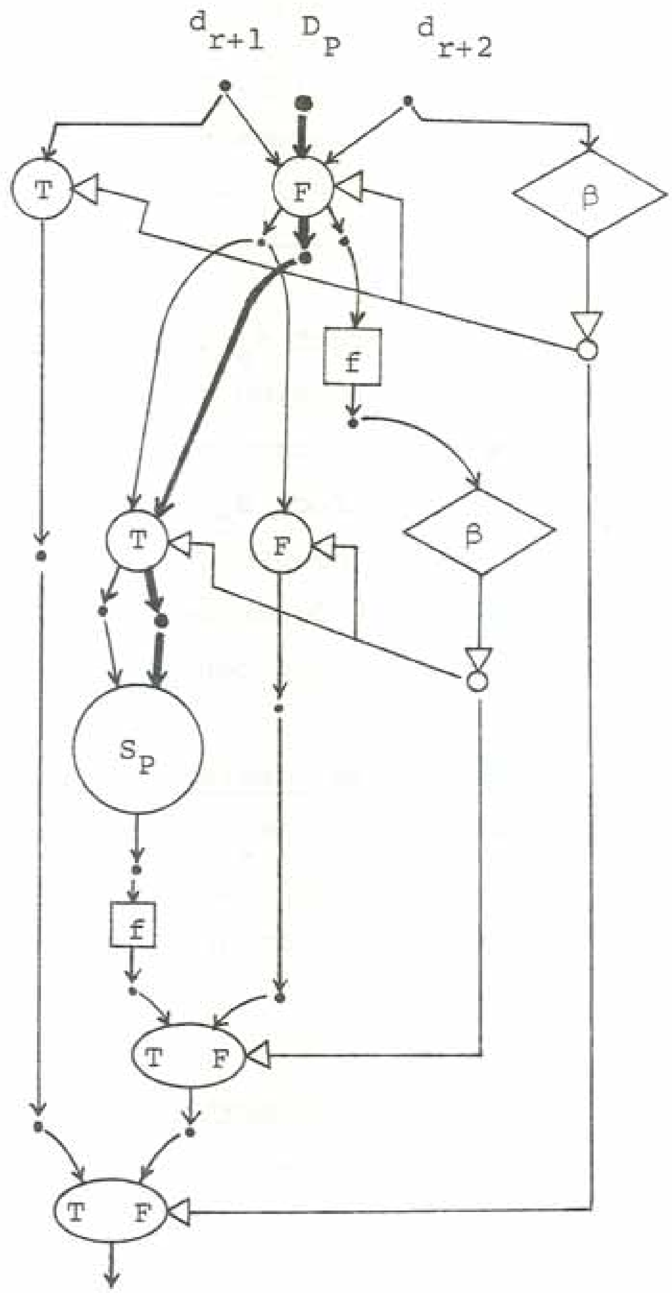


Figure 4.10 S_1 in Construction 4.3

(iv) $\beta(\delta_{r+2})=\underline{F}$ and $\beta(f\boxminus\delta_{r+2})=\underline{T}$, $P(\bar{x}_I)<\infty$. In this case S_P is activated and converges with output $\underline{f^{P(\bar{x}_I)}\boxminus\delta_{r+1}}$.
The output of S_1 is $\underline{f^{P(\bar{x}_I)+1}\boxminus\delta_{r+1}}$.

Construction of S_2 Construct S_P from $P(\bar{x})$ using Construction 4.2. Modify S_P to S_P' by adding conditional wfdfs'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 δ_{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) $\beta(\delta_{r+2}) = \underline{T}$
The T-branch of the outermost conditional wfdfs in S_2
The output of S_2 is δ_{r+1}.

(ii) $\beta(\delta_{r+2}) = \underline{F}$ and $\beta(f\boxminus\delta_{r+2}) = \underline{F}$
The F-branch of the conditional wfdfs which is contained in the F-branch of the outermost conditional wfdfs is taken. The output of S_2 is δ_{r+1}.

