CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Untangling unstructured programs
Authors	Oulsnam, Gordon
Publication date	1984
Original Citation	Oulsnam, G. 1984. Untangling unstructured programs. PhD Thesis, University College Cork.
Type of publication	Doctoral thesis
Link to publisher's version	http://library.ucc.ie/record=b1100709~S0
Rights	© 1984, Gordon Oulsnam - http://creativecommons.org/licenses/by-nc-nd/3.0/
Download date	2026-11-01 12:49:01
Item downloaded from	https://hdl.handle.net/10468/1659



UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

DP1984 OULS
239155

UNTANGLING UNSTRUCTURED PROGRAMS

by

Gordon Oulsnam

Submitted for the Degree of
Doctor of Philosophy to the
National University of Ireland.

The research contained in this thesis was begun in the Department of Computer Science University of Queensland Australia and was completed in the Department of Computer Science in the Faculty of Science at the University College Cork. The work has not been submitted in whole or in part to any other Institution for an Academic Award.

Date of submission: July 1984

Head of Department: Prof. P.G. O'Regan

Supervisor: Prof. P.G. O'Regan

G. Oulsnam.

UNIVERSITY COLLEGE CORK Library

CONTENTS

Summary

1. Introduction 1

2. Definitions 4

3. Basic Unstructured Forms 11

4. The Structuring Transforms 15

5. Effectiveness of the Transforms 24

6. Identifying Basic Unstructured Forms 30

7. The Structuring Process 41

8. Examples 50

9. Space and Time Overheads 61

Conclusion 64

Figures 65

Appendix 1 95

Proof of the forward path algorithm

Appendix 2 99

Abstract Data Type Specifications

Addendum

Unravelling Unstructured Programs

ACKNOWLEDGEMENTS

Thanks are due and gladly given to Prof. P.G. O'Regan for undertaking the task of supervision, and to Prof. F.H. Sumner, Manchester University, for his help in finding an External Examiner.

Thanks are also gladly given to my wife for her support and acceptance of neglect during the tortuous months of thesis preparation.

The customary plaudits for typing are omitted as all typing and art work was done by the author. In this regard a special word of thanks must go to Tipp-Ex Vertrieb GmbH & Co KG of Frankfurt West Germany without whose product this thesis would never have been completed and upon which so much of it depends!

Typed in Courier 10-point on an IBM Selectric Model 72.

SUMMARY

A method is presented for converting unstructured program schemas to strictly equivalent structured form. The predicates of the original schema are left intact with structuring being achieved by the duplication of the original decision vertices without the introduction of compound predicate expressions, or where possible by function duplication alone. It is shown that structured schemas must have at least as many decision vertices as the original unstructured schema, and must have more when the original schema contains branches out of decision constructs. The structuring method allows the complete avoidance of function duplication, but only at the expense of decision vertex duplication. It is shown that structured schemas have greater space-time requirements in general than their equivalent optimal unstructured counterparts and at best have the same requirements.

1. INTRODUCTION

This thesis presents a method for transforming unstructured program schemas into structured equivalents in D-chart format.[1] The form of the derived structured programs is such that the original unstructured forms can be easily recovered, thus revealing what overheads in space and time are inherent in the structured forms. The method enables the user to opt for minimization of time overheads, minimization of space overheads, or some intermediate compromise. A measure for the introduced overheads is given which can be used to compare the relative conceptual complexities of unstructured programs. A feature of the structuring method is that the number of introduced auxiliary Boolean variables, or flags, is kept to a minimum, and where such flags are introduced, they correspond exactly to some conditional expression, or predicate, in the original program. Thus the method preserves as far as possible the logic of the original algorithm.

The problem of transforming schemas into some standard form has been widely addressed in the literature. Methods based on yielding a schema in while-program form have been given by Jacopini,[2] Ashcroft and Manna,[3] Knuth and Floyd,[4] Bruno and Steiglitz,[1] Mills,[5] Kasai,[6] Williams,[7] Williams and Ossher[8] and Oulsnam.[9] The method of Jacopini was shown by Cooper[10] to yield in a trivial way a schema consisting of a single while statement enclosing a sequence of alternations based on introduced auxiliary variables. (This result was hardly surprising since it simply restates in computer science terminology what has been long known to mathematicians - that any general recursive computable function can be computed using at most one application

of the unbounded minimization, or μ, operator, the mathematical counterpart of the while construct.) Jacopini's conjecture that in general auxiliary variables would be necessary to transform arbitrary schemas into D-chart form was proved by Ashcroft and Manna,[3] Knuth and Floyd[4] and Bruno and Steiglitz.[1] Kasai[6] and Bainbridge[11] describe methods of reducing while-programs to minimal form, while the general capabilities and limitations of D-charts as a standard form were considered by Paterson, Kasami and Tokura[12] and Kosaraju.[13]

The necessity for auxiliary variables, coupled with the fact that schemas in while-program form were shown by Paterson *et al.*[12] to generally require some duplication of basic functions or predicates of the original schema, has led to consideration of more general standard control structures than D-charts. Wulf[14] has proposed the use of multi-level control structures, and further generalizations were analysed by Kosaraju[13] and Ledgard and Marcotty,[15] with some refinements of their results by Cherniavsky *et al.*[16] Proposals based on the non-duplication of the original schema's functions and predicates have been given by Urschler,[17] using a technique based on the back dominators of the original schema, and by Baker.[18] Of necessity both methods allow the use of *GOTO* statements although Urschler restricts these to backwards jumps only. A standard form based on binary trees has been proposed by Engeler[19] and Wegner.[20] The former allows only jumps to ancestor nodes in the tree, while the latter allows jumps in both directions. Proposals to convert schemas to recursive form have been made by Knuth and Floyd[4] and Urschler.[17] McCabe[21] and Williams[7] independently identified the basic forms of unstructuredness, while transformations based on the identification and elimination of these constructs have been given by

Williams,[7] Williams and Ossher[8] and Oulsnam.[9] Dijkstra,[22] echoed by Knuth,[23] has cautioned against expecting mechanical transformations of schemas to yield more comprehensible programs, while Knuth[23] has examined the problem of efficiency relating to programs translated to structured form. A comprehensive survey of program transformation systems is given in Partsch and Steinbrüggen.[24] Van Emden[25] has questioned the need for structured programming and has proposed a method for deriving programs directly with minimal function and predicate duplication.

The remainder of this thesis is organized as follows. Section 2 introduces some necessary definitions and concepts, Section 3 briefly reviews the basic forms of unstructuredness in schemas, while in Section 4 a method for their removal based on structured transforms is given. A proof of the effectiveness of the transforms in general is given in Section 5; Section 6 is concerned with the identification of the basic forms of unstructuredness in arbitrary schemas and Section 7 gives the full structuring algorithm. Section 8 is addressed to the practical application of the structuring method, while Section 9 is concerned with the space-time overheads incurred by structuring.

2. DEFINITIONS

Schemas

A schema is a labelled program flowgraph[26] which shows the control structure of the program whilst leaving the transformations on the program's variables to be determined by an interpretation of the labelling. Thus a schema represents a family of programs having a common control structure. A *program schema* is defined as the triple $G = (V,\Gamma,\Sigma)$ where V is a set of vertices or nodes, Γ is a mapping $\Gamma: V \to 2^V$, and Σ an alphabet of operators. If $u,v \in V$, $v \in \Gamma u$, and $\alpha \in \Sigma^*$ then $(u,v:\alpha)$ is an arc of G directed from u to v labelled by α. Vertex u is the *tail* of the arc, and v the *head*. If $v \in \Gamma u$ then $u \in \Gamma^{-1}v$. By extension $\Gamma^n u = \Gamma^{n-1}.\Gamma u$ and $\Gamma^{-n}v = \Gamma^{-n+1}.\Gamma^{-1}v$, for $n > 1$. The set of arcs $\{(u,v:\alpha) \mid v \in \Gamma u\}$ will be written $(u,\Gamma u:*)$, and the set of all arcs $\{(u,\Gamma u:*) \mid u \in V\}$ will be written E. Whenever convenient, G will be considered to be defined equivalently by (V,E,Σ). When the label of an arc is not of immediate concern it will be elided and the arc written in the abbreviated form (u,v); G similarly will be written (V,Γ) or (V,E).

$\Gamma^* u = \{u\} \cup \Gamma u \cup \Gamma^2 u \ldots$ is the set of vertices *reachable* from u. Similarly $\Gamma^{-*} u = \{u\} \cup \Gamma^{-1} u \ldots$ is the set of vertices that *reach* u. For $A \subset V$, $\Gamma A = \{v \mid v = \Gamma u,\ u \in A\}$, with corresponding definitions for $\Gamma^{-1}A$, $\Gamma^n A$, and so on. If $u,v \in V$ and $v \in \Gamma^* u$ then u is a *predecessor* of v, and v is a *successor* of u. If $v \in \Gamma u$ then u is an *immediate predecessor* of v, and v is an *immediate successor* of u.

A *path* $[u,v:\alpha]$ is a sequence of arcs $(u_0,u_1:\alpha_1) \ldots (u_{i-1},u_i:\alpha_i) \ldots (u_{n-1},u_n:\alpha_n)$ where $u_i \in \Gamma u_{i-1}$,

$u_0 = u$, $u_n = v$, $\alpha_i \in \Sigma^*$, and $\alpha = \alpha_1 \ldots \alpha_i \ldots \alpha_n$, for $1 \leq i \leq n$. The *length* of the path is n, the number of arcs on the path. An *elementary path* is one on which no vertex occurs more than once, while a *simple path* is one on which no arc occurs more than once. Any simple path $[u,u:\alpha]$ is a *cycle*, and a cycle of length one is called a *loop*.

Every arc of G is labelled with some string over Σ^*. The elements of Σ represent functions which operate on program variables. A program has three sets of variables: the input set X, the local set Y, and the output set Z. The operators of Σ over these variables are of three types. Let $f,g,p \in \Sigma$, then $f: X \times Y \rightarrow Y$ is a local function, $g: X \times Y \rightarrow Z$ is an output function, and $p: X \times Y \rightarrow \{\text{true, false}\}$ is a predicate. The logical negation of p is written $\bar{p}$. The composition of two local functions $f_1,f_2 \in \Sigma$, $f_2(X,f_1(X,Y))$, is written f_1f_2, which in programming terms corresponds to the sequencing operation: $Y \leftarrow f_1(X,Y)$; $Y \leftarrow f_2(X,Y)$. Similarly $g(X,f(X,Y))$ is written fg, while the conditional execution of a function *if* $p(X,Y)$ *then* $Y \leftarrow f(X,Y)$ is written pf.

Let $u,v,w \in V$, and $\alpha,\beta \in \Sigma^*$. The vertices of a schema form five equivalence classes. These are *start* with in-degree 0 and out-degree 1; *chain* with in-degree 1 and out-degree 1; *decision* with in-degree 1 and out-degree 2; *collector* with in-degree 2 and out-degree 1; *halt* with in-degree 1 and out-degree 0. A schema with vertices having higher in-degree than 2 can be put in the above form using a cascade of collectors, while a cascade of decisions can be used in place of vertices with out-degree greater than 2. By extension it can be assumed that every schema has exactly one start vertex and exactly one halt vertex. A chain vertex v

on a path $(u,v:\alpha)(v,w:\beta)$ can be elided by replacing the path by a new arc $(u,w:\alpha\beta)$. Unless otherwise stated schemas are assumed to have had all chain vertices elided by this form of path compression. The two arcs $(u,v:\alpha)$ and $(u,w:\beta)$ of a decision vertex are labelled $p\alpha$ and $\bar{p}\beta$ respectively, where p is the predicate associated with the decision u.

Notational conventions

Unless otherwise stated the following notational conventions are henceforth adopted throughout to avoid repetitious definitions: $G = (V,\Gamma,\Sigma)$ or (V,E,Σ), $u,v,\ldots \in V$, $\alpha,\beta,\ldots \in \Sigma^*$, $a,b,\ldots \in \Sigma$ denoting functions, $p,q,\ldots \in \Sigma$ denoting predicates, and $A,B\ldots \subseteq V$. All symbols may also be sub- or superscripted. λ is used to denote the empty string.

Subschemas

$G_A = (A,\Gamma_A,\Sigma)$ is a *subschema* of G when Γ_A is defined such that $\Gamma_A u \subseteq \Gamma u \cap A$, $u \in A$. Thus an arc (u,v) in G with $u,v \in A$ is not necessarily an arc of G_A. The *difference* $G \sim G_A$ is defined to be the subschema $G_D = (D,\Gamma_D,\Sigma)$ where $\Gamma_D u = \Gamma u - \Gamma_A u$ and $D = \{u \mid \Gamma_D u \neq \phi \text{ or } \Gamma_D^{-1} u \neq \phi\}$. Let G_A be a subschema of G, then the *predecessors* of G_A are defined by $\Gamma^- A = \{u \mid v \in \Gamma u \cap A \text{ and } (u,v) \notin E_A\}$. Thus if (u,v) is an arc of G but not of G_A, and v is in A, then u is a predecessor vertex of G_A. Vertex u may or may not be in A - see Figure 1. The *successors* of G_A are defined similarly by $\Gamma^+ A = \{v \mid u \in \Gamma^{-1} v \cap A,\ (u,v) \notin E_A\}$ - Figure 1. It will often prove convenient to augment a subschema G_A of G as follows. Let $S_A = \{s \mid s \in \Gamma^+ A\}$, $P_A = \{p \mid p \in \Gamma^- A\}$, then define $\hat{G}_A = (B,\Gamma_B,\Sigma)$ such that $B = A \cup S_A \cup P_A$, and $\Gamma_B u = \Gamma u \cap B$. $\hat{G}_A$ is the *augmented subschema* G_A with respect to G.

Paths and back arcs

An *advancing path* is any elementary path $[s,u:\alpha]$ where $s \in S$, the start set of G. A *forward path* is any elementary path $[s,h:\alpha]$ where $h \in H$, the halt set of G. Let $F_\alpha = [s,h:\alpha]$ be a forward path and let v be a vertex on F_α. If there exists a cycle $(v,w:\gamma)\ldots(u,v:\beta)$ then $(u,v:\beta)$ is a *back latch* of F_α. Deletion of the back latch eliminates the cycle. A cycle may contain back latches from several different partially overlapping forward paths - see Figure 2, for instance. A back latch of one forward path may itself lie on some other forward path - again see Figure 2. The notion of back latch is extended recursively as follows. The back latches of the forward paths of G are back latches of G. If G_F is that subschema of G which consists of just the forward paths of G, then the back latches of $G \sim G_F$ are also defined to be back latches of G - see Figure 3.

Let B be a subset of the back latches of G such that if G_B is the schema comprising the arcs of B then $G \sim G_B$ is acyclic. Further, if for all $b \in B$, $G \sim G_{B-\{b\}}$ is not acyclic then B constitutes a minimal set of back latches whose deletion from G leaves G acyclic. Such a set is called a *back arc set* of G, and the corresponding subschema G_B is called a *back arc subschema* of G. Back arc sets and subschemas are not necessarily unique. Any subschema $G \sim G_B$, where G_B is a back arc subschema of G, is called an *advancing path subschema* of G, and by definition is acyclic. Let G be a schema with some back arc set B. For each arc $(u,v:\alpha\beta)$ of B replace $(u,v:\alpha\beta)$ in G by the arcs $(u,y_{uv}:\alpha)$ and $(x_{uv},v:\beta)$ where $x_{uv},y_{uv} \notin V$. The resultant schema $\hat{G} = (\hat{V},\hat{E},\Sigma)$ is called the *augmented advancing path subschema* of G, where $\hat{V} = V \cup \{x_{uv},y_{uv} \mid (u,v) \in B\}$, and $\hat{E} = E - B + \{(u,y_{uv}), (x_{uv},v) \mid (u,v) \in B\}$.

Computation sequences

A *regular expression* over an alphabet Σ describes a *regular set* of strings $\subseteq \Sigma^*$, and is defined recursively as follows. The empty string λ and elements of Σ are regular expressions. If α and β are regular expressions then so are $\alpha.\beta$, $\alpha+\beta$, α^* and (α), where '+' is used here to denote set union. Usually the concatenation operator '.' and the pair () will be elided if no ambiguity is thereby introduced.

The *computation sequences* of a schema are defined as follows. Let $[u,v:\alpha]$ be a path in the schema, then α represents a possible computation sequence from u to v. The set of all such computation sequences from u to v is called the *end set* of u with respect to v and is written $E(u,v)$. Let $u \in A \subset V$ and $v \in B \subset V$, then the notion of end set is extended to include $E(A,v)$, the union over A of all $E(u,v)$, with corresponding extensions for $E(u,B)$ and $E(A,B)$. Following Kleene[27] it is known that end sets are regular sets over Σ. Let S be the start set and H the halt set of a schema G, and let $E(u,H)$ be abbreviated to $E(u)$, then $E(S)$ is the *computation sequence set* of G, and represents all possible terminating computations over the schema.

Any vertex u such that $\Gamma^*S \cap \{u\} = \emptyset$ is unreachable from S, so $E(u)$ contributes nothing to the computation sequence set. Any vertex v such that $\Gamma^*v \cap H = \emptyset$ cannot reach the halt set, with the result that $E(v)$ is undefined and represents a non-terminating computation of the schema. Vertices such as u and v do not lie on any path from S to H. For simplicity it is assumed throughout that all vertices that do not lie on any path from S to H have been elided from the schemas.

Let $(u,v:\alpha)$ be an arc of a schema G, then it follows from the definition of end set that $E(u) = \alpha E(v)$. If u is a decision vertex such that $(u,v:p\alpha)$ and $(u,w:\bar{p}\beta)$ are arcs of G, then $E(u) = p\alpha E(v) + \bar{p}\beta E(w)$. If there are paths from v to u such that $E(v,u) = \gamma$, then $E(u) = p\alpha\gamma E(u) + \bar{p}\beta E(w)$, from which it can be shown[28] that $E(u) = (p\alpha\gamma)^{*}\bar{p}\beta E(w)$.

Structured schemas

A *structured regular expression (sre)* is defined recursively as follows. The empty string is an sre, as are the elements of Σ. If $p \in \Sigma$ is a predicate, and α,β are sre's, then so too are (α), $\alpha.\beta$, $p.\alpha+\bar{p}.\beta$ and $\alpha(p\beta\alpha)^{*}\bar{p}$. An arc $(u,v:\alpha)$ corresponds to the program construct u: $Y \leftarrow \alpha(X,Y)$; *goto* v, where u,v are program labels. Similarly $\alpha.\beta$ corresponds to $Y \leftarrow \alpha(X,Y)$; $Y \leftarrow \beta(X,Y)$, and $p\alpha+\bar{p}\beta$ to *if* $p(X,Y)$ *then* $Y \leftarrow \alpha(X,Y)$ *else* $Y \leftarrow \beta(X,Y)$. The loop construct $\alpha(p\beta\alpha)^{*}\bar{p}$, which is equivalent to $(\alpha p\beta)^{*}\alpha\bar{p}$, corresponds to *repeat* $Y \leftarrow \alpha(X,Y)$; $P \leftarrow p(X,Y)$; *if* P *then* $Y \leftarrow \beta(X,Y)$ *until not* P; where $P \notin X$, Y, or Z. P is an introduced flag or auxiliary variable. If $\alpha = \lambda$, the no-op function that leaves Y unchanged, then $(p\beta)^{*}\bar{p}$ corresponds to *while* $p(X,Y)$ *do* $Y \leftarrow \beta(X,Y)$. If $\beta = \lambda$, then $\alpha(p\alpha)^{*}\bar{p}$ corresponds to *repeat* $Y \leftarrow \alpha(X,Y)$ *until* $\bar{p}(X,Y)$.

A schema is structured if and only if its computation sequence set is an sre.

If $E(u,v)$ is an sre for some $u,v \in V$, and the arc (u,v) - if it exists - is not the only simple path from u to v, then the simple paths from u to v can be replaced by a new single arc $(u,v:E(u,v))$. When all such replacements have been made the schema is in *reduced form*. If a schema is structured then it follows that

its reduced form consists of the single arc $(S,H:\sigma)$, where σ is a structured regular expression describing the schema's computation sequence set.

Unless otherwise stated it is assumed throughout that schemas are always in reduced form.

Regular expression notation

As stated above, the loop construct is represented by the regular expression $\alpha(p\beta\alpha)^*\bar{p}$ or equivalently $(\alpha p\beta)^*\alpha\bar{p}$. Unfortunately both forms of regular expression each contain two instances of α, whereas the loop construct only contains one. To make the notation reflect more closely the schema being represented the convention is adopted that both regular expressions will be written as $(\alpha p\beta)^+\bar{p}$.

3. BASIC UNSTRUCTURED FORMS

There are six basic unstructured forms (*buf's*) that can occur in a schema. These are:

jump into a decision - ID;
jump out of a decision - OD;
jump into the advancing path of a loop - IL;
jump out of the advancing path of a loop - OL;
jump into the back latch of a loop - IB;
jump out of the back latch of a loop - OB

and are depicted in Figure 6. The last two, IB and OB, are not independent forms but are morphologically identical to IL and OL respectively, as shown in Figure 7. Referring back to Figure 6, there are three possible positions for vertex *E* in the schema: on a path from the start vertex to vertex *A*; on a path from vertex *C* to the halt vertex; or on a path from the start vertex to the halt vertex which does not include any of the vertices *A*, *B*, or *C*. Analysis of all possible placings for vertex *E* with respect to each of the six *buf's* shows that unstructuredness always occurs in possibly overlapping combinations of the six unstructured subschemas depicted and named in Figure 8, and that none of the *buf's* can ever occur alone.

To see that none of the *buf's* can occur alone consider the ID construct shown in Figure 6. Vertex *E* must be reachable from the start vertex *S*, say. Since *S* only has out-degree 1, there must be a decision vertex, *D* say, somewhere on the path [*S*,*H*], where *H* is the halt vertex, and such that there is a path [*D*,*E*]. If *D* is a predecessor of *A* then there are paths [*S*,*D*][*D*,*A*] and [*S*,*D*][*D*,*E*]. But now *D* is the entry of a decision subschema with paths [*D*,*A*][*A*,*B*] and [*D*,*E*][*E*,*B*], with a

jump out at *A* to *C*, that is, an OD. If *D* is a successor of *C* then *D* is the exit vertex and *B* the entry of a cycle [*B*,*D*][*D*,*E*][*E*,*B*] with a jump in at *C* from *A* - an IL construct. If *D* occurs on [*A*,*C*] then *A* is the entry and *B* the exit of a decision subschema with paths [*A*,*B*] and [*A*,*D*][*D*,*E*][*E*,*B*] with a jump out at *D* to *C* - an OD construct. Finally, if *D* occurs on either [*A*,*B*] or [*B*,*C*] then structured subschemas result and there is no ID construct present. Similar arguments can be advanced for the other 'jump-in' constructs IL and IB.

Turning to the OD construct, the halt vertex *H* must be reachable from *E* and therefore there must be a collector *K* somewhere on [*S*,*H*] such that there is a path [*E*,*K*]. By considering all possible placings of *K* with respect to the OD construct instances of OL and ID are found to occur with the OD. Similarly, the other 'jump-out' constructs OL and OB can be shown not to occur in isolation.

Examination of each of the six possible forms of unstructuredness depicted in Figure 8 reveals that they each comprise pairs of *buf's* taken from the set ID, OD, IL and OL. Thus:

DD = ID + OD; DL = ID + IL;
LD = OD + OL; LL = IL + OL;
BL = ID + OL; LB = OD + IL.

That this is so for the first four forms is clear from the Figure. To see the result for BL observe that BL certainly contains an instance of OB, which as shown in Figure 7 is equivalent to OL, whilst the ID component, in the notation of Figures 8 and 6, is found by setting vertex 3 to *A*, 1 to *B*, 2 to *C* and the immediate predecessor of the BL construct to *E*. For LB, the IB construct is equivalent to IL, and the OD component is found by setting 3 to *A*, 4 to *B*, 2 to *C* and the immediate successor of the LB construct to *E*.

From the foregoing it is now evident that it is sufficient to consider just ID, OD, IL and OL as the basic units of unstructuredness whose removal will result in a structured schema. In fact it is sufficient to consider just ID, OD, and IL; structuring all instances of ID removes all occurrences of DD, DL, and BL, and then structuring all OD's removes LD and LB constructs to leave just LL's. These can then be removed by structuring just the IL's, so consideration of the OL forms is not necessary. Since each of the basic forms occurs in combination with each of the other three, any one of the forms ID, OD, IL, OL can be the form neglected in the structuring process, but for reasons to be given later it is advantageous to omit OL.

Although the various *buf's* have been depicted with just one jump in or out, a decision or cycle subschema may have several such jumps on any or all of its constituent paths and therefore gives rise to multiple and possibly overlapping instances of basic unstructured forms. Each *buf* can be considered in isolation and, as is easily seen, can only occur paired with one of the other *buf's*. The problem of structuring single jump entry or exit *buf's* is considered in the next Section whilst that of multiple overlapping *buf's* is taken up in Section 5.

McCabe[21] considered the *buf's* here named as ID, OD, IL and OL, but did not consider IB and OB. (As already noted, these are in any case not independent forms but are equivalent to IL and OL respectively.) McCabe[21] and Williams[7] independently derived the unstructured forms described here as DD, DL, LD and LL, but neither recognized the forms BL and LB. With these forms now included it can be seen from the foregoing that each *buf* is paired with one of the other three - as might

be expected from symmetry - to produce a unique unstructured single-entry single-exit subschema. Clearly no *buf* can be paired with itself to produce such a subschema, so the six forms DD, ..., LB are the paradigms of all possible forms of unstructuredness. Williams added a fifth form to DD, ..., LL termed parallel loops, but as Williams also recognized, this form is expressible in terms of the other four under the restriction of a single exit vertex.

4. THE STRUCTURING TRANSFORMS

Two schemas having identical functions and predicates are computationally equivalent if their computation sequence sets are described by the same regular set. The first step in the structuring process is therefore to recast the regular expressions describing the buf's into sre formats. The strategy for transforming an unstructured schema into structured form is then as follows:

1. If the schema is structured then stop.
2. Identify a buf and replace it with a computationally equivalent but structured schema. (Since buf's cannot occur alone a second buf will also be removed.)
3. Go to step 1.

In this Section the structured equivalents of the buf's are derived. Where possible two alternative transforms are given: one which avoids predicate duplication but necessarily incurs function duplication, called here Type 0, and one which avoids function duplication but necessarily incurs predicate duplication, called Type 1. The 0 and 1 represent the number of predicate duplications in the transform. For ID and IL both types of transform exist, but for OD and OL it will be shown that only Type 1 is possible. Proofs that the structuring algorithms can always be applied and always terminate will be given in the next Section.

Consider first the ID subschema shown in Figure 9. It is required to find a structured regular expression for the end set expression $E(N) + E(E)$. From the Figure it can be seen that

$$E(N) = c.E(A)$$
$$E(A) = q.e.E(C) + \bar{q}.d.E(B)$$
$$E(E) = b.E(B)$$
$$E(B) = a.E(C)$$
$$E(C) = f.E(X) \qquad (1)$$

It is required to solve these equations in a form which is an sre expressed in terms of $E(X)$. Bainbridge[11] has given three rules for solving end set equations to yield sre's. Letting x, y denote sre's and p a predicate these rules are:

1. if $E(v) = x.E(u)$ or $E(v) = x$
   then eliminate $E(v)$ by substitution;
2. if $E(v) = p.x.E(u) + \bar{p}.y.E(u)$
   then deduce $E(v) = (p.x + \bar{p}.y).E(u)$;
3. if $E(u) = p.x.E(u) + \bar{p}.y.E(u)$ or
   if $E(u) = p.x.E(u) + \bar{p}.y$
   then deduce
   $E(u) = (p.x)^*.\bar{p}.y.E(u)$ or
   $E(u) = (p.x)^*.\bar{p}.y$ respectively.

Bainbridge asserts that if application of these rules yields an sre then the sre is minimal with respect to a count of the number of occurrences of functions and predicates, but if a stage is reached where none of the rules applies then there is no sre solution for the end set equations. Applying these rules to equations (1) for ID gives

$$E(N) = c.(q.e + \bar{q}.d.a).E(B)$$
$$E(E) = b.a.E(B)$$
$$E(B) = f.E(X) \qquad (2)$$

which is in the required sre format and has also eliminated vertices A and C as chain vertices in the structured subschema. The structuring transform has thus effectively deleted the entry and exit vertices of ID by placing them on structured arcs in the

transformed schema whilst retaining the topological relationship between the vertices originally incident on the ID. Unlike equations (1), equations (2) contain one duplication of function *a*. The corresponding subschema is shown in Figure 9 as ID-0, the 0 denoting that the decision entry vertex of the ID has not been duplicated. Structuring has been achieved at the expense of a single function duplication.

For the IL depicted in Figure 10 the end set equations are:

$$\begin{aligned} E(N) &= f.E(A) \\ E(A) &= a.E(B) \\ E(E) &= b.E(B) \\ E(B) &= c.E(C) \\ E(C) &= q.e.E(A) + \bar{q}.d.E(X) \end{aligned} \qquad (3)$$

Following the example of ID, the intent is to eliminate A and C as the cycle entry and exit vertices and to find sre's for $E(N)$ and $E(E)$ in terms of $E(B)$, and $E(B)$ in terms of $E(X)$. By substitution for $E(A)$ and then $E(B)$ in $E(C)$ is obtained†

$$\begin{aligned} E(C) &= q.e.a.c.E(C) + \bar{q}.d.E(X) \\ &= (q.e.a.c)^*.\bar{q}.d.E(X) \end{aligned}$$

hence

$$\begin{aligned} E(B) &= c.(q.e.a.c)^*.\bar{q}.d.E(X) \\ &= (c.q.e.a)^+.\bar{q}.d.E(X) \end{aligned}$$

and then

$$\begin{aligned} E(N) &= f.a.E(B) \\ E(E) &= b.E(B) \\ E(B) &= (c.q.e.a)^+.\bar{q}.d.E(X) \end{aligned} \qquad (4)$$

---

† Direct substitution in the equation for $E(B)$ gives a form on which Bainbridge's Rules cannot be used.

Again structuring has been achieved at the expense of one function duplication, namely a in $E(N)$ and in $E(B)$, but without duplication of the cycle exit vertex q. The resulting structured subschema is shown in Figure 10 as IL-0, where it can be seen that the topological relationship between vertices incident on IL has been preserved.

Next consider OD, Figure 11. In this case an sre for $E(N)$ is required in terms of $E(B)$, and for $E(B)$ in terms of both $E(X)$ and $E(E)$. As seen from the Figure the end set equations are:

$$\begin{aligned} E(N) &= a.E(A) \\ E(A) &= q.e.E(C) + \bar{q}.d.E(B) \\ E(B) &= p.b.E(E) + \bar{p}.c.E(C) \\ E(C) &= f.E(X) \end{aligned} \tag{5}$$

Substituting for $E(A)$ and $E(C)$ gives

$$\begin{aligned} E(N) &= a.(q.e.f.E(X) + \bar{q}.d.E(B)) \\ E(B) &= p.b.E(E) + \bar{p}.c.f.E(X) \end{aligned} \tag{6}$$

which is not in sre format and none of the Bainbridge Rules can be applied further.

