B4-59

Automatic Generation of Path Covers

Antonia Bertolino[*], Martina Marré[**]

Index terms - Algorithmic complexity, automated testing tools, ddgraph, path cover, program testing, S_1-structured programs, unconstrained arcs, weakly incomparable arcs.

ABSTRACT

Among the fundamental problems computer program testing must deal with, there is that of selecting a significant sample of executions out of the potentially infinite execution domain. In this paper, structural testing is treated and, precisely, an algorithm is presented which finds a subset of program control flow paths satisfying the branch testing criterion, i.e. every program's branch is covered at least once. The minimal number of paths is found for acyclic structured programs. Being recursive, the algorithm is very simple.

The analysis is based on a reduced flowgraph representation of programs, called ddgraph, and uses two relationships, dominance and implication, defined between the arcs of a ddgraph. Specifically, these relationships make it possible to identify a subset of arcs, called unconstrained, having the property that, when the unconstrained arcs are exercised, the coverage of all the other arcs in the ddgraph is also guaranteed.

Properties of ddgraph arcs are extensively discussed. A proof of correctness and a theoretical evaluation of the algorithm are given. Application to the structural testing of real programs is straightforward and has been experimented in a prototype tool.

I. INTRODUCTION

Software testing [1] consists of the validation of computer programs through the observation of a meaningful sample of executions chosen amongst the potentially infinite execution domain. Thus, test planning requires selecting a suitable set of test paths to be exercised and finding the test input domain which corresponds to these paths. This selection can be based on program specifications, in which case we speak of *functional testing*, or on program code, *structural testing*. Which of the two strategies, functional and structural, is

[*] Istituto di Elaborazione della Informazione, CNR, Pisa, Italy.

[**] Dipartimento di Informatica, Università di Pisa, Italy and HP Labs, Pisa Science Center, Italy.

This work was supported in part by the project "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of the Italian National Research Council (CNR).

better than the other cannot be decided in general, but depends on the specific problem (very likely, the best thing is to apply both in successive stages).

Henceforth we shall concetrate our attention on the structural testing strategy, which permits the automatic selection of test paths. Structural testing, in turn, can be based on the program control flow or on its data flow. This paper deals with the problem of selecting a test path set based on the analysis of the program control flow. Particularly, the test strategy which is pursued is *branch testing* [11], which requires each branch in a program to be traversed at least once. Indeed, branch testing is the most widely applied among structural testing criteria, since it offers a favourable trade-off between testing costs and testing efficacy.

We present an algorithm which performs the automatic selection of a test path set satisfying the branch testing criterion for a given program (either structured or not structured). The path set which is found is minimum for acyclic structured programs; for generic structured programs, in practice, is not much greater than the minimum which is possible by iterating cycles at most once. This last feature derives from the decision to leave nondeterministic the choice of the next arc to be exercised at each iteration of the algorithm. The most appealing feature of the algorithm presented is its simplicity, thanks to its recursive structure.

Our analysis is conducted on a graphical representation of programs(1). Traditionally flowgraphs have been used; in this paper we introduce a somewhat novel representation model, called *ddgraph*, in which program branches are represented by arcs, thus reverting, in practice, the more typical usage in flowgraphs of associating program branches to nodes. Therefore, applied to ddgraphs, the branch testing criterion corresponds to finding a set of paths which exercises every ddgraph arc: this set is called a *path cover*. Moreover, we use two important relationships between ddgraph arcs, namely *dominance* and *implication*, which allow to identify a subset of ddgraph arcs, called *unconstrained arcs*. These arcs are the minimal arc subset having the property that, when a set of paths is selected which exercises all the unconstrained arcs in a ddgraph G, then this set constitutes a path cover for G, i.e. the traversal of every other arc is guaranteed.

Some previous works have studied the problem of finding path covers on flowgraphs. [14] gives a generalized optimal path-selection model for structural testing: it formulates the problem of selecting test paths as a zero-one integer programming problem (which is NP-complete). [13] discusses a number of path cover problems arising in program testing; particularly, the problem of covering every edge of a digraph G with a minimum

(1) Consequently, unfeasible paths, i.e. control flow paths which are exercised by no input data, could also be chosen; this is a limitation of every structural testing strategy, since the problem of determining if an input item exists which exercises a given path is undecidable [15].

number of paths is solved through application of a minimum flow method or a maximum matching method to a corresponding acyclic digraph G'. [12] proposes a heuristic method to find a path cover, yet without dealing with the number of the paths found. [6] gives an algorithm which solves a related, but different problem, that is finding a path that passes through all the program statements within a specified set, if one such path exists.

The paper is addressed to two different classes of readers, graph theoreticians and testing practitioners. Section II introduces the contents of the paper, giving a brief guideline to read it. Section III, which developes the basic terminology and notions later used to present the algorithm, is divided into 3 subsections: subsection A gives the definition of ddgraphs and some related notions; subsection B defines the notions of path cover, of unconstrained arcs and of "incomparability" between arcs; subsection C defines the notion of structuredness for ddgraphs. Section IV describes the algorithm and its functioning. Section V draws the conclusions, among which a table reporting the results of an experimentation of the algorithm.

II. HOW TO READ THIS PAPER

There are two different ways in which this paper can be read, depending on your purposes and your background.

If you are involved in graph theory and feel comfortable amongst lemmas, theorems and corollaries, you can read in sequence the whole paper: essentially, sections III contains a number of definitions and propositions which either extend to ddgraphs some results already acquired in flowgraphs or introduce some novel concepts; this material is then used in section IV to demonstrate that the algorithm presented "works well". We are quite confident that it will be not so hard a task for you to reach the end.

If you are a testing practitioner and do not love mathematical notation, don't feel frustrated by the hostile appearing of the following pages. The good news is that you can put in use the results of this paper in your practical job of testing real programs. If this is just what you expect from reading this paper, you can skip most part of it: just pick the subsection A of section III, the definition 7 of unconstrained arcs and the examples given below it, and finally the algorithm given in Pascal-like notation in the figures 11 and 12. Then follow the procedure below.

1) Extract the program ddgraph G.

2) Apply to G the algorithm FIND-A-TEST-PATH-SET; this in turn requires:

   2.1) using the dominance and the implication relationships between ddgraph arcs, derive the Dominator Tree and the Implication Tree for G;

   2.2) detect the unconstrained branches of G as $DTL \cap ITL$, where DTL is the set of leaves of the Dominator Tree and ITL is the set of leaves of the Implication Tree;

2.3) using the Dominator Tree and the Implication Tree, construct a path for each (yet uncovered) unconstrained arc;

2.4) repeat recursively steps 2.1, 2.2, 2.3 each time you encounter a discontinuity while performing step 2.3 above.

3) Trust us that the algorithm works: a proof of its correctness and its theoretical evaluation are given in this paper. If only *experimental testing* convinces yourself, the algorithm has also been implemented in a prototype tool, called BAT [4], which has been undergoing experimental validation for some period (see section V).

III. BASIC DEFINITIONS

A. DDgraphs and Sub-ddgraphs

A program's control flow is conveniently analyzed by means of a directed graph called *flowgraph*, which depicts all possible execution paths [7]. In this subsection we introduce a somewhat novel flowgraph representation of programs, called *ddgraph* (for decision-to-decision graph), which is particularly suitable for the purposes of pathwise testing. In fact, each arc in a ddgraph directly corresponds to a program's branch; thus program branch coverage is immediately measured in terms of arc coverage in the ddgraph.

Let us begin by recalling some basic notions of graph theory [2].

A *directed graph* or *digraph* $G=(V,E)$ consists of a set V of *nodes* or *vertices*, and a set E of *directed edges* or *arcs*, where a directed edge $e=(\mathrm{T}(e),\mathrm{H}(e))$ is an ordered pair of *adjacent* nodes, called *Tail* and *Head* of e, respectively: we say that e *leaves* $\mathrm{T}(e)$ and *enters* $\mathrm{H}(e)$. If $\mathrm{H}(e)\equiv\mathrm{T}(e')$, e and e' are called *adjacent* arcs. For a node n in V, *indegree*(n) is the number of arcs entering it and *outdegree*(n) is the number of arcs leaving it.

A *path* P of length q in G is a sequence $P=n_{j_0},e_{i_1},n_{j_1},\ldots n_{j_{q-1}},e_{i_q},n_{j_q}$, where $\mathrm{T}(e_{i_k})=n_{j_{(k-1)}}$ and $\mathrm{H}(e_{i_k})=n_{j_k}$, $k=1,\ldots,q$. We will also write $P=e_{i_1},\ldots,e_{i_q}$. A path P is *simple* if all its nodes are distinct.

An arc e *reaches* an arc e' (a node n *reaches* a node n') if there exists a path in G from e to e' (from n to n').

Let $P=e_{i_1},\ldots,e_{i_q}$ be a path in G. We say that a path P' is a *subpath* of P if $P'=e_{i_{j_1}},\ldots,e_{i_{j_r}}$, where for $h=1,\ldots,r$: $j_h\in\{1,\ldots,q\}$ and for $h=1,\ldots,r-1$: $j_h\leq j_{h+1}$.