(iii) $\beta(\delta_{r+2}) = \underline{F}$, $\beta(f\boxminus\delta_{r+2}) = \underline{T}$, and $P(\bar{x}_I)=\infty$
The T-branch of the conditional wfdfs which contains S_P'

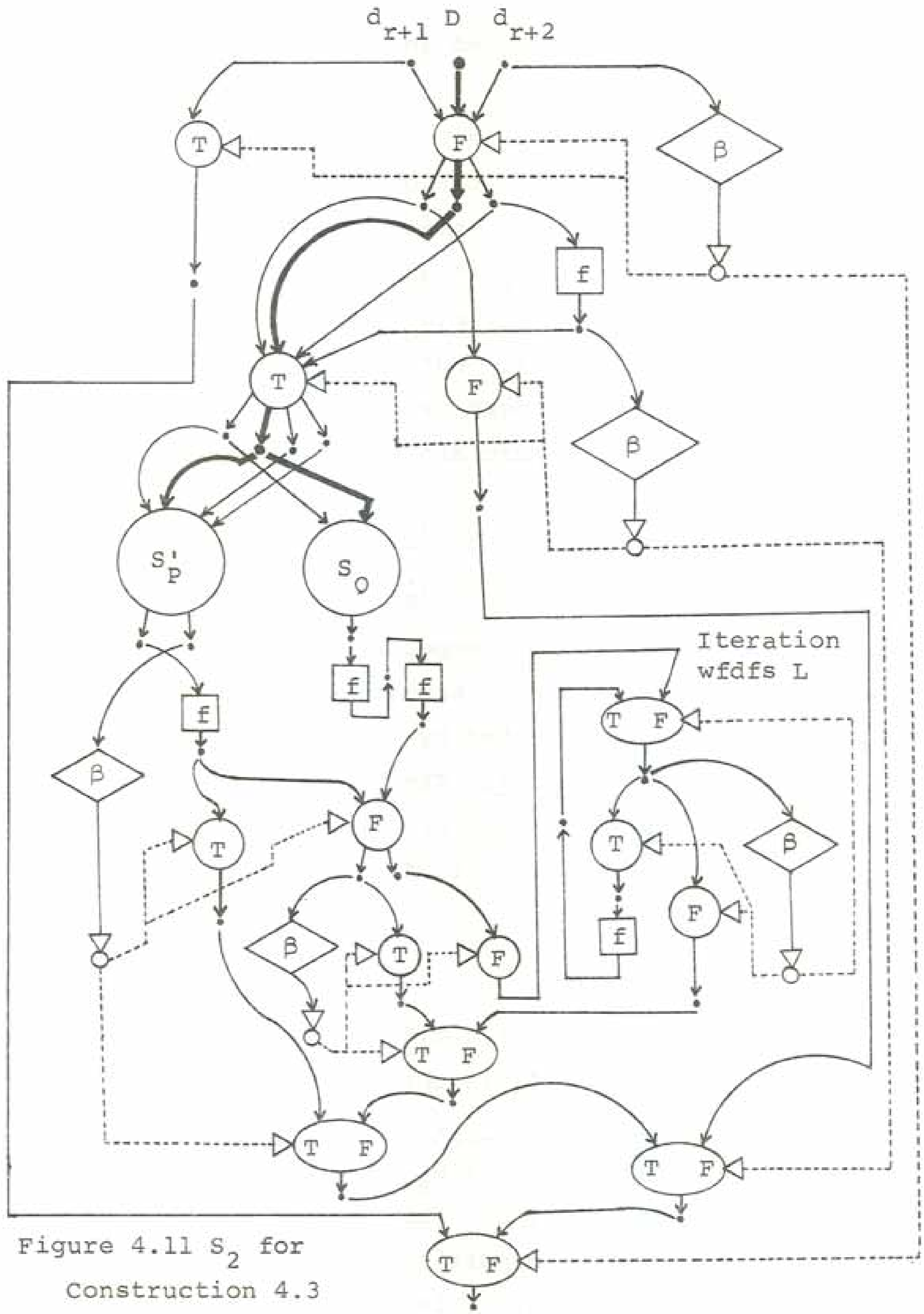


Figure 4.11 S_2 for Construction 4.3

is taken. S_P' is activated and diverges. S_2 diverges.