To achieve sre format it is necessary to expand terms with the general format

$$r.f.E(U) + \bar{r}.g.E(V)$$

into the factored sre form

$$(r.f + \bar{r}.g).(r'.E(U) + \bar{r}'.E(V))$$

where r' is an introduced predicate whose computation yields the same Boolean value as that originally computed for r. Given that r' can be so computed, then being a predicate its computation will not alter the local or output variable sets Y and Z respectively of the schema. Now the only possible computation

sequences of the factored sre are $r.f.r'\mathrm{E}(U)$ and $\bar{r}.g.\bar{r}'.\mathrm{E}(V)$ which respectively produce the same sequences for the values of Y and Z as $r.f.\mathrm{E}(U)$ and $\bar{r}.g.\mathrm{E}(V)$. Thus the unfactored and factored expressions are computationally equivalent under the assumption concerning r'. Remembering that r is a predicate and f, g are local functions of the schema such that

$$r: \quad X \times Y \to \{true, false\}$$
$$f, g: X \times Y \to Y$$

it follows that r' cannot in general be replaced by a recomputation of r because the local functions f and g could have subsequently changed the arguments of r. Thus it is necessary to record for later use the value of r at its point of computation by the introduction of an auxiliary variable, or flag, which is distinct from the variables of the schema. To preserve computational equivalence the flag is introduced in the following way.

At a decision vertex the associated predicate p, say, is computed as before but its value is immediately assigned to a flag P, say, uniquely associated with p. It is this flag that is used, rather than the original predicate, as the discriminant in choosing the exit path from the decision vertex. In programming terms

if p then ... else ...

is replaced by

P ← p; if P then ... else ...

Whereas in the original schema the value of a predicate is known only at its point of computation, the introduction of a corresponding flag preserves the predicate's value until recomputed. In the sequel the convention is adopted that schema predicates will be denoted by p, q, r, ... and the corresponding uniquely

associated flags by P, Q, R $\cdots$. It will also prove convenient to allow direct assignment of truth values to the flags. Again, such assignments do not affect the original schema's computations. Bainbridge's Rules can now be extended to allow for the introduction of flags as follows, where 1 denotes *true*, 0 denotes *false*, and ϕ denotes the non-existent or null string:

4. if $E(u) = p.a.E(v) + \bar{p}.b.E(w)$ then introduce
   $E(u) = (P \leftarrow p).(P.a_1 + \bar{P}.b_1).$
   $(P.a_2.E(v) + \bar{P}.b_2.E(w))$
   where
   $a_1.a_2 = a$ and $b_1.b_2 = b$;
5. from $(P \leftarrow 1).(P.a + \bar{P}.b)$ deduce a, and from
   $(P \leftarrow 0).(P.a + \bar{P}.b)$ deduce b;
6. if x, y are sre's that do not contain assignments to P, then from
   $P.x.P.y$ deduce $P.x.y$,
   $P.x.\bar{P}.y$ " ϕ,
   $\bar{P}.x.\bar{P}.y$ " $\bar{P}.x.y$,
   $\bar{P}.x.P.y$ " ϕ;
7. from $(P + \bar{P}).E(u)$ deduce $E(u)$;
8. from $\phi.x$ deduce ϕ, and from
   $x.\phi$ deduce ϕ.

Returning to the end set equations (5), $E(B)$ can be recast in the form

$$E(B) = (P \leftarrow p).(P.b_1 + \bar{P}.c).(P.b_2.E(E) + \bar{P}.E(C))$$

where $b_1.b_2 = b$, $c_1 = c$, and $c_2 = \lambda$. In order to provide a common sre factor for $E(A)$ - and hence $E(N)$ - $E(C)$ can be expanded into the computationally equivalent form

$$(P \leftarrow 0).(P.b_2.E(E) + \bar{P}.E(C)),$$

which explains the choice of $c_2 = \lambda$ in the sre for $E(B)$. Now follows after some collection of terms

$$E(N) = a.E(A)$$
$$= a.[q.e.(P \leftarrow 0) + \bar{q}.d.(P \leftarrow p).(P.b_1 + \bar{P}.c)].E(B')$$
$$E(B') = P.b.E(E) + \bar{P}.f.E(X) \qquad (6')$$

This is the desired sre format but in terms of a new B' in place of the original B. The corresponding subschema is shown in Figure 11 as OD-1, with $b_2 = b$ and $b_1 = \lambda$. This choice is made because the new arc $(B',E;P.b)$ is then closely similar to the original jump-out arc $(B,E;p.b)$.

Structuring has removed the buf's entry and exit vertices, but in this case the resulting sre can be expressed only in terms of duplication of a predicate - that which provided the jump-out construct. The original jump-out decision vertex is incorporated in the new structured arc (N,B') as the entry to the construct $P + \bar{P}.c$ while the duplicated vertex B' takes over the role of tail to the original arc (B,E). Again, the topological relationship of the vertices incident upon the buf has been preserved.

Now consider OL, Figure 12. For this buf

$$E(N) = f.E(A)$$
$$E(A) = a.E(B)$$
$$E(B) = p.b.E(E) + \bar{p}.c.E(C)$$
$$E(C) = q.e.E(A) + \bar{q}.d.E(X) \qquad (7)$$

As for OD, it will be found that no sre solution can be formulated using function replication alone. Introducing flags for p and q, and after substituting for $E(C)$ and $E(B)$, $E(A)$ takes the form

$$a.(P \leftarrow p).(P.b.E(E) + \bar{P}.c.(Q \leftarrow q).(Q.e.E(A) + \bar{Q}.d.E(X))$$

The first disjunctive term can be extended to the form

$$P.(Q \leftarrow 0).\bar{Q}.b.E(E)$$

for reasons that will shortly become apparent.

Applying the expansion formula

$$P.u.x + \bar{P}.v.y = (P.u + \bar{P}.v).(P.x + \bar{P}.y)$$

gives

$$E(A) = a.(P \leftarrow p).(P.(Q \leftarrow 0) + \bar{P}.c.(Q \leftarrow q)).$$
$$[P.\bar{Q}.b.E(E) + \bar{P}.(Q.e.E(A) + \bar{Q}.d.E(X))]$$

which can be recast in the computationally equivalent form

$$E(A) = a.(P \leftarrow p).(P.(Q \leftarrow 0) + \bar{P}.c.(Q \leftarrow q)).$$
$$[Q.e.E(A) + \bar{Q}.(P.b.E(E) + \bar{P}.d.E(X))]$$

Defining

$$x = a.(P \leftarrow p).(P.(Q \leftarrow 0) + \bar{P}.c.(Q \leftarrow q))$$
$$E(B') = P.b.E(E) + \bar{P}.d.E(X)$$

gives

$$E(A) = x.(Q.e.E(A) + \bar{Q}.E(B'))$$
$$= x.E(A'), \text{ say.}$$

Thus

$$E(A') = Q.e.x.E(A') + \bar{Q}.E(B')$$
$$= (Q.e.x)^{*}.\bar{Q}.E(B')$$

from which

$$E(A) = x.(Q.e.x)^{*}.\bar{Q}.E(B')$$
$$= (x.Q.e)^{+}.\bar{Q}.E(B')$$

and finally

$$E(N) = f.(x.Q.e)^{+}.\bar{Q}.E(B')$$
$$E(B') = P.b.E(E) + \bar{P}.d.E(X) \qquad (8)$$

Thus structuring requires the duplication of the jump-out decision at B, with the original vertex now contained on the structured arc (N,B') and duplicated at B'. The resulting subschema is depicted as OL-1 in Figure 12. As for all other structured forms of the buf's the topological relationship of the incident vertices is preserved.

Whilst OD-1 and OL-1 each contain one duplicated decision vertex, neither contains a duplicated function. It is also possible to structure both ID and IL without function duplication at the expense of one duplicated decision vertex as shown by ID-1 and IL-1 in Figures 11 and 10 respectively. Thus each of the basic unstructured forms OD, ID, IL and OL can be structured at the expense of at most one duplicated decision vertex and no function duplication, but only ID and IL can be structured by function duplication alone.

As noted on Section 3, the six paradigms of unstructuredness are composed of pairs of buf's:

DD = ID + OD; DL = ID + IL; LD = OD + OL;
LL = IL + OL; BL = ID + OL; LB = OD + IL

and all of these except LD can be structured without predicate duplication by a suitable application of either ID-0 or IL-0. However, as LD consists of OD + OL it can only be structured at the expense of one introduced decision vertex using either OD-1 or OL-1.

It was remarked in Section 3 that of the four buf's it was only necessary to consider structuring transforms for three of them and the preferred three were ID, OD, and IL. The reason for this is now clear. Whereas the transforms for these three buf's require at most one flag each, that for OL requires two, and is in any case the most complex of all the transforms.

It remains to be established under what conditions, if any, the transforms developed in this Section can be applied in the presence of overlapping buf's. This is taken up in the next Section.

5. EFFECTIVENESS OF THE TRANSFORMS

A transform is considered to be effective only if it results in another valid schema and if it gives a reduction in the total number of buf's left in the schema.

In the previous Section it was assumed in the derivation of the structuring transforms that there was no overlap between the buf's. The effect of overlap is to introduce decision or collector vertices on what would otherwise be arcs of buf's, and then there is no guarantee that the structuring transforms can still be applied effectively. In this Section it is shown that whilst some forms of overlap can invalidate certain transforms, nonetheless for every schema there is always at least one transform that can be applied effectively, thus enabling every schema to be progressively transformed into structured format.

Consider ID, Figure 9, and the ID-1 transform. A collector on path $[A,B]$ of ID gives rise to a second instance of ID. Applying the ID-1 transform to the jump-in at B from E still effectively removes the first instance of ID but now leaves $[N,B]$ not as a structured arc but as an instance of ID on the path originally labelled $q.e$. A second application of ID-1 effectively removes this ID buf. By extension it is possible to effectively remove any number of overlapping ID's arising from collectors on the original path $[A,B]$. Similarly, ID-1 is effective with respect to the original arc (E,B) in the presence of any number of collectors on paths $[B,C]$ and $[A,C]$. Clearly ID-1 is also equally effective in the presence of any number of decisions on the original paths $[A,B]$, $[A,C]$ and $[B,C]$,

or indeed for any combination of decisions and collectors. Thus ID-1 can always be applied effectively to any ID buf.

ID-0 is also effective with respect to introduced vertices on the original arcs (A,B) and (A,C) of ID, but not for vertices on (B,C) because ID-0 duplicates this arc. However, consider an introduced collector B' on $[B,C]$ and let B' have an external immediate predecessor E'. The ID comprising vertices $AB'E'C$ can be regarded as the original ID having a collector B on its path $[A,B']$ for which as already noted ID-0 is effective. This argument can be extended to any number of collectors on $[B,C]$ of the original ID. In the presence of decision vertices on $[B,C]$ ID-0 is ineffective but, as already seen, ID-1 can be used instead. Hence

Lemma 1. There is always an effective transform for the removal of ID constructs from any schema.

Consider OD and OD-1, Figure 11. The presence of collector vertices on any path of OD introduces instances of ID which by Lemma 1 can always be removed effectively so it is sufficient to consider introduced decision vertices alone. Each such vertex introduces one additional instance of OD. Now the $[N,B]$ path of OD-1 contains two nested decision subschemas, namely $q.e.(P\leftarrow 0) + \bar{q}.d.(P\leftarrow p).(P + \bar{P}.c)$ and $P + \bar{P}.c$. Each decision vertex on paths $[A,B:q.e]$ and $[A,C:\bar{q}.d]$ therefore gives rise to one OD construct on the otherwise structured path $[N,B]$ of OD-1 whereas each such vertex on path $[B,C:\bar{p}.c]$ of OD necessarily produces two OD's in OD-1 - one for each decision subschema on $[N,B]$. Thus OD-1 is effective for decisions on paths $[A,B]$ and $[A,C]$ of OD, but not $[B,C]$. However, let B' be that decision on $[B,C]$ which has vertex C as an immediate successor, and let E' be its other immediate

successor. Subschema $AB'E'C$ is now an instance of OD with decision vertices on its $[A,B']$ path, for which as already noted OD-1 is effective. Hence:

Lemma 2. In the absence of ID constructs there is always an effective transform for the removal of OD constructs from any schema.

Summarizing thus far, ID-1 can always be applied on any ID such that the jump-in vertex is the most immediate successor of the ID subschema's entry vertex, and OD-1 can always be applied on any OD such that the jump-out vertex is the immediate predecessor of the OD's exit collector. Lemmas 1 and 2 together assert that it is always possible to transform an unstructured schema to one that contains no instances of ID or OD. Thus it is now necessary to consider schemas containing only IL and OL buf's. Since, as noted in Section 3, neither buf can occur in combination with itself, the only possible remaining unstructured form is IL + OL, and the effective removal of one component ensures the removal of the other.

Consider IL, Figure 10, and the transform IL-0. Since path $[B,X]$ of IL-0 contains a cycle with label $(c.q.e.a)^+.\bar{q}$ it follows that any vertex introduced on $[B,C:c]$ or $[C,A:q.e]$ of IL leaves IL-0 as an effective transform, with the introduced buf's being transferred to path $[B,X]$ of IL-0. However, IL-0 is not effective in respect of vertices introduced on path $[A,B:a]$ of IL due to the duplication of the path. The path $[B,X]$ of IL-1 contains a cycle $[(Q.a+\bar{Q}).c.(Q+q).e]^+.\bar{Q}$ with nested decision subschema $Q.a + \bar{Q}$. As for IL-0, IL-1 remains effective in the presence of introduced vertices on $[B,C]$ and $[C,A]$ in IL, but not for $[A,B:a]$ which due to the nested decision cause two buf's in IL-1 - one for the cycle and one for the nested decision subschema.

However, as will now be shown, for every schema comprising LL forms alone there is at least one IL for which there is no vertex on the path $[A,B:a]$. Since by assumption all ID and OD forms have been removed there can be no vertices on path $[C,A]$ of IL because these imply the presence of LB's and BL's which in the absence of OD's and ID's respectively cannot exist. Thus all decisions are cycle exits and all collectors are cycle entries with the result that all vertices necessarily lie on the forward path $[s,h]$. Let d be that decision vertex which is the most immediate successor decision of the start vertex s, and let (d,c) be the corresponding back latch. The cycle $[c,d](d,c)$ can comprise IL constructs only, since by definition $[c,d]$ is decision-free apart from d.

Consider the IL constructs of this cycle and let B' be that collector on $[c,d]$ which has c as one of its immediate predecessors and E', say, as its other. Now $cB'E'd$ can be regarded as an IL buf with no vertices on its path $[A,B']$, that is, on (c,B') and so can be structured using either IL-0 or IL-1. The argument can be repeated on the resulting schema, hence:

Lemma 3. *In the absence of ID and OD constructs there is always an effective transform for the removal of at least one IL construct from any schema.*

Lemmas 1–3 taken together lead to the principal result of this thesis:

Theorem. *It is always possible to put an unstructured schema into a computationally equivalent structured form using only the transforms ID-0, ID-1, OD-1, IL-0 and IL-1. If function duplication is not required then ID-1, OD-1 and IL-1 comprise the minimum set of transforms necessary.*

A note on λ-paths.

λ-paths may be present in the original schema, or may arise during the structuring process in the presence of overlapping buf's. The presence of a λ-path in a buf may enable simplification of the equivalent structured form by eliminating the need for a duplicate function, a duplicate test on a flag, or even the flag itself.

For instance, consider ID, Figure 9. If $a = \lambda$ then ID-0 becomes simply:

$$E(N) = c.(q.e + \bar{q}.d).E(B)$$
$$E(E) = b.E(B)$$
$$E(B) = f.E(X)$$

with no function or predicate duplication necessary, and no need for any flags. ID-1 is redundant in this case.

Next consider IL, Figure 10. Setting $a = \lambda$ makes IL-0

$$E(N) = f.E(B)$$
$$E(E) = b.E(B)$$
$$E(B) = (c.q.e)^{+}.\bar{q}.d$$

where again there is no predicate or function duplication, or any introduced flag. IL-1 is redundant.

Finally consider OD, Figure 11. Setting $c = \lambda$ makes the nested decision $(P + \bar{P}.c)$ superfluous to yield

$$E(N) = a.(q.e.(P \leftarrow 0) + \bar{q}.(P \leftarrow p)).$$
$$(P.b.E(E) + \bar{P}.f.E(X))$$

In this case the flag P must be retained, because although the original jump-out predicate p has now dropped out of the decision subschema due to the presence of the λ-path, nonetheless p must not be

computed after the preceding computation of q has returned 'true'.

It is assumed throughout the rest of this thesis that all such simplifications due to the presence of λ-paths are duly made, and done so usually without further reference.

6. IDENTIFYING BASIC UNSTRUCTURED FORMS

Throughout the development of the structuring transforms in the earlier Sections it was assumed that it was always possible to identify the basic unstructured forms and, in particular, to identify them in such a way that structuring could be carried out in the prescribed order of ID's, OD's and IL's. In this Section algorithms are developed to justify that assumption.

Although the recognition of buf's seems straightforward when looking at carefully drawn schemas, their recognition is less obvious when one is faced with the description of the schema in the form of, say, an adjacency list of vertices. Thus some algorithmic technique is required. The first requirement in any algorithmic method is to partition the vertices (other than *start* and *halt*) into four equivalence classes: the entry vertices of decision subschemas, the exits of decisions, the entries of cycles, and the exits of cycles. Given these partitions, decision subschemas are identified by finding for each entry decision the set of exit collectors such that there are exactly two disjoint simple paths from the entry to the exit collectors. Each such decision - collector pair, together with the two associated simple paths, constitutes a decision subschema. Cycle subschemas can be identified in an analogous manner.

However, it is not possible in general to partition all vertices uniquely into the required equivalence classes. Consider, for instance, the schemas depicted in Figure 4b. Partitioning the vertices of schema *A* into decision entries, decision exits, cycle entries and cycle exits gives respectively {*a*,*e*}, {*b*,*f*}, {*d*},

{c}, whilst for the schema *B* the respective partitions are {a,e}, {d,f}, {b}, {e} - yet the schemas are identical having the same vertex set *V* and arc function Γ. The ambiguity surrounding vertices *b*, *c*, *d* and *e* arises as follows. There are two forward paths in the schema (as depicted by the vertically drawn paths in forms *A* and *B*) with back latches (*c*,*d*) and (*e*,*b*) respectively. Both back latches lie on the only cycle in the schema, namely *d*-*e*-*b*-*c*-*d*, and the removal of either leaves the schema acyclic. Treating {(*c*,*d*)} as the back arc set suggests the partition associated with schema *A*, whereas selecting {(*e*,*b*)} suggests that for schema *B*. Either group of partitions is equally acceptable for the purposes of applying the structuring transforms, although some particular interpretation of the schema may make one group preferred over the other.

Since there can be no ambiguity in an acyclic schema the foregoing example shows that the first step in identifying buf's must be to determine a back arc set for the schema, whose deletion would leave the schema acyclic. Having found such a set, the heads of the back arcs are designated cycle entry collectors and all other collectors of the schema as decision exits. Partitioning the decision vertices is then straightforward, as will be shown later in this Section.

Defining back arc sets

One way to define back arcs of a schema $G = (V,E)$ might be by means of a depth first traversal of the schema, beginning at the start vertex. Let $T = (V,E_T)$, $E_T \subseteq E$, (or equally $T = (V,\Gamma_T)$) be a depth first spanning tree of *G* and let the vertices of *T* (and hence of *G*) be numbered upwards in pre-order (number the root, number the left subtree, number the right subtree) with vertex *u* being numbered *N*(*u*), say. The arcs of *G* can

UNIVERSITY COLLEGE CORK Library

now be partitioned into four equivalence classes:

tree arcs = $\{(u,v) \mid (u,v) \in E_T\}$,

forward arcs = $\{(u,v) \mid N(u) < N(v)$ and $(u,v) \notin E_T\}$,

cross arcs = $\{(u,v) \mid N(u) > N(v)$ and $u \notin \Gamma_T^* v\}$,

back arcs = $\{(u,v) \mid N(u) > N(v)$ and $u \in \Gamma_T^* v\}$.

Back arcs can be distinguished from cross arcs by renumbering the vertices of T in reverse pre-order (number the root, number the right subtree, number the left subtree) with vertex u numbered $R(u)$, say, to yield $R(u) < R(v)$ for cross arcs but $R(u) > R(v)$ for back arcs.

Unfortunately this straightforward method of defining back arcs can cause failure to recognize even the simplest of decision subschemas. Consider for instance the schema of Figure 4c, where the vertices have been numbered in a possible depth first search order. This numbering yields (4,2) as the back arc and then 1-4 and 1-2-3-4 as the two paths of a rather improbable decision subschema, instead of the much simpler decision paths 1-2 and 1-4-2, with back arc (3,4). The intuitive preference for the simpler decision subschema is justified by the fact that it lies wholly on forward paths of the schema, whereas the alternative one does not. The reason for the failure of the depth first search to find the back arc (3,4) was that it did not fully traverse all forward paths before exploring other paths.

A suitable definition of back arcs in terms of forward paths is the following. Let F be a forward path of a schema G, and let L be the set of back latches of F. Each back latch is by definition associated with a distinct cycle in G, and no two back latches lie on

one cycle and no other. For suppose that (u,v) and (w,x) are two back latches lying on the same cycle but no other. Then this cycle is composed of paths $[u,v]$, $[v,w]$, $[w,x]$, $[x,u]$. But v,x both lie on F so either $[v,x]$ or $[x,v]$ is also a path. Suppose the former, then there exists a distinct cycle $[u,v]$, $[v,x]$, $[x,u]$ which does not include backlatch (w,x), contrary to assumption. Similarly if $[x,v]$ is a path then there is a cycle $[w,w]$ which does not include (u,v).

Each backlatch of F is therefore in a back arc set B of G. The remaining members of B are defined recursively by considering the not necessarily connected subschema $G \sim F$, regarding F as a schema. Since, as shown above, back latches of different forward paths can lie on a single cycle, and can be part of some other forward path, it is evident that B will depend upon the order in which forward paths are found.

Finding forward paths

A suitable goal-oriented depth first search algorithm for finding forward paths is the following. Starting from some initial vertex u, a depth first search is carried out until either a halt vertex h is found, in which case the depth first path $[u,h]$ is accepted as an elementary path from u to h, or no further progress is possible. To improve computational efficiency, the set of goal vertices is extended from the halt set to include all those vertices already found to lie on some elementary path from u to h.

Specifically, let $F = (V_F, E_F)$, or equivalently let $F = (V_F, \Gamma_F)$, where $F \leq G$, and let $H \leq V_F$ be the set of halt vertices in G. It is required to construct F from the given H such that F is a maximal acyclic forward path subschema of G. Let U, $U \leq V-H$, denote the set of

vertices whose membership of F is yet to be determined. The required forward path function *dfs* is defined by

$$dfs: \quad V \to 2^V \times 2^E$$

where

$dfs(u) \to U - \{u\};$
  for all $x.x \in \Gamma u \cap U$ *do*
    $dfs(x);$
  for all $x.x \in (\Gamma u - \{u\}) \cap V_F$ *do*
    $(V_F, E_F) \leftarrow (V_F + \{u\},\ E_F + \{(u,x)\});$

It is shown in Appendix 1 that, in the modified notation of Hoare's[29] axiomatics,

$$\{u \in U \wedge I\} \quad dfs(u) \quad \{\Gamma^* u \cap U = \emptyset \wedge I\}$$

where

$$I: \quad I_0 \wedge I_1$$
$$I_0: \ (U \subset V) \wedge (V_F \subseteq V) \wedge (H \subseteq V_F) \wedge (U \cap V_F = \emptyset)$$
$$I_1: \ (\forall w,x,y.w,x,y \in V).$$
$$[(x,y \in V_F \Rightarrow x,y \in \Gamma_F^{-*} H) \wedge (y \in \Gamma_F x \Rightarrow x \notin \Gamma_F^* y)$$
$$\wedge (w \notin V_F \Rightarrow w \notin \Gamma_F x)]$$

Invariant I guarantees that F is acyclic, while the condition $\Gamma^* u \cap U = \emptyset$ guarantees that all paths from u have been considered and hence that F is maximal with respect to u.

It now follows that under the pre-condition

$$\{(s \in U) \wedge I \wedge (V_F = H) \wedge (E_F = \emptyset)\}$$

$dfs(s)$ yields

$$\{\Gamma^* s \cap U = \emptyset \wedge I\}$$

from which is easily deduced

$$\{s \in V_F \Rightarrow s \in \Gamma_F^{-*} H\}$$

with $F = (V_F, E_F)$ being a maximal forward path subschema

of G. In general F will depend on the order of selection of the vertices in $dfs(s)$.

If G contains a set of start vertices S then all forward paths are found from

$$\begin{aligned} &\textit{for all } s.s \in S \textit{ do} \\ &\quad dfs(s). \end{aligned}$$

The correctness of this is easily seen by introducing a new vertex $t \notin V$ and extending Γ to include $S = \Gamma t$. Then $dfs(t)$ over $V \cup \{t\}$ yields $F' = (V'_F, E'_F)$, say, from which $F = (V'_F - \{t\}, E'_F - \{(t, \Gamma t)\})$.

Finally the back latch set L of F is given by

$$L = \{(u,v) \mid \quad v \in V_F \wedge (u,v) \notin E_F\}$$

which is a subset of the back arc set B of G. As noted earlier, B is found by recursively applying the dfs algorithm to $G \sim F$.

Identifying decision subschemas

To identify decision subschemas in a schema $G = (V,E)$ first find the back arc set B of G as described above and then define four sets of vertices

C: the set of collectors in V;
D: the set of decisions in V;
E: $\{v \mid (u,v) \in B\}$
X: $\{u \mid (u,v) \in B \wedge u \in D\}$

E is the set of cycle entry collectors and X the set of cycle exit decisions. Whilst the head vertex of every back arc is necessarily a collector, not every tail is necessarily a decision - see Figure 4a for instance, where (e,a) is a back arc and e is a collector. Thus it remains to find the complete set of cycle exit decisions as well as identifying decision subschemas.

For the purposes of identifying basic unstructured forms it is required to find *simple decision subschemas* where by *simple* is meant that the subschema properly contains no other with the same entry. An algorithm will now be given to determine for each vertex $d \in D$ whether or not d is a decision entry vertex, and if so, simple decision subschemas for which it is the entry vertex. Applying the algorithm to each element of D results in finding all simple decision subschemas, and the unique partitioning of all decision and collector vertices into the four classes: decision entries, decision exits, cycle entries and cycle exits. The algorithm does not, however, identify simple cycles.

Algorithm 6.1. To find for each decision vertex of a schema G whether or not it is a decision entry vertex and if so the simple decision subschemas for which it is the entry.

Input. The advancing path subschema G_A of a schema G, and the sets C,D,E,X: being respectively the collectors, decisions, cycle entries and cycle exits of G.

Output. For each vertex d, $d \in D$, a list L_d of topologically ordered sets, each set comprising the vertices of a simple decision subschema with d as its entry.

Method.

0. Define $U = D - X$.
1. *If* $U = \emptyset$ *then* stop.
2. Choose d, $d \in U$, and define $\Gamma_A d = \{x,y\}$.
   $L_d \leftarrow \emptyset$; $U \leftarrow U - \{d\}$;
   Define $R_d = (\Gamma_A^* x \cap \Gamma_A^* y) \cap (C - E)$ and let R_d be topologically ordered. [R_d comprises the collectors reachable from d by two or more paths.]
3. *If* $R_d = \emptyset$ *then*
   begin $X \leftarrow X \cup \{d\}$; *goto* step 1 *end.*
4. Let $c = first\ (R_d)$, where *first* (S) returns the first element of ordered set S.

$R_d \leftarrow R_d - \Gamma_A^* c$.

$V_d \leftarrow \phi$. [V_d is a set used to hold vertices of a simple decision subschema.]

Define $F = \{c\} \cup (\Gamma_A^* d - \Gamma_A^* c)$ and let V_d, F be topologically ordered.

5. *If* $F = \phi$ *then goto* step 7.
6. Let u = *first* (F).

   If $c \in \Gamma_A^* u$ *then*

   begin $V_d \leftarrow V_d \cup \{u\}$; $F \leftarrow F - \{u\}$ *end*

   else

   $F \leftarrow F - \Gamma_A^* u$.

   goto step 5.
7. $L_d \leftarrow L_d, V_d$.

   If $R_d = \phi$ *then goto* step 1 *else goto* step 4.

Notes.

1. R_d contains all decision exit collectors reachable from d by two or more distinct paths. If R_d is empty then there are no such collectors and d must therefore be a cycle exit.
2. Since R_d is topologically ordered, [d, *first*(R_d)] must be a simple decision subschema. For suppose the contrary, then there exists some collector u, say, which lies on either [x, *first*(R_d)] or on [y, *first*(R_d)] where $\Gamma_A d = \{x,y\}$. In either case u must topologically precede *first*(R_d) which is in contradiction to the definition of *first*.
3. The preceding argument explains why $\Gamma_A^* c$ is subtracted from R_d in Step 4 rather than just c.
4. In Step 6, if u, $u \in F$ cannot reach c, then neither can any vertex reached by u, hence $\Gamma_A^* u$ rather than u is subtracted from F in this case.

As suggested by the above algorithm, a decision entry may have more than one simple decision subschema associated with it. As an example consider the schema of Figure 5 where decision d_0 has the simple decision

subschemas (d_0,d_1,d_2,c_1) and (d_0,d_1,d_2,c_2). Conversely there may also be a collector associated with two overlapping subschemas. Again referring to Figure 5, c_3 has (d_1,c_1,c_2,c_3) and (d_2,c_1,c_2,c_3) as overlapping simple decision subschemas.

It is to be noted that whilst a simple decision does not properly contain another with the *same* entry vertex it may properly contain another with a *different* entry. Simple decisions that do not properly contain others are called *basic decision subschemas*, and it is these that comprise the basic unstructured forms ID and OD. It is not necessary, however, to find all basic decisions in the forward path subschema before commencing structuring. It is sufficient at each stage merely to identify the topologically last basic decision and then structure it, repeating the process until all basic decisions have been structured.

Identification of the last basic decision is straightforward. Let U be a set of decision entry vertices in the forward path subschema arranged in topological order, and let u, $u \in U$ be the last decision vertex of U. Then each simple decision subschema with entry u is also a basic decision. The proof is immediate: suppose that V_u is a simple decision with u as its entry but which is not basic. Then V_u must contain a decision entry vertex which is topologically later in U than u, contrary to the assumption that u was the last.□ Any of the simple decisions of u can be chosen as the last basic decision, but in view of the desire to structure IDs first, one with no cycle exits (if it exists) would be chosen preferentially.

Identifying cycle subschemas

To identify cycle subschemas of a schema G for the purposes of structuring, the decision subschemas on the augmented advancing path subschema $\hat{G}$ of G are progressively detected and removed by structuring as described above. Since $\hat{G}$ is acyclic, after the removal of all decision subschemas $\hat{G}$ will be reduced to the form of a tree, $\hat{T}$ say. Reinstating the back arcs of G onto $\hat{T}$ restores the cycles of G, to give a schema R, say.

