An Applicable Family of Data Flow Testing Criteria

PHYLLIS G. FRANKL AND ELAINE J. WEYUKER

***Abstract*—A test data adequacy criterion is a predicate which is used to determine whether a program has been tested "enough." An adequacy criterion is applicable if for every program there exists a set of test data for the program which satisfies the criterion. Most test data adequacy criteria based on path selection fail to satisfy the applicability property because, for some programs with unexecutable paths, no adequate set of test data exists.**

In this paper, we extend the definitions of the previously introduced family of data flow testing criteria to apply to programs written in a large subset of Pascal. We then define a new family of adequacy criteria called feasible data flow testing criteria, which are derived from the data flow testing criteria. The feasible data flow testing criteria circumvent the problem of nonapplicability of the data flow testing criteria by requiring the test data to exercise only those definition-use associations which are executable. We show that there are significant differences between the relationships among the data flow testing criteria and the relationships among the feasible data flow testing criteria.

We also discuss a generalized notion of the executability of a path through a program unit. A script of a testing session using our data flow testing tool, ASSET, is included in the Appendix.

***Index Terms*—Data flow analysis, software testing, software validation.**

I. Introduction

An important problem in software testing is deciding when to stop. An *adequacy criterion* is a predicate which is used to determine whether a program has been tested "enough." Several software test data adequacy criteria are based on the idea that one cannot consider a program to be adequately tested if certain sequences of statements have never been executed by any test data. These methods generally associate a *test set* T, which is a subset of the input domain of the specification of a program P, with the set Π of paths through P's flow graph, which are executed when the program is run with inputs from T. The test set T, or equivalently the set of paths Π, is said to satisfy criterion C for programs P ("T is C-adequate for P") if and only if each of the sequences required by C is a subpath of one of the paths in Π.

The most well known of these criteria are statement testing, branch testing, and path testing, which require that the test data cause every node, branch, or path, respectively, in the program's flow graph to be executed [11], [12]. Unfortunately, statement and branch testing can fail to expose many common errors, and path testing is usually infeasible since programs with loops have infinitely many paths [7], [10]. Several criteria which are based on analysis of the program's control flow and which are stronger than branch testing but weaker than path testing have been proposed [11], [15], [23].

Recently, a number of test data adequacy criteria which are based on data flow (DF) analysis, some of which "bridge the gap" between branch testing and path testing, have been proposed and studied [1], [9], [14], [16], [18], [19]. Tools based on some of them have been implemented [2], [3], [6], [13]. These criteria are based on the intuition that one should not feel confident that a variable has been assigned the correct value at some point in the program if no test data cause the execution of a path from the assignment to a point where the variable's value is subsequently used.

All of these criteria suffer from the weakness that for programs with unexecutable paths it may be impossible for any test set to satisfy the given adequacy criterion. For example, consider a program containing the statement "**for** $i := 1$ **to** 5 **do** S." For each $n \geq 0$, there is at least one path through the program's flow graph which traverses the loop n times. However, those paths corresponding to $n \neq 5$ can never be executed. Such a program could not be adequately tested using the path testing criterion, even if it were possible to do exhaustive testing. Experience using our tool, ASSET, has shown that, for many programs, unexecutable paths make it impossible for any test to satisfy a given DF testing criterion [2], [3]. This is clearly an undesirable situation.

An adequacy criterion C satisfies the *applicability property* if and only if for every program P there exists some test set which is C-adequate for P [22]. One would expect a "good" adequacy criterion C to satisfy the applicability property. However, the statement testing, branch testing, path testing, and DF testing criteria all fail to satisfy the applicability property. Furthermore, for each of them it is undecidable whether a test set exists which adequately tests a given program.