(iv) $\beta(\delta_{r+2}) = \underline{F}$, $\beta(f\boxtimes\delta_{r+2}) = \underline{T}$, and $P(\bar{x}_I)<\infty$

As in Case (iii), the T-branch of the conditional wfdfs 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 $P(\bar{x})$, 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:

<u>Case 1</u> $\exists j$, $1\leq j\leq P(\bar{x}_I)$, $\beta(f^j\boxtimes\delta_{r+1}) = \underline{F}$

From the structure of S_P', the outputs of S_P' are $\underline{f^{P(\bar{x}_I)}\boxtimes\delta_{r+1}}$ and $f\boxtimes\delta_{r+2}$ respectively. Under I $\beta(f\boxtimes\delta_{r+2})$ has outcome $\underline{T}$. After S_P' terminates, the output of S_P' which is $f\boxtimes\delta_{r+2}$ is tested by a decider labelled by β. This test has outcome $\underline{T}$. The T-branch of the conditional wfdfs which is controlled by the decider is taken. The output of this conditional wfdfs, hence of S_2, is $\underline{f^{P(\bar{x}_I)+1}\boxtimes\delta_{r+1}}$.

<u>Case 2</u> $\forall j$, $1\leq j\leq P(\bar{x}_I)+1$, $\beta(f^j\boxtimes\delta_{r+1}) = \underline{T}$

The outputs of S_P' in this case are $\underline{f^{P(\bar{x}_I)}\boxtimes\delta_{r+1}}$ and δ_{r+2}. Under I, $\beta(\delta_{r+2})$ has outcome $\underline{F}$. After S_P' terminates, the output of S_P' which is δ_{r+2} is tested by a decider labelled by β. This decision has outcome $\underline{F}$. The F-branch of the conditional wfdfs controlled by this decider is taken. This F-branch contains a conditional wfdfs which is controlled by another decider labelled by β. This second decider tests the data value $\underline{f^{P(\bar{x}_I)+1}\square\delta}_{r+1}$ and has outcome $\underline{T}$. The output of the conditional wfdfs controlled by this second decider, and hence of S_2, is $\underline{f^{P(\bar{x}_I)+1}\square\delta}_{r+1}$.

<u>Case 3</u> $\forall j,\ 1\le j\le P(\bar{x}_I),\ \beta(f^j\square\delta_{r+1}) = \underline{T}$,

$$\beta(\underline{f^{P(\bar{x}_I)+1}\square\delta}_{r+1}) = \underline{F}$$

In this case the outputs of S_P' are $\underline{f^{P(\bar{x}_I)+1}\square\delta}_{r+1}$ and δ_{r+2}. After S_P' converges, these two outputs of S_P' are tested by deciders labelled by β. Both of these tests have outcome $\underline{F}$. Iteration wfdfs L is activated with input data value $\underline{f^{Q(\bar{x}_I)+2}\square\delta}_{r+1}$. The output of S_2 is the output of L. Let $\tilde{j}$ be the <u>least</u> integer, $\tilde{j}\ge 0$, such that under I,

$$\beta(\underline{f^{Q(\bar{x}_I)+2+\tilde{j}}\square\delta}_{r+1}) = \underline{F}$$

The output of S_2 under I is

$$\underline{f^{Q(\bar{x}_I)+2+\tilde{j}}\square\delta}_{r+1}\ .$$

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

$$P(\bar{x}_I) + 1 \geq Q(\bar{x}_I) + 2 \quad .$$

Under Case (iv).3, $\forall j,\ 1 \leq j \leq P(\bar{x}_I)$, $\beta(f^j \amalg \delta_{r+1}) = \underline{T}$

$$\beta(f^{P(\bar{x}_I)+1} \amalg \delta_{r+1}) = \underline{F} \ .$$

$$\tilde{j} = P(\bar{x}_I) - Q(\bar{x}_I) - 1 \quad \text{by definition, and}$$

$$\text{the output of } L = \underline{f^{Q(\bar{x}_I)+2+(P(\bar{x}_I)-Q(\bar{x}_I)-1)} \amalg \delta_{r+1}}$$

$$= \underline{f^{P(\bar{x}_I)+1} \amalg \delta_{r+1}} \quad ,$$

and S_1, S_2 have the same output.