A path $P=n_{j_0},e_{i_1},n_{j_1},\ldots n_{j_{q-1}},e_{i_q},n_{j_q}$, is a *cycle* if $n_{j_q}=n_{j_0}$. A *simple cycle* is a simple path that is also a cycle. An *acyclic* digraph is a digraph that has no cycles.

A (*rooted*) *tree* $T=(V,E)$ is a digraph, in which one distinguished node, called the *root*, is the Head of no arcs; every node except the root is the Head of exactly one arc and there exists a (unique) path from the root to each node. If there is an arc $e=(n_i,n_j)$ in T, n_i is said the *parent* of n_j and n_j is said a *child* of n_i. Tree nodes of zero outdegree are said *leaves*.

In the remainder of this subsection we shall introduce some specific terminology and the basic concepts which the paper is laid on. The following is our definition of ddgraphs.

Definition 1: *DDgraph*
A *ddgraph* is a digraph $G=(V,E)$ with two distinguished arcs e_0 and e_k (which are the unique entry arc and exit arc, respectively), such that any other arc in G is reached by e_0 and reaches e_k, and such that for each node n in V, $n \neq \mathrm{T}(e_0)$, $n \neq \mathrm{H}(e_k)$, (indegree(n) + outdegree(n)) > 2, while indegree(T(e_0))=0 and outdegree(T(e_0))=1, indegree(H(e_k))=1 and outdegree(H(e_k))=0.

In figure 1 and figure 2 we present two ddgraphs: G_1 with distinguished arcs e_0 and e_{10} and G_2 with with distinguished arcs e_0 and e_{14}, respectively.

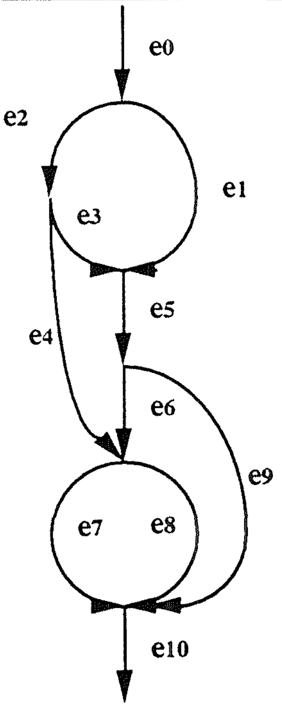

fig. 1

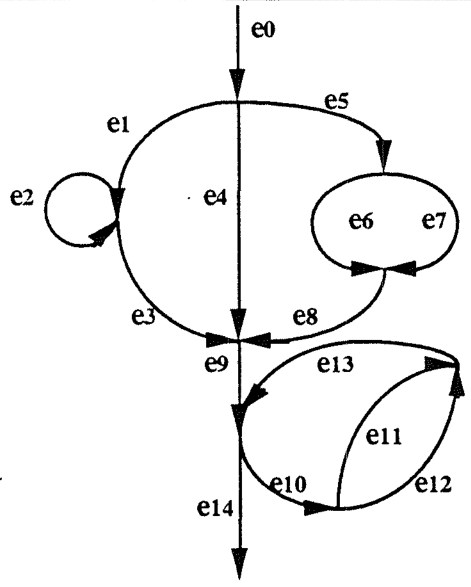

fig. 2

DDgraph's arcs are associated to program's branches. A branch is here defined as a strictly sequential set of program statements(2), of which the first [the last] statement may emanate from [terminate at]: i) the BEGIN clause [the END clause] of the program; ii) the evaluation of a conditional expression; iii) the join of separate control flow paths. In some

(2) i.e., a sequence of program statements not containing alterations of the control flow.

cases, an arc is introduced which does not correspond to a piece of code, but nevertheless represents a possible course of the program control flow (e.g. the implicit ELSE part of an IF statement).

The points i), ii) and iii) above are associated to ddgraph's nodes; particularly, ii) individuates a node with outdegree≥2 which corresponds to a branching in the program control flow; iii) individuates a node with indegree≥2 which corresponds to a joining in the program control flow (for example below a conditional statement) and may be empty (i.e., not associated to any piece of code).

Obviously, for a strictly sequential program, the ddgraph G=(V,E) will consist of just one arc, i.e. $E=\{e_0 \equiv e_k\}$ and $V=\{\mathrm{H}(e_0), \mathrm{T}(e_0)\}$. This is called the *trivial* ddgraph.

Note that, in practice, ddgraphs revert the more typical usage in flowgraphs of associating program blocks to nodes.

An important concept found in the literature [7, p.55] is the *dominance* relation, which imposes a partial ordering on the nodes of a flowgraph. Since in ddgraphs program branches are associated to arcs, we here are interested in applying the dominance relationship to the arcs instead of to the nodes. Therefore, here below we give our definition of dominance between arcs in a ddgraph.

Definition 2: *Dominance*
Let G=(V,E) be a ddgraph with distinguished arcs e_0 and e_k. An arc e_i *dominates* an arc e_j if every path P from the entry arc e_0 to e_j contains e_i.

Several algorithms have appeared in the literature to find the dominators in a digraph, for example [9].

By applying the dominance relationship between the arcs of a ddgraph, we can obtain a tree (whose nodes represent the ddgraph arcs) rooted at e_0, called *Dominator Tree* (*DT*(G)). For each pair (e_i, e_j) of adjacent nodes in DT, e_i=Parent(e_j) is the *immediate dominator* of e_j. The immediate dominator e_i of an arc e_j is a dominator of e_j with the property that any other dominator of e_j also dominates e_i. Notice that each arc (different of e_0) has exactly one immediate dominator.

A *dominance path* P_{DT} in DT(G) is a sequence $P_{DT}=e_{i_1},\dots,e_{i_q}$, where for $j=1,\dots,q-1$: e_{i_j}=Parent($e_{i_{j+1}}$).

In figure 3, the DT of the ddgraph G_2 in figure 2 is shown. $P_{DT}=e_0,e_9,e_{10},e_{11}$ is a dominance path in DT(G_2).

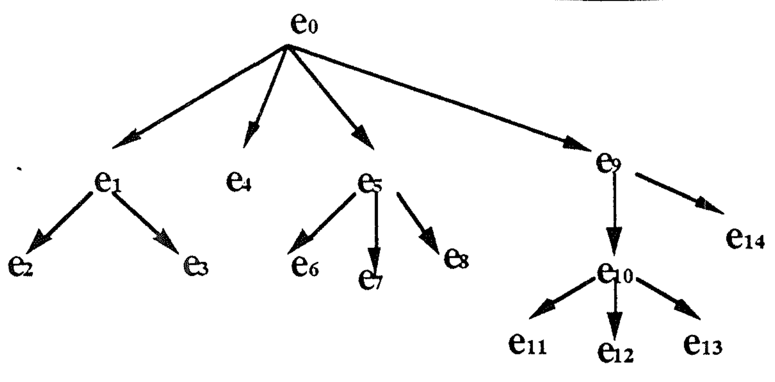

fig. 3

Following, we introduce the "symmetric" relation of *implication* between arcs in a ddgraph.

Definition 3: *Implication*
An arc e_i *implies* an arc e_j if every path P from e_i to the exit arc e_k contains e_j.

The implied arcs in a ddgraph G with distinguished arcs e_0 and e_k can be found as the dominators of the ddgraph G' having distinguished arcs e'_0 and e'_k, in which every arc e' is obtained by reverting a corresponding arc e in G (i.e. $H(e')=T(e)$ and $T(e')=H(e)$), e'_0 corresponds to the reverse arc of e_k and e'_k corresponds to the reverse arc of e_0.

By applying the implication relationship between the arcs of a ddgraph, we can obtain a tree (whose nodes represent the ddgraph arcs) rooted at e_k, called *Implied Tree* (*IT*(G)). For each pair (e_i, e_j) of adjacent nodes in IT(G), e_i=Parent(e_j) is *immediately implied* by e_j. An arc e_i is immediately implied by an arc e_j if e_j implies e_i and any other arc which implies e_i also implies e_j. Notice that each arc (different of e_k) is immediately implied by exactly one arc.

An *implication path* P_{IT} in IT(G) is a sequence $P_{IT}=e_{i_1},...,e_{i_q}$, where for $j=1,...,q-1$: $e_{i_{j+1}}$=Parent(e_{i_j}).

In figure 4, the IT of the ddgraph G_2 in figure 2 is shown. $P_{IT}=e_1,e_3,e_9,e_{14}$ is an implication path in IT(G_2).