These cycles may be in structured form or in the form of possibly overlapping instances of LL (IL + OL), and BL (ID + IL). It might be expected that LB (OD + IL) could occur as well, but in fact it cannot because both of the constituent paths of the OD component of any LB construct in G are necessarily contained in the advancing path subschema $\hat{G}$ of G, and therefore the OD is removed before R is constructed. The ID component of BL on the other hand has a back arc as the first arc on each of its constituent paths, and so cannot occur in $\hat{G}$.

Detection of each structured cycle is straightforward. If (u,v) is a back arc then, recalling that R is in reduced form, if (v,u) is an arc of R then u-v-u is a structured cycle, and can be replaced by a single arc, making u and v chain vertices, which are then elided to preserve reduced form.

The unstructured cycles of R are now found as follows. Let (u,v) be a back arc of G in R, then the elementary path $[v,u]$ in R together with (u,v) constitutes a cycle. By analogy with decision subschemas, a *simple* cycle is defined to be one which does not properly contain another having the same exit decision, whilst a *basic*

cycle is one that does not properly contain another with a different exit decision. However, before identifying IL constructs for the purposes of structuring it is first necessary to identify and remove the ID constructs of BL. This, and the final identification and removal of the IL constructs, is taken up in the next Section.

7. THE STRUCTURING PROCESS

General Strategy

The first step is to identify a back arc set of the schema G, and then consider the augmented advancing path subschema $\hat{G}$ derived from G by cutting the back arcs and introducing new vertices at the cut ends of the arcs. Since by definition $\hat{G}$ is acyclic, it comprises at most the ID and OD bufs. To remove these, first identify the topologically last basic decision in $\hat{G}$. (Where a choice among such decisions exists, choose the one with the least number of cycle exits.) Structure this decision to produce $\hat{G}_1$, say, and repeat the structuring process on $\hat{G}_1$ to produce a sequence of progressively more structured schemas, until all ID and OD constructs have been removed, that is, until all decision entry vertices have been reduced to chain vertices. At each stage any chain vertices produced are elided to keep the schemas in reduced form. Since cycle exit decisions are not decision entries, no cycle exit will be elided in the structuring process.

The resultant schema is in the form of a tree, $\hat{T}$ say. Disregarding the cut back-arcs and their introduced end vertices yields the structured advancing-path tree, T say, of G whose vertices (apart from *start* and *halt*) comprise only the cycle entries and exits of G. Each leaf of T (other than *halt*) is necessarily a cycle exit decision, and by a process to be described below can be pruned to yield a tree, T_1 say. T_1 has the same property as T regarding its leaves, and so the pruning process can continue until a tree in the form of a single non-branching path is obtained. At this stage any remaining IL and OL constructs can be removed by identifying and removing the ILs, to yield the final

structured form of G.

The structuring method.

To illustrate the various steps in the structuring algorithm a detailed example of a hypothetical schema designed for the purpose will be worked out. The application of the structuring algorithm to some practical problems is done in the next Section.

Consider the schema G shown in Figure 13 for which $S = \{0\}$, $H = \{10\}$, and $V = \{v \mid 0 \leq v \leq 21\}$. The desired forward path $G_F = (V_F, E_F)$ of G with respect to start vertex 0 and halt vertex 10 is initialised by setting $V_F = H$, $E_F = \phi$, and the set of unconsidered vertices $U = V - V_F$. Clearly the invariant

$$I_0: \quad (U \subset V) \wedge (V_F \subseteq V) \wedge (U \cap V_F = \phi) \wedge (H \subseteq V_F)$$

is satisfied, as is also in a vacuuous manner invariant

$$I_1: \quad (\forall x, y . x, y \in V_F).[(x, y \in \Gamma_F^{-*} H) \wedge (y \in \Gamma_F x \; x \notin \Gamma_F^{*} y)]$$

the latter being the condition for G_F to be acyclic. The precondition for $dfs(0)$ to be executed, namely $I \wedge (0 \in U)$, is met so after $dfs(0)$ terminates G_F will be as depicted in Figure 14, omitting the cut back arcs and their introduced end vertices. Arcs not in G_F but incident on it such that the head vertices are in V_F are the back latches of G_F and are therefore back arcs of G. These are: (6,1), (19,2), (21,11), and (15,12).

The next step is to form $G_1 = G \sim G_F$, and if this is non-empty to repeat the process of finding forward paths and their associated back latches. For the present example G_1 is not empty and is as depicted in Figure 15, from which it can be seen that G_1 comprises two disjoint subschemas collectively having $S_1 = \{5, 9, 13\}$ and $H_1 = \{1, 2, 12, 13\}$. Applying dfs over S_1

with $G_{F1} = (H_1, \phi)$, $V_{F1} = H_1$ and $U = V_1 - V_{F1}$ yields the forward path G_{F1} for G_F, shown in Figure 16. G_{F1} has two disjoint components but only the larger has back latches, these being (20,14) and (20,17). Repeating the process to form $G_2 = G_1 \sim G_{F1}$ yields a subschema which has no back latches, so the search for the back arcs of G ends. The back latches of G_F and G_{F1} together comprise the back arc set of G.

The next step in the structuring algorithm is to construct the augmented path subschema $\hat{G}$ of G by cutting the back arcs and introducing new vertices at their ends. $\hat{G}$ is shown in Figure 17. Omitting the cut back arcs gives the advancing path subschema G_A from which it is required to identify and structure the topologically last decision subschema in $\hat{G}$, repeating the process until none remains.

One possible topological ordering of the vertices of G_A is:

0 1 2 3 4 5 7 11 12 13 8 9 6 10
14 15 16 17 21 18 19 20.

In the notation of the previous Section, the back arc, collector, decision, cycle entry and cycle exit sets are respectively

B: {(6,1), (19,2), (21,11), (15,12), (20,14), (20,17)}
C: {1, 2, 7, 11, 12, 8, 6, 14, 17, 18}
D: {3, 4, 5, 13, 9, 15, 16, 21, 19, 20}
E: {1, 2, 11, 12, 14, 17}
X: {15, 21, 19, 20}.

With six cycle entries but only four cycle exits identified it is clear that two further cycle exits remain to be found.

Applying Algorithm 6.1 gives

0. $U \leftarrow D - X$
   $= \{3, 4, 5, 13, 9, 16\}$
1. $U \neq \emptyset$ hence
2. Choose d as 16, the last decision in U. Then
   $U \leftarrow U - \{d\}$
   $= \{3, 4, 5, 13, 9\}$
   $L_{16} \leftarrow \emptyset$ and
   $\Gamma_A 16 = \{17, 21\}$ so that
   $R_{16} \leftarrow (\Gamma_A^* 17 \cap \Gamma_A^* 21) \cap (C - E)$
   $= (\{18, 19, 20\}) \cap (\{7, 8, 18\})$
   $= \{18\}$.

   Thus 18 is identified as the only decision exit corresponding to decision entry 16. To find the vertices of the decision subschema:
3. $R_{16} \neq \emptyset$ so
4. $c \leftarrow 18$
   $R_{16} \leftarrow R_{16} - \Gamma_A^* 18$
   $= \emptyset$
   $V_{16} \leftarrow \emptyset$
   $F \leftarrow \{c\} \cup \{\Gamma_A^* d - \Gamma_A^* c\}$
   $= \{16, 17, 21, 18\}$

   from which repeated application of
5,
6. eventually gives
   $V_{16} = \{16, 17, 21, 18\}$
   the vertex set of a topologically last decision subschema headed by 16.
7. $L_{16} \leftarrow V_{16}$ and as
   $R_{16} = \emptyset$ there are no other decision subschemas headed by 16.

Since 17 is a collector and 21 a decision the decision subschema comprises one ID and one OD construct. As shown in Section 5, structuring effectively deletes the entry and exit vertices 16 and 18 respectively to

leave G_A in the form shown in Figure 18, in part.

Returning to Algorithm 6.1, the next decision to be considered is 9 so that

2. $U \leftarrow \{3, 4, 5, 13\}$
   $L_9 \leftarrow \emptyset$ and
   $\Gamma_A 9 = \{6, 10\}$ so that
   $R_9 \leftarrow (\{6\} \cap \{10\}) \cap (\{7, 8, 18\})$
3. $R_9 = \emptyset$ and hence
   $X \leftarrow \{9, 15, 21, 19, 20\}$.
   Thus vertex 9 has been identified as a cycle exit.

Continuing with Algorithm 6.1, 13 is likewise found to be a cycle exit, so

$X \leftarrow \{13, 9, 15, 21, 19, 20\}$.

Next, decision vertex 5 is found to have 6 as its only collector, the paths of the decision being (5,6) and 5-7-8-9-6. Structuring the decision, which comprises two ID's and one OD, effectively removes 5 and 6 to yield the schema partially depicted in Figure 19. The next decision vertex, 4, heads a structured decision [4,7] and so 4 and 7 are elided as chain vertices following replacement of [4,7] by a single arc. Finally, vertex 3 is found to be a decision entry with paths (3,8) and 3-11-12-13-8, which after structuring the ID's at 11 and 12 and then the OD at 13 gives the schema depicted in Figure 20, in part. The final form of G_A is thus a tree, T say, which after adding the cut back arcs and their introduced end vertices gives the schema $\hat{T}$ depicted in Figure 21.

Referring to Figure 21, it can be seen that vertex 13 is the only branch vertex of T whilst vertex 9 is the tail of the only back latch (9,1) of the forward path of T. Vertices 13 and 9 are precisely those which were added to the cycle exit set X in Step 3 of Algorithm 6.1. It is easy to see that for any schema G a deci-

sion vertex d is added to X only if it is not the head vertex of a decision subschema and is not the tail of a back latch of G. Thus all such vertices d are necessarily branch vertices of the derived structured tree T of the advancing path subschema G_A of G. In the present example, vertex 9 is a branch vertex for which one of its subschemas (the arc 9-6) has been elided in the structuring process, leaving 9 as the tail of the derived back latch (9,1) which has subsumed the original arcs (9,6) and (6,1).

It remains to explain how to structure G_A following the reinstatement of the back arcs, some of which as noted already may have been extended by the elision of their tails during the structuring of G_A.

Let R denote the schema obtained by reinstating the back arcs of G on the structured form of G_A. If R has a tree T as its advancing path subschema rather than just the forward path $[s,h]$, then instances of BL, and hence ID, are present in R. If d is a leaf vertex of T other than the halt vertex then d is necessarily a decision vertex and therefore the head of two back arcs (d,b_1) and (d,b_2), say. (If d were a leaf collector of T it would be the exit of some decision subschema in G_A contrary to the fact that all such subschemas have been removed from G_A.) Both b_1 and b_2 must lie on the path $[s,d]$ (otherwise they would not have been treated as back arcs by the forward path algorithm) with b_1 say such that $[b_1,b_2]$ is an elementary path in T. $(d,b_1)[b_1,b_2]$ is one path of a decision subschema D and (d,b_2) the other, with an ID component at b_1. The path $[b_1,b_2]$ may itself comprise both decision and collector vertices so D contains the one ID construct at b_1 and possibly several ID and OD constructs over the vertices between b_1 and b_2 on $[b_1,b_2]$.

Structuring D eliminates d and b_2 together with (d,b_2) from R and replaces (d,b_1) by an arc (p,b_1) where $d \in \Gamma p$. Vertex p is never the same one as b_2 since if it were then b_2-d-b_2 would be a structured cycle and therefore would have been replaced earlier by a single arc in R. Let R_1 be the resultant schema with advancing path tree T_1. If p is not on the forward path of T_1 it is a decision vertex leaf of T_1 and as such is the exit decision of two nested cycles. The inner cycle may or may not be structured. If it is, then the outer cycle surely may not be structured also.† Figure 22 illustrates the possibilities. After eliding any structured cycles associated with p, a new schema R' say is obtained for which the process of pruning decision leaves from the underlying advancing path tree T' can be repeated until all such leaves have been elided. What remains is a schema in the form of a single forward path $[s,h]$ together with back arcs. Since there is now only a single forward path there can be no remaining instances of ID and hence BL, thus only IL and OL constructs remain to be structured.

The IL constructs are easily identified and structured as follows. Let the vertices on the forward path $[s,h]$ of R' be numbered in ascending order - the topological numbering given to the vertices for Algorithm 6.1 will suffice - with vertex v being numbered $N(v)$, and let D be the ordered set of decision vertices so numbered. Choose a vertex $d \in D$ such that d is the lowest numbered vertex of D, and let $c \in \Gamma_R, d$ such that $N(c) < N(d)$. Then the cycle $(d,c)[c,d]$ contains IL constructs alone. For suppose the contrary, then there must be a decision vertex d' on $[c,d]$ and hence

---

† Because p would lie on a cycle from which the halt vertex h was unreachable.

such that $N(d') < N(d)$, contradicting the fact that d is the lowest numbered vertex in D. Structuring the cycle effectively deletes d and c from R', and the structuring process can then be repeated until R' comprises just the derived structured arc (s,h).

Returning to the example, let R be the schema derived from G_A by reinstating the back arcs, then since G_A is a tree there are instances of BL in R. Figure 23(i) shows G_A with the back arcs from the leaf vertex 20 restored. Vertex 14 corresponds to b_1 and 17 to b_2. Since decision vertex 15 lies on [14,17], the decision subschema headed by 20 comprises one instance of ID at 14 and one of OD at 15. Structuring the decision removes 20 and 17. Figures 23(ii) to (v) show the progressive steps in pruning the leaves of the advancing path tree of R. In this instance two nested structured cycles 2-13-2 and 1-2-13-9-1 are left after removing all ID constructs, so the schema when put into reduced form is structured. The steps sketched in Figure 23 are set out fully in Figures 24 to 28.

The *sre* for the original schema G is thus

$$a[b(X,W,U\leftarrow 0)(\theta\zeta T\varepsilon\kappa)^{+}\bar{T}\gamma\bar{S}f]^{+}S$$

where

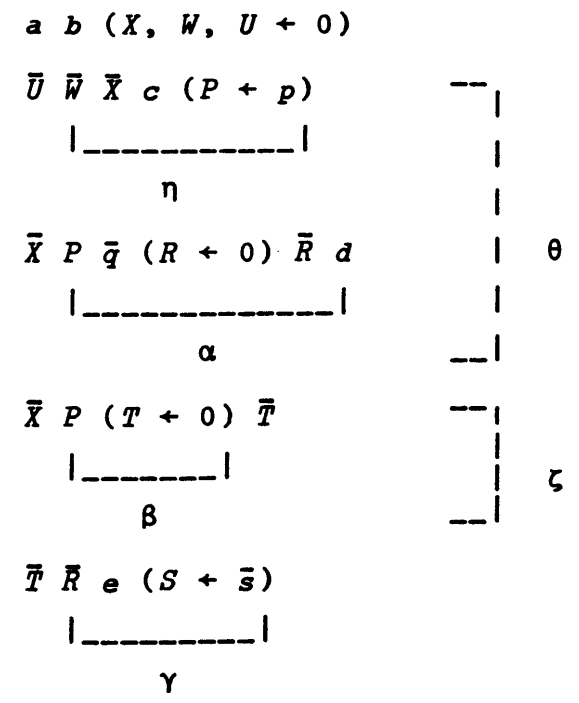
$$\kappa = U+\bar{U}\delta[W+\bar{W}k(X\leftarrow x)]$$
$$\theta = U(P\leftarrow 0)+\bar{U}\eta(X+\bar{X}\alpha)$$
$$\eta = W(P,X\leftarrow 0)+\bar{W}[X+\bar{X}c(P\leftarrow p)]$$
$$\zeta = X(Y\leftarrow y)(\bar{T}\leftarrow 0)+\bar{X}\beta[\bar{T}+T(Y\leftarrow 0)]$$
$$\varepsilon = Y(V,U\leftarrow 0)+\bar{Y}i(U\leftarrow\bar{u})[U+\bar{U}(V\leftarrow\bar{v})]$$
$$\delta = V(W\leftarrow\bar{w})+\bar{V}j(W\leftarrow 0)$$
$$\gamma = R(S\leftarrow 0)+\bar{R}e(S\leftarrow\bar{s})$$
$$\beta = P(T\leftarrow 0)+\bar{P}h(T\leftarrow\bar{t})$$
$$\alpha = P[q(R\leftarrow r)+\bar{q}(R\leftarrow 0)](R+\bar{R}d)+\bar{P}g.$$

Note that $\alpha, \ldots, \kappa$ are all *sre's*.

As an illustration of the consequences of structuring consider the following computation sequence in the original schema G:

$$a\ b\ c\ p\ \bar{q}\ d\ e\ \bar{s}$$

For the structured form of G this becomes



Thus the order of evaluation of the original functions and predicates is

$$a\ b\ c\ p\ \bar{q}\ d\ e\ \bar{s}$$

as expected, but the structured schema requires seven assignments to, and thirteen tests on, predicate variables as against no assignments and just three tests for the unstructured schema. Structuring is achieved only at a price.

8. EXAMPLES

In this Section the structuring method is applied to some practical problems.

Example 1.

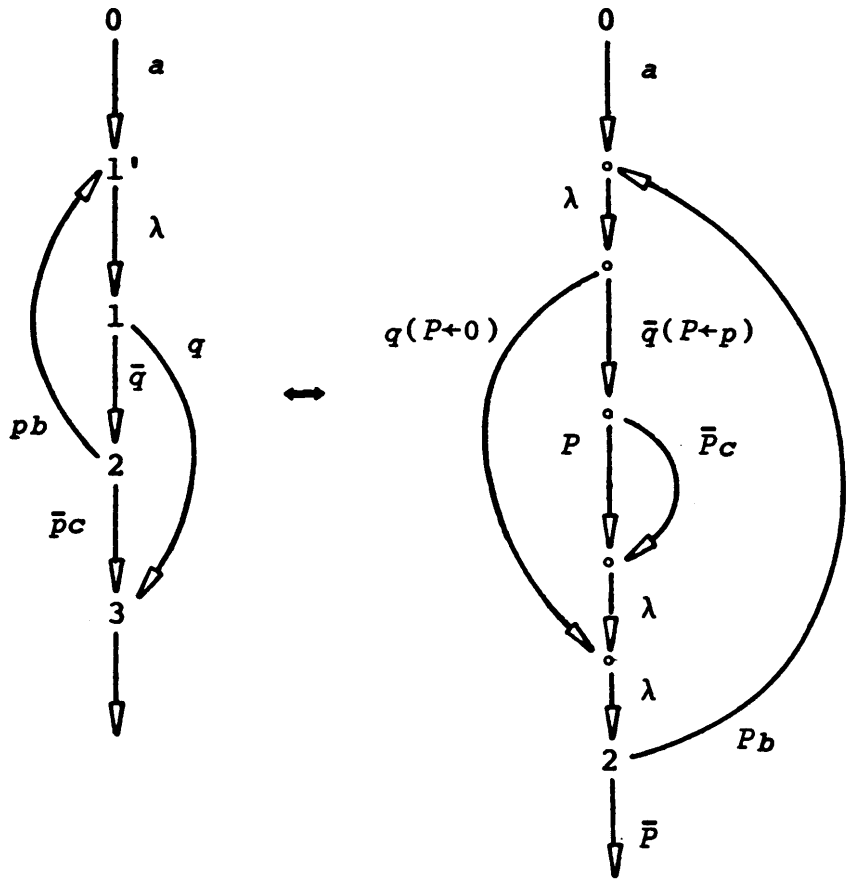
First consider the problem of searching an unordered list for a given key where it is not guaranteed that the key sought is present in the list. It is required to return found false if the key is not present and found true together with the entry index value if the key is present. Assuming that the list indices range from 1 to size inclusive then a typical solution is

```
0:  found ← false;
    entry ← 1;
1':
1:  while entry ≤ size do
      2: if list[entry] = key then
           found ← true;
           goto 3;
         else
           entry ← entry + 1;
3:  {found ⇒ list[entry] = key}
```

where indentation is used to indicate statement groupings. Define

```
a:  found ← false;  entry ← 1;
p:  list[entry] ≠ key
q:  entry > size
b:  entry ← entry + 1;
c:  found ← true;
```

then the schema for the search algorithm is as shown below and, as can be seen, comprises one instance of LD. Structuring the OD component yields the structured form shown which has the computation sequence set



$$a.[(\bar{q}.(P{\leftarrow}p).(P + \bar{P}.c) + q.(P{\leftarrow}0)).P.b]^{+}.\bar{P}$$

The structured form of the algorithm can be recovered by substituting for the labels and writing NotFinished for *P*. However, to avoid constructs such as ••• *until not* NotFinished, the algorithm will be written in terms of the flag $\bar{P}$: Finished rather than *P*. After transcribing expressions such as NotFinished ← false to Finished ← true, the final structured form of the algorithm becomes as shown below. Although longer than its equivalent unstructured form the algorithm nonetheless retains the same predicates and the same tests on them, albeit duplicated for Finished.

```
found ← false;
entry ← 1;
repeat
    if entry ≤ size then
        Finished ← list[entry] = key;
        if Finished then
            found ← true;
    else
        Finished ← true;
    if not Finished then
        entry ← entry + 1
until Finished;
{found ⇒ list[entry] = key}.
```

Structured Algorithm for List Search.

Example 2.

As a second example consider the problem of processing the nodes of a non-empty binary tree in post-order. An iterative solution is given below in terms of abstract data type (adt) operations on the tree and a stack.

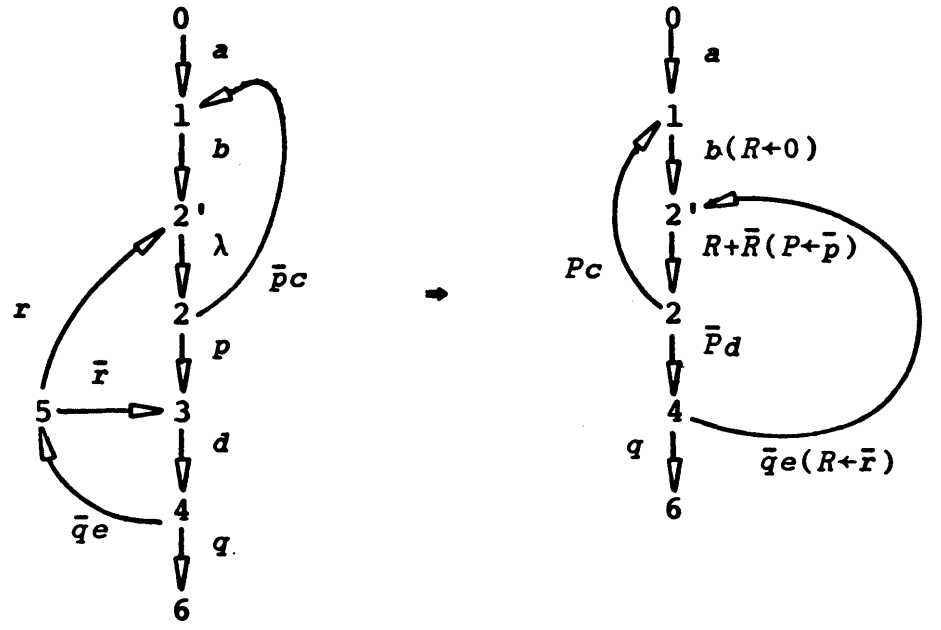
```
{t is a non-empty binary tree}
    s ← create_stack;
1:  while not empty_tree(left(t))do  {go left}
        push(s, <t, goright>);
        t ← left(t);
2':
2:  if not empty_tree(right(t)) then {go right}
        push(s, <t, goback>);
        t ← right(t);
        goto 1;
3:  process(root(t));
```

```
4:  if not empty_stack(s) then {go back up}
        <t, action> ← top(s);
        pop(s);
5:      if action = goright then
            goto 2
        else
            goto 3;
6:
```

Let

```
a:  s ← create_stack;
b:  while not empty_tree(left(t)) do
        push(s, <t, goright>);
        t ← left(t);
c:  push(s, <t, goback>);
    t ← right(t);
d:  process(root(t));
e:  <t, action> ← top(s);
    pop(s);
p:  empty_tree(right(t))
q:  empty_stack(s)
r:  action = goright
```

then the schema for the post-order traversal algorithm is as shown below. The back arcs are easily found to be (5,2'), (5,3) and (2,1), while the advancing path tree is seen to have one branch vertex 4, and one decision vertex leaf, 5. To prune the leaf back arcs (5,2) and (5,3) are reinstated to give a decision sub-schema with paths 5-3 and 5-2'-2-3 which, because 2' is a collector and 2 a decision, comprises one ID and one OD. Structuring this in the prescribed order of of ID and OD and eliminating redundant constructs which arise due to the presence of λ-paths yields the LL schema also shown below. Structuring the LL component then gives the structured regular expression



$$a.(P\leftarrow 1).[[(P.b.(R\leftarrow 0) + \bar{P}.e.(R\leftarrow \bar{r})).$$
$$(R + \bar{R}.(P\leftarrow \bar{p})).P.c]^{+}.\bar{P}.d.(P\leftarrow 0).\bar{q}]^{+}.q$$

The term $(P\leftarrow 0)$ is redundant since it will be executed only when P is false. Writing EmptyRight for $\bar{P}$ and BackUp for R finally gives the structured iterative algorithm below for the post-order traversal of a binary tree.

```
s ← create_stack;
EmptyRight ← false;
repeat
    repeat
        if not EmptyRight then
            while not empty_tree(left(t)) do
                push(s, <t, goright>);
                t ← left(t);
            BackUp ← false
        else
            <t, action> ← top(s);
            pop(s);
            BackUp ← action ≠ goright;
        if not Backup then
            EmptyRight ← empty_tree(right(t));
```

```
            if not EmptyRight then
                push(s, <t, goback>);
                t ← right(t);
        until EmptyRight;
        process(root(t));
    until empty_stack(s).
```

Structured Post-Order Traversal Algorithm.

Example 3.

As a third example consider the unstructured algorithm given below for merging two sorted files *l* and *r* into a single sorted file *s*. It is assumed that *l* and *r* are sorted in ascending order and that both are non-empty initially. The algorithm uses the following adt file operations

eof(*f*):	return 'true' if *f* is empty, 'false' otherwise;
get(*f*):	if not eof(*f*) then return <*f'*, *i*> where *i* is the first item on *f* and *f'* is the remainder of *f*, otherwise undefined;
put(*f*,*i*):	append item *i* to file *f* and return the updated file;
copy(f_1,f_2):	return the composite file $f_1.f_2$.

```
   {~eof(l) ∧ ~eof(r)}
   <l,u> ← get(l);
   <r,v> ← get(r);
1', 1",
1: if u ≤ v then
       s ← put(s,u);
2:     if eof(l) then
           s ← copy(put(s,v),r)
```

```
        else  {~eof(l)}
            <l,u> ← get(l);
            goto 1
    else  {u > v}
        s ← put(s,v);
3:      if eof(r) then
            s ← copy(put(s,u),l)
        else
            <r,v> ← get(r);
            goto 1;
4:
```

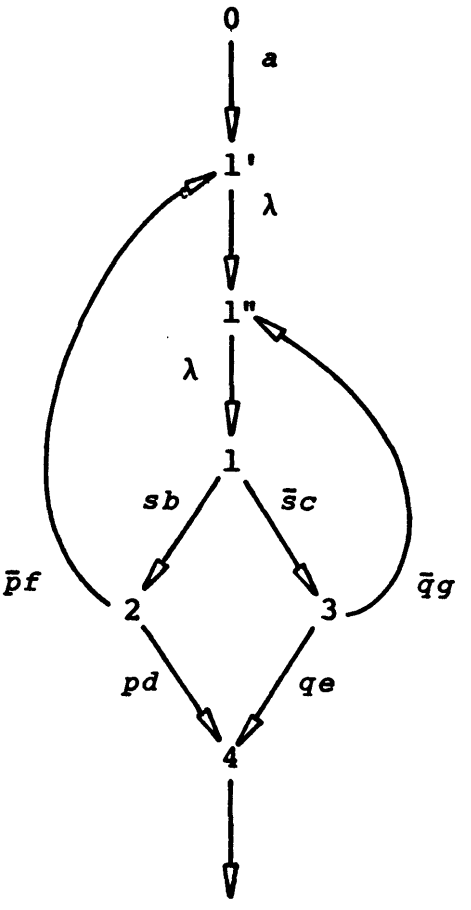
Algorithm for File Merge

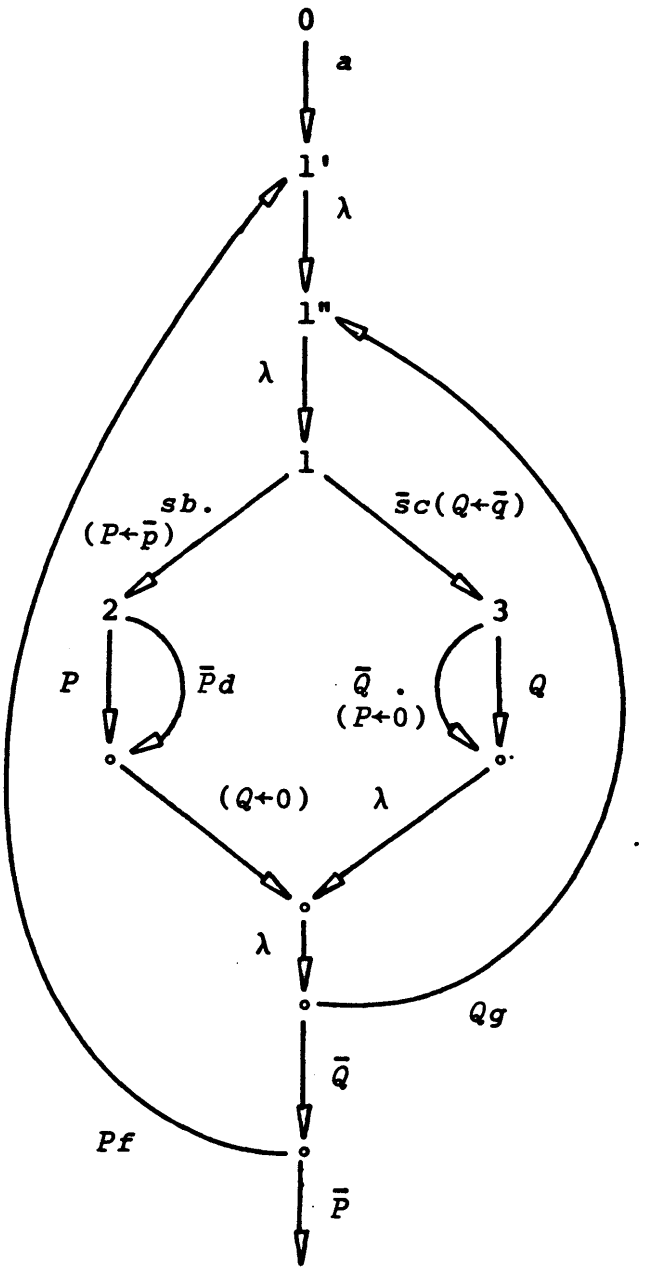
Defining

```
a:   <l,u> ← get(l);  <r,v> ← get(r);
b:   s ← put(s,u);
c:   s ← put(s,v);
d:   s ← copy(put(s,v),r)
e:   s ← copy(put(s,u),l)
f:   <l,u> ← get(l);
g:   <r,v> ← get(r);
p:   eof(l)
q:   eof(r)
```

then the resulting schema *G* is as shown below. Since arcs (1',1") and (1",1) are both λ-paths, 1' and 1" can be interchanged without any change to the computation sequences of the schema. This could be of advantage to avoid what might otherwise be an IL construct at a later stage in the structuring process.