One way to enforce the applicability of a criterion C is to restrict the class of programs considered to A_C, the set of programs for which there exists a C-adequate test set. Unfortunately, for each DF testing criterion C (as well as the other criteria mentioned above), A_C excludes many "typical" programs. Furthermore, it is undecidable whether a given program belongs to A_C. These drawbacks lead us to reject this approach. Instead, we define a new

Manuscript received September 15, 1986. This research was supported in part by the National Science Foundation under Grant DCR8501614 and by the Office of Naval Research under Contract N00014-85-K-0414.

P. G. Frankl is with the Department of Electrical Engineering and Computer Science, Polytechnic University, Brooklyn, NY 11201.

E. J. Weyuker is with the Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, New York, NY 10012.

IEEE Log Number 8823071.

family of adequacy criteria by modifying the old criteria so as to ensure applicability.

In this paper, we define a new family of adequacy criteria, which are derived from the DF testing criteria proposed in [18], [19] and which satisfy the applicability property. Roughly speaking, for each of these new criteria, a test is adequate if and only if it comes "as close as possible" to satisfying the corresponding DF testing criterion. These criteria will be defined precisely, and the relationships between them will be explored in Section III. In Section II, we summarize the theory of DF testing, extending it to apply to programs written in Pascal. In Section IV, we define and discuss a generalization of the new family of criteria which takes into account information about the context in which the subprogram being tested is called.

II. Definitions of the DF Testing Criteria

A family of test data adequacy criteria, based on analysis of the DF characteristics of the program being tested, was defined in [18]. These criteria, which we call *data flow testing criteria*, or DF testing for short, were originally defined for a very simple universal programming language consisting of assignment statements, conditional and unconditional transfer statements, and I/O statements. They require that the test data exercise certain paths from a point in a program where a variable is defined to points where the variable is subsequently used. A tool, ASSET, which performs DF testing on programs written in such a language, is described in [2].

In order to make DF testing more practical, we have extended it to apply to a large subset of Pascal and have enhanced ASSET accordingly. The basic ideas behind DF testing apply to testing programs written in other imperative languages, but for precision it is necessary to specify a particular syntax. We now summarize the extended theory of DF testing.

We apply DF testing to an individual subprogram, i.e., a main program, a procedure, or a function. To execute a procedure or function P, we must call it from a *driver program*. Thus, to test a procedure or function P, we need a *test-set/driver-program* pair (T, D) where D is a program which might call P and T is a subset of the input domain of the specification for D. Obviously, the path (or paths) through P which is executed when a particular test case is input to D will depend on D, as well as on the test case. We will often omit reference to the driver program when it is obvious which driver program is calling the subprogram. Similarly, we may omit reference to the driver program if it simply reads in the arguments to the subprogram in order and then calls the subprogram once.

As a technical convenience, we assume that the subprogram being tested has no **goto** statements, no **with** statements, no variant records, no functions having **var** parameters, no procedural or functional parameters, and no conformant arrays. It would not be difficult to relax these assumptions. We also assume that in every conditional statement the Boolean expression which determines the flow of control has at least one occurrence of a variable or a call to the function *eof* or to the function *eoln*.