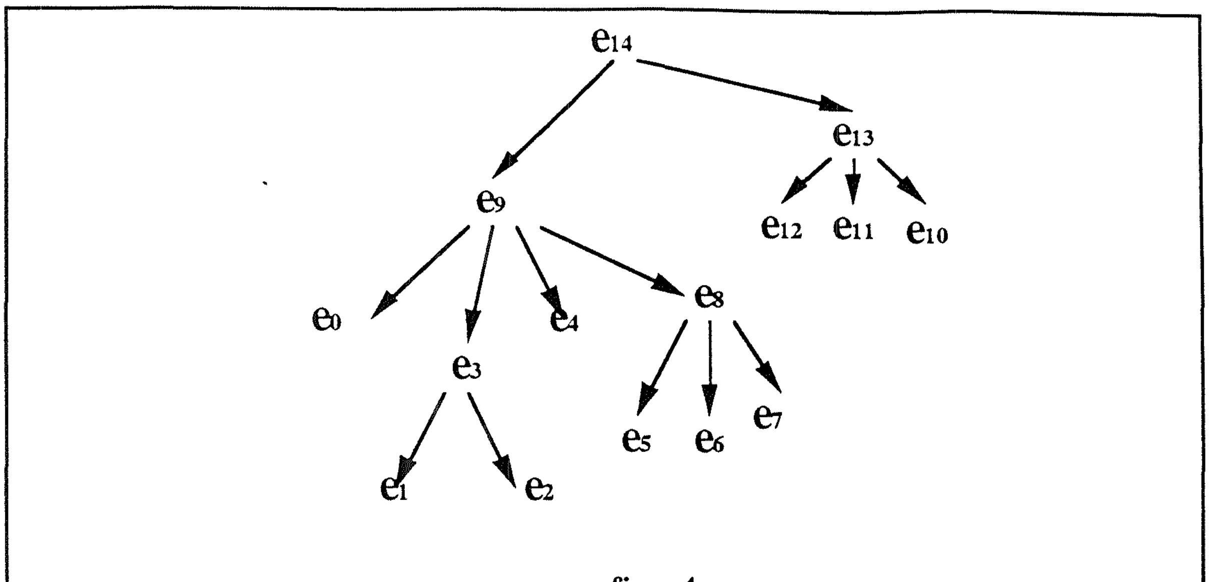

fig. 4

DTs and ITs can contain *discontinuities*, as defined below.

Definition 4: *Discontinuity*
Given a dominance path [an implication path] $P_T = e_{i_1},\dots,e_{i_j},e_{i_{j+1}},\dots,e_{i_q}$, we say that a *discontinuity* between e_{i_j} and $e_{i_{j+1}}$ exists when $H(e_{i_j}) \neq T(e_{i_{j+1}})$, i.e. when two arcs are represented by adjacent vertices in DT(G) (in IT(G)), but they are not adjacent in G.

For example, there exists a discontinuity between e_0 and e_9 in the DT(G_2) of figure 3.

Dominance paths and implication paths can be used to construct ddgraph paths. The following proposition states an important relationship: a path on the DT(G) (or on the IT(G)) always corresponds to a path on the ddgraph G provided that possible discontinuities in P_{DT} or in P_{IT} be "filled" with a subpath in G. For example, a path $P=e_0,e_4,e_9,e_{10},e_{11}$ on DT(G_2) corresponds to the dominance path $P_{DT}=e_0,e_9,e_{10},e_{11}$ given above.

Proposition 1

Let G be a ddgraph, DT(G) its dominance tree and IT(G) its implication tree.

i) If there is a path P_{DT} in DT(G) from e_0 to e, $P_{DT}=e_0,e_{i_1},\dots,e_{i_q},e$, then there exists a path P from e_0 to e in the ddgraph G and P has the following form: $P=(e_{i_0}=e_0),P_0,e_{i_1},P_1,\dots,e_{i_q},P_q,(e_{i_{q+1}}=e)$, where for $j=0, \dots, q$, P_j is a path from $H(e_{i_j})$ to $T(e_{i_{j+1}})$, possibly empty.

ii) If there is a path P_{IT} in IT(G) from e to e_k, $P_{IT}=e,e_{i_1},\dots,e_{i_q},e_k$, then there exists a path P from e to e_k in the ddgraph G and P has the following form:

$P=(e_{i_0}=e), P_0, e_{i_1}, P_1, \ldots, e_{i_q}, P_q, (e_{i_{q+1}}=e_k)$, where for j=0, ..., q, P_j is a path from H(e_{i_j}) to T($e_{i_{j+1}}$), possibly empty.

Proof

i) G is a ddgraph, then there exists at least a path from e_0 to e. From the definition of dominance, for each path P from e_0 to e in G and for each arc e_i in P, the arc Parent(e_i) is in P and precedes e_i.

ii) similar to i). ♦

Let $G=(V,E)$ be a digraph with a unique entry arc e_0 and a unique exit arc e_k. The procedure REDUCE in figure 5 transforms G into a ddgraph G' with distinguished arcs e_0 and e_k. REDUCE eliminates each (possible) node n in G with indegree(n)=1 and outdegree(n)=1 (which cannot exist in G') by "compacting" the arc e_i of G entering n and the arc e_j of G leaving n with an only arc $e_{i\text{-}j}$ in G'. Since each arc can be reduced at most one time, procedure REDUCE can be implemented in $O(|E|)$ time .

```
Procedure REDUCE (in G=(V,E):digraph; out G'=(V',E'): ddgraph;);
begin
   E':= E;
   V':= V;
   while ∃ei,ej ∈ E' such that ei≠ej, T(ej)=H(ei), indegree(H(ei))=1,
           outdegree(T(ej))=1
   do begin
           V':= (V'-{H(ei)});
           E':= (E'-{ei,ej})∪{ei-j}, where T(ei-j)=T(ei) and H(ei-
j)=H(ej)
   end
end procedure.
```

fig. 5

Next, we define sub-ddgraphs and suggest an algorithm to derive a desired sub-ddgraph from a ddgraph. Sub-ddgraphs will be useful to derive the paths in G which fill the possible discontinuities in the dominance and implication paths (see proposition 1 above). A sub-ddgraph of a ddgraph G from e_a to e_b is formed by all the paths from H(e_a) to T(e_b), which do not use the arcs e_a and e_b.

Definition 5: *Sub-ddgraph*

Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k. Let e_a and e_b be arcs in E. The *sub-ddgraph* of G between e_a and e_b, written sub-ddgraph(G,e_a,e_b), is the ddgraph obtained after reducing via the procedure REDUCE (in Figure 5) the digraph $G'=(V',E')$, where

- e_a and e_b are not in E';
- e_0' and e_k' are the distinguished arcs in E', with $H(e_0')=H(e_a)$ and $T(e_k')=T(e_b)$ in V' and with $H(e_k')$ and $T(e_0')$ two new different nodes in V' (not in V);
- if $e \in E$ and there exists a path P from $H(e_a)$ to $T(e_b)$ in G, not containing neither e_a and e_b, such that e is in P, then $e \in E'$ and $T(e)$ and $H(e)$ are in V'.

In the following, for brevity, we shall continue to adopt for the distinguished arcs e_0' and e_k' of G' the corresponding names of the arcs e_a and e_b in G, of course understanding that the sub-ddgraph of G with distinguished arcs e_a and e_b is always obtained following the above reduction rules.

The sub-ddgraph of G_2 in figure 2, with distinguished arcs e_6 and e_{13} is presented in figure 6.

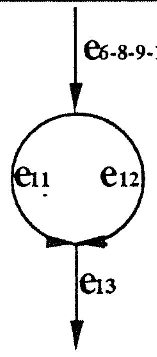

fig. 6

Properties

i) A sub-ddgraph is a ddgraph.

ii) Sub-ddgraph(G,e_0,e_k)=G.

iii) If $e_0 \neq e_a$ or $e_k \neq e_b$ then $|E'|<|E|$.

In figure 7 the algorithm SUB-DDGRAPH is presented, which finds the sub-ddgraph G^* of a given ddgraph $G=(V,E)$ with distinguished arcs e_a and $e_b \in E$ in $O(|E|)$ time. In a first step (first *repeat*), the algorithm finds the arcs reachable from the head of the arc e_a not using the arc e_a. In a second step (second *repeat*), from this set of arcs, it selects those that reach the tail of the arc e_b not using the arc e_b. Also, the new distinguished nodes are generated, and the sub-ddgraph is finally obtained applying the procedure REDUCE.