The back arcs of *G* are (2,1') and (3,1") while the forward path includes two OD constructs. Applying OD-1 to the jump-out arc (2,1') first, followed by a further application of OD-1 yields the schema also shown below.





Observing that $P.(Q\leftarrow 0).\bar{Q}.P.f$ can be rewritten as $P.f.(Q\leftarrow 0).\bar{Q}$ and $Q.\lambda.Q.g$ as $Q.g.\lambda.Q$ without affecting any other computation sequence, the sre for the merge files problem becomes

$$a.[[s.b.(P\leftarrow\bar{p}).(P.f + \bar{P}.d).(Q\leftarrow 0) + (Q\leftarrow\bar{q}).(Q.g + \bar{Q}.e.(P\leftarrow 0))).Q]^{+}.\bar{Q}.P]^{+}.\bar{P}$$

from which the corresponding interpretation is:

```
<l,u> ← get(l);
<r,v> ← get(r);
repeat
    repeat
        if u ≤ v then
            s ← put(s,u);
            EofL ← eof(l);
            if EofL then
                s ← copy(put(s,v),r)
            else
                <l,u> ← get(l);
            EofR ← true
        else
            s ← put(s,v);
            EofR ← eof(r);
            if EofR then
                s ← copy(put(s,u),l);
                EofL ← true
            else
                <r,v> ← get(r)
    until EofR
until EofL
```

where EofL = $\bar{P}$ and EofR = $\bar{Q}$.

Although correct the algorithm can scarcely be regarded as more intelligible than the unstructured form. Indeed, the setting of EofR to 'true' when $u \leq v$, simply to enable escape from the inner loop, is positively misleading.

This example shows that the structuring method is not guaranteed to give sre's whose interpretations are clearer or more 'logical' in some sense than the unstructured original. In the present example the loss of clarity arises because of the overloading of meaning on the inner loop exit test for $\bar{Q}$, that is, EofR.

If the loop exit test is reached by the sequence $s.b. \cdots .(Q \leftarrow 0)$ then Q = false (EofR = true) means 'finished' if P = false also (EofL = true), otherwise it means 'continue to compare files'! Specifically, at the inner loop exit test point the following meanings hold

Q:	continue comparisons
$\bar{Q} \wedge P$:	" "
$\bar{Q} \wedge \bar{P}$:	finished.

The last condition suggests that the two loops be combined into one and a new flag R: Finished be introduced to give

$$a.[(s.b.(P \leftarrow \bar{p}).(P.f.(R \leftarrow 0) + \bar{P}.d.(R \leftarrow 1)) + \bar{s}.c.(Q \leftarrow \bar{q}).(Q.g.(R \leftarrow 0) + \bar{Q}.e.(R \leftarrow 1)).\bar{R}]^{+}.R$$

from which the term $(R \leftarrow 0)$ can be deleted and placed immediately after a. In this form the sre represents the same computation sequence set as the unstructured schema with regard to the latter's functions and variables and, like the original, is also symmetric. It is clear that the structuring transforms alone could not have produced this result for the reason that the final stages of development were dependent upon the particular interpretation given to the schema.

The final form of the File Merge Algorithm is

```
<l,u> ← get(l);
<r,v> ← get(r);
Finished ← false;
repeat
    if u ≤ v then
        s ← put(s,u);
        EofL ← eof(l);
        if EofL then
            s ← copy(put(s,v),r);
            Finished ← true
```

```
        else  {~EofL}
            <l,u> ← get(l)
    else  {u > v}
        s ← put(s,v);
        EofR ← eof(r);
        if EofR then
            s ← copy(put(s,v),l);
            Finished ← true
        else
            <r,v> ← get(r)
until Finished.
```

Structured Algorithm for File Merge.

9. SPACE AND TIME OVERHEADS

In developing and proving the effectiveness of the structuring transforms one important question was left unasked: how efficient are the resulting schemas in respect of space and time? This question is now taken up and it will be shown that structuring can only ever be achieved at a price: increased space requirements in the form of function duplication, introduced flags and assignments to the flags; increased time requirements in the form of duplicate tests on predicates; or some combination of the two.

It is desirable to have some general measure of the overheads incurred which is independent of particular interpretations of the schemas. Such a measure can of course give no guidance in general on the efficiency of the transformations for particular interpretations where for example reductions in overheads may be possible through optimizations over local functions or introduced flags, but could nonetheless be useful as a means of comparing the results produced by the structuring process. One such measure is the space-time hierarchy for embedded graphs described by Lipton, Eisenstat and DeMillo[30] and refined by them in DeMillo *et al.*[31] This measure can be defined informally as follows. Let $G = (V,E)$ be a schema in which the vertices V represent functions and predicates and the arcs E the flow of control between them. (This definition is different from that used elsewhere in this thesis.) Let $\text{dist}(u,v)$ be the minimum path length, calculated as the number of arcs between the vertices $u,v \in V$, with $u \neq v$. G is said to be embedded in a strictly equivalent schema $G' = (V',E')$ with respect to space S and time T if S is the largest number of

duplications of any function of predicate of G contained in G', and T is the least value satisfying $dist(u',v') \leq T \times dist(u,v)$, where u',v' in G' correspond to u,v in G. Thus $S = s$ means that there are s occurrences of some function in G' as against one in G and no other function in G' has more occurrences than s. $T = t$ means that two distinct functions or predicates having one arc in between them in G have t arcs in between them in G', and no other pairs of distinct functions or predicates in G have a greater separation than t in G'.

Returning to the structuring transforms and noting that each introduced reference (test or assignment) to a flag adds an arc to the embedding schema, the space-time measures of DeMillo *et al.* can be derived directly from the computation sequence sets of the embedded and embedding schemas. Thus for ID, Figure 9, the computation sequence sets for ID and its two transforms are

	ID	ID-0	ID-1
(1)	$c.q.e.f$	$c.q.e.f$	$c.(Q \leftarrow q).Q.e.Q.f$
(2)	$c.\bar{q}.d.a.f$	$c.\bar{q}.d.a.f$	$c.(Q \leftarrow q).\bar{Q}.d.\bar{Q}.a.f$
(3)	$b.a.f$	$b.a.f$	$b.(Q \leftarrow 0).\bar{Q}.a.f$

from which it can be seen that $T = 1$ for all sequences in ID-0 but $T = 3$ for ID-1 in consequence of sequences (1) and (2). The space overheads can be seen by re-writing the computation sequences as

ID	ID-0
$c.(q.e + \bar{q}.d.1{:}a).f$	$c.(q.e + \bar{q}.d.a).2{:}f$
$b.\uparrow 1$	$b.a.\uparrow 2$

where n: denotes the target of a *goto* action ↑n. Thus $S = 2$ for ID-0, whilst for ID-1 $S = 1$ with respect to the original functions and predicates of ID since

the DeMillo *et al.* space measure takes no account of introduced flags or assignments to them. The S, T values over all transforms are found similarly and are given in the table below:

ID-0	2, 1	IL-0	2, 1
ID-1	1, 3	IL-1	1, 3
OD-1	1, 3	OL-1	1, 4

Thus the Type-0 transforms require an increase of space alone (from function duplication), whereas the Type-1 transforms require an increase in time alone (from predicate duplication in the form of flags).

Because S and T are defined in terms of extreme values rather than total ones, they are generally not additive over successive applications of the transforms, so that the space-time penalties incurred in producing any particular structured schema have to be computed from that schema rather than from the transforms used to produce it. Nonetheless it is clear that since structuring must use at least one of the Type-0 or Type-1 transforms some space or time overheads are necessarily incurred in general, although the presence of λ-paths may in some instances reduce these to nothing by making some form of duplication redundant. Thus $a = \lambda$ in the paradigm for ID means that there is no function duplication in ID-0 and that the decision subschema $Q + \bar{Q}.a$ is redundant in the form $Q + \bar{Q}.\lambda$ in ID-1, thereby removing the need for predicate duplication. Conversely, the transforms do not introduce space-time savings which, because of the restrictive nature of structured control mechanisms, is as to be expected.

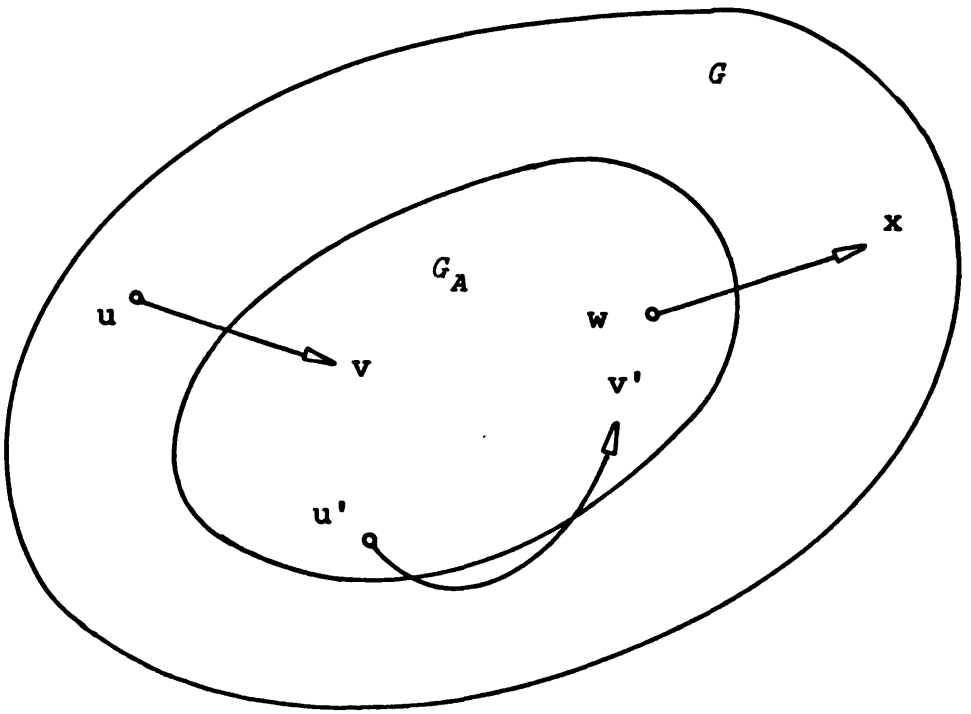
CONCLUSION

It has been shown that an arbitrarily complex unstructured program schema can always be put into structured form using just three transforms on basic unstructured forms. These transforms require in general either a duplication of a predicate or, under special conditions, a duplication of a function instead. In the presence of λ-paths in the buf undergoing transformation it may be possible to avoid function or predicate duplication but in general structuring always requires some loss of computational efficiency compared with optimal unstructured schemas.

Algorithms for the identification of buf's have been given and an ordering in which the buf's should be removed has been proposed.

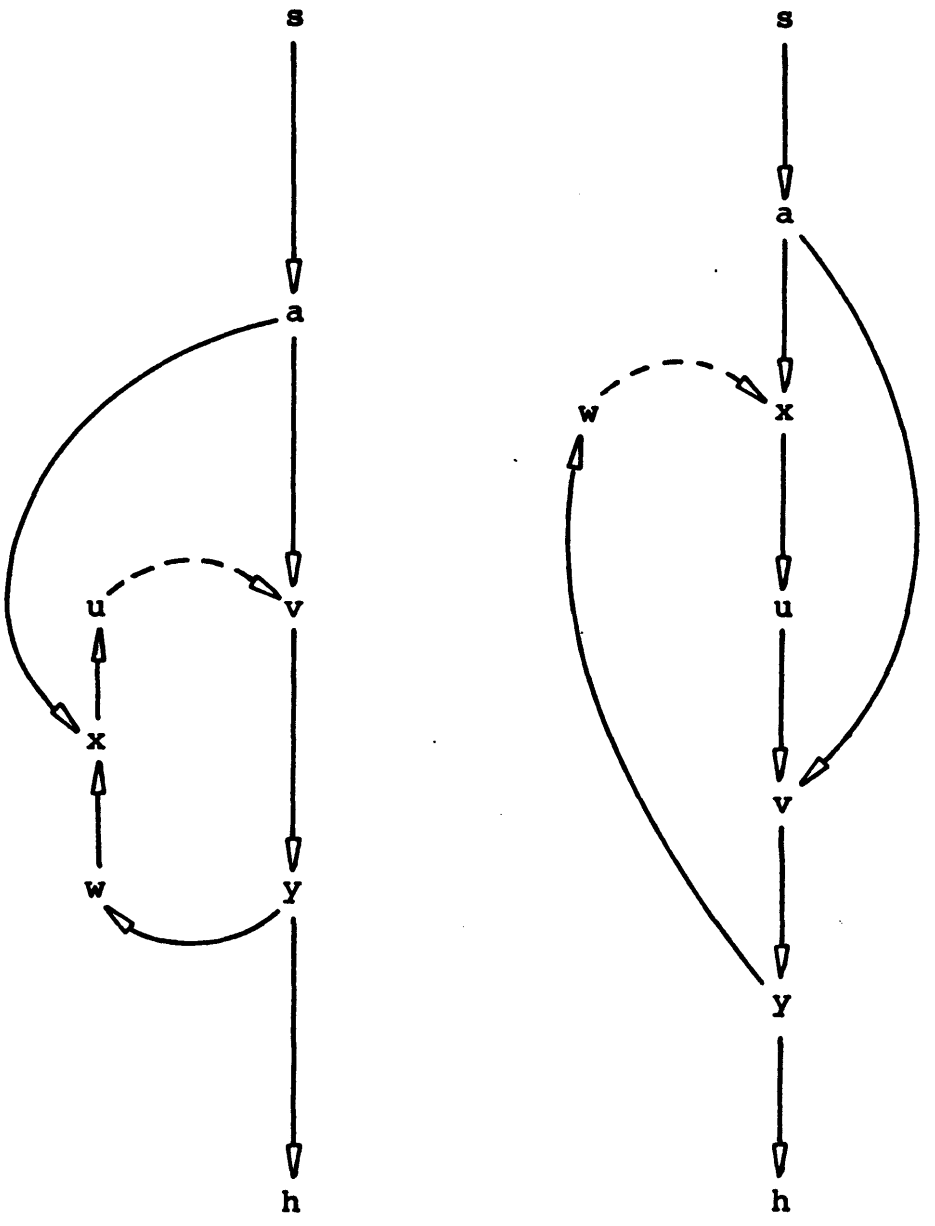
Application of the transformational method to some practical problems shows that clear well formed structured algorithms can be produced, but it has also been shown that the method does not necessarily produce such results under all conditions and that further transformations based on considerations of particular interpretations may be required to achieve clear logical structured programs.

Finally it must be said that the structured schemas produced can at best be only as good as the unstructured originals - tangled nonsense cannot be transformed into logical poetry.



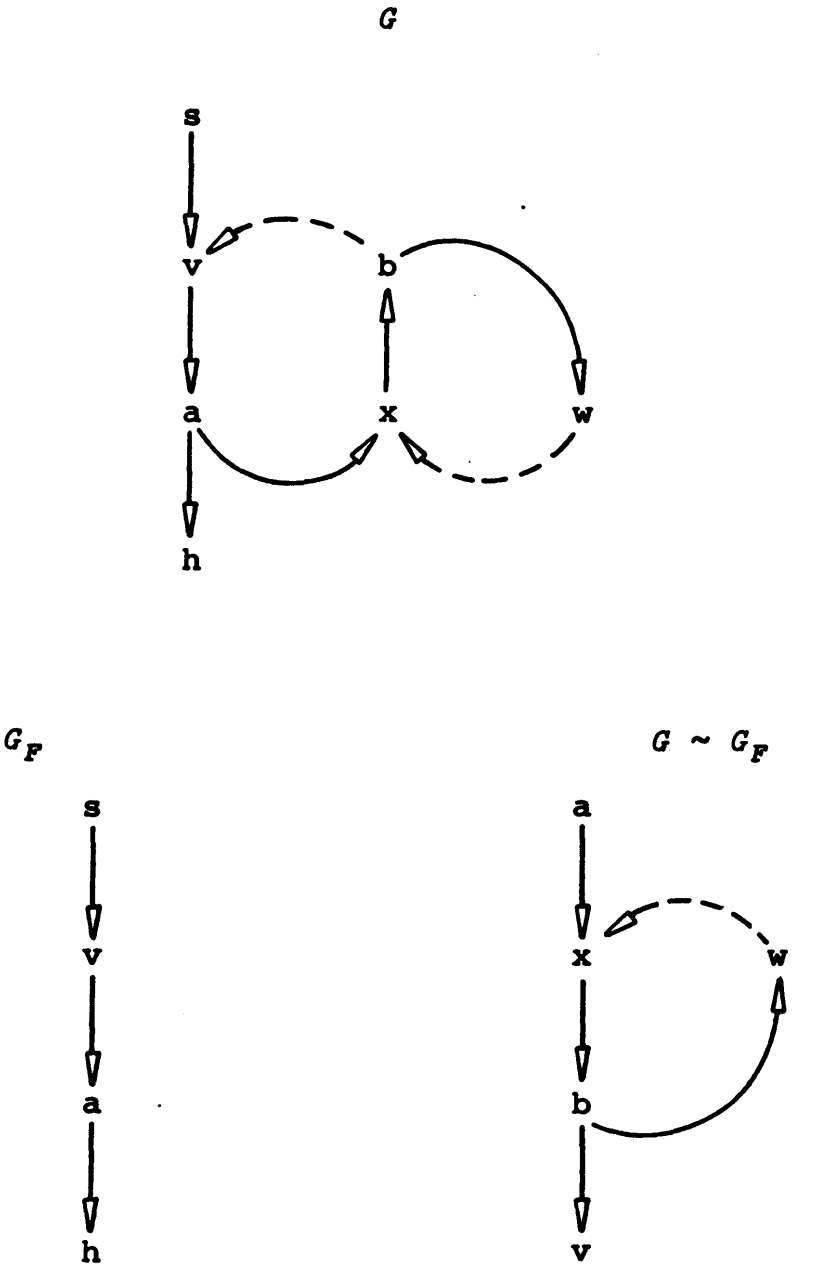
Predecessor vertices of G_A	u, u'
Successor vertices of G_A	v', x

Figure 1.



(u,v) is a back latch of s-a-v-y-h
(w,x) is a back latch of s-a-x-u-v-y-h

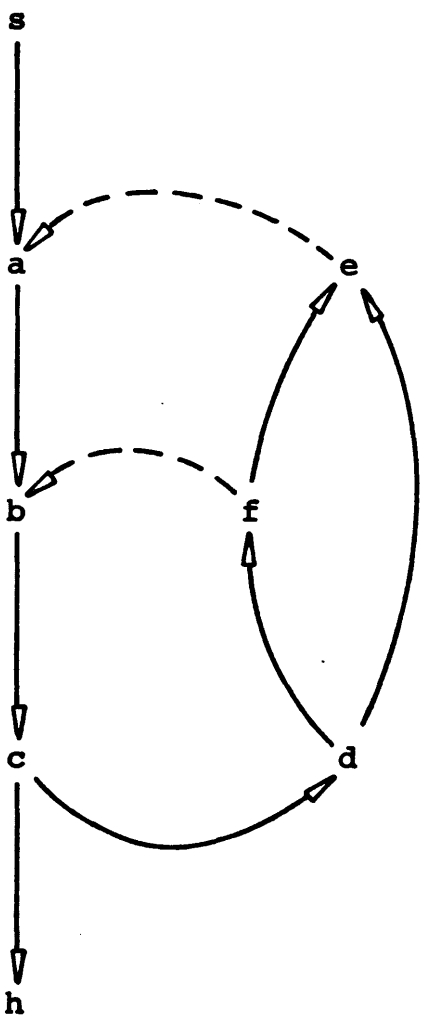
Figure 2.



(w,x) is a back latch of G and $G \sim G_F$

(b,v) is a back latch of G.

Figure 3.



Back arcs	(e,a), (f,b)
Decision entries	d
Decision exits	e
Cycle entries	a, b
Cycle exits	f, c

Figure 4a.

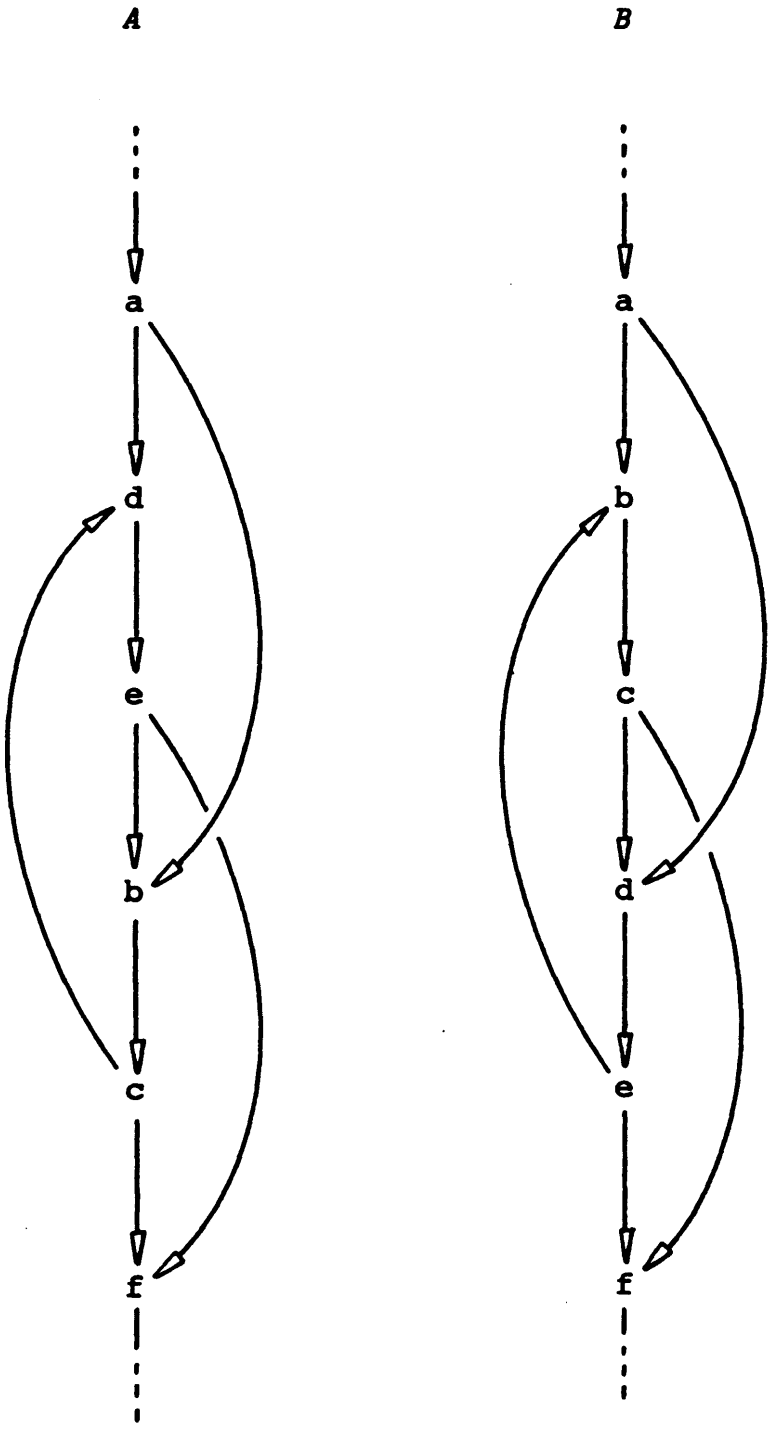


Figure 4b.

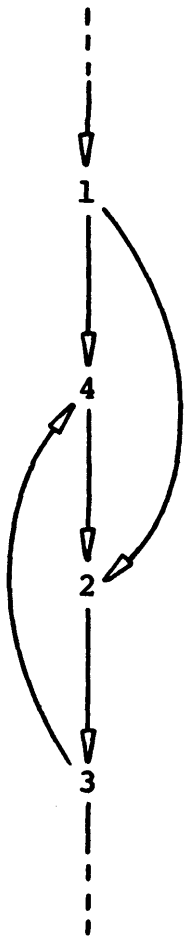
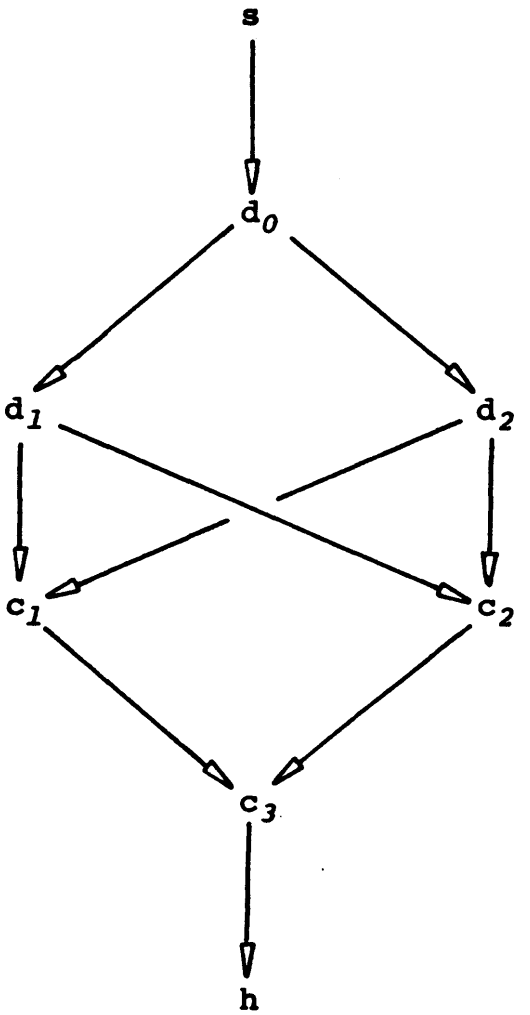


Figure 4c.

UNIVERSITY COLLEGE LONDON Library



Overlapping simple decisions

(d_0, d_1, d_2, c_1) with (d_0, d_1, d_2, c_2)

(d_1, c_1, c_2, c_3) with (d_2, c_1, c_2, c_3)

Figure 5.

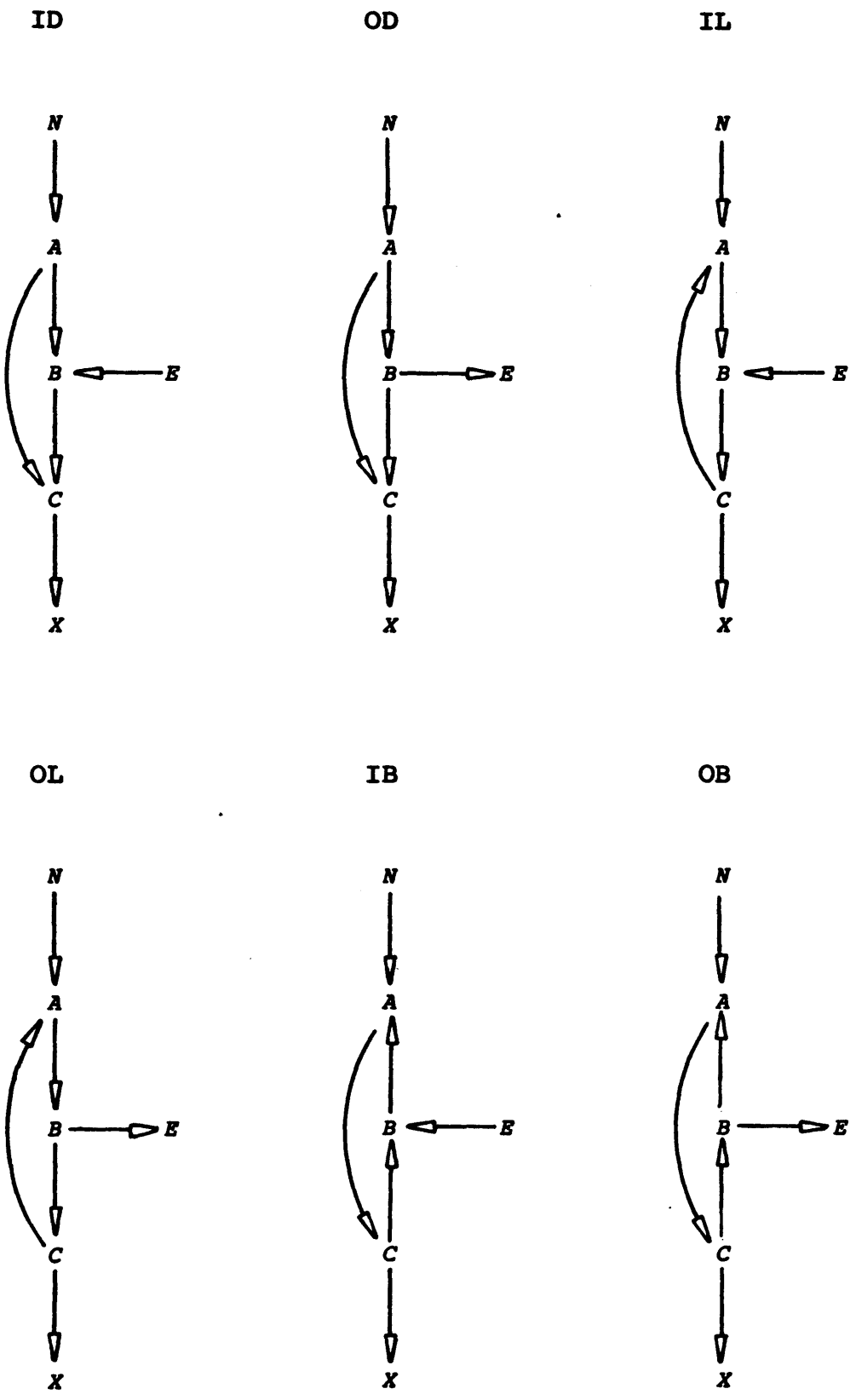


Figure 6.