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

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.

$$\bar{x}_I = \bar{x}_o \quad \text{by definition of I}$$

$$P(\bar{x}_I) = Q(\bar{x}_I)$$

$$Q(\bar{x}_I)+2 > P(\bar{x}_I)+1$$

The output of S_1 under I is $\underline{f^{P(\bar{x}_I)+1} \amalg \delta_{r+1}}$.

The output of S_2 under I is $\underline{f^{Q(\bar{x}_I)+2+\tilde{j}} \amalg \delta_{r+1}}$.

S_1, S_2 are not equivalent under I.

Thus <u>if $P(\bar{x}_0) = Q(\bar{x}_0)$ for some $\bar{x}_0 \in N^r$, S_1 and S_2 are not</u>

equivalent.

End of Construction 4.3.

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:

$$T = \{ (S_1, S_2) \mid S_1, S_2 \text{ are constructed using Construction 4.3, given two polynomials } P(\bar{x}), Q(\bar{x}) \text{ such that for all } \bar{x}_o \in N^r, P(\bar{x}_o) \geq Q(\bar{x}_o). \}$$

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 $P(\bar{x})=Q(\bar{x})+1$, however, S_2 is not open. The first decision made when L is activated in S_2 under a free interpretation I is $\beta(f^{Q(\bar{x}_I)+2} \square \delta_{r+1})$. This decision always follows the decision $\beta(f^{P(\bar{x}_I)+1} \square \delta_{r+1})$. If L is activated under I, this latter decision has outcome F. Since $P(\bar{x})=Q(\bar{x})+1$, 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.