The initialization steps requires $O(|E|+|V|)$ time. The first repeat selects each arc at most one time, and then it works in $O(|E|)$ time. Analogously, the second repeat. The procedure REDUCE is $O(|E|)$ time. Since G is a ddgraph, the number of nodes grows bounded by the number of arcs, and then the algortihm SUB-DDGRAPH is $O(|E|)$ time.

```
Procedure SUB-DDGRAPH (in G=(V,E):ddgraph; in $e_a$,$e_b$:arcs in E;
                              out G*=(V*,E*):ddgraph)
begin
    V':=∅;   E':=∅;
    for each n∈V do label(n):=false;
    for each e∈E do used(e):=false;
    label(H($e_a$)):=true; Q:={H($e_a$)}; used($e_a$)=true;
    repeat
       select and remove n∈Q;
       V':=V'∪{n};
       for each e∈E such that not(used(e)) and T(e)=n do
          begin
          used(e):=true;
          E':=E'∪{e};
          if label(H(e))=false then
             begin
             label(H(e)):=true;
             Q:=Q∪{H(e)};
             end
          end
    until Q=∅;
    V'':=∅;  E'':=∅
    for each n∈V' do label(n):=false;
    for each e∈E' do used(e):=false;
    label(T($e_b$)):=true; Q:={T($e_b$)}; used($e_b$)=true;
    repeat
       select and remove n∈Q;
       V'':=V''∪{n};
       for each e∈E' such that not(used(e)) and H(e)=n do
          begin
          used(e):=true;
          E'':=E''∪{e};
          if label(T(e))=false then
             begin
             label(T(e)):=true;
             Q:=Q∪{T(e)};
             end
          end
    until Q=∅;
    new($v_1$); new($v_2$);
    V'':=V''∪{$v_1$,$v_2$};
    $e_0$':=($v_1$,H($e_a$)); $e_k$':=(T($e_b$),$v_2$);
    E'':=E''∪{$e_0$',$e_k$'};
    REDUCE(G''=(V'',E''), G*=(V*,E*));
end procedure.
```

fig. 7

B. Path Covers on DDgraphs

In this subsection we define path covers on ddgraphs. Finding a set of test paths which is a path cover for a ddgraph is the fundamental problem of branch testing. With this aim, we introduce the notion of unconstrained arcs in a ddgraph. Unconstrained arcs form a subset of ddgraph's arcs which has proved very useful in the analysis of program structure for the purposes of pathwise testing, in that, a path set which covers all the unconstrained arcs also covers all the arcs in the ddgraph. This fundamental property of unconstrained arcs is stated further in Theorem 1; its demonstration has been the subject of a separate paper [3].

Definition 6: *Path Cover*
Let $G=(V,E)$ be a ddgraph. A set of paths $\mathbf{P}=\{P_1,...,P_n\}$ is a *path cover* for G if for each arc $e \in E$ there exists at least a path P_i in $\mathbf{P}$ containing e.

A path cover $\mathbf{P}$ for a ddgraph G is a *minimum path cover* if there does not exist a path cover $\mathbf{P}'$ for G with $|\mathbf{P}'|<|\mathbf{P}|$[3].

The set of paths $\mathbf{P}=\{P_1,P_2,P_3,P_4\}$ is a path cover for the ddgraph G_1 of figure 1 with:

P_1: $e_0\ e_2\ e_3\ e_5\ e_6\ e_9\ e_{10}$;

P_2: $e_0\ e_1\ e_5\ e_6\ e_8\ e_{10}$;

P_3: $e_0\ e_2\ e_4\ e_9\ e_{10}$;

P_4: $e_0\ e_1\ e_5\ e_7\ e_{10}$.

$\mathbf{P}$ is not minimum, since $\mathbf{P}'=\{P'_1,P'_2,P'_3\}$ is another path cover for G_1 having $|\mathbf{P}'|<|\mathbf{P}|$ with:

P'_1: $e_0\ e_1\ e_5\ e_7\ e_{10}$;

P'_2: $e_0\ e_2\ e_4\ e_8\ e_{10}$;

P'_3: $e_0\ e_2\ e_3\ e_5\ e_6\ e_9\ e_{10}$.

Definition 7: *unconstrained arcs*
An arc e_u is *unconstrained* if e_u dominates no other arc and is implied by no other arc in G.

(3) The cardinality of a set S, denoted by $|S|$, gives the number of elements in S.

By the same definition, we can obtain the set UE(G) of unconstrained arcs of G simply as $DTL \cap ITL$, where DTL is the set of leaves of DT(G) and ITL is the set of leaves of IT(G).

Said in another way, an arc e in a ddgraph G is unconstrained if for any other arc e' in G there is at least a path from e_0 to e_k containing e' and not containing e.

Let us note that, by ddgraph's definition, UE(G)$\neq\emptyset$ for any ddgraph G. For the trivial ddgraph is UE=$\{e_0\}$.

For the ddgraph G_2 in Figure 2 we have:

DTL(G_2) = $\{e_2,e_3,e_4,e_6,e_7,e_8,e_{11},e_{12},e_{13},e_{14}\}$,

ITL(G_2) = $\{e_0,e_1,e_2,e_4,e_5,e_6,e_7,e_{10},e_{11},e_{12}\}$,

and thus the set of unconstrained arcs of G_2 in figure 2 is:

UE(G_2) = DTL(G_2)$\cap$ITL(G_2) = $\{e_2,e_4,e_6,e_7,e_{11},e_{12}\}$.

Here below we enunciate the fundamental theorem of unconstrained arcs; as already said, its proof can be found in [3].

Theorem 1: *the fundamental property of unconstrained arcs*

Let $G=(V,E)$ be a ddgraph. Then:

i) A set of paths covering all the unconstrained arcs is a path cover for G;

ii) The set of unconstrained arcs is minimum among the sets of arcs with property (i).

The following proposition states that the unconstrained arcs of a sub-ddgraph G' derived from a ddgraph G are unconstrained also in G. More precisely, the two possible cases are distinguished, i.e. an arc of G also belongs to G', or else more arcs of G, at least one of which unconstrained, are compressed into an only arc of G'.

Proposition 2

Let $G=(V,E)$ be a ddgraph. Let e_0' and e_k' be two arcs in E such that there exists a discontinuity between e_0' and e_k' in DT(G) or in IT(G). Let $G'=(V',E')$ be the sub-ddgraph of G with distinguished arcs e_0' and e_k'. Then

i) if $e\in$ UE(G') and $e\in E\cap E'$, then $e\in$ UE(G);

ii) if $e\in$ UE(G') and $e\in E'$ but $e\notin E$ (i.e., e is obtained by a reduction of a sequence of arcs $e_1,\dots,e_r$ in E), then there exists at least an arc e_i, $i\in\{1,\dots,r\}$ such that $e_i\in$ UE(G).

Proof

Since there exists a discontinuity between e_0' and e_k' in DT(G) or in IT(G), there exist at least two distinct paths from T(e_0') to H(e_k') in G and also in G'. We only present a proof of ii). The proof of i) can be obtained as a particular case of this one.

ii) Suppose that the unconstrained arc e in E' is the reduction of the sequence of arcs $e_{i_1},\dots,e_{i_r}$ in E. We will see that there exist e_{i_j}, $j\in\{1,\dots,r\}$ such that for each $e'\in E$, there exists a path from e_0 to e_k using the arc e' and not the arc e_{i_j}.

If the node v=T(e_{i_r}) has outdegree(v)>1 then e_{i_r} is an unconstrained arc in G. If the node v=H(e_{i_1}) has indegree(v)>1 then e_{i_1} is an unconstrained arc in G.

Since G is a ddgraph, for each internal node, outdegree(v)+indegree(v)>2. Suppose that the conditions before do not hold, i.e. indegree(H(e_{i_1}))=1 and outdegree(T(e_{i_r}))=1; therefore outdegree(H(e_{i_1}))>1 and indegree(T(e_{i_r}))>1. Let v be the last node in the sequence e_{i_1},...,e_{i_r} with the property that indegree(v)=1. If v=H(e_{i_j})=T($e_{i_{j+1}}$), then indegree(H($e_{i_{j+1}}$))>1 and outdegree(T($e_{i_{j+1}}$))>1. Let e' be an arc leaving T($e_{i_{j+1}}$), and e'' be an arc entering H($e_{i_{j+1}}$). (Figure 8).

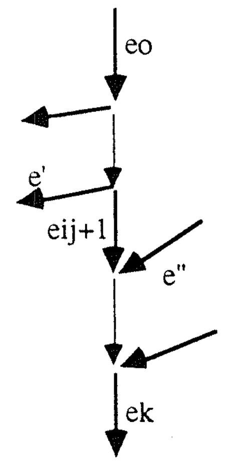

fig. 8

We will see that for each arc $e_G \in E$, $e_G \neq e_{i_{j+1}}$, there exists a path from e_0 to e_k such that e_G is in the path and $e_{i_{j+1}}$ is not. Suppose that we know a path p from e_0 to e_k such that e_G and $e_{i_{j+1}}$ are in the path. We have two cases:

• $p=e_0,\dots,e_G,\dots,e_{i_{j+1}},\dots,e_k$.

Since $e_{i_1},\dots,e_{i_r}$ is reduced to e in G', does not exist a path in G from e' to e_{i_s}, for each $s \in \{1,\dots,r\}$. Let $p_{e'e_k}$ be a path from e' to e_k in G. If $e_{i_{j+1}}$ is not in $p_{e'e_k}$, we already found a path from e_0 to e_k containing e_G and not $e_{i_{j+1}}$: $p'=e_0,\dots,e_G,\dots,\mathrm{T}(e_{i_{j+1}}),p_{e'e_k}$.