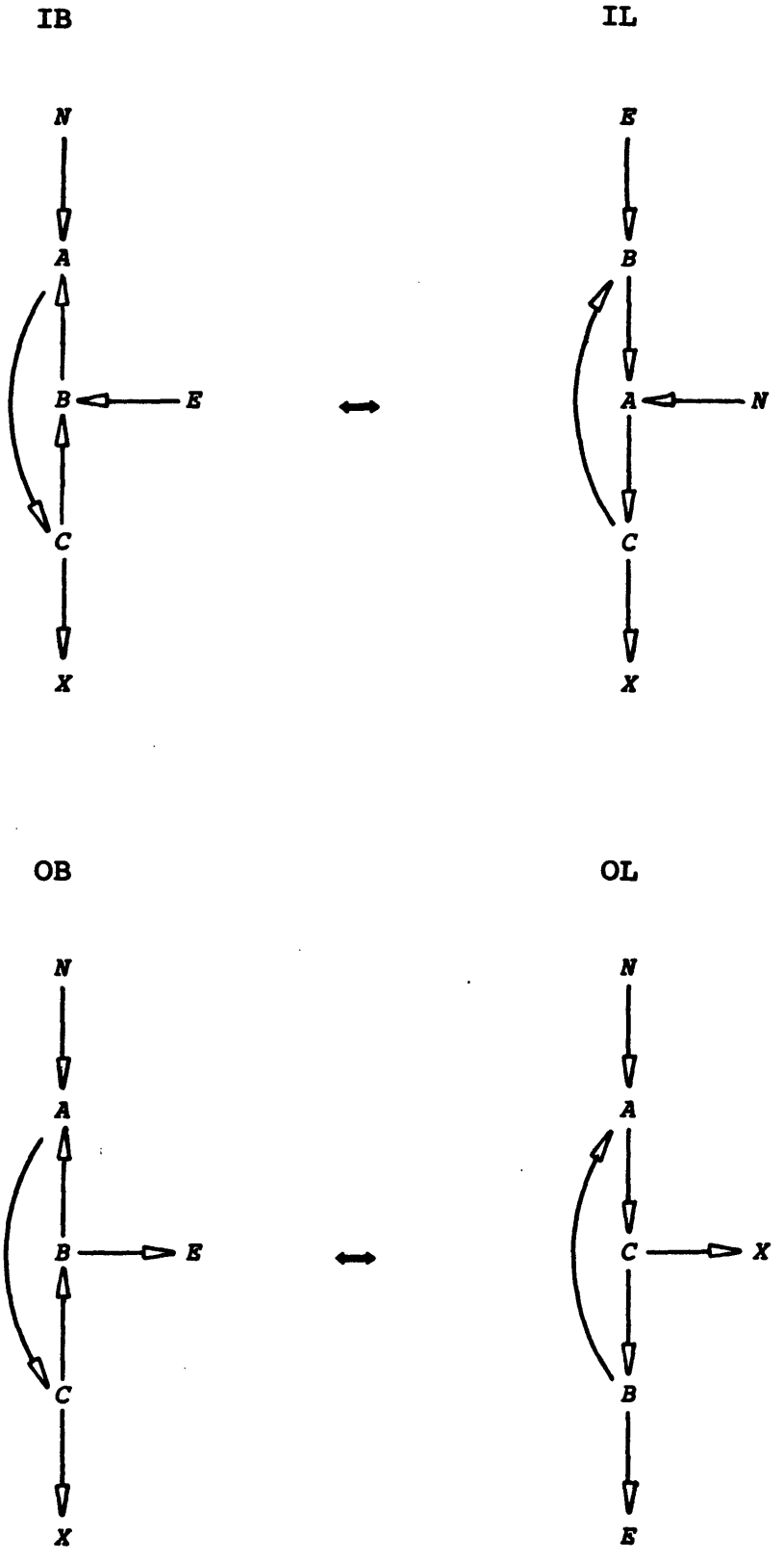


Figure 7.

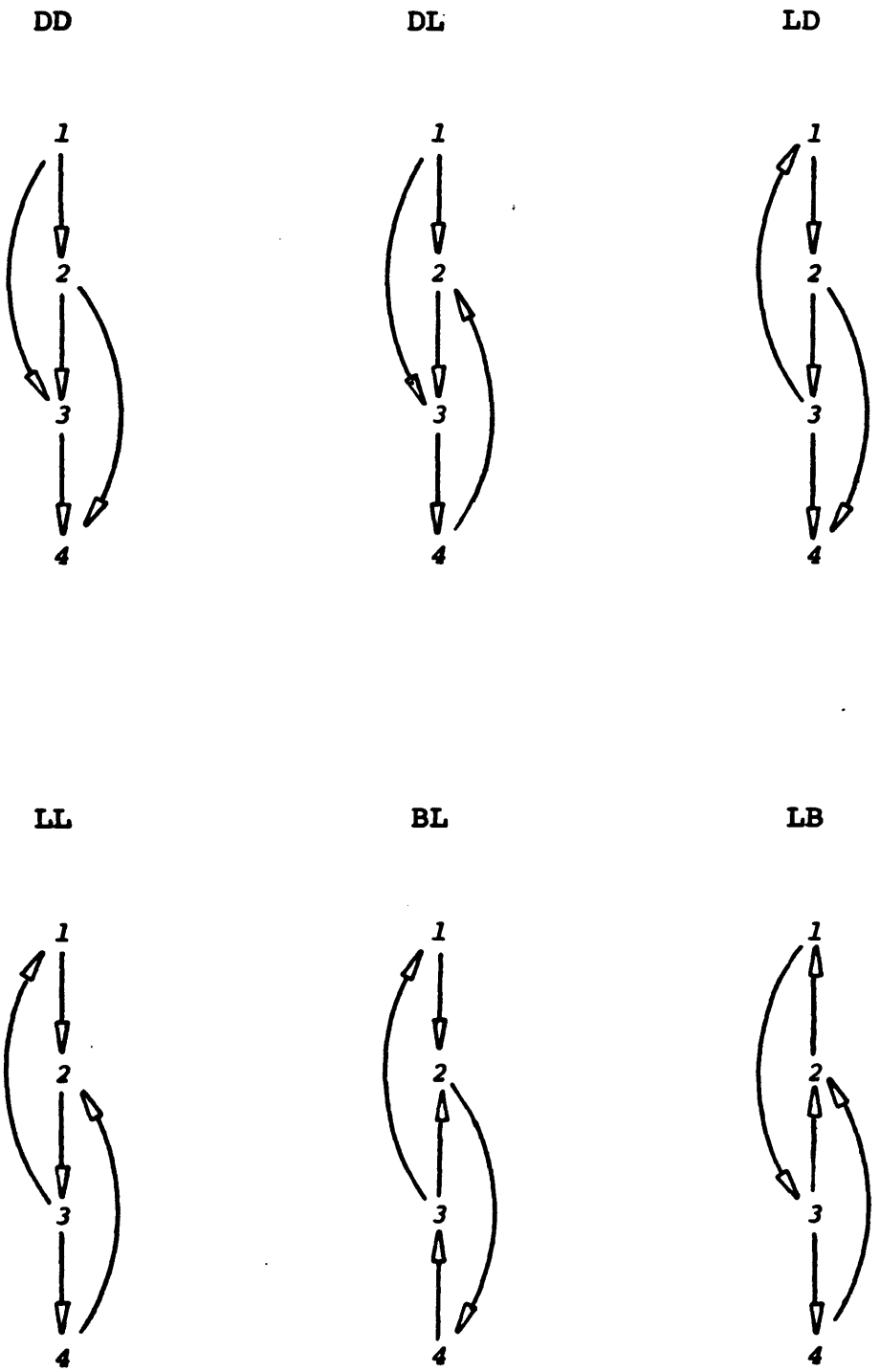
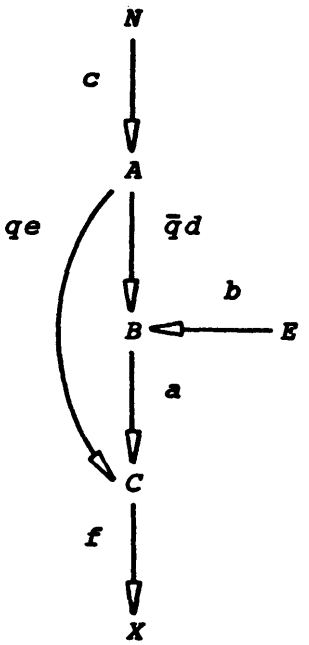
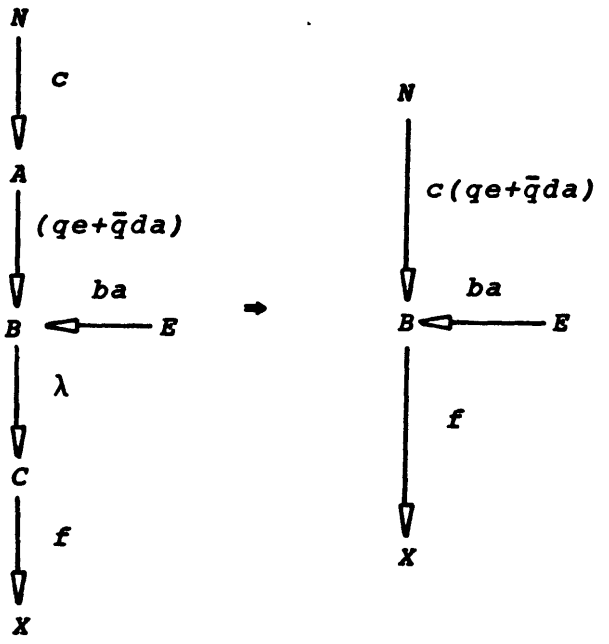


Figure 8.

ID



ID-0



ID-1

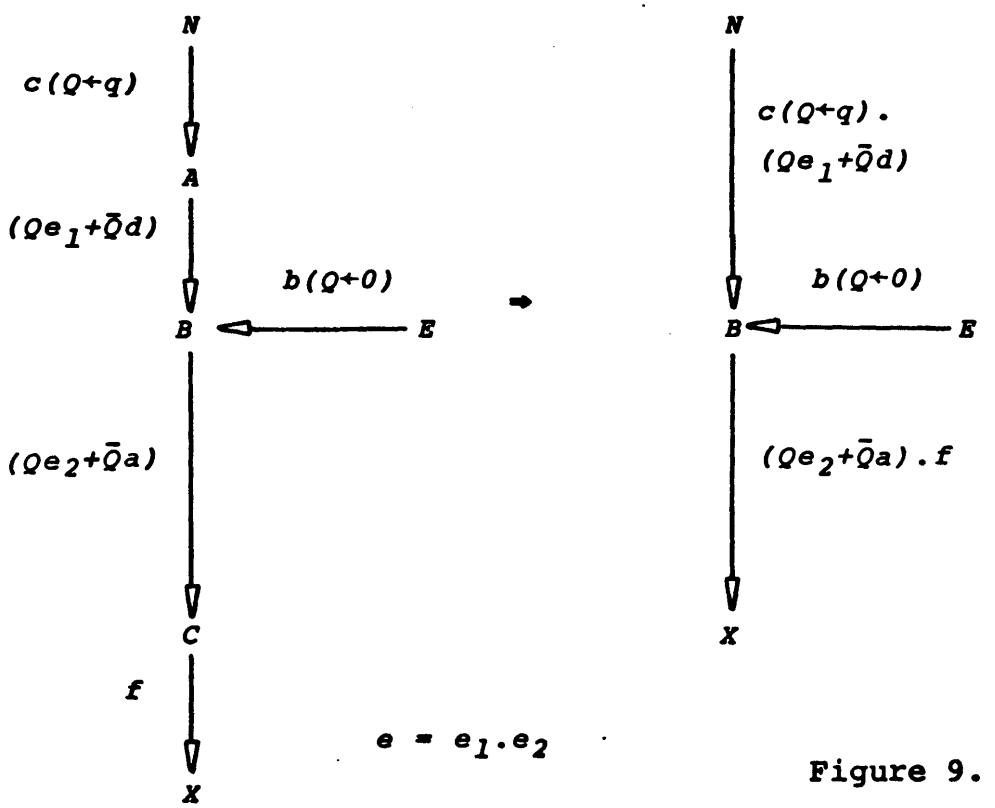
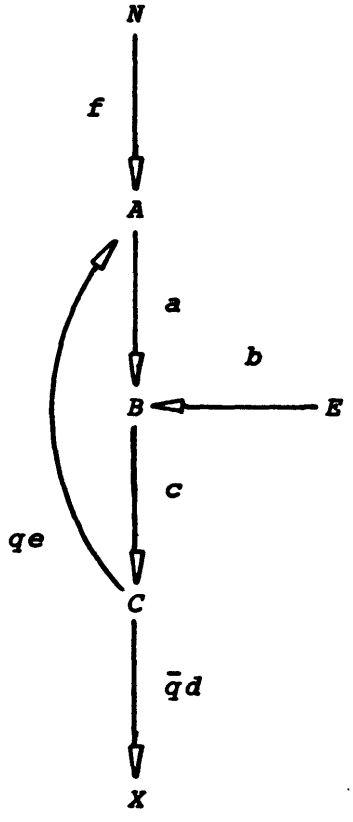
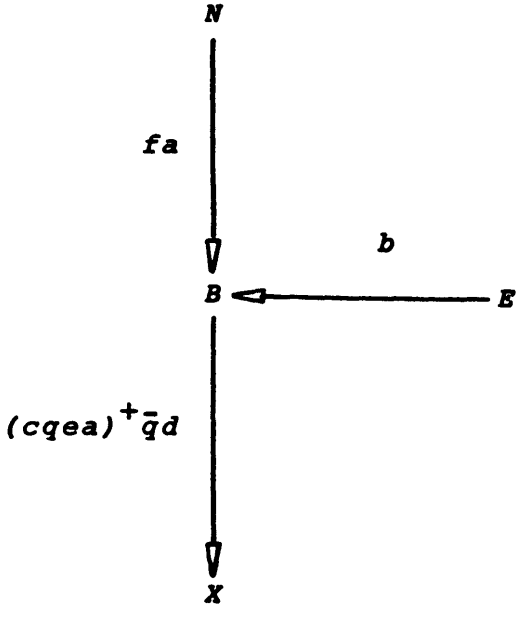


Figure 9.

IL



IL-0



IL-1

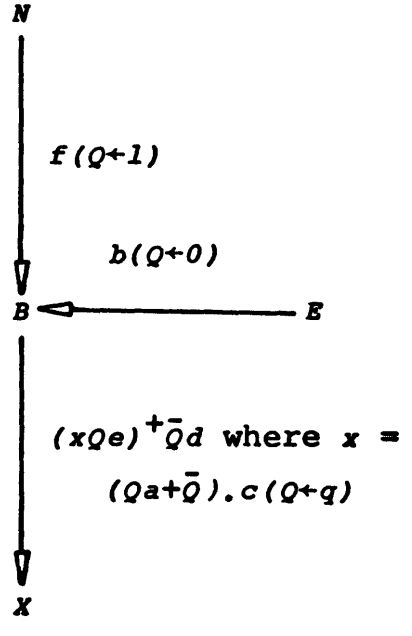
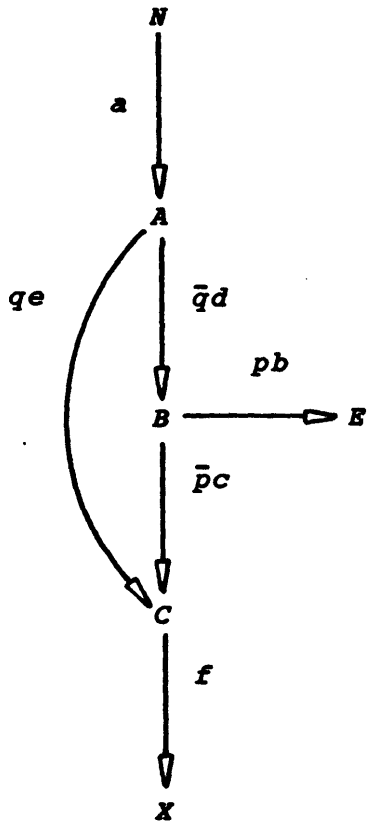


Figure 10.

OD



OD-1

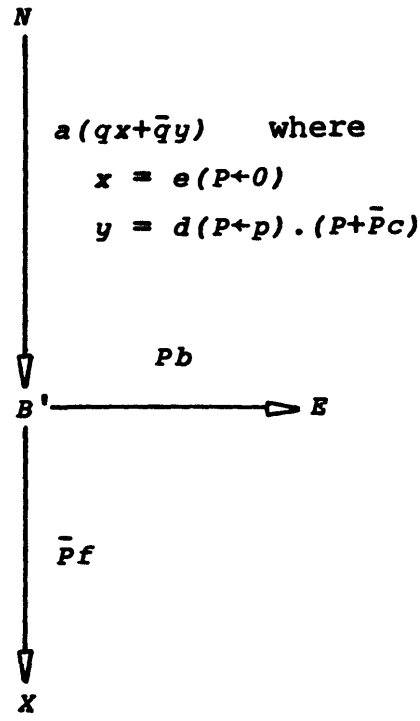
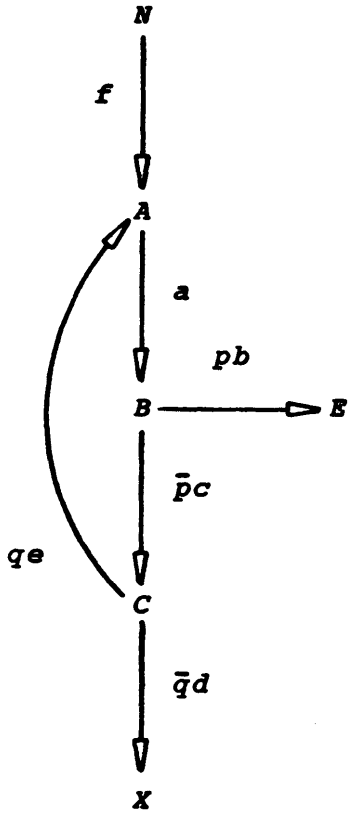


Figure 11.

OL



OL-1

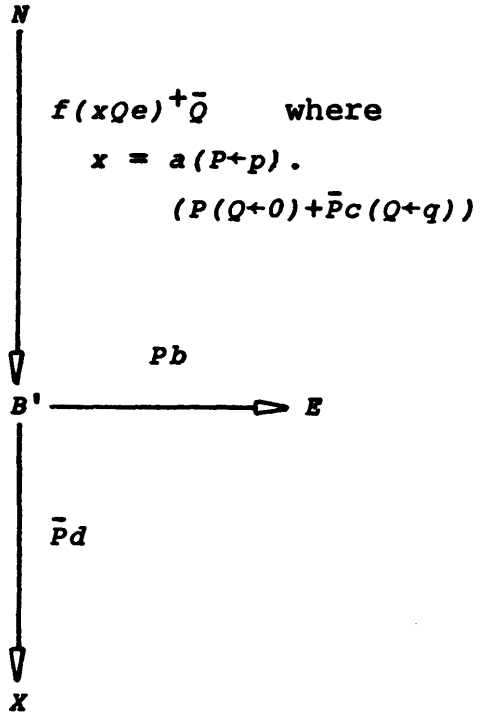


Figure 12.

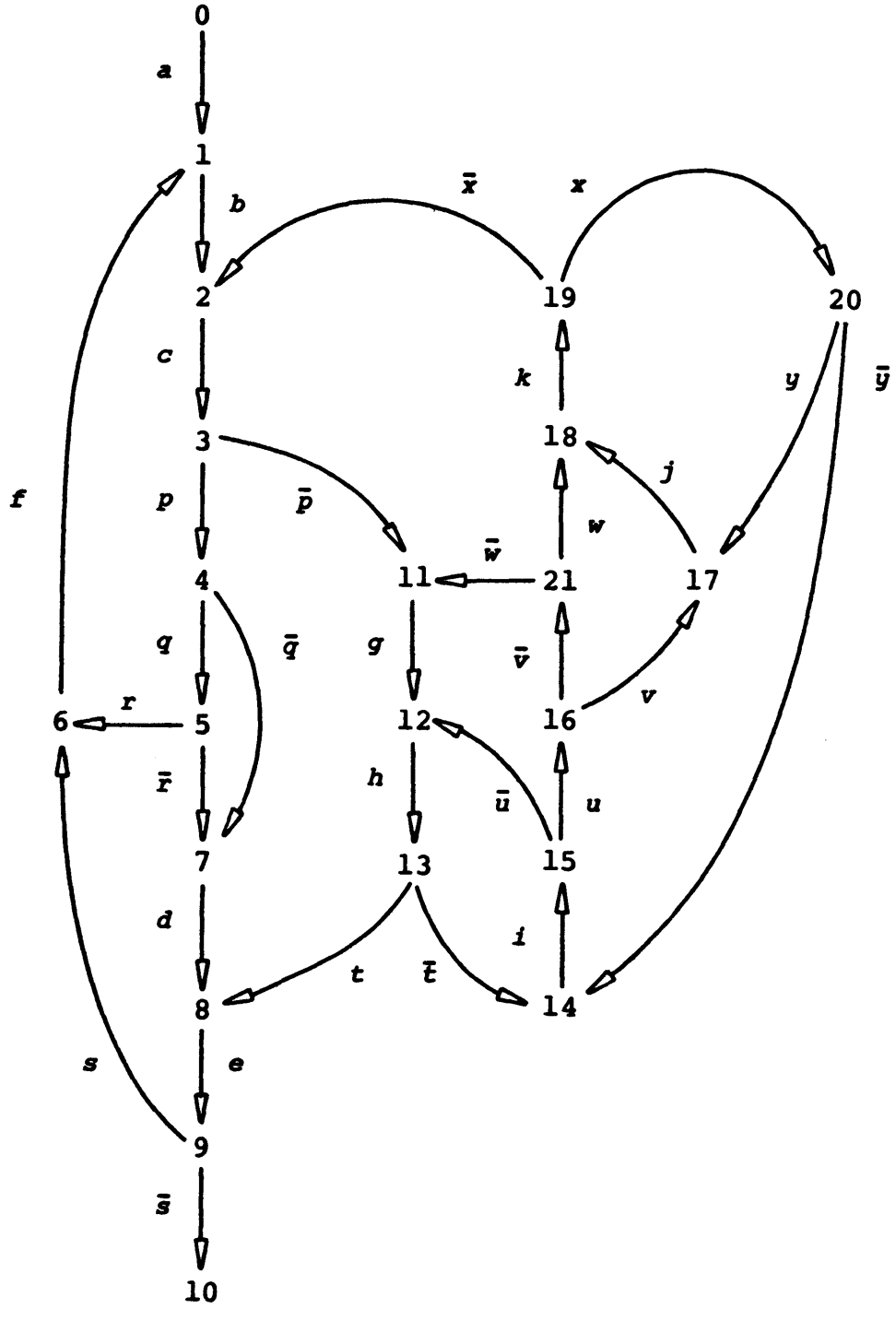
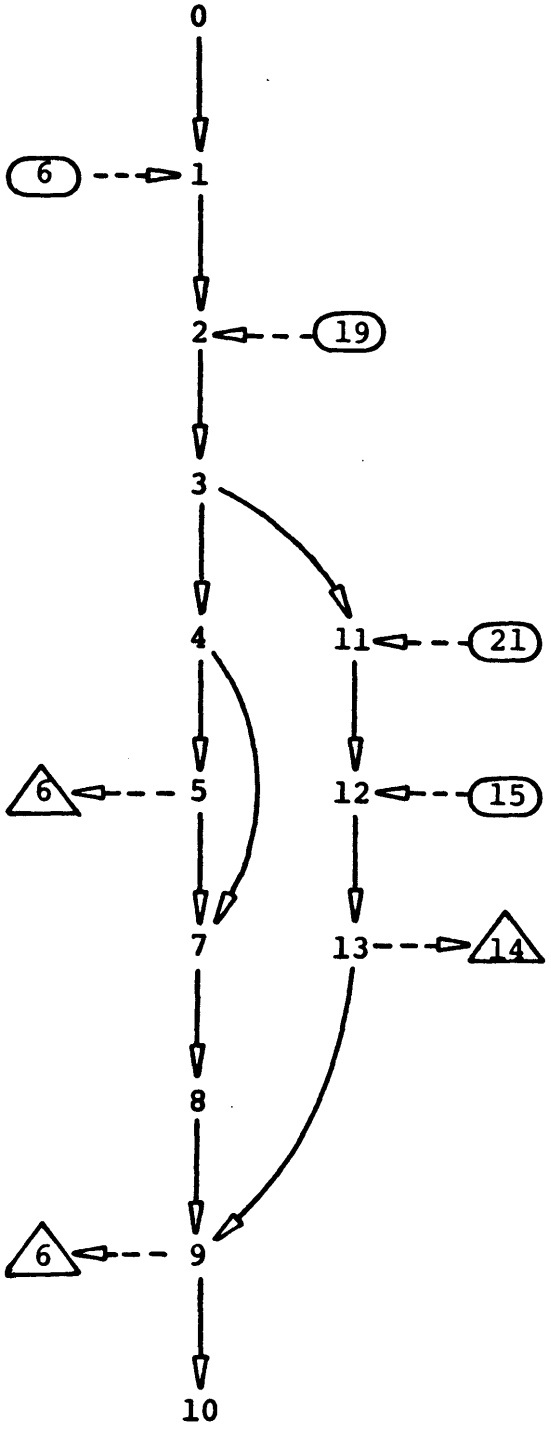

0
a
1
b
$\bar{x}$
x
2
19
20
c
k
y
$\bar{y}$
3
18
j
f
p
$\bar{p}$
w
$\bar{w}$
4
11
21
17
q
$\bar{q}$
g
$\bar{v}$
r
v
6
5
12
16
$\bar{r}$
h
$\bar{u}$
u
7
13
15
d
i
t
$\bar{t}$
8
14
s
e
9
$\bar{s}$
10

Figure 13.

G_F



Back arcs:
(6,1), (19,2), (21,11),
(15,12).

Cycle entries:
1, 2, 11, 12.

Cycle exits:
15, 19, 21.

Figure 14.

$G_1 = G \sim G_F$

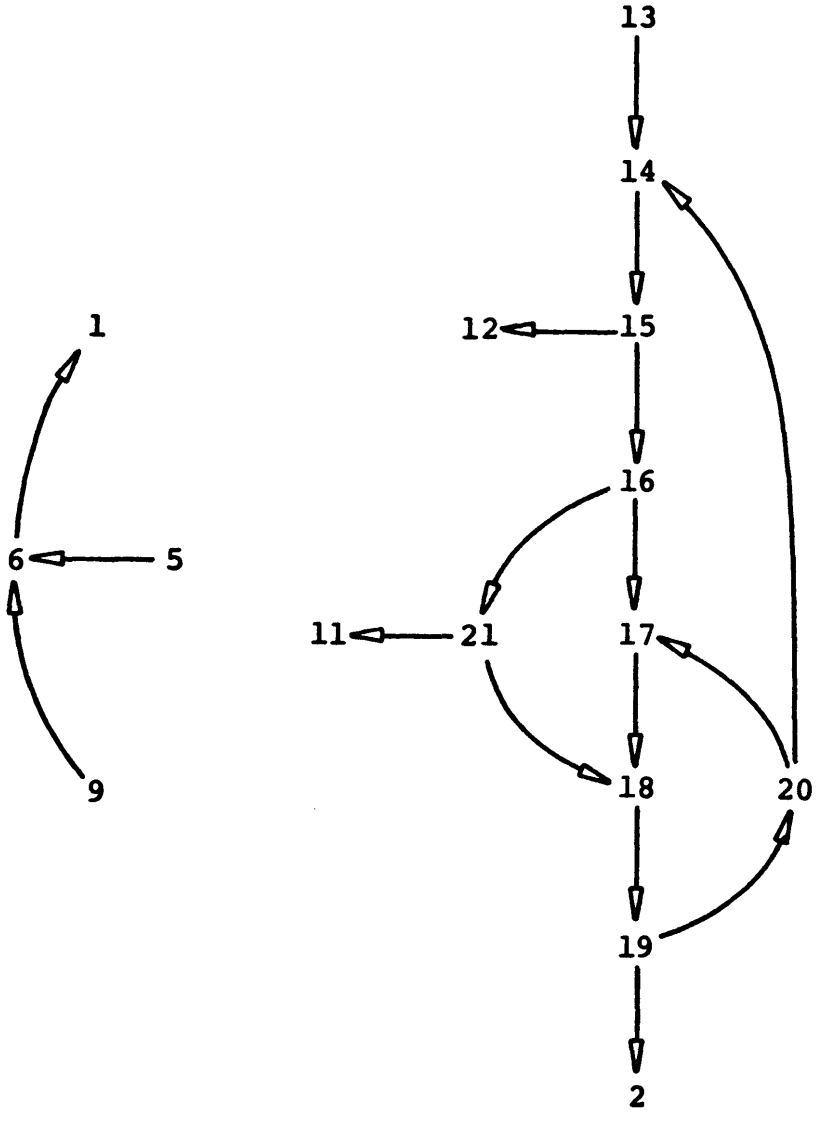
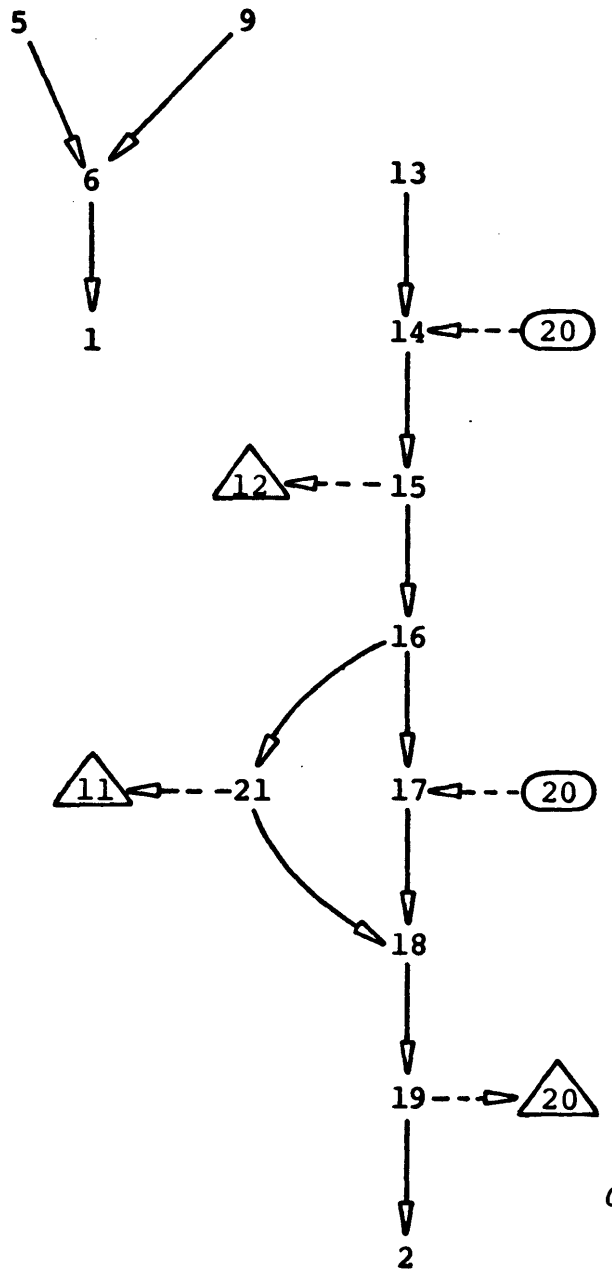


Figure 15.

G_{1F}



$G_2 = G_1 \sim G_{1F} = G_{2F}$

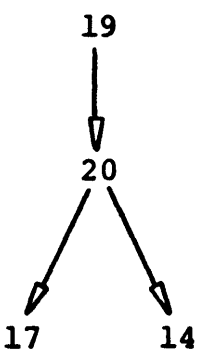


Figure 16.