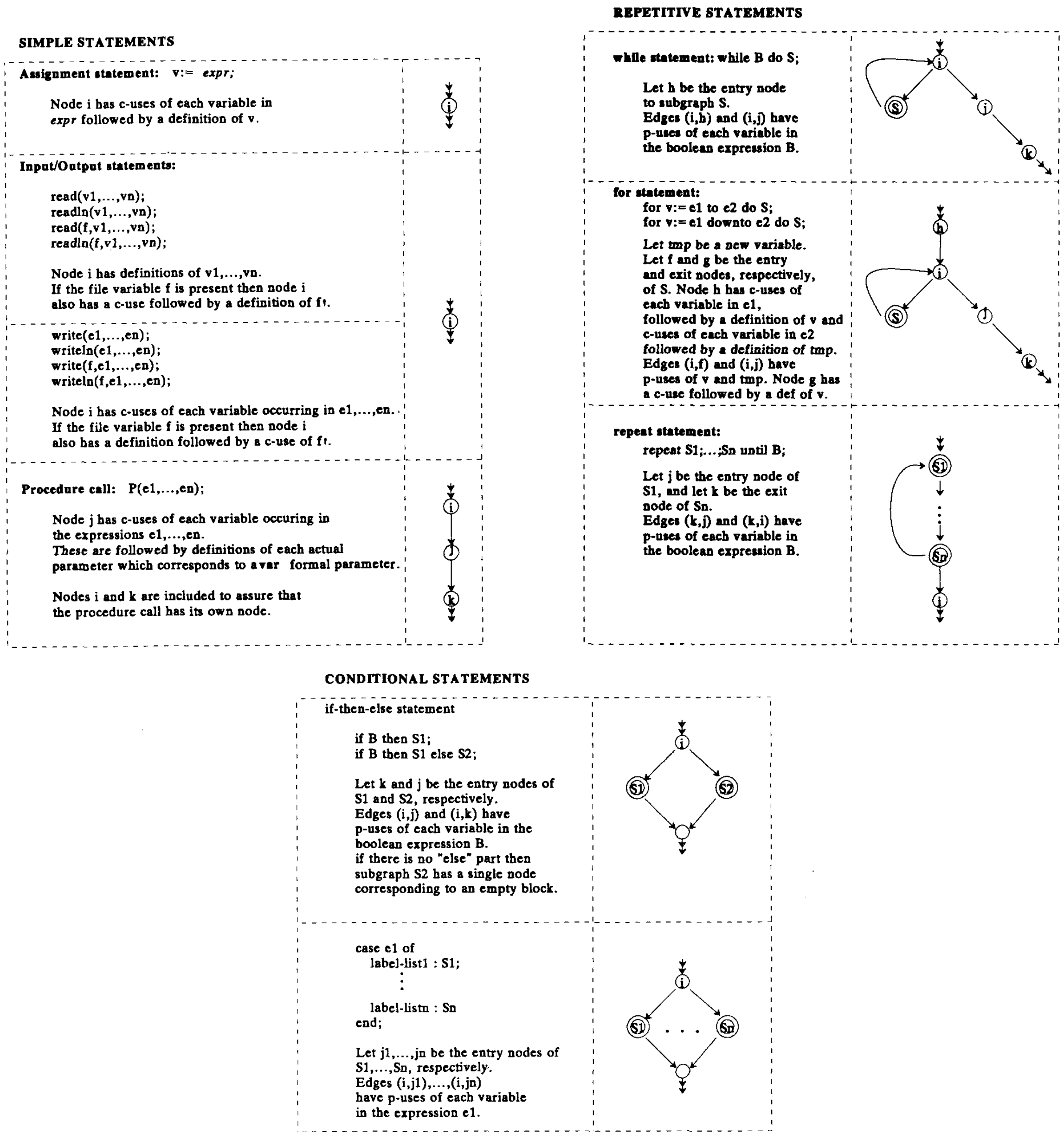
A subprogram can be uniquely decomposed into a set of disjoint blocks of statements. A *block* is a maximal sequence of simple statements having the properties that it can only be entered through the first statement and that, whenever the first statement is executed, the remaining statements are executed in the given order. The subprogram to be tested is represented by a *flow graph* in which the nodes correspond to the blocks of the subprogram and edges indicate possible flow of control between blocks. As a technical convenience, some nodes which correspond to empty sequences of statements may also be added to the flow graph. Fig. 1 shows the subgraphs corresponding to statements in the language. The subprogram's flow graph is obtained by merging the exit node of each statement with the entry node of the following statement. An entry node preceding the first statement of the procedure and an exit node succeeding the last statement are added.

DF analysis was originally used for compiler optimization [8], [20]. It generally classifies each variable occurrence as being a *definition*, in which a value is stored in a memory location, a *use*, in which a value is fetched from a memory location, or an *undefinition*, in which the value and the location become unbound. For our purposes, we will also distinguish between two different types of use. The first type directly affects the computation being performed or outputs the result of some earlier definition. We call such a use a *computation use*, or a *c-use*. Of course, a c-use may indirectly affect the flow of control through the subprogram. In contrast, the second type of use directly affects the flow of control through the subprogram, and thereby may indirectly affect the computations performed. We call such a use a *predicate use* or *p-use*.