Otherwise, ($e_{i_{j+1}} \in p_{e'e_k}$) then $e_{i_1},\dots,e_{i_{j+1}}$ is a subpath of $p_{e'e_k}$. Since e is an unconstrained arc in G', there exists an alternative path $p_{\mathrm{T}(e)e_k}$ from T(e) to e_k in G' (and also in G) not containing the arc $e_{i_{j+1}}$. Then $p'=e_0,\dots,e_G,\dots,\mathrm{T}(e_{i_1}),p_{\mathrm{T}(e)e_k}$ is a path from e_0 to e_k containing e_G and not $e_{i_{j+1}}$.

• $p=e_0,\dots,e_{i_{j+1}},\dots,e_G,\dots,e_k$. The proof is similar.♦

The rest of this subsection settles a number of definitions and observations about *incomparability* between ddgraph's arcs, which will be useful later on (precisely, in theorem 3) to establish an upper limit to the number of paths constructed by the algorithm presented in section IV.

Definition 8: *Incomparable*
Let $G=(V,E)$ be a ddgraph, with $e,e' \in E$ and $e \neq e'$. If there does not exist a path in G containing both e and e', then e and e' are called *incomparable.*

Next we present a relaxation of the above definition which will be useful further on to evaluate the functioning of the algorithm (see Theorem 3).

Definition 9: *Weakly Incomparable*
Let $G=(V,E)$ be a ddgraph, and the arcs e and $e' \in E$, with $e \neq e'$. If:
i) does not exist a simple path from e_0 to e_k in G containing both e_0 and e_k, and
ii) if there exists a (not simple) path
$P=n_0,e_0,n_1,\ldots,n_r,e,n_{r+1},\ldots,n_{r'},e',n_{r'+1},\ldots,n_{q-1},e_k,n_q$
such that there exist i_1, $i_2 \in \{0,\ldots,q\}$, with $i_1 \leq r$ and $i_2 \geq r'+1$, $n_{i_1}=n_{i_2}$ and $n_{i_1},\ldots,n_r,e,n_{r+1},\ldots,n_{r'},e',n_{r'+1},\ldots,n_{i_2}$ is a simple cycle, then there exists a subpath of P from e_0 to e_k containing e [e'] but not e' [e],
then e and e' will be called *weakly incomparable* .

Note that if G is an acyclic ddgraph, $e,e' \in E$ and $e \neq e'$, then e and e' are weakly incomparable if and only if they are incomparable.

In the ddgraph G_2 of figure 2, the arcs e_1 and e_4 are incomparable; the arcs e_{11} and e_{12} are weakly incomparable but are not incomparable, e.g. the not simple path

$P=e_0\ e_4\ e_9\ e_{10}\ e_{11}\ e_{13}\ e_{10}\ e_{12}\ e_{13}\ e_{14}$

contains both of them; the arcs e_{11} and e_{13} are not weakly incomparable, since, even if there is not a simple path from e_0 to e_k containing both, for any not simple path containing both, if we open e_{11} [e_{13}], we can never obtain a subpath of P from e_0 to e_k containing e_{13} [e_{11}].

Given a ddgraph G, a *largest weakly incomparable* arc set is a maximum set of (weakly incomparable) arcs, as the following definition formally states.

Definition 10: *Largest Weakly Incomparable Arc Set*
Let $G=(V,E)$ be a ddgraph. A *largest weakly incomparable (LWI) arc set* E_{LWI} for G is a subset of E satisfying:

i) for each $e,e' \in E_{LWI}$, $e \neq e'$, then e and e' are weakly incomparable,

ii) $|E_{LWI}| = \max\{|E'|$: for each $e,e' \in E'$, if $e \neq e'$ then e and e' are weakly incomparable$\}$.

Notice that given a ddgraph, there is at least one LWI unconstrained arc set, and possibly it is not unique: $E_{LWI_1}=\{e_1,e_2,e_4,e_6,e_7,e_{11},e_{12}\}$ and $E_{LWI_2}=\{e_2,e_3,e_4,e_6,e_7,e_{11},e_{12}\}$ are LWI arc sets for the ddgraph G_2 of figure 2.

In general, as can be seen from the above example, a LWI arc set can contain both unconstrained and not unconstrained arcs; therefore, for a generic ddgraph G, $|E_{LWI} \cap UE|$ can vary. For example, for the ddgraph G_3 of figure 9, $E_{LWI_1}=\{e_1,e_2,e_5,e_6\}$ and $E_{LWI_2}=\{e_2,e_9,e_{11},e_{12}\}$ are LWI arc sets and $|E_{LWI_1} \cap UE|=3$, while $|E_{LWI_2} \cap UE|=4$.

However, it can be easily shown that if G is an acyclic ddgraph, then E_{LWI} only contains unconstrained arcs and, more precisely, the proposition 3 below shows that a not unconstrained arc e can be part of E_{LWI} only if it dominates or is implied by an unconstrained arc which is "within a cycle".

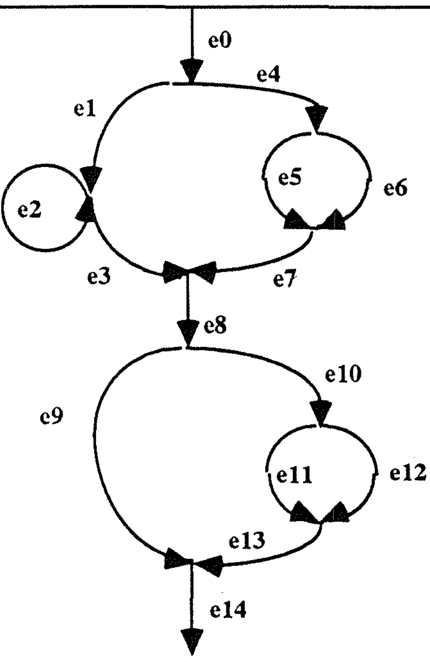

fig. 9

Proposition 3

Let $G=(V,E)$ be a ddgraph. Let E_{LWI} be a LWI arc set for G and let $e \notin UE(G)$ be in E_{LWI}. Then either e dominates e' or e is implied by e', where $e' \in UE(G)$ and there does not exist a simple path from e_0 to e_k in G containing e'.

Proof

It can be demonstrated (it has been done in [3]) that an arc $e \notin UE(G)$ dominates or is implied by at least an arc $e' \in UE(G)$. Without loss of generality, let us suppose that e dominates e'. Being e' unconstrained, there must exist a path P'' from T(e') to e_k which does not contain e'. Let us call e'' the arc in P'' which leaves T(e'). Obviously, e also dominates e'' and any other arc which is weakly incomparable with e is also weakly incomparable with e' and e'', while e' and e'' are weakly incomparable by construction. By contradiction, if there exist a simple path from e_0 to e_k in G containing e', it would also contain e, hence e and e' would not be weakly incomparable, and the set $E'_{LWI} = (E_{LWI} - \{e\}) \cup \{e', e''\}$ would have $|E'_{LWI}| > |E_{LWI}|$ against the hypothesis of E_{LWI} being a LWI arc set for G. ♦

C. S_1-structures

Traditionally, structured programs are informally defined in terms of GOTO-less programs, based on the IF-THEN-ELSE and the WHILE-DO control structures. A broader and rigorous theory of structured programs has been recently introduced [16]: different classes of program structuredness can be defined; very briefly, flowgraphs are S_n-structured if they can be constructed by repeatedly composing the class S_n of *basic* flowgraphs. Particularly, S_1-structured flowgraphs precisely correspond to the programs normally referred as structured in the literature, based on the control statements SEQUENCE, IF-THEN-ELSE, WHILE-DO and REPEAT-UNTIL. In this subsection we present a constructive definition of S_1-structured ddgraphs based on this theory.

A strictly sequential program(4) is represented by the trivial ddgraph. Then the trivial ddgraph is a S_1-ddgraph. Similarly, the other S_1-basic ddgraphs are immediately identified (see fig. 10).

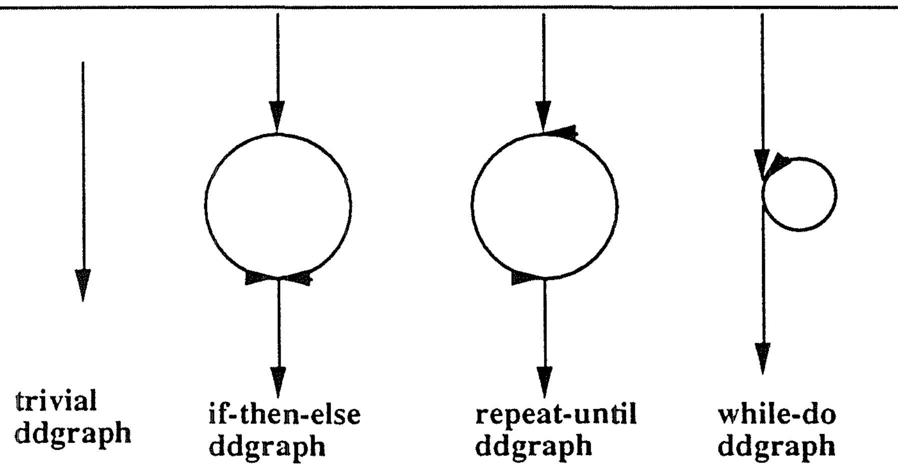

fig. 10

The class of S_1-structured ddgraphs is the class of those ddgraphs that can be constructed by the repeated application of a composition operation, starting from the basic S_1-ddgraphs, as formally stated by the two following definitions.