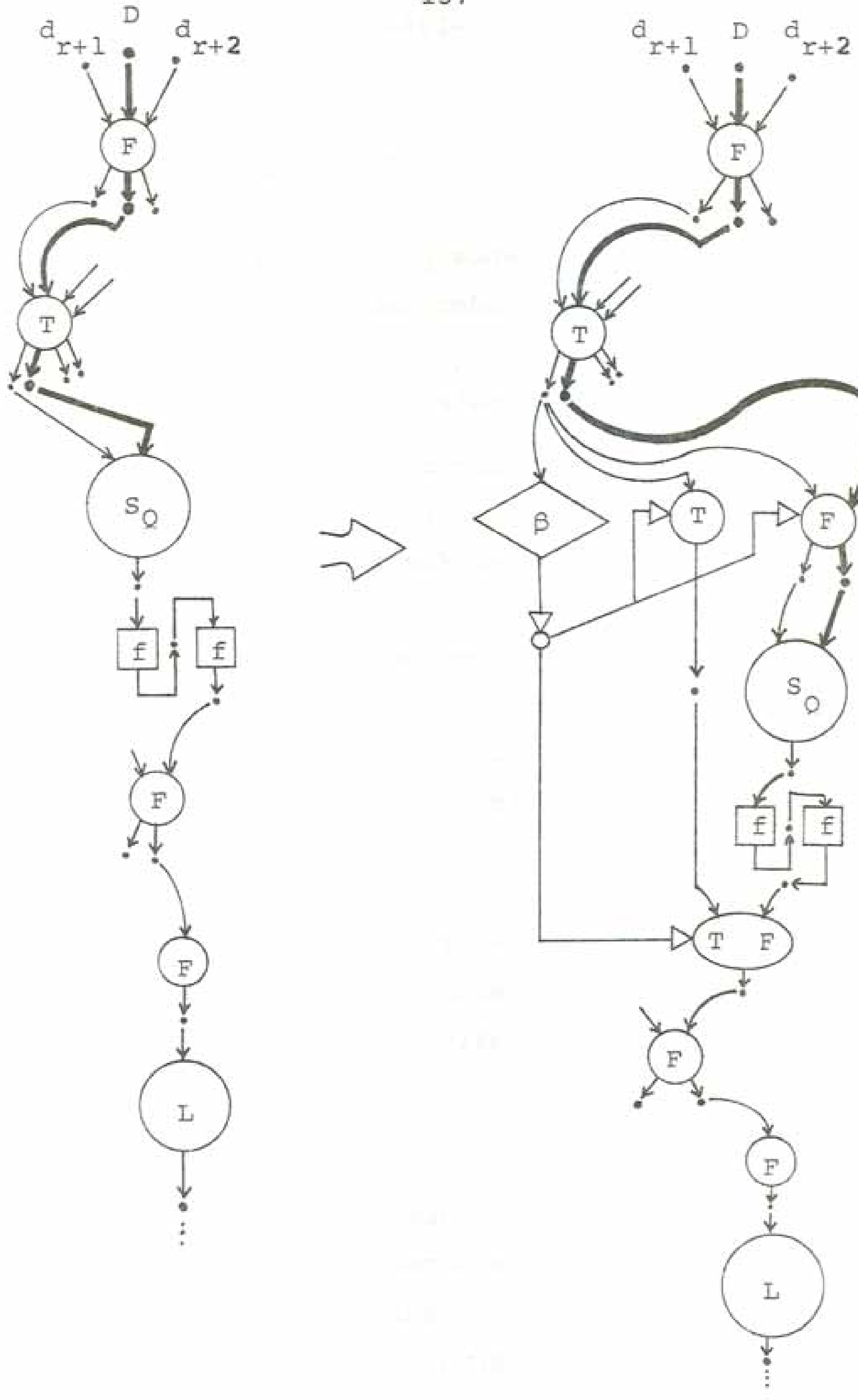


Figure 4.12 Modifying S_2 to S_2' in Theorem 4.2

To prove the undecidability of equivalence in open and complete wfdfs's, we have to modify S_2 to an equivalent open and complete wfdfs S_2' . The modification consists of replacing S_Q in S_2 by a conditional wfdfs controlled by the decision $\beta(\delta_{r+1})$ (Figure 4.12) . If this decision has outcome $\underline{F}$, the output of the conditional wfdfs is the output of S_Q, which is $\underline{f^{Q(\bar{x}_I)} \boxplus \delta}_{r+1}$. If the decision has outcome $\underline{T}$, the output is δ_{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 equivalent.

Under Case (iv), if $\beta(\delta_{r+1})=\underline{F}$ 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 $\exists \bar{x}_o$ such that $P(\bar{x}_o)=Q(\bar{x}_o)$.

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

$$\text{for } 1\le j\le P(\bar{x}_I), \quad \beta(f^j \boxplus \delta_{r+1}) = \underline{T} ,$$

$$\beta(\underline{f^{P(\bar{x}_I)+1} \boxplus \delta}_{r+1}) = \underline{F}$$

$$\beta(\delta_{r+1}) = \underline{T}$$

Since the input to L is δ_{r+1}, the body of L is entered at least once. As in Construction 4.3, the output of L is $\underline{f^{\tilde{j}+1} \boxplus \delta}_{r+1}$, where $\tilde{j}$ is the least integer, $\tilde{j}\ge 0$, such that

$\beta(\underline{f^{\tilde{j}+1} \boxdot \delta}_{r+1}) = \underline{F}$. The set of decision outcomes listed above implies that the output of L is $\underline{f^{P(\bar{x}_I)+1} \boxdot \delta}_{r+1}$, which is alos 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 $P(\bar{x}_o) > Q(\bar{x}_o)$ for all $\bar{x}_o$. 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 wfdfs's. The equivalence problem for open and complete wfdfs's is thus undecidable.

Q.E.D.

<u>Corollary 4.3-1</u> Openness is undecidable in complete wfdfs's.
<u>Proof</u>:

In Theorem 4.3 the equivalence problem for the pairs of open and complete wfdfs's $\{ (S_1, S_2') \}$ is undecidable. For every such pair we can construct a wfdfs Z which activates S_1 and S_2', makes a decision with the output of S_1, 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. α 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 $\underline{T}$ (or $\underline{F}$). 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.