$\hat{G}$

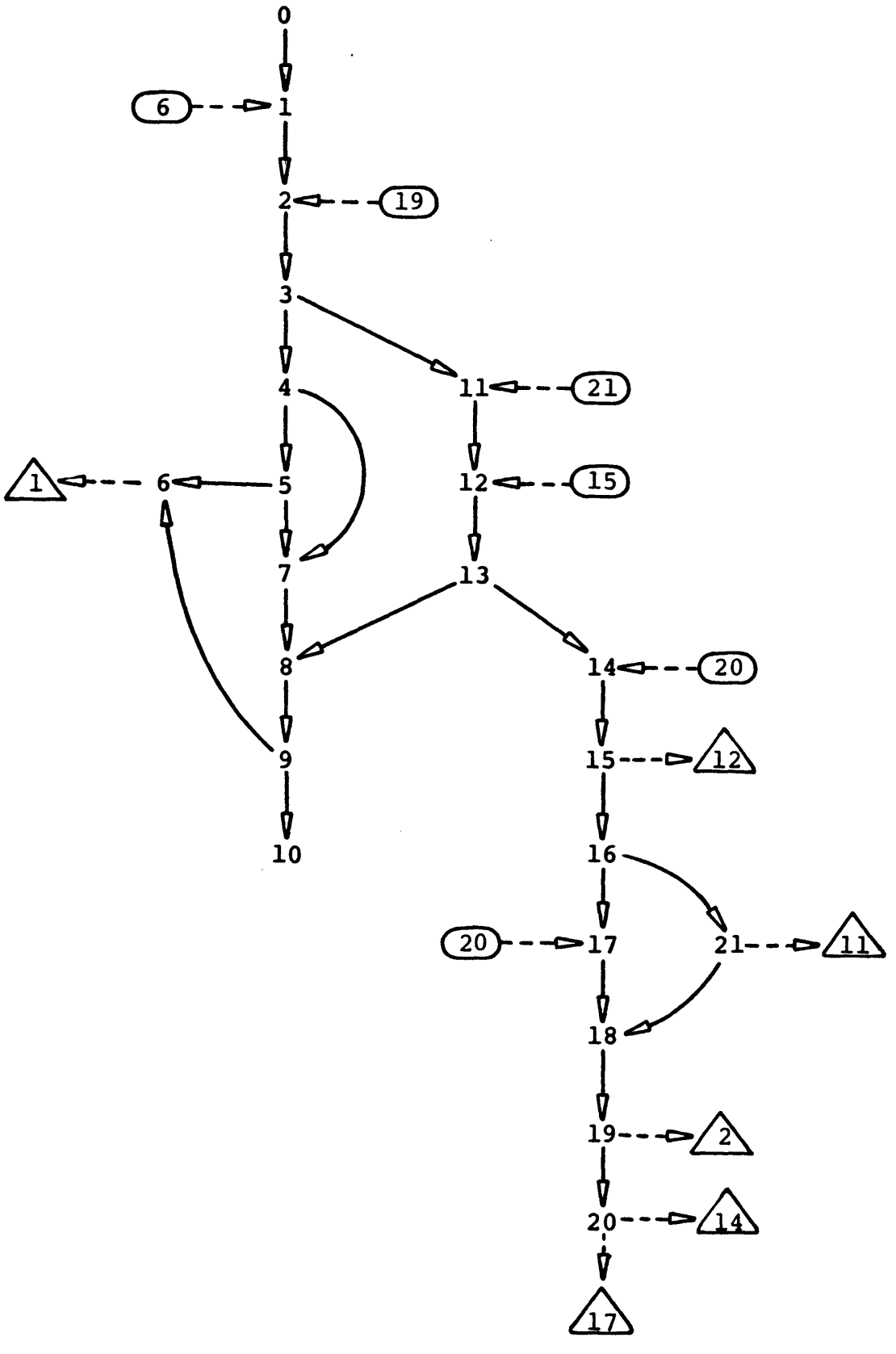
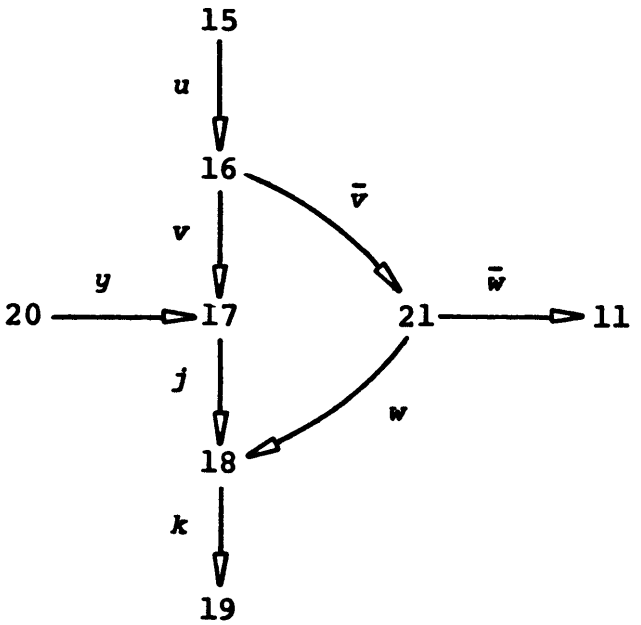


Figure 17.


15
u
16
v̄
v
y
w̄
20
17
21
11
j
w
18
k
19

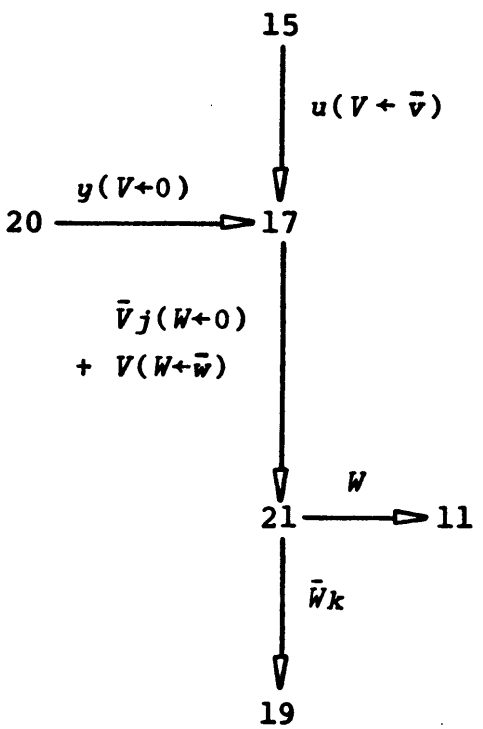

15
u(V ← v̄)
y(V←0)
20
17
V̄j(W←0)
+ V(W←w̄)
W
21
11
W̄k
19

Figure 18.

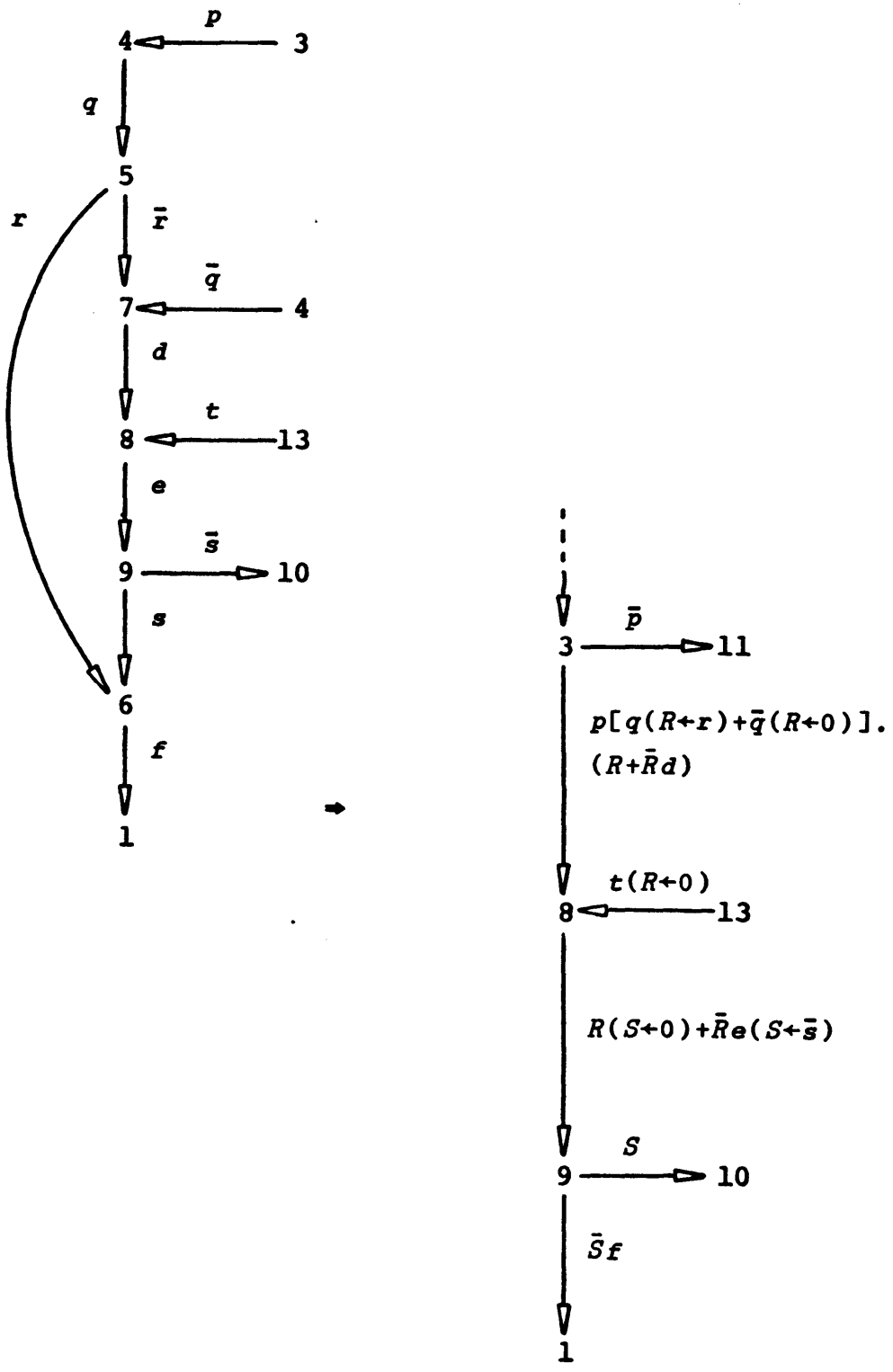
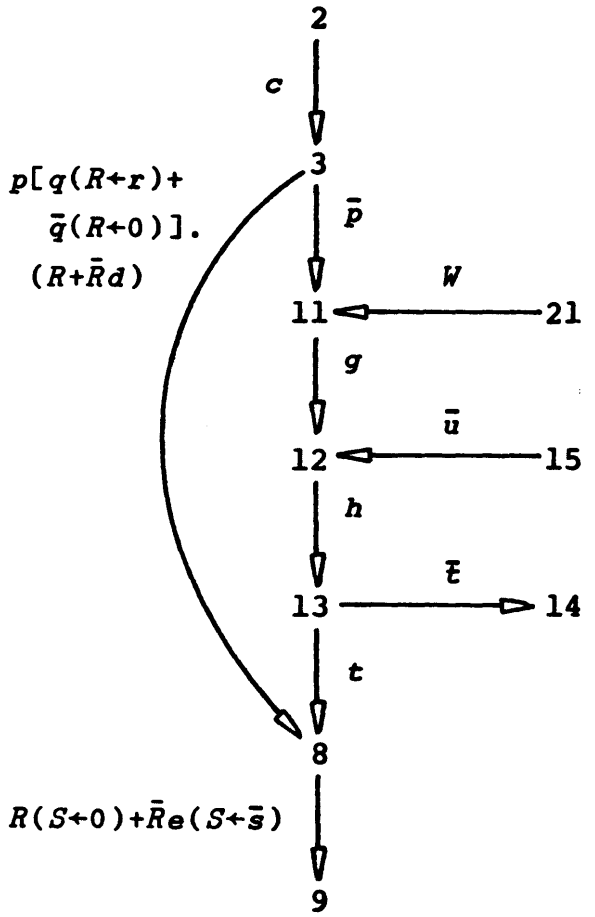

p
4
3
q
5
r
r̄
q̄
7
4
d
t
8
13
e
s̄
9
10
s
6
f
1
p̄
3
11
p[q(R←r)+q̄(R←0)].
(R+R̄d)
t(R←0)
8
13
R(S←0)+R̄e(S←s̄)
S
9
10
S̄f
1

Figure 19.


2
c
3
p[q(R←r)+
q̄(R←0)].
(R+R̄d)
p̄
W
11
21
g
ū
12
15
h
t̄
13
14
t
8
R(S←0)+R̄e(S←s̄)
9

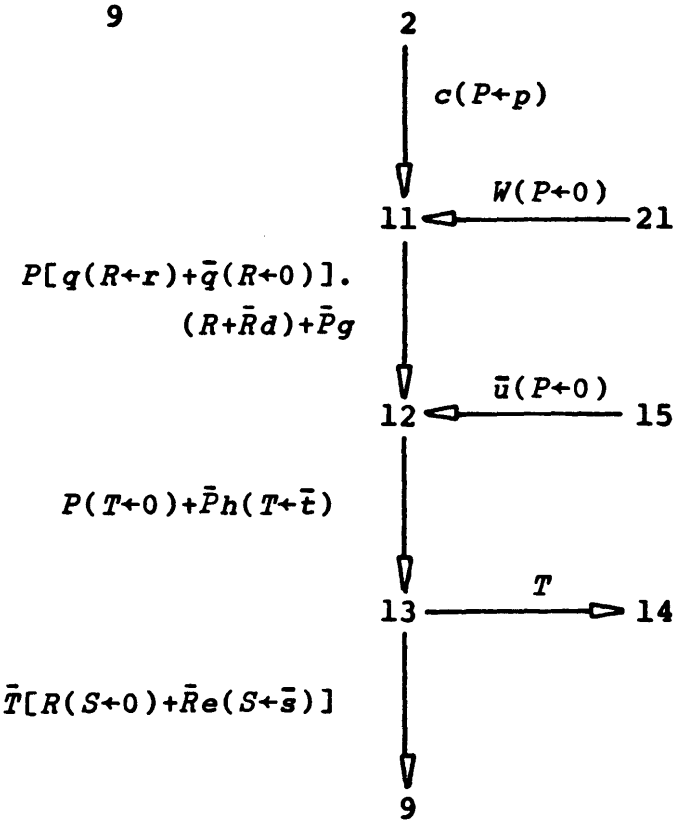

2
c(P←p)
W(P←0)
11
21
P[q(R←r)+q̄(R←0)].
(R+R̄d)+P̄g
ū(P←0)
12
15
P(T←0)+P̄h(T←t̄)
T
13
14
T̄[R(S←0)+R̄e(S←s̄)]
9

Figure 20.

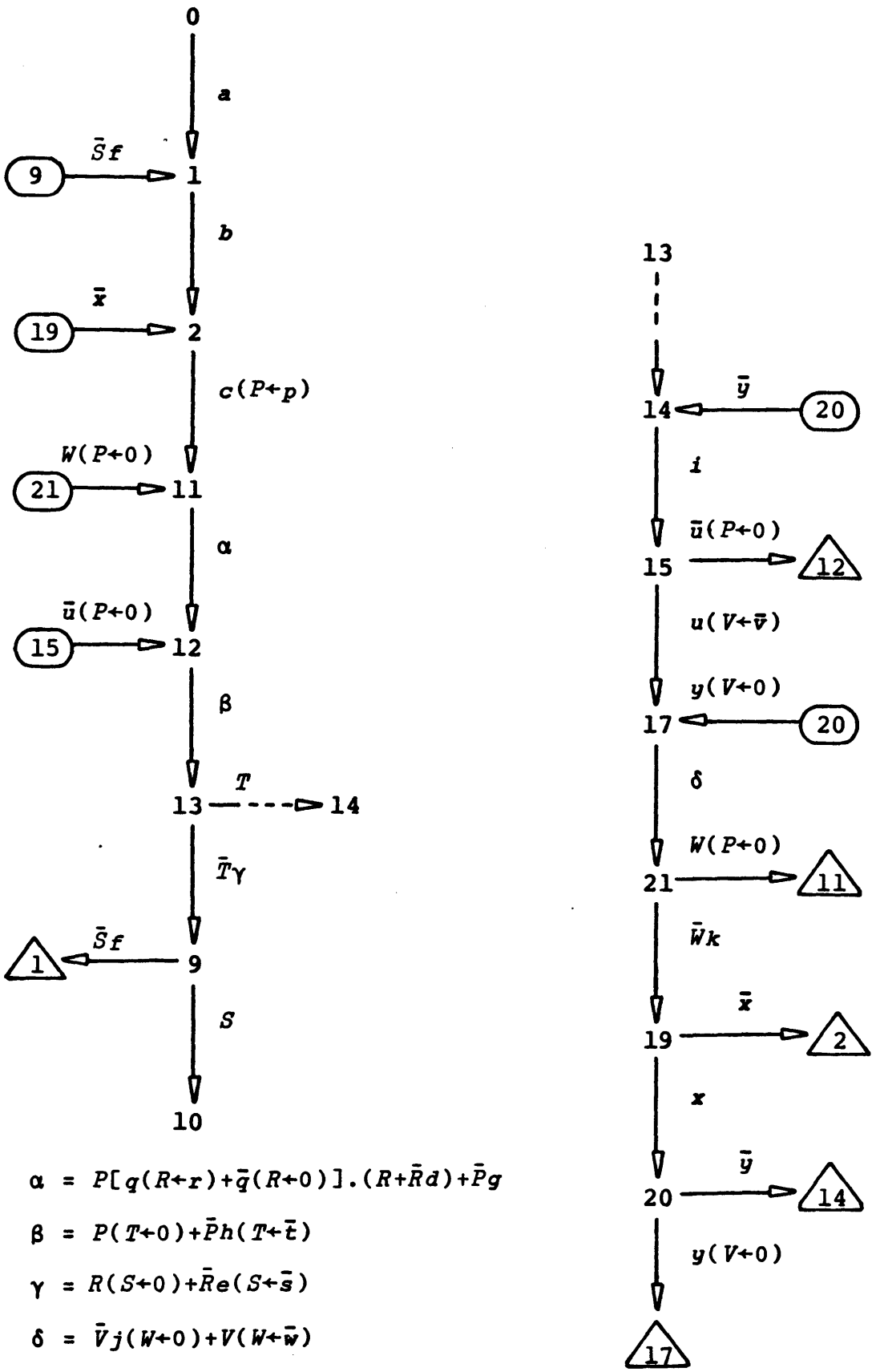

$\hat{T}$
0
a
$\bar{S}f$
9
1
b
$\bar{x}$
19
2
c(P←p)
W(P←0)
21
11
α
$\bar{u}$(P←0)
15
12
β
T
13
14
$\bar{T}\gamma$
$\bar{S}f$
1
9
S
10
13
$\bar{y}$
14
20
i
$\bar{u}$(P←0)
15
12
u(V←$\bar{v}$)
y(V←0)
17
20
δ
W(P←0)
21
11
$\bar{W}k$
$\bar{x}$
19
2
x
$\bar{y}$
20
14
y(V←0)
17
α = P[q(R←r)+$\bar{q}$(R←0)].(R+$\bar{R}$d)+$\bar{P}$g
β = P(T←0)+$\bar{P}$h(T←$\bar{t}$)
γ = R(S←0)+$\bar{R}$e(S←$\bar{s}$)
δ = $\bar{V}$j(W←0)+V(W←$\bar{w}$)

Figure 21.

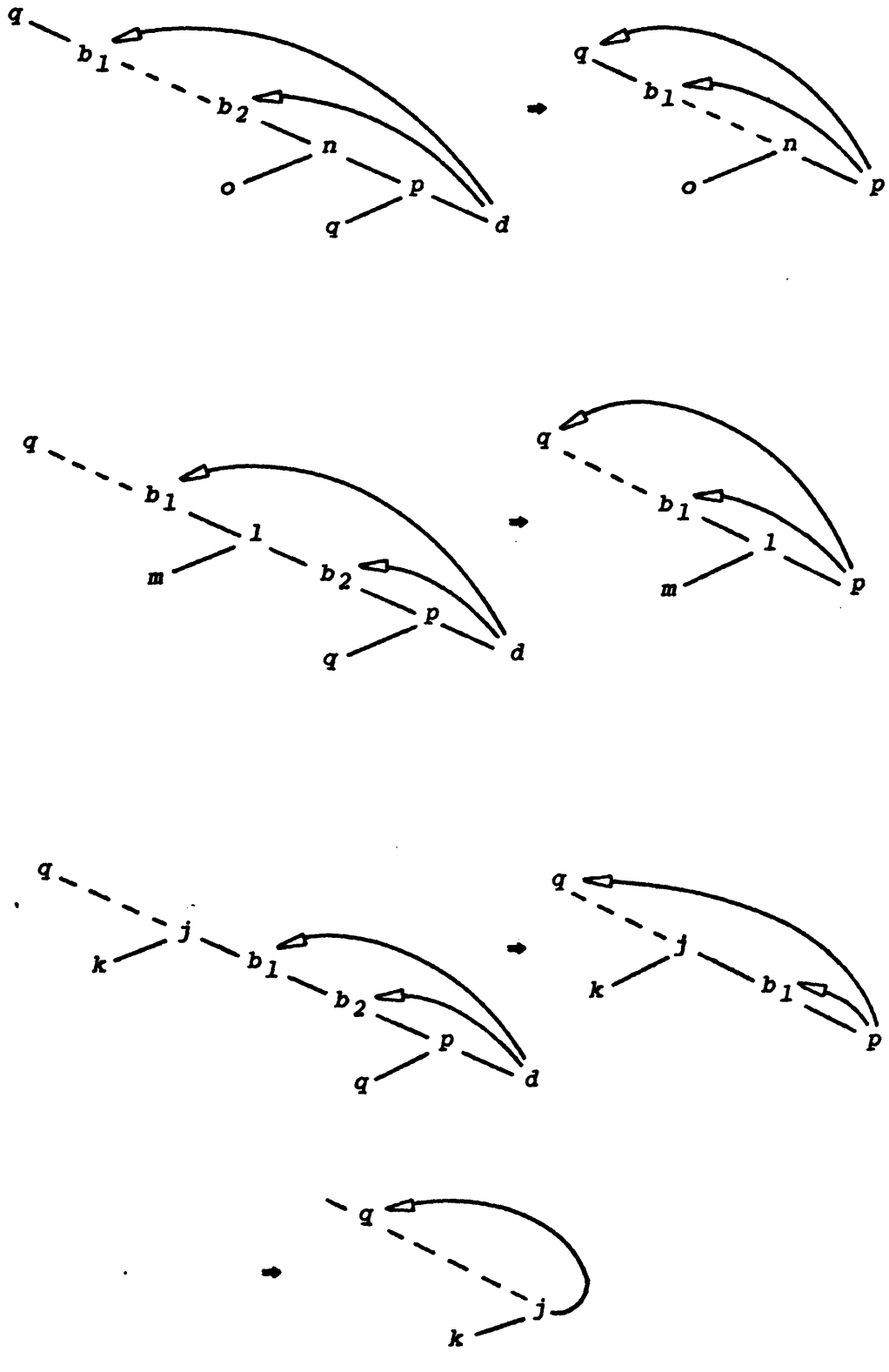
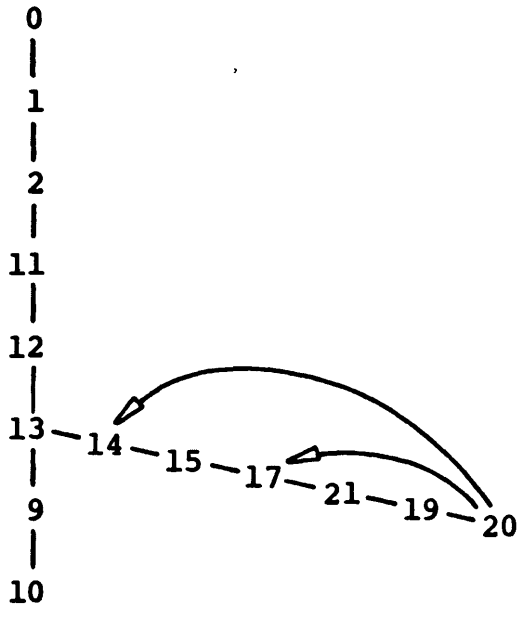
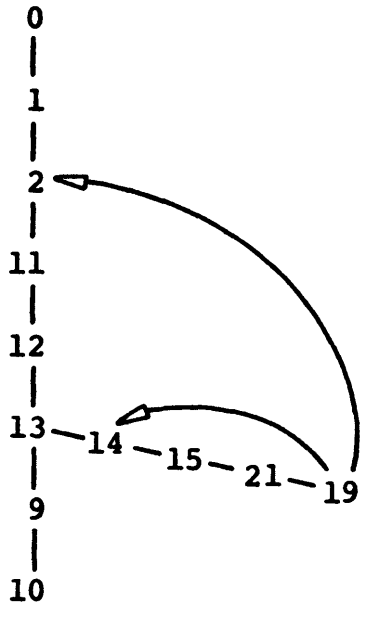

q
b1
b2
n
o
p
q
d
q
b1
n
o
p
q
b1
l
m
b2
p
q
d
q
b1
l
m
p
q
j
k
b1
b2
p
q
d
q
j
k
b1
p
q
j
k

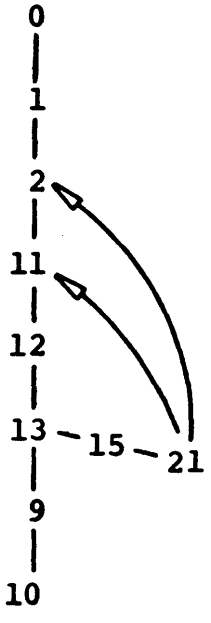
Figure 22.



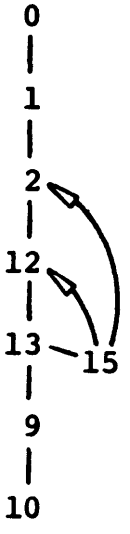
(i)



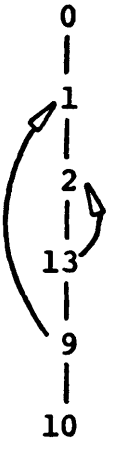
(ii)



(iii)

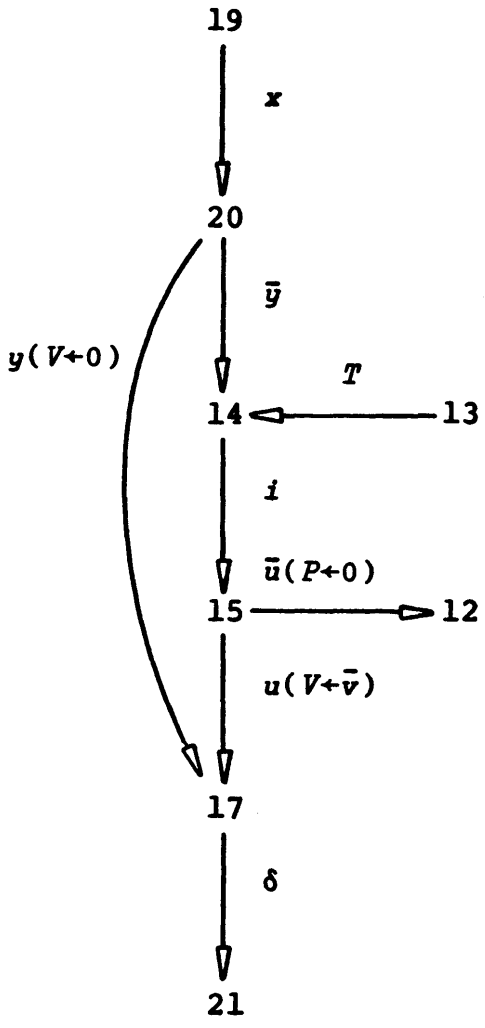


(iv)



(v)

Figure 23.



$$\varepsilon = Y(V\leftarrow 0)(U\leftarrow 0) + \bar{Y}i(U\leftarrow\bar{u})[U+\bar{U}(V\leftarrow\bar{v})]$$

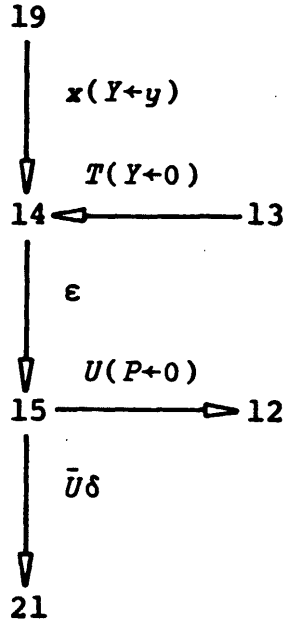
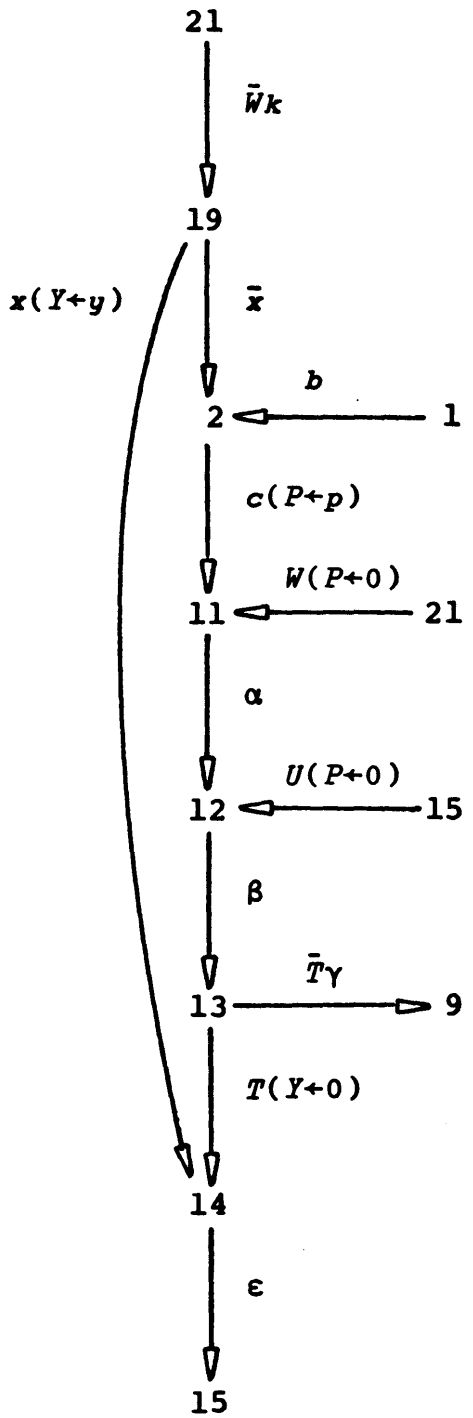


Figure 24.



$$\zeta = X(Y\leftarrow y)(\bar{T}\leftarrow 0)+ \bar{X}\beta[\bar{T}+T(Y\leftarrow 0)]$$

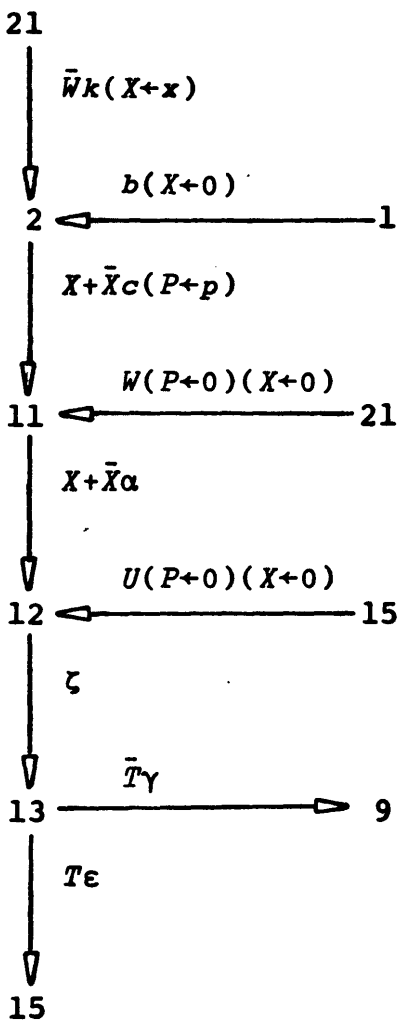
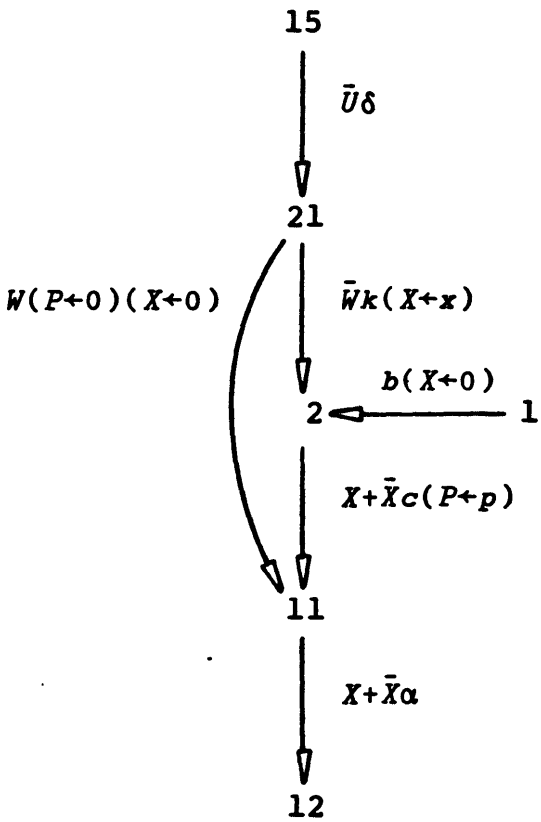


Figure 25.