(4) i.e. a program only composed of SEQUENCE statements.

Definition 11: *Composition of ddgraphs*
Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k, $G'=(V',E')$ be a ddgraph with distinguished arcs e_0' and e_k'. Let e be an arc in E. The ddgraph $G(G'$ in $e)$ is defined as the ddgraph $G''=(V'',E'')$ with distinguished arcs e_0'' and e_k'', which is obtained substituting the arc e in G by the graph G'. G'' is called the *composition* of G with G' in e.

Let us observe that, by construction, G'' is a ddgraph.

Definition 12: *S_1-ddgraphs*
The class of *S_1-ddgraphs* is the smallest class of ddgraphs that satisfies the following conditions:
i) every basic ddgraph is a S_1-ddgraph;
ii) if G and G' are S_1-ddgraphs, and e is in G, then $G(G'$ in $e)$ is a S_1-ddgraph.

Proposition 4

Let $G=(V,E)$ be a S_1-structured ddgraph with distinguished arcs e_0 and e_k, which is constructed by composition of G_1 with G_2 in e. Then there exist two S_1-structured ddgraphs, G'_1 and G'_2, and an arc e' in G'_1, such that G is constructed by composition of G'_1 with G'_2 in e', and such that the ddgraph G'_2 is a basic ddgraph.

Proof

Being S_1-structured, we can consider the ddgraph G_2 as composed by two S_1-structured ddgraphs H'_1 and H'_2 in the arc e_h.

Then, we can obtain a S_1-structured ddgraph H' by the composition of G'_1 with H'_1 in the arc e. In a second step, it is possible to construct a ddgraph H, by composition of H' with H'_2 in the arc e_h. Then H is the ddgraph G.

Now, if H'_2 is a basic ddgraph, we have a proof of the theorem, where G'_1 is the ddgraph G_1, G'_2 is the ddgraph H'_2 and the arc e' is e_h. Otherwise we can iterate the procedure before on the ddgraph H'_2, until the obtained ddgraph is a S_1-basic one. Since any structured ddgraph is obtained by a finite number of compositions of simpler structured ddgraphs, this procedure eventually ends.♦

IV. THE ALGORITHM

Every definition and proposition given until this point have laid the ground for the presentation of the following algorithm, which finds a test path set suitable for the branch testing of a program represented by a ddgraph.

The algorithm FIND-A-TEST-PATH-SET (FTPS for short) constructs a path cover $\mathcal{P}=\{P_1,...,P_n\}$ for a given ddgraph $G=(V, E)$. At each iteration, the algorithm selects a still unused unconstrained arc e_u and finds a path P_u from the entry arc e_0 to the exit arc e_k, using the arc e_u. More precisely, each path from e_0 to e_k is constructed by the function FIND-A-PATH, by concatenating the unique dominance path P_{DT} in DT(G) from the entry node e_0 to the leaf e_u with the unique implication path P_{IT} in IT(G) from the leaf e_u to the exit arc e_k. Let $P_{DT}=e_0,e_{d1},...,e_{dr},e$, and $P_{IT}=e,e_{i1},...,e_{ir},e_k$. By proposition 1, there exists a path in G from e_0 to e_k having the following form:

$P_u=e_0,p_{0d_1},e_{d_1},p_{d_1d_2},...,e_{d_r},p_{d_re},e,p_{ei_1},e_{i_1},p_{i_1i_2},...,e_{i_r},p_{i_r,k},e_k,$

where each p_{ij} is the empty path if there is not a discontinuity between e_i and e_j in the corresponding tree, otherwise it is a (not empty) path from e_i to e_j in G. Thus, whenever a discontinuity between e_i and e_j is detected, the algorithm must construct the subpath p_{ij} . To this purpose, a recursive call to FIND-A-PATH is done, to construct a path from e_i to e_j in the sub-ddgraph of G with distinguished arcs e_i and e_j, using a (possibly not used) unconstrained arc. Note that the paths found may be cyclic, but, by construction, each cycle within each path will be iterated at most once.

As usually in this paper:

DT(G) and IT(G) denote the dominator and the implied tree of G, respectively.

UE denotes the set of the unconstrained arcs of G.

A path is a sequence of adjacent arcs; moreover, the algorithm will make use of the following operations over paths:

- $<e>$ returns the path formed by the only arc e;
- *p1+p2* returns a path composed by the path *p1* followed by the path *p2*, provided that the Head of the last arc in *p1* coincides with the Tail of the first arc in *p2*;
- *p1-p2* returns the path obtained by eliminating from *p1* all the arcs in *p2* provided that the arcs in *p2* are the last arcs in the path *p1*.

An auxiliary set of arcs NOT_USED is defined, which, at each moment, contains the still unexercised arcs in UE. Hence, anytime the algorithm must select an arc in UE, it chooses preferably an arc in the set NOT_USED, if there is any.

The following functions are used:

- FIND-A-PATH(G,*ea*,*eb*,NOT_USED), which returns a path in the sub-ddgraph(G,*ea*,*eb*,) from *ea* to *eb*, containing, if it is possible, an arc belonging to the set NOT_USED. As side effect, it modifies the set NOT_USED, taking away this (these) selected arc(s). It performs the recursion of the algorithm;
- *parent*(e,T), which returns the parent of the arc e in the tree T;
- *Leaves*(T), which returns the set of leaves of the tree T;
- *disc*(e,e',T), which returns true if there exists a discontinuity between the arcs e and e', i.e H(e)≠T(e) in E, and false otherwise, where e = parent(e',T).

```
Algorithm FIND-A-TEST-PATH-SET(in G: ddgraph):set of paths;
    begin
        NOT_USED:=UE;
        P:=∅; {P, initially empty, will return the test path set found}
        while |NOT_USED|>0 do
        {finding a new path p}
            p:=FIND-A-PATH(G, e0, ek, NOT_USED);
            P:= P∪{p};
        return(P);
    end algorithm.
```

fig. 11

```
Function FIND-A-PATH (in G:ddgraph; in ea,eb:arc; out NOT_USED:set
of arcs): path;
begin
        G':=sub-ddgraph(G,ea,eb);
        DT':=DT(G');
        IT':=IT(G');
        UE':=Leaves(DT')∩Leaves(IT');

{selecting an unconstrained arc $e_u$}
        if (UE'∩NOT_USED)≠∅
          then begin
                select $e_u$∈(UE'∩NOT_USED);
                NOT_USED:=NOT_USED-{e};
                end
          else select e∈UE';

{finding a path using $e_u$}
        p:=<$e_u$>;

  {finding a dominator path from $e_a$ to $e_u$}
        $e_i$:=$e_u$;
        while ($e_i$≠$e_a$)
          do begin
             $e_p$:=parent($e_i$,DT');
             if disc($e_p$,$e_i$,DT)
                then p:=FIND-A-PATH(G,$e_p$,$e_i$,NOT_USED)−<$e_i$>+p
                else p:=<$e_p$>+p;
             $e_i$:=$e_p$;
             end;

  {finding an implied path from $e_u$ to $e_b$}
        $e_i$:=$e_u$;
        while ($e_i$≠$e_b$)
          do begin
             $e_p$:=parent($e_i$,IT);
             if disc($e_p$,$e_i$,IT)
                then p:=p−<$e_i$>+FIND-A-PATH(G,$e_i$,$e_p$,NOT_USED)
                else p:=p+<$e_p$>;
             $e_i$:=$e_p$;
             end;

  return(p);
  end function;
```

fig. 12

The construction of each path requires $O(|UE|\ |E|)$ time; in fact, the first time an unconstrained arc e_u is chosen, at most other m unconstrained arcs can be contained within the path passing by e_u, i.e. m recursive calls to FIND-A-PATH can be made, with $m<|UE|<|E|$; each call to FIND-A-PATH costs $O(|E|)$, since the $O(|E|)$ procedure SUB-

DDGRAPH is called and the DT and IT of the derived sub-ddgraph must be constructed, each requiring $O(|E|)$ time[5] (Actually, at each iteration of the algorithm, the number of arcs in the derived sub-ddgraph decreases, thus $|E|$ is an upper bound). Then, $(|UE|-(m+1))<|UE|$ unconstrained arcs remain to be covered. Hence the algorithm FTPS is $O(|UE|^2\ |E|)$ time.

In the rest of this section, we shall perform a theoretical analysis of the algorithm. Theorem 2 below proves the termination and the correctness of the algorithm.