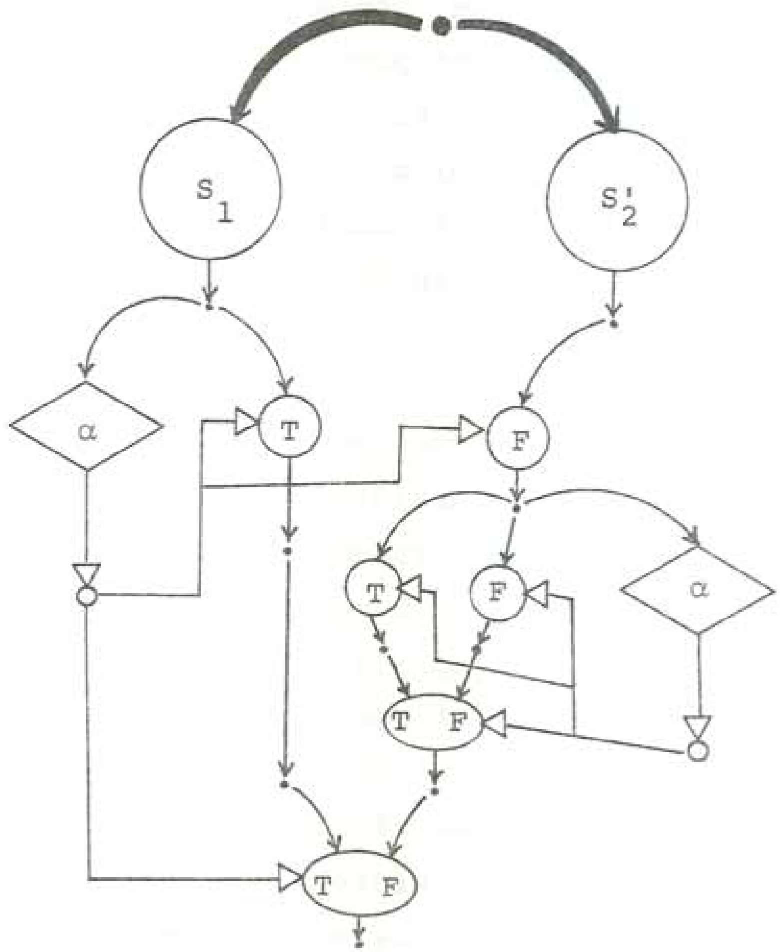


Figure 4.13 Complete wfdfs Z for Corollary 4.3-1

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 wfdfs'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 α-free wfdfs's are proper subclasses of free flowchart schemas. The equivalence problems for free and α-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 α-free wfdfs's is thus an approach to the solution of the equivalence problem in free program schemas.

BIBLIOGRAPHY

[1] Ashcroft, E. & Manna, A. The translation of 'goto' programs to 'while' programs. Information Processing 71 North Holland Publishing Co., Amsterdam 1972.

[2] Chandra, A.K. On the properties and applications of program schemas. Stan-CS-73-336, Stanford University, Computer Science Department, March 1972.

[3] Constable, R.L. & Gries, D. On classes of program schemata. SIAM J. Computing 1:1. March 1972

[4] de Bakker, J.W. & Scott, D. A theory of programs unpublished notes, IBM Seminar, Vienna 1969.

[5] Dennis, J.B. First version of a data flow procedure language. Proceedings of the Symposium on Programming. University of Paris, April 1974.

[6] Dennis, J.B. & Fosseen, J. Introduction of data flow schemas. Computation Structures Group Memo 81, MIT Project MAC, September 1973.

[7] Dennis, J.B. & Misunas, D.P. A computer architecture for highly parallel signal processing. Proceedings of the 1974 ACM National Conference, Association for Computing Machinery, New York, November 1974.

[8] Engeler, E. Structure and meaning of elementary programs. Symposium on Semantics of Algorithmic Languages. Lecture Notes in Mathematics 188, Springer-Verlag 1971.