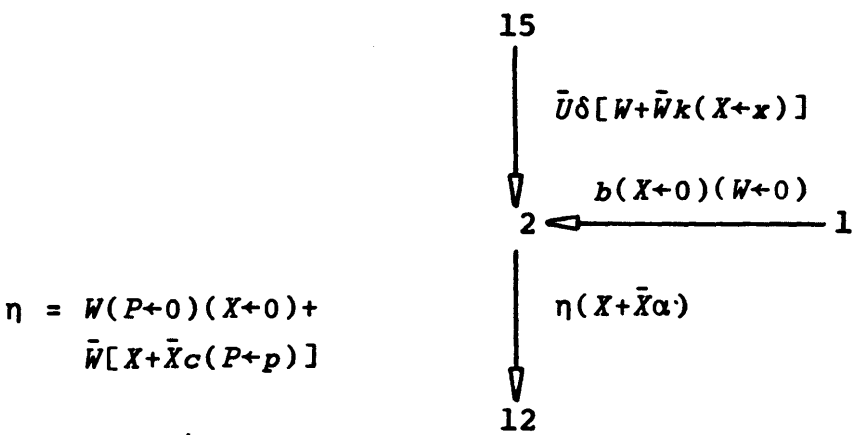
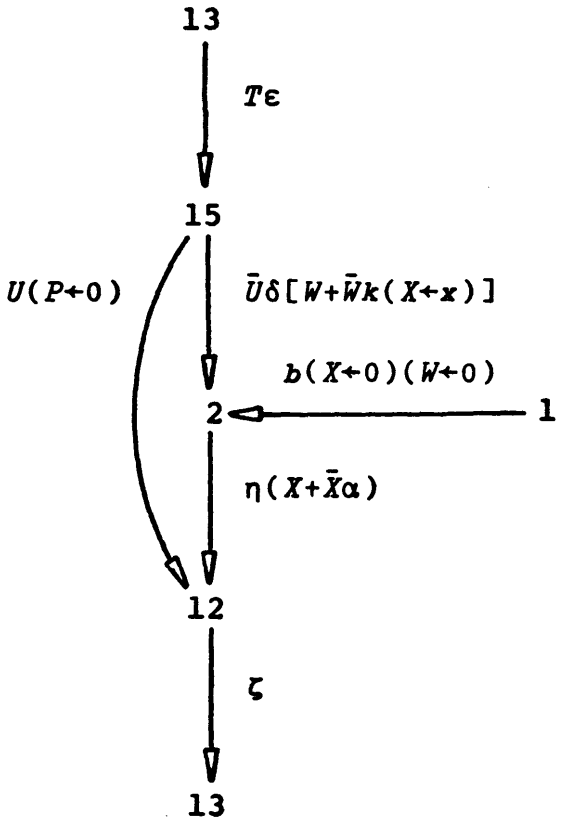


Figure 26.

UNIVERSITY COLLEGE Library



$\theta = U(P\leftarrow 0)+\bar{U}\eta(X+\bar{X}\alpha)$

$\kappa = U+\bar{U}\delta[W+\bar{W}k(X\leftarrow x)]$

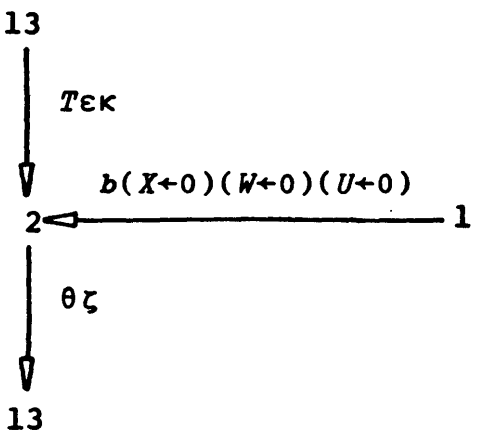


Figure 27.

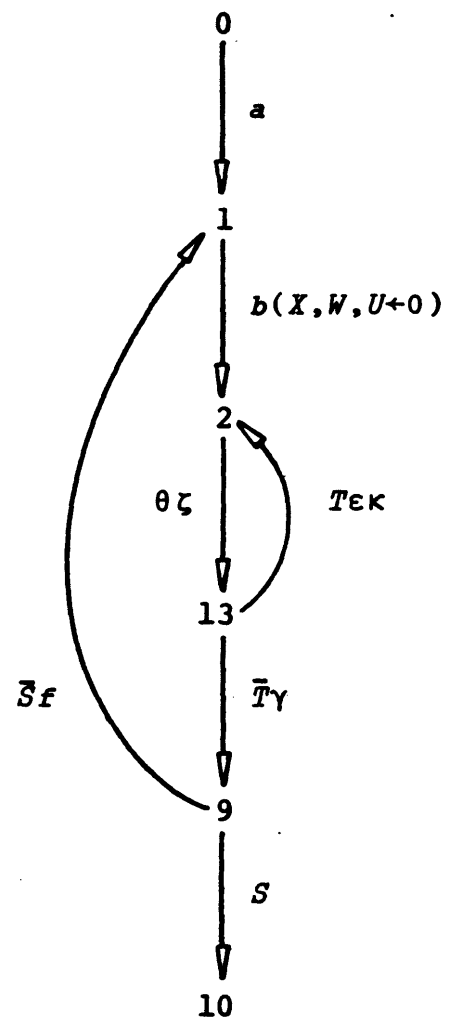


Figure 28.

APPENDIX 1.

Proof of the forward path algorithm

Let $G = (V,\Gamma)$, let $G_F = (V_F,E_F)$ or (V_F,Γ_F) be an acyclic subschema of G, with halt set H, and let U represent the set of vertices whose membership of any acyclic path terminating in H is yet to be determined. Define

I_0: $(U \subset V) \wedge (V_F \subseteq V) \wedge (U \cap V_F = \phi) \wedge (H \subseteq V_F)$

I_1: $(\forall x,y.x,y \in V_F).[(x,y \in \Gamma_F^{-*} H) \wedge (y \in \Gamma_F x \rightarrow x \notin \Gamma_F^{*} y)]$

I : $I_0 \wedge I_1$

Invariant I asserts that G_F is an acyclic path terminating at H and is a subschema of G. Define

dfs: $V \rightarrow 2^V \times 2^E$

such that

dfs(u) →

```
1:  U ← U - {u};
2:  for all v ∈ Γu∩U do
3:      dfs(v);
4:  for all v ∈ Γu∩V_F do
5:      (V_F,E_F) ← (V_F∪{u}, E_F∪{(u,v)});
6:
```

It is asserted that $I \wedge (u \in U)$ {dfs(u)} adds to G_F all paths $[u,v]$ where $v \in V_F$ and no other vertex on $[u,v]$ is in V_F, and preserves the acyclicity of G_F.

The proof is by induction. Let the predicates at labels 1, 2, ··· in the algorithm be denoted by P_1, P_2, ··· . There are three mutually exclusive possibilities to consider: $v \in \Gamma u \cap V_F$, $(v \in \Gamma u) \wedge (v \notin U \cup V_F)$, and $v \in \Gamma u \cap U$. Consideration of the first two forms

the basis step in the proof, and that of the third the induction step.

Basis.

(i) Consider all v such that $v \in \Gamma u \cap V_F$. Then

$$P_1: \quad I \wedge (u \in U) \vdash u \notin V_F$$
$$P_2: \quad I \wedge (u \notin U \cup V_F) \wedge (v \in \Gamma u \cap V_F) \vdash (v \notin U) \wedge (u \neq v).$$

As v is not in U, statement 3 is not executed, so

$$P_4: \quad P_2.$$

P_4 includes the precondition for statement 5 to be executed, so

$$P_5: \quad P_4 \vdash (u \neq v) \wedge (\forall y.y \in V_F)(u \notin \Gamma_F^* y) \vdash (v \in V_F) \wedge (u \notin \Gamma_F^* v) \wedge (v \in \Gamma u).$$

P_5 asserts that arc (u,v) exists in G, that it is not a loop, and that there are no paths from any vertex of G_F to u. Thus adding (u,v) to E_F and u to V_F in statement 6 cannot introduce a cycle. Hence

$$P_6: \quad I \wedge (u \notin U) \wedge (\forall v.v \in \Gamma u \cap V_F)(v \notin U).$$

The first conjunctive term establishes the invariance of I, whilst the second and third terms state that u and its successors in V_F are left by $dfs(u)$ in $V - U$, that is, their membership or otherwise of G_F is established by $dfs(u)$. In this case, of course, u and its successors in V_F are left in V_F.

(ii) Consider all v such that $(v \notin U \cup V_F) \wedge (v \in \Gamma u)$, that is, v is a successor of u and is known not to be in G_F. Then

$$P_1: \quad I \wedge (u \in U) \vdash (u \notin V_F)$$
$$P_2: \quad I \wedge (u \notin U \cup V_F) \wedge (v \in \Gamma u) \wedge (v \notin U \cup V_F) \vdash (v \notin \Gamma u \cap U) \wedge (v \notin \Gamma u \cap V_F).$$

Statements 3 and 5 are not executed as their preconditions are not satisfied. Hence

$$P_6: \quad I \wedge (u \notin U) \wedge (\forall v.v \in \Gamma u-(U \cup V_F))(v \notin U)$$

Again the invariance of I is established, with u and any of its immediate successors not in either U or V_F being left as non-members of U.

Induction.
For all v such that $v \in \Gamma u \cap U$ assume that

$$I \wedge (v \in U) \ \{dfs(v)\} \ I \wedge (v \notin U) \wedge (\Gamma v \cap U = \phi).$$

Then

$$P_1: \quad I \wedge (u \in U) \vdash u \notin V_F$$
$$P_2: \quad I \wedge (u \notin U \cup V_F) \wedge (v \in \Gamma u \cap U) \vdash (u \neq v)$$

Since the precondition for statement 3 is satisfied

$$P_3: \quad I \wedge (v \in U) \wedge (u \notin U \cup V_F) \wedge (u \neq v).$$

Before inferring P_4 it is necessary to show that $dfs(x)$ terminates for all x, $x \in U$. To see that it does, note that a call on $dfs(x)$ immediately deletes x from U thus ensuring that $dfs(x)$ cannot be called more than once. Since U is a finite set, the number of calls on *dfs* is also finite. As both *for* statements in *dfs* have a limited range, *dfs* must terminate. Further, vertex w not in U prior to a call on *dfs* cannot ever become an argument to later calls on *dfs*. Thus $w \notin U$ is invariant over *dfs*. Hence from the induction hypothesis

$$P_4: \quad I \wedge (\forall v.v \in \Gamma u)[(v \notin U) \wedge (\Gamma v \cap U = \phi)] \wedge (u \notin U \cup V_F).$$

Consider all v such that $v \in \Gamma u \cap V_F$. Then

$$P_5: \quad P_4 \wedge (v \in \Gamma u \cap V_F) \vdash (u \neq v) \wedge (v \in \Gamma u) \wedge (v \in V_F) \wedge (u \notin \Gamma_F^* v)$$

which after statement 5 gives

$$P_6: \quad I \wedge (u \notin U) \wedge (\forall v.v \in \Gamma u \cap V_F). \ [(v \notin U) \wedge (\Gamma v \notin U)]$$

Returning to P_4, consider the only possible remaining condition for v, namely that $(v \notin U \cup V_F) \wedge (v \in \Gamma u)$. As the precondition for statement 5 is not satisfied

$$P_6: \quad P_4 \vdash$$
$$I \wedge (u \notin U) \wedge (\forall v.v \in \Gamma u - V_F).$$
$$[(v \notin U) \wedge (\Gamma v \notin U)].$$

Combining the results for P_6 over all $v.v \in \Gamma u$ is finally obtained

$$P_6: \quad I \wedge (u \notin U) \wedge (\Gamma u \notin U).$$

Since $dfs(u)$ terminates only after all calls $dfs(v)$, $v \in \Gamma u$ have terminated, it follows by induction from the third conjunctive term of P_6 that

$$I \wedge (u \in U) \quad \{dfs(u)\} \quad I \wedge (\Gamma^* u \notin U). \quad \square$$

REFERENCES

1. J. Bruno and K. Steiglitz, The expression of algorithms by charts, *Journal of the ACM* 19 517-525 (Oct. 1972).
2. C. Boehm and G. Jacopini, Flow diagrams, Turing Machines and languages with only two formation rules, *Communications of the ACM* 9 366-371 (Sep. 1966).
3. E. Ashcroft and Z. Manna, The translation of 'GOTO' programs to 'WHILE' programs, in *Information Processing 71*, ed. by C.V. Freiman, Vol. 1, pp. 250-255. Amsterdam, North-Holland (1972).
4. D.E. Knuth and R. W. Floyd, Notes on avoiding GOTO statements, *Information Processing Letters* 1 23-31 (1971); corrections 1 177 (1972).
5. H.D. Mills, Mathematical foundations for structured programming, Federal Systems Division, IBM Corp., Gaithersburg, MD, FSC 72-6012 (1972).
6. T. Kasai, Translatability of flowcharts into While programs, *Journal of Computer and Systems Sciences* 9 177-195 (Oct. 1974).
7. M.H. Williams, Genarating structured flow diagrams: the nature of unstructuredness, *The Computer Journal* 20 45-50 (Feb. 1977); 20 381-383 (Nov. 1977).
8. M.H. Williams and H.L. Ossher, Conversion of unstructured flow diagrams to structured form, *The Computer Journal* 21 161-167 (May 1978).
9. G. Oulsnam, Cyclomatic Numbers do not measure complexity of unstructured programs, *Information Processing Letters* 9 207-211 (Dec. 1979).
10. D.C. Cooper, Boehm and Jacopini's reduction of flow charts, *Communications of the ACM* 10 263, 463 (1967).

11. E.S. Bainbridge, Minimal while programs, in *Lecture Notes in Computer Science*, ed. by A. Mazurkiewicz, Vol. 45 pp. 180-186. Springer-Verlag, Berlin (1976).
12. W.W. Paterson, T. Kasami and N. Tokura, On the capabilities of While, Repeat and Exit statements, *Communications of the ACM* 16 503-512 (Aug. 1973).
13. S.R. Kosaraju, Analysis of structured programs, *Journal of Computer and Systems Sciences* 9 232-255 (Dec. 1974).
14. W.A. Wulf, Programming without the GOTO, in *Information Processing 71* Vol. 1, pp. 408-413, ed. by C.V. Freiman, North-Holland, Amsterdam (1972).
15. H. Ledgard and M. Marcotty, A genealogy of control structures, *Communications of the ACM* 18 629-639 (Nov. 1975).
16. J.C. Cherniavsky, J. Keohane and P.B. Henderson, A note concerning top down program development and restricted exit control structures, *Information Processing Letters* 9 8-12 (Jul. 1979).
17. G. Urschler, Automatic structuring of programs, *IBM Journal of Research and Development* 19 181-194 (Mar. 1975).
18. B.S. Baker, An algorithm for structuring flow-graphs, *Journal of the ACM* 24 98-120 (Jan. 1977).
19. E. Engeler, Structure and meaning of elementary programs, in *Lecture Notes in Mathematics*, ed. by E. Engeler, Vol. 188, pp. 89-101. Springer-Verlag Berlin (1971).
20. E. Wegner, Tree-structured programs, *Communications of the ACM* 6 704-705 (Nov. 1973).
21. T.J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* SE-2 308-320 (Dec. 1976).
22. E.W. Dijkstra, Go To statement considered harmful, *Communications of the ACM* 11 147-148, 538, 541 (Mar. 1968).

23. D.E. Knuth, Structured programming with go to statements, *ACM Computing Surveys* 6 261-301 (Dec. 1974).

24. H. Partsch and R. Steinbrügger, Program transformation systems, *ACM Computing Surveys* 15 199-236 (Sep. 1983).

25. M.H. van Emden, Programming with verification conditions, *IEEE Transactions on Software Engineering* SE-5 148-159 (Mar. 1979).

26. D.C. Luckham, D.M.R. Park and M.S. Paterson, On formalized computer programs, *Journal of Computer and Systems Sciences* 4 220-249 (Aug. 1970).

27. S.C. Kleene, Representation of events in nerve sets, in *Automata Studies*, ed. by C.E. Shannon and J. McCarthy, pp. 3-40. Princetown University Press, Princeton, New Jersey (1956).

28. P.J. Denning, J.B. Dennis and J.E. Qualitz, *Machines, Languages and Computation*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).

29. C.A.R. Hoare, An axiomatic basis for computer programming, *Communications of the ACM* 12 576-580 (Oct. 1969).

30. R.J. Lipton, S.C. Eisenstat and R.A. DeMillo, Space and time hierarchies for classes of control structures and data structures, *Journal of the ACM* 23 720-732 (Oct. 1976).

31. R.A. DeMillo, S.C. Eisenstat and R.J. Lipton, Space-time tradeoffs in structured programming: an improved combinatorial embedding theorem, *Journal of the ACM* 27 123-127 (Jan. 1980).

Unravelling Unstructured Programs

G. Oulsnam*
Department of Computer Science, University of Queensland, St. Lucia, Queensland 4067, Australia

A method is presented for converting unstructured program schemas to strictly equivalent structured form. The predicates of the original schema are left intact with structuring being achieved by the duplication of the original decision nodes without the introduction of compound predicate expressions, or, where possible, by function duplication alone. It is shown that structured schemas must have at least as many decision nodes as the original unstructured schema, and must have more when the original schema contains branches out of alternation constructs. The structuring method allows the complete avoidance of function duplication, but only at the expense of decision node duplication. It is shown that structured schemas always require an increase in space–time requirements, and it is suggested that this increase can be used as a complexity measure for the original schema.

1. INTRODUCTION

This paper presents a method for transforming unstructured program flowgraphs into structured equivalents in D-chart format.[1] The form of the derived structured programs is such that the original unstructured programs can be easily recovered, thus revealing what overheads in space and time are inherent in the structured forms. The method enables the user to opt for minimization of time overheads, minimization of space overheads, or some intermediate compromise. A measure for the introduced overheads is given which can be used to compare the relative conceptual complexities of unstructured programs. A feature of the structuring method is that the number of introduced auxiliary Boolean variables, or flags, is kept to a minimum, and where such flags are introduced, they correspond exactly to some conditional expression, or predicate, in the original program. Thus the method preserves as far as possible the logic of the original program.

The problem of transforming flowgraphs into some standard form has been widely addressed in the literature. Methods based on yielding a flowgraph in while-program form have been given by Jacopini,[2] Ashcroft and Manna,[3] Knuth and Floyd,[4] Bruno and Steiglitz,[1] Mills,[5] Kasai,[6] Williams,[7] Williams and Ossher[8] and Oulsnam.[9] Jacopini's method was shown by Cooper[10] to yield in a trivial way a flowgraph consisting of a single while statement enclosing a sequence of alternations based on introduced auxiliary variables. Jacopini's conjecture that in general auxiliary variables would be necessary to transform arbitrary flowgraphs into D-chart form was proved by Ashcroft and Manna,[3] Knuth and Floyd[4] and Bruno and Steiglitz.[1] Kasai[6] and Bainbridge[11] describe methods of reducing while-programs to minimal form, while the general capabilities and limitations of D-charts as a standard form were considered by Paterson, Kasami and Tokura[12] and Kosaraju.[13]

The necessity for auxiliary variables, coupled with the fact that flowgraphs in while-program form were shown by Paterson *et al.*[12] to generally require some duplication of basic functions or predicates of the original flowgraph, has led to consideration of more general standard forms than D-charts. Wulf[14] proposed the use of multi-level control structures and further generalizations were analysed by Kosaraju[13] and Ledgard and Marcotty,[15] with some refinements of their results by Cherniavsky *et al.*[16] Proposals based on the non-duplication of the original flowgraph's functions and predicates have been given by Urschler[17] (using a technique based on back-dominators of the original flowgraph), and Baker.[18] Of necessity both methods allow the use of GOTO statements although Ref. 17 restricts these to backward jumps only. A standard form based on binary trees has been proposed by Engeler[19] and Wegner.[20] The former allows only jumps to ancestor nodes in the tree, while the latter allows jumps in both directions. Proposals to convert flowgraphs to recursive form have been made by Knuth and Floyd[4] and Urschler.[17] McCabe[21] and Williams[7] independently identified the basic forms of unstructuredness, and transformations based on the identification and elmination of these constructs have been given by Williams,[7] Williams and Ossher[8] and Oulsnam.[9] Dijkstra,[22] echoed by Knuth,[23] has cautioned against expecting mechanical transformation of flowgraphs to yield more comprehensible programs, while Knuth[23] has examined the problem of efficiency relating to programs translated to standard—or structured—form. Van Emden[24] has dismissed the need for structured programming altogether and proposes a method for deriving programs directly with minimal function and predicate duplication.

The remainder of this paper is organized as follows. Section 2 introduces some necessary definitions and concepts, Section 3 briefly reviews the basic forms of unstructuredness in flowgraphs while in Section 4 a method for their removal based on structured transforms is given. The proof of the effectiveness of the structuring algorithm is given in Section 5. Section 6 contains an example of the use of the method and the paper concludes in Section 7 with a discussion on the space–time efficiency of the structuring transforms.

2. SCHEMAS

The method of structuring to be introduced in Section 4 involves transformations on program flowgraphs or

*Present address: Department of Computer Science, University College, Cork, Eire.

CCC-0010-4620/82/0025-0379 $04.50

© Heyden & Son Ltd, 1982

schemas.[25] This Section briefly reviews schemas and an associated algebra for describing them.

A schema shows the control structure of the program whilst leaving the details of the program's computation to be defined as an interpretation of the schema. A schema therefore represents a family of distinct programs sharing a common control structure. Each program of a schema is considered to operate on three types of variables:

input variables	$x1, \ldots, xa$
local variables	$y1, \ldots, yb$
output variables	$z1, \ldots, zc$

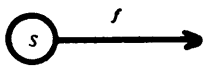
which are represented collectively by X, Y and Z respectively. The operations of the program on these variables are of two types:

functions	$f1(X, Y), \ldots, fm(X, Y)$
predicates	$p1(X, Y), \ldots, pm(X, Y)$

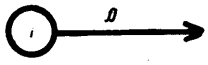
Functions map their arguments into either Y or Z, while predicates map theirs into {true, false}. The composition of functions such as $fi(X, fj(X, Y))$ is denoted by $fj(X, Y).fi(X, Y)$, where the full point (.) denotes the sequencing operator. The logical negation of a predicate $pi(X, Y)$ is denoted by $\bar{p}i(X, Y)$. For both functions and predicates the argument list usually will be elided.

The specification of the variables, functions and predicates for a particular program is called an interpretation of the schema. The transformational structuring process described in this paper is independent of such interpretations.

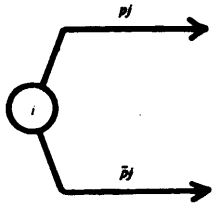
Schemas are constructed by composition of the following statements.

START: 

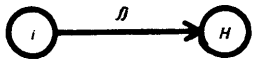
which is to be understood as an abbreviation for S: $Y := f(X)$, where S is a program (node) label.

ASSIGN: 

which denotes i: $Y := fj(X, Y)$

TEST: 

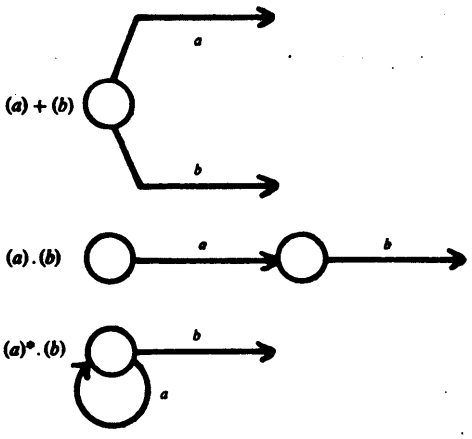
which denotes i: IF $pj(X, Y)$ GOTO the left branch target node ELSE GOTO the right branch target node.

HALT: 

which denotes i: $Z := fj(X, Y)$

Every schema consists of exactly one START and one HALT statement, and any number of uniquely labelled ASSIGN and TEST statements such that every statement lies on some path from START to HALT. The node of a TEST statement is called a decision node, and that of an ASSIGN statement a collecting node.

In addition to the geometrical representation of schemas it is advantageous to have an algebraic representation as an aid to the transformational process. Following Kleene[26] it is known that the computations associated with a flowgraph schema can be represented by a regular set. The regular expressions of the set are derived by regarding the schema as a finite state generator (fsg) whose states correspond to the nodes of the flowgraph and whose transitions correspond to traversal of flowgraph edges. Each transition causes the function or predicate identifier of the corresponding edge to be appended to the fsg's output string. Any string output by the fsg in going from the START node to the HALT node represents a possible computation sequence of the schema, and the set of all such strings represents the schema's computation sequence set. The regular set operators of union (+), concatenation (.) and Kleene star closure (*) are related to schema operations and statements as follows:



Here, 'a' and 'b' denote strings of function and predicate identifiers and the star closure postfix operator means 'concatenated zero or more times'. The string grouping operators (,) will be elided where their omission does not cause ambiguity.

A complete forward path in a schema is any path that begins at the START node and ends at the HALT node without going through the same node twice. An edge of the flowgraph on some complete forward path is called a forward edge, and an edge which is not a forward edge is called a backward edge. Any path which does not include a backward (forward) edge is called a forward (backward) path.

The end set $E(i, j)$ of a node 'i' with respect to a not necessarily distinct node 'j' is defined[27] as the set of strings that the fsg would output in traversing all paths from 'i' to 'j'. Thus $E(S, H)$ is the computation sequence

© Heyden & Son Ltd, 1982

set of the schema. For brevity E(i, H) will be written E(i). By convention E(H) = ( ), the empty string.

Structured schemas

A structured regular expression (sre) is defined recursively as follows:

(1) Functions and predicates are sre's.
(2) If x and y denote any two sre's and p is a predicate then the following are also sre's: (a) a sequence $x.y$ (b) a decision $(p.x + \bar{p}.y)$ (c) a loop $(x.p.y)^*.x.\bar{p}$ or equivalently $x.(p.y.x)^*.\bar{p}$.

The familiar WHILE–DO construct is a special case of the loop in which x = ( ), whilst the REPEAT–UNTIL construct is obtained by setting y = ( ) instead. For loops, $x.p$ is a forward path with respect to the loop's entry and exit nodes whilst $p.y$ is a backward path. A loop here is what Dijkstra reportedly[23] termed a $n + \frac{1}{2}$ loop.

It is to be noted that whilst the flowgraph for a loop contains only one instance each of x and y, the corresponding regular expressions given in 2(c) above each contain two occurrences of x. In fact the second of the two expressions can be written in programming terms as:

x; WHILE p DO BEGIN y; x END;

showing that it is always possible (but only at the expense of duplicating the function on the forward path) to express a loop in terms of the WHILE... DO construct. Throughout this paper the $(n + \frac{1}{2})$ loop is taken as the terminal form for a structured loop since it contains both the WHILE... DO and REPEAT... UNTIL constructs as special cases and, as just seen, can always be converted to WHILE... DO format if so desired.

A schema is structured if and only if its computation sequence set E(S) is a sre.

3. THE BASIC FORMS OF UNSTRUCTUREDNESS

There are six basic unstructured forms (buf's) that can occur in a schema: jump into a decision—ID; jump out of a decision—OD; jump into the forward path of a loop—IL; jump out of a forward path of a loop—OL; jump into the backward path of a loop—IB; and jump out of the backward path of a loop—OB. These are depicted in Fig. 1. The last two, IB and OB, are additional to the forms considered by McCabe.[21] Referring to Fig. 1, there are three possible placings for the node E: on a path from the START node S to node A; on a path from node C to the HALT node H; on a path from S to H which does not include nodes A, B or C. Analysis of all possible placings of node E with respect to each of the six buf's shows that unstructuredness always occurs in possibly overlapping combinations of the six unstructured subgraphs depicted and named in Fig. 2, and that none of the basic forms can ever occur by itself. For example, the schema of Fig. 7 comprises one instance each of LD, DL and LL. McCabe[21] and Williams[7] independently derived the forms described here as DD, DL, LD and LL. Williams[7] added a fifth form called parallel loops but, as he recognized, this form is

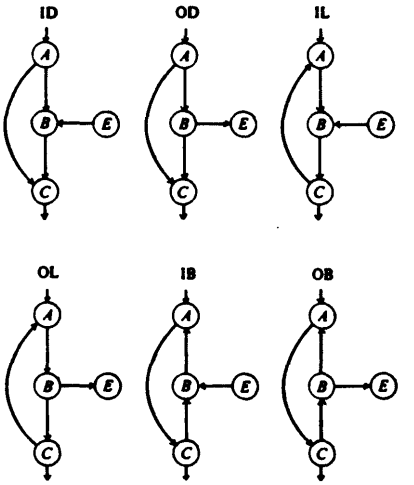


Figure 1. The six basic unstructured forms.