Theorem 2: *Termination and Correctness of the algorithm*

Let $G=(V,E)$ be a ddgraph with distinguished arcs e_0 and e_k. Then the algorithm FTPS terminates and returns a set of paths from e_0 to e_k covering all arcs in G.

Proof

Termination: The set of leaves is a finite set. In each iteration of each *while*, e will be Parent[e], then it is impossible not to reach the root. The recursion ends because e_0 and e_k are not unconstrained arcs: in a recursion call the sub-ddgraph has a strictly lesser number of arcs than the ddgraph, and then the algorithm will finish in a finite number of steps.

Correctness: Let P be the set of paths returned by the algorithm.

The algorithm has two different *while*s: one for the construction of a path from e_0 to the unconstrained selected arc e, and one for the construction of a path from e to e_k. The first *while* finds a path in the dominance tree. The base case is when two adjacent arcs in the path are adjacent arcs in the graph, and then the path in the graph is trivial. Otherwise, i.e. two adjacent arcs in the path are not adjacent arcs in the graph (*disc(Parent[e],e)*), the algorithm calls the recursion function in order to find a path from Parent[e] to e. Notice that from proposition 2, in the corresponding sub-ddgraph there exists at least one unconstrained arc, hence the recursion step can be made. Similarly, the second *while* finds a path in the implication tree.

When the algorithm ends, each unconstrained arc in G is covered by at least a path in P. From theorem 1, a set of paths covering the set of unconstrained arcs also covers the graph G.♦

Notice that, by the structure of the algorithm, the number n of paths found is not greater than $|UE|$. The following theorem demonstrates that, in the case of S_1-structured ddgraph G, n is generally lower than $|UE|$, depending on the number of weakly incomparable unconstrained arcs in G. Formally:

[5] Actually, [9] gives the most efficient algorithm for finding dominators which is $O(|E|\ \alpha|E|)$ time, where α is the Ackermann inverse function.

Theorem 3: *Number of Paths in a S_1-structured DDgraph*

Let G be a S_1-structured ddgraph, UE be the set of unconstrained arcs of G and $\mathcal{P}=\{P_1,...,P_n\}$ be the set of n paths constructed by the algorithm FTPS.

Then $n \leq |E_{LWI} \cap UE|_{\min} = \min\{|E_{LWI} \cap UE|: E_{LWI} \subseteq E$ is a LWI arc set of $G\}$.

Proof

Notice that at each iteration of the FIND-A-PATH function in the algorithm, the choice of an unconstrained arc is nondeterministic. This proof will show that the maximum number of paths constructed by FTPS is not greater than $|E_{LWI} \cap UE|_{\min}$, as the LWI arc set of G varies. The proof is inductive on the construction of the structured ddgraph G.

(**Basic Step**) Suppose that the ddgraph G is one of the S_1-basic ddgraphs (Figure 8).

• G is the *trivial ddgraph*: the theorem is obvious.

• G is the *If-Then-ElseDdgraph*: FTPS constructs $n=2$ paths, one for the *if-branch*, and another for the *else-branch*. On the other hand, $UE=E_{LWI}=\{$*if-branch*, *else-branch*$\}$, and the theorem follows.

• G is the *Repeat-UntilDdgraph*: FTPS constructs a unique path, selecting the (only unconstrained arc) *until-branch*. Since $UE \cap E_{LWI}=\{$*until-branch*$\}$, the theorem follows.

•*While-DoDdgraph* Case FTPS constructs a unique path, selecting the (only unconstrained arc) *while-branch*. Since $UE \cap E_{LWI}=\{$*while-branch*$\}$, the theorem follows.

(**Inductive Step**) Let us consider the ddgraph $G=(V,E)$ with distinguished arcs e_0 and e_k, which is constructed by composition of G_1 with G_2 in e. Let UE^1 (UE^2) be the unconstrained arc set for G_1 (G_2) and $E^1{}_{LWI}$ ($E^2{}_{LWI}$) be a largest weakly incomparable arc set such that $|UE^1 \cap E^1{}_{LWI}|$ is minimum. Moreover, let n_1 (n_2) be the (maximum) number of paths constructed for the ddgraph G_1 (G_2) by FTPS.

By proposition 4 we can suppose that the ddgraph G_2 is S_1-basic. Then there are four possible cases:

• G_2 is the *trivial ddgraph*: the theorem is obvious because $G_1=G$(6).

• G_2 is the *If-Then-ElseDdgraph*: Let $UE^2=E^2{}_{LWI}$ be $\{e_1,e_2\}$; it can be easily seen that e_1 and e_2 are unconstrained branches also in the ddgraph G. Let E_e be a largest weakly incomparable arc set in G_1 containing the arc e.

Obviously, $|UE^1 \cap E^1{}_{LWI}|_{\min} \leq |UE \cap E_{LWI}|_{\min} \leq |UE^1 \cap E^1{}_{LWI}|_{\min}+2$. Let us examine the 3 cases possible.

(6) Actually, G and G_1 are isomorph.

i) $|UE \cap E_{LWI}|_{min} = |UE^1 \cap E^1_{LWI}|_{min} + 2$.

i) is true only if $|E_e| = |E^1_{LWI}|$ and $e \notin UE^1$; then, $n = n_1 + 2$. In fact, the arc e in G_1 is covered by the subpath constructed by FIND-A-PATH when choosing an unconstrained arc e' which either is dominated by or implies e and such that e' is "within a cycle" (see proposition 3). In G, the worst situation in the nondeterministic choice by the function FIND-A-PATH of a yet uncovered unconstrained branch is that e' is chosen after having already constructed a path through e_1 and another path through e_2 (remember that $e_1, e_2 \in UE(G)$). Therefore, at most 2 more paths are found and the theorem follows.

ii) $|UE \cap E_{LWI}|_{min} = |UE^1 \cap E^1_{LWI}|_{min} + 1$.

ii) is true only if $|E_e| = |E^1_{LWI}|$ and $e \in UE^1$; then, $n = n_1 + 1$. In fact, since n_1 paths at most cover all the branches in G_1, n_1 paths at most also cover all the branches in G and at least one of $\{e_1, e_2\}$. Hence just another path for covering one yet uncovered branch in $\{e_1, e_2\}$ is necessary and the theorem follows.

iii) $|UE \cap E_{LWI}|_{min} = |UE^1 \cap E^1_{LWI}|_{min}$.

iii) is true only if $|E_e| < |E^1_{LWI}|$, whether $e \in UE^1$ or $e \notin UE^1$; then $n = n_1$. In fact, let S_1 be the sub-ddgraph of G_1 from T(e) to H(e), and $r = |UE^1 \cap E^{S_1}_{LWI}|$, where $E^{S_1}_{LWI}$ is a LWI arc set in S_1: by construction, $E^{S_1}_{LWI}$ contains the arc e. Suppose that of the r arcs in $(UE^1 \cap E^{S_1}_{LWI})$, there are r_1 which are incomparable with e. Suppose also that in $(UE^1 \cap E^1_{LWI})$ there are m_1 arcs which are not weakly incomparable with e (and m_2 which are w.i. with e). Notice that for each arc e' of the m_1 arcs which are not weakly incomparable with e, there must exist either a simple path from e_0 to e_k containing both e' and e or there must exist a not simple path containing both e' and e, such that the condition ii) of Definition 9 is not satisfied: these two types of path will be called *≈simple* in the following.

By the hypothesis of $|E_e| < |E^1_{LWI}|$, $m_1 > r_1$ (otherwise $|UE^1 \cap E_e| = (r_1 + m_2) \geq |UE^1 \cap E^1_{LWI}|$). Since the m_1 arcs considered $\in E^1_{LWI}$, but are not not weakly incomparable with e, they are incomparable one with the other; therefore, among the n_1 paths constructed on G_1 by the algorithm, there must be m_1 distinct paths, covering them one at a time. Further on, we can deduce that there must be at least $m_1 \geq (r_1+1)$ ≈simple paths reaching e (and hence S_1).

Reasoning now on G, we have that (r_1+1) ≈simple paths reach S, where S is the sub-ddgraph of G from T(e) to H(e) (after having substituted e with G_2). By the policy of FTPS of choosing a yet uncovered unconstrained arc whenever possible, at least (r_1+1) different incomparable unconstrained arcs can be covered in S, i.e. the n_1 paths which would cover every arc of G_1 are still sufficient to cover every arc of G, Q.E.D..

• G_2 is the *Repeat-UntilDdgraph*: The proof is similar to the case below.

• G_2 is the *While-DoDdgraph*:

Let e' be the *while-do* branch in G_2. We can obtain a largest weakly incomparable arc set of G adding the arc e' to E^1_{LWI}, and eliminating the arc e. Then $|E_{LWI}|=|E_{LWI}|+1$, if $e\notin E_{LWI}$, or $|E_{LWI}|=|E_{LWI}|$, if $e\in E_{LWI}$. By hypothesis, e' is an unconstrained arc for G. Then UE is obtained from UE^1 adding the arc e', and eliminating the arc e. Then $|UE|=|UE^1|+1$, if $e\notin UE$, or $|UE|=|UE^1|$, if $e\in UE$. Then $|E_{LWI}\cap UE|=|E^1_{LWI}\cap UE^1|+1$, if $e\notin UE$, or $|E_{LWI}\cap UE|=|E^1_{LWI}\cap UE^1|$, if $e\in UE$.