We will associate a sequence of definitions and c-uses with each node in the flow graph and will associate a set of p-uses with each edge in the flow graph. Fig. 1 shows the classification of variable occurrences in the language's statements. In addition, the entry node is considered to have a definition of each parameter, each nonlocal variable which occurs in the subprogram, and the input buffer *input*↑, which may implicitly occur in calls to the standard procedures/functions *read*, *readln*, *eoln*, and *eof*. The exit node has an *undefintion* of each local variable, a c-use of each variable parameter, a c-use of each nonlocal variable, and a c-use of the input buffer *input*↑.

We now discuss how DF analysis is handled for structured variables. Since it is not possible, in general, to determine the particular array element which is being defined or used in an occurrence of an array variable, any definition of the variable $a[e]$ will consist of a c-use of each variable occurring in the expression e, followed by a definition of a. Any use of $a[e]$ will consist of uses of all of the variables occurring in e, followed by a use of a.

Similarly, we will treat pointers purely syntactically, making no attempt to perform DF analysis on dereferenced pointers. If p is a pointer variable, a definition of



SIMPLE STATEMENTS

Assignment statement: v:= *expr;*

Node i has c-uses of each variable in *expr* followed by a definition of v.

Input/Output statements:

read(v1,...,vn);
readln(v1,...,vn);
read(f,v1,...,vn);
readln(f,v1,...,vn);

Node i has definitions of v1,...,vn.
If the file variable f is present then node i also has a c-use followed by a definition of f↑.

write(e1,...,en);
writeln(e1,...,en);
write(f,e1,...,en);
writeln(f,e1,...,en);

Node i has c-uses of each variable occurring in e1,...,en.
If the file variable f is present then node i also has a definition followed by a c-use of f↑.

Procedure call: P(e1,...,en);

Node j has c-uses of each variable occuring in the expressions e1,...,en.
These are followed by definitions of each actual parameter which corresponds to a **var** formal parameter.

Nodes i and k are included to assure that the procedure call has its own node.

REPETITIVE STATEMENTS

while statement: while B do S;

Let h be the entry node to subgraph S.
Edges (i,h) and (i,j) have p-uses of each variable in the boolean expression B.

for statement:
for v:=e1 to e2 do S;
for v:=e1 downto e2 do S;

Let tmp be a new variable.
Let f and g be the entry and exit nodes, respectively, of S. Node h has c-uses of each variable in e1, followed by a definition of v and c-uses of each variable in e2 followed by a definition of tmp.
Edges (i,f) and (i,j) have p-uses of v and tmp. Node g has a c-use followed by a def of v.

repeat statement:
repeat S1;...;Sn until B;

Let j be the entry node of S1, and let k be the exit node of Sn.
Edges (k,j) and (k,i) have p-uses of each variable in the boolean expression B.

CONDITIONAL STATEMENTS

if-then-else statement

if B then S1;
if B then S1 else S2;

Let k and j be the entry nodes of S1 and S2, respectively.
Edges (i,j) and (i,k) have p-uses of each variable in the boolean expression B.
if there is no "else" part then subgraph S2 has a single node corresponding to an empty block.

case e1 of
  label-list1 : S1;
  ⋮
  label-listn : Sn
end;

Let j1,...,jn be the entry nodes of S1,...,Sn, respectively.
Edges (i,j1),...,(i,jn) have p-uses of each variable in the expression e1.

Fig. 1. Control flow and DF for statement in the language. *Si*○ denotes the subgraph corresponding to statement *Si*.