[9] Fosseen, J. Representation of algorithms by maximally parallel schemata. S.M.Thesis, Department of Electrical Engineering and Computer Science, June 1972.

[10] Garland, S. & Luckham, D. Program schemes, recursion schemes, and formal languages. JCSS 7:2, April 1973.

[11] Hewitt, C. & Paterson, M.S. Comparative Schematology. Record of Project MAC Conference on Concurrent Systems and Parallel Computation, ACM New York, 1970.

[12] Isaman, D.L. Data structuring operations for parallel processors. Forthcoming Ph.D. Thesis.

[13] Karp, R.M. ¢ Miller, R.E. Parallel program schemata JCSS 3:2, May 1969.

[14] Keller, R.M. Parallel program schemata and maximal parallelism. JACM 20:3 &20:4, July & October 1973.

[15] Kosinski, P. A data flow language for operating system programming. SIGPLAN Notices, 8:9, September 1973.

[16] Knuth, D.E. & Floyd, R.W. Notes on avoiding "go to" statements. Information Processing Letters 1, North-Holland Publishing Co., The Netherlands, 1971.

[17] Lesser, V.R. The design of an emulator for a parallel machine language. Proceedings of ACM SIGPLAN-SIGMICRO Interface Meeting, June 1973.

[18] Linderman, J.P. Productivity in parallel computation schemata. MAC-TR-111, MIT Project MAC, December 1973.

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

[20] Post, E. A variant of a recursively unsolvable problem. Bulletin of the American Mathematical Society, 52, 1946.

[21] Qualitz, J.E. Weakly productive computation schemata. S.M.Thesis, Department of Electrical Engineering & Computer Science, MIT, May 1972.

[22] Qualitz, J.E. Equivalence problems for monadic schemas. Ph.D. Thesis, Department of Electrical Engineering & Computer Science, MIT, May 1975.

[23] Rodriguez, J.D. A graph model for parallel computation. MAC-TR-64, MIT Project MAC, September 1969.

[24] Rumbaugh, J.E. A parallel asynchronous computer architecture for data flow programs. Ph.D. Thesis, Department of Electrical Engineering & Computer Science, MIT, May 1975.

[25] Rutledge, J. On Ianov's program schemata. JACM 11:1, January 1964.

[26] Slutz, D.R. The flow graph schemata model of parallel computation. MAC-TR-53, MIT Project MAC, September 1968.

Appendix

Proof of Lemma 3.2 (Section 3.1.2)

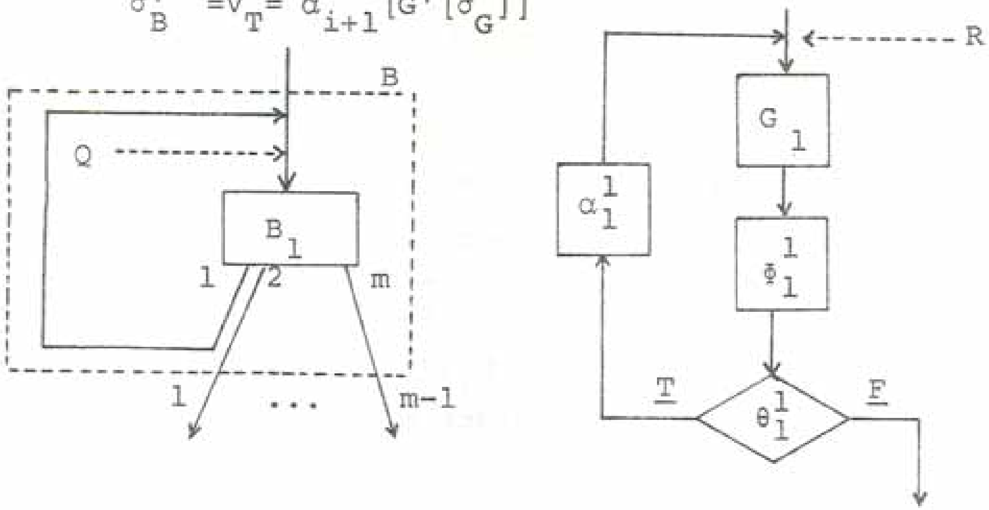
Let B be a nf-block formed by loop formation from a nf-block B_1, as shown in Figure A.1.