On the other hand, the number of paths cannot grow if e is an unconstrained arc (essentially, the arc e' replaces the arc e). The number of paths can grow in one is the arc e is not unconstrained. Then the result follows.♦

Particularly, if G is an acyclic ddgraph, $n=|E_{LWI}|$ and, (remembering that two arcs of an acyclic ddgraph are weakly incomparable if and only if they are incomparable) according to the Dilworth's theorem(7), the number n of paths constructed by the algorithm FTPS is the minimum possible.

For a generic structured ddgraph, the number of paths varies, depending on the policy implemented to select the next unconstrained arc to be covered (see also the example below); anyhow, it would be not much greater than the minimum possible by iterating each cycle at most once.

Theorem 3 does not hold for a not S_1-structured ddgraph. For example, in the ddgraph G_1 of figure 1, the set of unconstrained arcs is $UE=\{e_1,e_3,e_4,e_7,e_8,e_9\}$ and the LWI arc sets are $E'_{LWI}=\{e_1,e_3,e_4\}$ and $E''_{LWI}=\{e_7,e_8,e_9\}$. Thus, $|E_{LWI}\cap UE|_{\min}=3$.

The algorithm could select:
e_1 and later e_7, then P_2: $e_0\ e_1\ e_5\ e_6\ e_7\ e_{10}$;
e_3 and later e_8, then P_1: $e_0\ e_2\ e_3\ e_5\ e_6\ e_8\ e_{10}$;
e_4, then P_3: $e_0\ e_2\ e_4\ e_7\ e_{10}$;
e_9, then P_4: $e_0\ e_1\ e_5\ e_9\ e_{10}$;
and $n=4>|E_{LWI}\cap UE|_{\min}$.
But the algorithm could also select:
e_9 and later e_3, then P'_1: $e_0\ e_2\ e_3\ e_5\ e_9\ e_{10}$;
e_8 and later e_4, then P'_2: $e_0\ e_2\ e_4\ e_8\ e_{10}$;
e_7 and later e_1, then P'_3: $e_0\ e_1\ e_5\ e_6\ e_7\ e_{10}$;
thus obtaining a minimum path cover for G_1.

(7) **Dilworth's theorem** [5]

Let G be an acyclic ddgraph. The number of paths in a minimum path cover is:

$m = \max\{|E'|: E'\subseteq E$ and, for each $e,e'\in E'$, with $e\neq e'$, e and e' are incomparable$\}$.

The same example also shows how different implementations of the policy of selection of the next unconstrained arc (actually left nondeterministic in our algorithm) can bring to different path sets, with possibly different cardinality. Particularly, in the above example, the first set of paths is built choosing the yet uncovered unconstrained arc having the lowest associated index, while the second one is built choosing the highest associated index.

V. CONCLUSIONS

We have dealt with the problem of selecting a meaningful set of test paths which satisfies the branch testing criterion. We have used a flowgraph representation model for computer programs called ddgraph and we have individuated a subset of ddgraph arcs, the unconstrained arcs, having the property that a path set which covers all of them is a path cover.

Based on this property, we have presented an algorithm which can be applied to a generic program; it finds a minimal path cover for acyclic S_1-structured programs, and a "nearly minimal" path cover for S_1-structured programs. Very briefly, the algorithm builds one path at a time, each path being composed by a number of subpaths, and each subpath, in turn, containing one unconstrained arc. The selection of the next unconstrained arc is nondeterministic (actually, preference is given to yet uncovered arcs) . Obviously, one could fix a policy to select a more suitable unconstrained arc, for example one which is "within a loop", in order to further lessen the number of paths found.

Being recursive, the algorithm is very simple. It works in $O(|UE|^2\,|E|)$ time. However, a modified version of the same algorithm only working for structured programs can be easily derived which works in $O(|E| + |UE|\,|E|)$ time. Intuitively, in this case, it would not be necessary to construct neither a sub-ddgraph nor the DT nor the IT at each discontinuity (which costs $O(|E|)$ time), since the DT and the IT of the complete S_1-structured ddgraph can always be used.

Besides, the algorithm has been presented within a theoretical, extensive framework, thanks to which termination and correctness results have been proved and an analysis of the functioning have been done.

We want to stress that the results in the paper are of straightforward usability in the structural testing, while preparing the test plan, and, in general during the static analysis of program structure. Indeed, the work developed within the paper is aimed at program testers who should find it useful in view of its immediate applicability. In fact, the algorithm presented has been implemented within a prototype tool, called BAT [4], which works for C [8] programs. An experimental evaluation of the functioning of the algorithm is shown in Table 1. Obviously, the time of the algorithm depends on the control flow structure of the program analyzed, and not on the number of statements. Indicatively, we give with each entry the number of arcs in the derived ddgraph, the number of unconstrained arcs, the Mc-Cabe cyclomatic number [10] c(8), the number of paths found, the number of discontinuities encountered while performing the path search, and the time spent. The latter was measured in terms of the time spent on the basic *while-do* ddgraph (figure 10). Some realistic cases are shown; as it can be expected, the time depended heavily on the number of discontinuities. Note the last entry in the table, referring to a very intricate program (c=49!) we used to perform a *stress testing* of BAT.

Table 1

$\lvert E\rvert$	$\lvert UE\rvert$	c	n	*Discontinuities*	**Time**(9)
3	1	2	1	0	1
4	2	2	2	0	1.3
16	10	11	10	0	5
8	4	4	3	1	27
19	7	9	4	3	96
10	6	5	2	5	121
19	13	10	2	9	510
25	11	11	6	10	407
100	50	49	8	149	$\sim 10^6$

ACKNOWLEDGEMENTS

We thank Marco Giromini and Amedeo Astrea who designed and built the BAT (prototype) tool.

(8) As known, $c=(\lvert E\rvert-\lvert V\rvert+2)$

(9) The unit of time approximately corresponded to 430 microseconds on a DEC Station 5000/125, also comprehending system time.

REFERENCES

[1] W. R. Adrion, M. A. Branstad and J. C. Cherniavsky, "Validation, Verification and Testing of Computer Software", *ACM Comp. Surveys*, vol. 14, No. 2, pp.159-192, June 1982.

[2] C. Berge, *Graphs and Hypergraphs*, North-Holland, 1973.

[3] A. Bertolino, "Unconstrained Edges and Their Application to Branch Analysis and Testing of Programs", to appear on *The Journal of Systems and Software* .

[4] A. Bertolino, M. Giromini, "Easy Branch Testing", *Proc. of the Int. Conf. on Achieving Quality in Software (AQuIS '91)*, pp. 251-258, Pisa, Italy, April 22-24,1991.

[5] R. P. Dilworth, "A decomposition theorem for partially ordered sets", *Annals Math.*, vol. 51, No. 1, 161-166, Jan. 1950.

[6] H. N. Gabow, S. N. Maheshwari, and L. J. Osterweil, "On Two Problems in the Generation of Program Test Paths", *IEEE Trans. on Software Engineering*, vol.SE-2, No. 3, 227-231, Sept. 1976.

[7] M. S. Hecht, *Flow Analysis of Computer Programs,* North Holland, 1977.

[8] B. W. Kernighan, and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.

[9] T. Lengauer, and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph", *ACM Trans. on Programming Languages and Systems*, vol. 1, pp.121-141, 1979.

[10] T. J. McCabe, "A Complexity Measure", *IEEE Trans. on Software Engineering*, vol. SE 2, No. 4, p.308-320, December 1976.

[11] E. F. Miller, "Software Testing Technology: An Overview", in *Handbook of Software Engineering* (C. R. Vick and C. V. Ramamoorthy, eds.), Van Nostrand Reinhold Company, 1984.

[12] E. F. Miller, M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural Techniques of Program Validation", *Digest COMPCON74*, p. 161-164, 1974.

[13] S. C. Ntafos, and S. Louis Hakimi, "On Path Cover Problems in Digraphs and Applications to Program Testing", *IEEE Trans. on Software Engineering*, vol.SE-5, No. 5, 520-529, Sep. 1979.

[14] H. S. Wang, S.R. Hsu, and J.C. Lin, "A Generalized Optimal Path Selection Model for Structural Program Testing", *The Journal of Systems and Software* , vol. 10, pp. 55-63, 1989.

[15] E. Weyuker, "Translatability and Decidability Questions for Restricted Classes of Program Schemas", *SIAM Journal on Computers*, vol. 8, No. 4, pp. 587-598, 1979.

[16] R.W. Whitty, N.E. Fenton and A.A. Kaposi, "A rigorous approach to structural analysis and metrication of software", *Software and Microsystems,* vol. 4, No. 1, pp.2-16, February 1985.