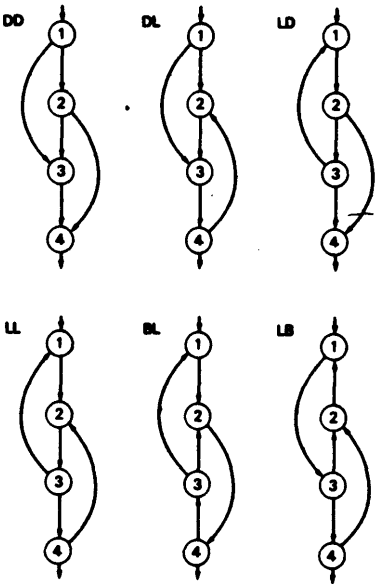


Figure 2. The six forms of unstructuredness.

expressible in terms of the other four under the restriction of a single HALT node.

Examination of each of the six forms DD... LB of Fig. 2 reveals that they are constructed from pairs chosen from the basic forms ID, OD, IL and OL. Thus DD = ID + OD; DL = ID + IL; LD = OD + OL; LL = IL + OL; BL = ID + OL; LB = OD + IL. (For instance, for

© Heyden & Son Ltd, 1982

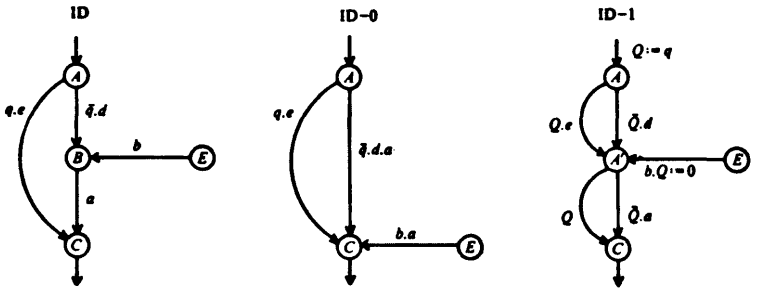


Figure 3. Jump into a decision and its structured forms.

BL the ID component is obtained with node 1 equivalent to B, node 2 to C, node 3 to A and the immediate predecessor of node 1 to E. The OL component comprises the loop 2–4–3–2 with node 2 equivalent to A, node 3 to C, node 4 to B and the immediate successor node of the BL construct to E.) From this it follows that it is sufficient to consider just ID, OD, IL and OL as the basic units of unstructuredness whose removal will result in a structured schema. In fact, since none of these can occur alone in a schema, it is sufficient to consider any three of them as the minimum set for removal.

4. THE STRUCTURING TRANSFORMS

Two schemas having identical functions and predicates are computationally equivalent if their computation sequence sets are described by the same regular set. The first step in the structuring process is therefore to recast the regular expressions describing the buf's into sre formats. The strategy for transforming an unstructured schema into structured form is then as follows. (1) Identify a buf and replace it with a computationally equivalent but structured subgraph. (Since buf's cannot occur alone, a second buf will also be removed.) (2) Repeat the process until a structured schema is obtained.

In this Section the structured equivalents of the buf's are derived, whilst a proof that the structuring procedure can always be applied and will always terminate is given in Section 5.

Consider first ID, Fig. 3, for which it is required to find an sre for $E(A) + E(E)$. From Fig. 3 it is seen that

$$\begin{aligned} E(A) &= q.e.E(C) + \bar{q}.d.E(B) \\ E(E) &= b.E(B) \\ E(B) &= a.E(C) \end{aligned} \quad (1)$$

Bainbridge[11] has given three rules for solving end set equations to yield sre's. Letting x, y denote sre's and p a predicate these are:

1. if $E(v) = x.E(u)$ or $E(v) = x$, then eliminate $E(v)$ by substitution;
2. if $E(v) = p.x.E(u) + \bar{p}.y.E(u)$ then deduce $E(v) = (p.x + \bar{p}.y).E(u)$;
3. if $E(v) = p.x.E(v) + \bar{p}.y.E(u)$ or $E(v) = p.x.E(v) + \bar{p}.y$ then deduce $E(v) = (p.x)^*.\bar{p}.y.E(u)$ or $E(v) = (p.x)^*.\bar{p}.y$ respectively.

Bainbridge asserts that if application of these rules yields a sre then the sre is minimal with respect to a count of the number of occurrences of functions and predicates, but if a stage is reached where none of the rules can be applied then there is no sre solution. Applying these rules to the end set equations for ID to eliminate $E(B)$ gives:

$$\begin{aligned} E(A) &= (q.e + \bar{q}.d.a).E(C) \\ E(E) &= b.a.E(C) \end{aligned} \quad (2)$$

which is in the required sre format. However, unlike Eqns (1), Eqns (2) contain one duplication of identifier 'a'. The flowgraph corresponding to Eqns (2) is shown in Fig. 3 as ID-0, the 0 denoting no duplication of the predicate 'q'.

For IL, Fig. 4, the end set equations are:

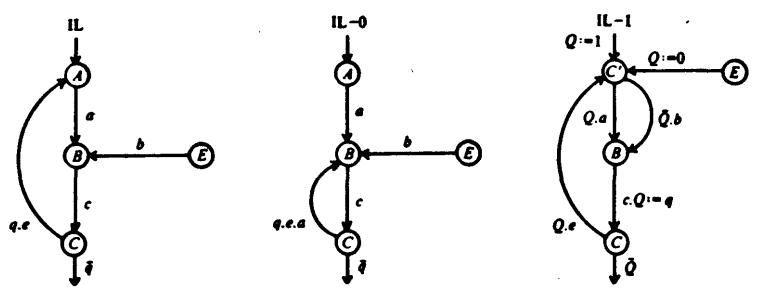


Figure 4. Jump into a loop and its structured forms.

© Heyden & Son Ltd, 1982

$$E(A) = a.E(B)$$
$$E(E) = b.E(B)$$
$$E(B) = c.E(C)$$
$$E(C) = q.e.E(A) + \bar{q}.$$

Eliminating $E(B)$ and $E(C)$ gives

$$E(A) = a.c.(q.e.E(A) + \bar{q})$$

from which can be shown

$$E(A) = a.(c.q.e.a)^*.c.\bar{q}$$
$$= a.E(B'), \text{ say,}$$

and

$$E(E) = b.E(B')$$

to give IL-0, Fig. 4. (Actually IL-0 can be obtained directly by simply substituting for $E(A)$ in the equation for $E(C)$.) Again it has been necessary to duplicate just edge 'a' to achieve sre format.

Now consider OD, Fig. 5. In this case a sre for $E(A)$ is required since B–E is an outgoing edge from the decision. The end set equations are:

$$E(A) = q.e.E(C) + \bar{q}.d.E(B)$$
$$E(B) = p.b.E(E) + \bar{p}.c.E(C) \qquad (3)$$

Since an expression for $E(A)$ is required in terms of $E(C)$ and $E(E)$ it is necessary to eliminate $E(B)$, but none of the Bainbridge rules can be applied to achieve this. Thus OD cannot be structured by function replication alone, so predicate replication must be considered instead.

Predicate replication is achieved by introducing auxiliary predicate variables, or flags, with identifiers distinct from those of the schema's functions, predicates and variables. In order to preserve the schema's computations over its variables, the flags are introduced in the following way. At a TEST statement node, the TEST predicate 'p' is computed as before, but its value is immediately assigned to a flag 'P' uniquely associated with the predicate. It is this flag that is used, rather than the original predicate, as the discriminant in choosing the exit path from the TEST node. In programming terms,

IF p THEN ... ELSE ...

is replaced by

$P := p$; IF P THEN ... ELSE ...

Whereas in the original schema the value of a predicate is known only at its point of computation (since subsequent functions will in general change the values of the predicate's arguments), the introduction of a corresponding flag preserves the predicate's value until that value is recomputed. In the sequel the convention is adopted that schema predicates will be denoted by p, q, r, ... and the corresponding uniquely associated flags by P, Q, R ....

It will also prove convenient to allow direct assignment of truth values to flags. Again, such assignments do not affect the original schema's computations. Bainbridge's rules can now be extended to allow for the introduction of flags as follows, where '1' denotes TRUE, '0' denotes FALSE and '@' denotes the non-existent or null string:

4. if $E(u) = p.a.E(v) + \bar{p}.b.E(w)$ introduce $E(u) = (P := p).(P.a + \bar{P}.b).(P.E(v) + \bar{P}.E(w))$
5. from $(P := 1).(P.x + \bar{P}.y)$ deduce x and from $(P := 0).(P.x + \bar{P}.y)$ deduce y
6. if x, y do not contain assignments to P, then from $P.x.P.y$ deduce $P.x.y$ and from $P.x.\bar{P}.y$ deduce @
7. from $(P + \bar{P}).E(u)$ deduce $E(u)$
8. from @ $.x$ and $x.$ @ deduce @.

In each of rules 4–7, P and $\bar{P}$ can be interchanged.

Returning to the end set Eqns (3) for OD, the term $q.e.E(C)$ can be recast as

$$q.e.(P := 0).(P.E(E) + \bar{P}.E(C))$$

and $E(B)$ as

$$(P := p).(P.b + \bar{P}.c).(P.E(E) + \bar{P}.E(C))$$

to yield the sre:

$$E(A) = (q.e.(P := 0) + \bar{q}.d.(P := p).(P.b + \bar{P}.c)).$$
$$(P.E(E) + \bar{P}.E(C))$$

The corresponding flowgraph is shown in Fig. 5 as OD-1, the '1' indicating that the TEST statement at node B has been duplicated at B'. (In the Figure the assignment $P := p$ has been 'pushed back' onto the A–B path for the sake of giving a slightly simpler flowgraph.)

Now consider OL, Fig. 6. For this buf

$$E(A) = a.p.b.E(E) + a.\bar{p}.c.(q.e.E(A) + \bar{q}).$$

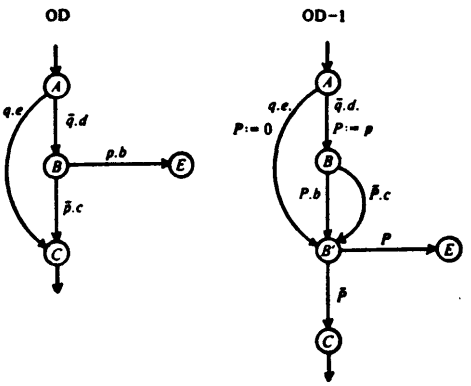


Figure 5. Jump out of a decision.

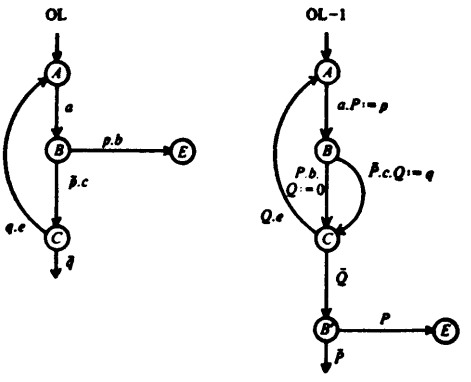


Figure 6. Jump out of a loop.

© Heyden & Son Ltd, 1982

Introducing flags for 'p' and 'q', and after some rearrangement of terms, then

$$E(A) = a.(P := p).(\bar{P}.c.(Q := q).Q.e.E(A) + P.b.E(E) + \bar{P}.c.(Q := q).\bar{Q}).$$

By expansion, the three disjunctive terms in parentheses can be written respectively as:

$$(P.b.(Q := 0) + \bar{P}.c.(Q := q)).Q.e.E(A)$$
$$P.b.(Q := 0).\bar{Q}.P.E(E) \quad \text{and}$$
$$\bar{P}.c.(Q := q).\bar{Q}.\bar{P}$$

Using the expansion formula

$$P.u.P.x + \bar{P}.v.\bar{P}.y = (P.u + \bar{P}.v).(P.x + \bar{P}.y)$$

on the last two expressions, and then collecting terms, gives the sre

$$E(A) = a.(P := p).(P.b.(Q := 0) + \bar{P}.c.(Q := q)).(Q.e.E(A) + \bar{Q}.(P.E(E) + \bar{P}))$$

depicted as OL-1 in Fig. 6, where the original decision node B has been duplicated at B'.

Whilst OD-1 and OL-1 each contain one duplicated decision node, neither contains the duplication of a function. It is also possible to structure both ID and IL without function duplication at the expense of one duplicated decision node as shown by ID-1 and IL-1 in Figs 3 and 4 respectively.

Thus each of the basic unstructured forms OD, ID, IL and OL can be structured at the expense of at most one duplicated decision node and no function duplication, but only ID and IL can be structured by function duplication alone.

As noted in Section 3, the six paradigms of unstructuredness are composed of pairs of basic unstructured forms:

DD = ID + OD; DL = ID + IL; LD = OD + OL;
LL = IL + OL; BL = ID + OL; LB = OD + IL

and all of these except LD can be structured without decision node duplication by a suitable application of either ID-0 or IL-0. However, as LD consists of OD + OL it can only be structured at the expense of one introduced decision node using either OD-1 or OL-1. Since the former requires only one flag to the latter's two, OD-1 is the preferred choice.

It remains to be established under what conditions, if any, the transforms can be applied in the presence of overlapping buf's. This is taken up in the next Section.

5. EFFECTIVENESS OF THE TRANSFORMS

A transform is considered to be effective only if it results in another valid schema and if it gives a reduction in the total number of buf's left in the schema.

In the previous Section it was assumed in the derivation of the structuring transforms that there was no overlap between the buf's. The effect of overlap is to introduce decision or collecting nodes on what would otherwise be edges of buf's, and there is then no guarantee that the structuring transforms can still be applied effectively. In this Section it is shown that whilst some forms of overlap can invalidate certain transforms, nonetheless for every schema there is always at least one transform that can be applied effectively, thus proving that every schema can be progressively transformed into structured format.

Consider ID, Fig. 3, and transform ID-1. The introduction of a collecting node on any of the edges A–B, A–C or B–C of ID gives rise to one additional instance of ID. Application of ID-1 leaves the introduced ID intact, but still effectively removes the original ID. In fact it can be seen from Fig. 3 that ID-1 remains effective in the presence of any number of collecting nodes on the edges of ID and also for any number of introduced decision nodes.

Similarly, ID-0 is effective with respect to introduced nodes on edges A–B and A–C of ID, but, because of edge duplication, not for nodes on B–C. Consider an introduced collecting node B' on B–C, and let B' have an external immediate predecessor E'. The subgraph $AB'CE'$ can be regarded as an ID form with node B a collecting node on the A–C edge for which, as already seen, ID-0 is effective. This argument can be extended to any number of collecting nodes on edge B–C of the original ID. With regard to introduced decision nodes on edge B–C, the use of ID-0 is inappropriate as at best it would give rise to duplicated predicates—precisely what ID-0 was designed to avoid. However, as noted above, ID-1 can be used instead. Hence:

Lemma 1. There is always an effective transform for the removal of ID constructs from any schema.

Consider OD and OD-1, Fig. 5. The presence of collecting nodes on the edges A–B, A–C or B–C introduces instances of ID. From Lemma 1 it is known that these instances can always be removed effectively, so it remains to consider introduced decision nodes alone. It can be seen from Fig. 5 that OD-1 remains effective for decision nodes on A–B and A–C, but not for those on B–C. (Since E can always be chosen to make B–E an edge, there is no need to consider nodes on B–E.) Let B' be the decision node on B–C which has node C as an immediate successor, and let E' be its other immediate successor. Subgraph $AB'CE'$ is now an instance of OD with decision nodes on its A–B' edge, for which OD-1 effective. Hence:

Lemma 2. In the absence of ID constructs there is always an effective transform for the removal of OD constructs.

Lemmas 1 and 2 together assert that it is always possible to transform an unstructured schema to one that contains no instances of ID or OD. Thus it is now necessary to consider schemas containing only IL and OL constructs. Since, as noted in Section 3, neither IL nor OL occur in combination with themselves, the only possible remaining constructs are of the form IL + OL, and the effective removal of one component guarantees the removal of the other.

Consider IL and its transforms, Fig. 4. Any nodes introduced on the B–C or C–A edges leave both IL-0 and IL-1 as effective transforms. Collecting nodes on A–B introduce further instances of IL, and there is always one of these that has no collecting nodes on its corresponding A–B edge. Thus IL-0 and IL-1 can always be applied effectively to this IL construct in the absence of decision nodes on the A–B edge.

It remains only to consider decision nodes on the A–B path. Let these nodes be $B1, \ldots, Bn$ with corresponding external target nodes $E1, \ldots, En$. Since by assumption

© Heyden & Son Ltd, 1982

all instances of ID and OD have been removed, all the edges $B1$–$E1$, ... , Bn–En are backward edges, and together give rise to other instances of IL + OL, that is, of LL. Since the schema is finite, there must be at least one LL that has no backward edge leading out of it. (The first instance of LL encountered on a forward path from START is an example.) But an LL construct which has no such edge cannot have an (introduced) decision node on any of its edges and, in particular, its IL component cannot have an introduced decision node on its A–B edge. As already noted, both IL-0 and IL-1 can be applied effectively to this instance of IL.

Lemma 3. In the absence of ID and OD constructs, there is always an effective transform for the removal of IL constructs, and hence of OL constructs as well.

Lemmas 1–3 taken together lead to the principal result of this paper.

Theorem. It is always possible to put an unstructured schema into a computationally equivalent structured form using only the transforms ID-0, ID-1, OD-1, IL-0 and IL-1.

In fact, as is easily shown, the result can be strengthened to use ID-1, OD-1 and IL-1 as the minimum set if the avoidance of decision node duplication is not a requirement.

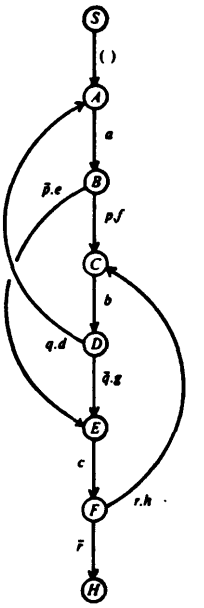


Figure 7. Generalized Flynn's Problem No. 5.

6. AN EXAMPLE

Consider the schema shown in Fig. 7. This is a slightly generalized version of a schema which Tausworthe[28] calls Flynn's Problem No. 5. To recover the original problem from Fig. 7 it is merely necessary to set 'f', 'g' and 'h' to ( ), the empty string. It can be seen that the schema comprises one instance each of LD (OL + OD), DL (ID + IL), and LL (IL + OL). Two structuring strategies might be: (1) avoid function duplication; (2) avoid decision node duplication (as far as possible). For the purposes of illustration the second approach will be chosen here.

Since decision node duplication is to be avoided it is required to apply the Type-0 transforms of ID and IL wherever possible, but the presence of LD in the schema suggests that some decision node duplication is unavoidable.

Consider first the ID construct comprising nodes B, C, E with external node F. The presence of decision node D on path C–E precludes the use of ID-0. Next consider the IL comprising nodes C, E, F with external node B. The decision node D on path C–E again prevents the use of a Type-0 transform—this time IL-0. Next consider the IL comprising nodes A, C, D with external node F. Now it is decision node B on path A–C that prevents the use of IL-0 and so there is no effective Type-0 transform available.

Applying ID-1 removes the DL construct to leave a schema with one LL and one LD for which again no Type-0 transform is effective. Applying ID-1 and finally

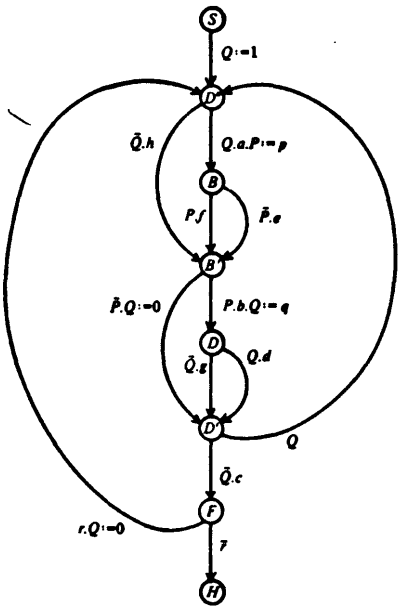


Figure 8. Structured form of Flynn's Problem.

© Heyden & Son Ltd, 1982

IL-1 gives the structured schema shown in Fig. 8. The end set equations for this schema are:

$$E(S) = (Q := 1).E(D^*)$$

$$E(D^*) = (Q.a.(P := p).(P.f + \bar{P}.e) + \bar{Q}.h).$$
$$(P.b.(Q := q).(Q.d + \bar{Q}.g) + \bar{P}.(Q := 0)).$$
$$(Q.E(D^*) + \bar{Q}.c.(r.(Q := 0).E(D^*) + r))$$

from which it can be seen that the assignment $Q := 0$ is redundant in the expression $\bar{Q}.c.r.(Q := 0).E(D^*)$ and so can be eliminated. Thus the structured schema contains one duplication of decision node 'p' and two of 'q', together with four assignments to flags. Whether or not the structured schema is more perspicuous than the original is left to the reader's judgement.

7. SPACE-TIME OVERHEADS

In developing and proving the effectiveness of the transforms, two important questions were left unasked: how are the basic unstructured forms identified in a general flowgraph, and how efficient are the resulting flowgraphs in respect of space and time?

With regard to the first question, whilst decision and loop subgraphs are easily identified in suitably drawn flowgraphs their presence is less obvious in arbitrary ones. The identification of such subgraphs is a major topic outside the scope of this paper. The interested reader is referred to standard texts such as Schaefer[29] and Aho and Ullman[30] where suitable techniques and further references can be found.

On the question of efficiency, it is desirable to have some general measure for program schemas which is independent of particular interpretations. Such a measure could of course give no guidance in general on the efficiency of the transformations for particular interpretations of schemas, but could be useful as a means of comparing the results produced by the structuring process. One such measure is the space–time hierarchy for embedded graphs described by Lipton, Eisenstat and DeMillo (LED)[31] and refined by them in Ref. 32. This measure can be defined informally as follows. Let $G = (V, E)$ be a flowgraph in which the nodes V represent functions and predicates and the edges E the flow of control between them. (This definition is different from that used elsewhere in this paper—see Fig. 9 for the depiction of ID and ID-1 in this form.) Let $dG(u, v)$ be the minimum path length, calculated as the number of edges, between two nodes u, v of V, with $u \neq v$. G is said to be embedded in a strictly equivalent flowgraph $G^* = (V^*, E^*)$ with respect to space S and time T if S is the largest number of duplications of any function or predicate of G contained in G^*, and T is the least value satisfying $dG^*(u^*, v^*) \leq T.dG(u, v)$. Thus $S = n$ means that there are n occurrences of some function in G^* as against one in G and no other function in G^* has more occurrences than n. $T = m$ means that two distinct functions (or predicates) having one edge between them in G have m edges between them in G^*, and no other pairs of distinct functions in G have a greater separation than m in G^*. The embedding is denoted by $G \leq (S, T)G^*$.

Returning to the structuring transforms, and noting that each introduced reference (assignment or test) to a flag adds a node to the LED graph of a buf, it can be shown that ID $\leq$ (2, 1) ID-0 and ID $\leq$ (1, 3) ID-1, and similarly for IL. For OD the relationship is OD $\leq$ (1, 3) OD-1 while for OL it is OL $\leq$ (1, 4) OL-1. Thus the Type-0 transforms require an increase of space alone (from function duplication), whereas the Type-1 transforms require an increase in time alone (from the introduction of flags).

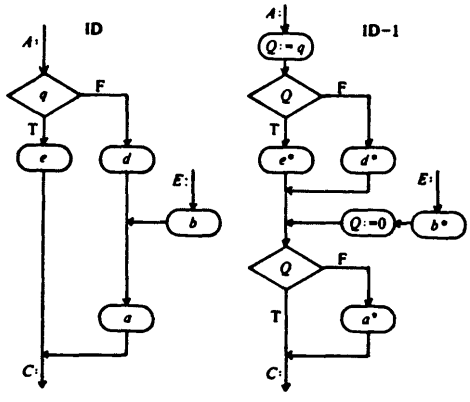


Figure 9. Embedding of ID in ID-1.

Because S and T are defined in terms of extreme values rather than total ones, they are not necessarily additive over successive applications of the transforms. Thus for the generalized Flynn's Problem, denoting the original and final flowgraphs by G and G^* respectively gives $G \leq (1, 3)G^*$ despite three applications of Type-1 transforms.

Intuitive concepts of the relative complexities of the basic unstructured forms and their structured counterparts are to some extent reflected by $\leq (S, T)$ but the important topological distinction between duplicated functions and duplicated decision nodes is lost. It might be worthwhile replacing S by (S, D) where S is unchanged in meaning and D is computed like S but in respect of decision nodes only. For this purpose a test on a flag in G^* is regarded as the same action as a test on the corresponding predicate in G. Thus for the generalized Flynn's Problem is now obtained $G \leq ((1, 3), 3)G^*$ indicating no duplicated functions, a maximum of three occurrences of at least one decision node (the TEST on Q) and an increased time factor (path length) of three for at least one path. This is the price to be paid for achieving structured form.

CONCLUSION

It has been shown that any unstructured schema can be put into strictly equivalent structured form using simple transforms on basic unstructured forms. Further, the transforms used do not rely on the introduction of compound predicate expressions and therefore preserve the original schema's logic as closely as possible. However, it must be said that the structuring process presented here is no substitute for good design. The transforms developed in this paper might help to unravel some knotty problems, but they cannot produce logical poetry from tangled nonsense.

© Heyden & Son Ltd, 1982

REFERENCES

1. J. Bruno and K. Steiglitz, The expression of algorithms by charts, *Journal of the ACM* **19** 517–525 (Oct. 1972).
2. C. Boehm and G. Jacopini, Flow diagrams, Turing machines, and languages with only two formation rules, *Communications of the ACM* **9**, 366–371 (Sep. 1966).
3. E. Ashcroft and Z. Manna, The translation of 'GOTO' programs to 'WHILE' programs, in *Information Processing 71*, ed. by C. V. Freiman, Vol. 1, pp. 250–255. Amsterdam, North-Holland (1972).
4. D. E. Knuth and R. W. Floyd, Notes on avoiding GOTO statements, *Information Processing Letters* **1**, 23–31 (1971); corrections **1**, 177 (1972).
5. H. D. Mills, Mathematical foundations for structured programming, Federal Systems Division, IBM Corp., Gaithersburg, MD, FSC 72-6012 (1972).
6. T. Kasai, Translatability of flowcharts into While programs, *Journal of Computer and Systems Sciences* **9**, 177–195 (Oct. 1974).
7. M. H. Williams, Generating structured flow diagrams: the nature of unstructuredness, *The Computer Journal* **20**, 45–50 (Feb. 1977); **20**, 381–383 (Nov. 1977).
8. M. H. Williams and H. L. Ossher, Conversion of unstructured flow diagrams to structured form, *The Computer Journal* **21**, 161–167 (May 1978).
9. G. Oulsnam, Cyclomatic numbers do not measure complexity of unstructured programs, *Information Processing Letters* **8**, 207–211 (Dec. 1979).
10. D. C. Cooper, Boehm and Jacopini's reduction of flow charts, *Communications of the ACM* **10**, 263, 463 (1967).
11. E. S. Bainbridge, Minimal while programs, in *Lecture Notes in Computer Science*, ed. by A. Mazurkiewicz, Vol. 45, pp. 180–186. Springer-Verlag, Berlin (1976).
12. W. W. Paterson, T. Kasami and N. Tokura, On the capabilities of While, Repeat and Exit statements, *Communications of the ACM* **16**, 503–512 (Aug. 1973).
13. S. R. Kosaraju, Analysis of structured programs, *Journal of Computer and Systems Sciences* **9**, 232–255 (Dec. 1974).
14. W. A. Wulf, Programming without the GOTO, in *Information Processing 71*, Vol. 1, pp. 408–413, ed. by C. V. Freiman, North-Holland, Amsterdam (1972).
15. H. Ledgard and M. Marcotty, A genealogy of control structures, *Communications of the ACM* **18**, 629–639 (Nov. 1975).
16. J. C. Cherniavsky, J. Keohane and P. B. Henderson, A note concerning top down program development and restricted exit control structures, *Information Processing Letters* **9**, 8–12 (Jul. 1979).
17. G. Urschler, Automatic structuring of programs, *IBM Journal of Research and Development* **19**, 181–194 (Mar. 1975).
18. B. S. Baker, An algorithm for structuring flowgraphs, *Journal of the ACM* **24**, 98–120 (Jan. 1977).
19. E. Engeler, Structure and meaning of elementary programs, in *Lecture Notes in Mathematics*, ed. by E. Engeler, Vol. 188, pp. 89–101. Springer-Verlag, Berlin (1971).
20. E. Wegner, Tree-structured programs, *Communications of the ACM* **6**, 704–705 (Nov. 1973).
21. T. J. McCabe, A complexity measure, *IEEE Transactions on Software Engineering* **SE-2**, 308–320 (Dec. 1976).
22. E. W. Dijkstra, Go To statement considered harmful, *Communications of the ACM* **11**, 147–148 [the second occurrence of these pages], 538, 541 (Mar. 1968).
23. D. E. Knuth, Structured programming with go to statements, *ACM Computing Surveys* **6**, 261–301 (Dec. 1974).
24. M. H. van Emden, Programming with verification conditions, *IEEE Transactions on Software Engineering* **SE-5**, 148–159 (Mar. 1979).
25. D. C. Luckham, D. M. R. Park and M. S. Paterson, On formalized computer programs, *Journal of Computer and Systems Sciences* **4**, 220–249 (Aug. 1970).
26. S. C. Kleene, Representation of events in nerve sets, in *Automata Studies*, ed. by C. E. Shannon and J. McCarthy, pp. 3–40. Princeton University Press, Princeton, New Jersey (1956).
27. P. J. Denning, J. B. Dennis and J. E. Qualitz, *Machines, Languages and Computation*, Prentice-Hall, Englewood Cliffs, New Jersey (1978).
28. R. C. Tausworthe, *Standardized Development of Computer Software*, Prentice-Hall, Englewood Cliffs, New Jersey (1977).
29. M. Schaefer, *A Mathematical Theory of Global Program Optimization*, Prentice-Hall, Englewood Cliffs, New Jersey (1973).
30. A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts (1977).
31. R. J. Lipton, S. C. Eisenstat and R. A. DeMillo, Space and time hierarchies for classes of control structures and data structures, *Journal of the ACM* **23**, 720–732 (Oct. 1976).
32. R. A. DeMillo, S. C. Eisenstat and R. J. Lipton, Space-time tradeoffs in structured programming: an improved combinatorial embedding theorem, *Journal of the ACM* **27**, 123–127 (Jan. 1980).

Received October 1981

© Heyden & Son Ltd, 1982