Generate blocks G_1, α_i^1's and β_i^1's to simulate B_1. Let G' be the subschema constructed from G_1, α_i^1's and β_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 σ_B and σ_G be 2 memory states which are equivalent wrt V_T. B, if entered with σ_B, reaches the point Q (Figure A.1) n times and $B[\sigma_B]=(\sigma_B', i)$ iff G', if entered with σ_G, reaches the point R (Figure A.1) n times,

and for $1 \le k \le i$, $\beta_k^1[G'[\sigma_G]]=\underline{F}$, $\beta_{i+1}^1[G'[\sigma_G]]=\underline{T}$,

and $\sigma_B' =V_T= \alpha_{i+1}^1[G'[\sigma_G]]$



B 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 σ_B, reach the point Q only once and exit at its i-th exit with σ'_B.

From the block structure of B, and the facts that G_1, α^1_i's and β^1_i's simulate B_1:

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

(ii) $\sigma'_B =_{V_T}= \alpha^1_{i+1}[G_1[\sigma_G]]$

(iii) For $1 \le k \le i$, $\beta^1_i[G_1[\sigma_G]]=\underline{F}$, $\beta^1_{i+1}[G_1[\sigma_G]]=\underline{T}$

From the branching control outcomes in (iii), the structure of G', and the non-interference condition for G_1, α^1_i's and β^1_i's:

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

(v) for $1 \le k \le i$, $\beta^1_i[G'[\sigma_G]]=\underline{F}$, $\beta^1_{i+1}[G'[\sigma_G]]=\underline{T}$, and

$$\sigma'_B =_{V_T}= \alpha^1_{i+1}[G'[\sigma_G]]$$

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

$$\text{for } 1 \le k \le i,\ \beta^1_k[G_1[\sigma_G]]=\underline{F},\ \beta^1_{i+1}[G_1[\sigma_G]]=\underline{T}, \text{ and}$$

$$G'[\sigma_G] =_{V_T}= G_1[\sigma_G]$$

Since G_1, α^1_i's and β^1_i's simulate B_1,

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

$$\sigma'_B =_{V_T}= \alpha^1_{i+1}[G_1[\sigma_G]] =_{V_T}= \alpha^1_{i+1}[G'[\sigma_G]]$$

From the structure of B, the point Q in B is reached only once, and $B[\sigma_B] = (\sigma'_B, i)$

<u>Induction Hypothesis</u>: Theorem A.1 holds for $n=\ell$.

<u>Induction Step</u>: $n=\ell+1$

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

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

(ii) B is reentered with state σ''_B and reaches the point Q ℓ more times.

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

Since G_1, α^1_i's and β^1_i's simulate B_1,

$$\sigma''_B =_{V_T}= \alpha^1_1[G_1[\sigma_G]]$$

$$\beta^1_1[G_1[\sigma_G]] = \underline{T}$$

From the block structure of G', if G' is entered with σ_G, G' is reentered with σ''_G after reaching the point R once, and

(i) $\sigma''_G =_{V_T}= \alpha^1_1[G_1[\sigma_G]] =_{V_T}= \sigma''_B$

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

(iii) G' exits after reaching the point R ℓ more times.

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

(i) G', when entered with σ''_G, reaches the point R ℓ more times. B, when entered with state σ''_B, reaches the point Q ℓ more times.

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

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

Q.E.D.

<u>Proof of Lemma 3.2</u>:

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 $\sigma_B =_{V_T} \sigma_G$, then:

(i) B terminates on σ_B iff G terminates on σ_G.

(ii) If B terminates and $B[\sigma_B] = (\sigma'_B, i)$, then G terminates and for $1 \le k \le i$, $\beta^1_i[G[\sigma_G]] = \underline{F}$, $\beta^1_{i+1}[G[\sigma_G]] = \underline{T}$

$$\sigma'_B =_{V_T} \alpha^1_{i+1}[G[\sigma_G]]$$

Q.E.D.