$p\uparrow$ consists of a c-use of p followed by a definition of $p\uparrow$, and a use of $p\uparrow$ consists of a use of p followed by a use of $p\uparrow$. Since it is not possible to determine statically the memory location to which a pointer points, we will ignore the definitions and uses of $p\uparrow$.

Each field of a record is treated as an individual variable. Any unqualified occurrence of a record is treated as an occurrence of each field of the record. Occurrences of file variables in I/O statements are handled by considering the effect of the statement on the file buffer.

Note that our model of data flow may not reflect the actual DF in the subprogram being tested completely accurately. For example, we have made no attempt to perform any interprocedural data flow analysis, have ignored dereferenced pointers, have made no attempt to disambiguate array references, and have ignored potential aliasing and side effects. In an optimizing compiler, it is imperative that conservative assumptions be made about the flow of data, lest a code transformation which changes the semantics of the program be performed. In the context of DF testing, however, such caution is not strictly necessary. On the other hand, it seems reasonable to expect that more accurate DF analysis will force the selection of better test data. In [3], we compare the test data needed to test programs adequately when each array is treated as a single entity to the test data required to test transformed programs adequately in which array references are disambiguated and each element of the array is treated as an individual entity. More exploration of the tradeoff between the difficulty of performing accurate DF analysis and the quality of the resulting test data is needed.

We are interested in tracing the flow of data *between* nodes, and thus define a c-use of a variable x in node i to be a *global c-use* if the value of x has been assigned in some block other than block i. Let x be a variable occurring in a subprogram. A path $(i, n_1, \cdots, n_m, j)$, $m \geq 0$, containing no definitions or undefinitions of x in nodes $n_1, \cdots, n_m$ is called a *definition clear path with respect to* x (def-clear path wrt x) from node i to node j and from node i to edge (n_m, j). A node i has a *global definition* of a variable x if it has a definition of x and there is a def-clear path wrt x from node i to some node containing a global c-use or edge containing a p-use of x. Since every p-use is associated with a potential transfer of control from one node to another, there is no need to distinguish between p-uses and global p-uses.

We restrict the class of subprograms to which DF testing applies to those subprograms P satisfying the following two properties.

1) No-Syntactic-Undefined-P-use Property (NSUP): For every p-use of a variable x on an edge (i, j) in P, there is some path from the start node to edge (i, j) which contains a global definition of x.

2) Non-Straight-Line Property (NSL): P has at least one conditional or repetitive statement.

Note that the NSL property guarantees that at least one node in P's flow graph has more than one successor and that at least one variable has a p-use in P.

The subprogram's *def-use graph* is obtained from the flow graph by associating with each node i the sets *c-use* (i) = {variables which have global c-uses in block i} and *def* (i) = {variables which have global definitions in block i} and associating with each edge (i, j) the set *p-use* (i, j) = {variables which have p-uses on edge (i, j)}. We also define sets of nodes *dcu* (x, i) = {nodes j such that $x \in$ c-use(j) and there is a def-clear path with respect to x from i to j} and *dpu* (x, i) = {edges (j, k) such that $x \in$ p-use(j, k) and there is a def-clear path with respect to x from i to (j, k)}. These definitions are summarized in Fig. 2.

Thus, if $x \in$ def(i) and $j \in$ dcu(x, i), then x has a global definition in node i and a c-use in node j, and there is a definition clear path with respect to x from node i to node j. Therefore, it may be possible for control to reach node j with the variable x having the value which was assigned to it in node i.

A *definition-c-use association* is a triple (i, j, x) where i is a node containing a global definition of x and $j \in$ dcu(x, i). A *definition-p-use association* is a triple $(i, (j, k), x)$ where i is a node containing a global definition of x and $(j, k) \in$ dpu(x, i). A *simple path* is one in which all nodes, except possibly the first and last, are distinct. A *loop-free path* is one in which all nodes are distinct. A path $(n_1, \cdots, n_j, n_k)$ is a *du-path* with respect to a variable x if n_1 has a global definition of x and either

1) n_k has a global c-use of x and $(n_1, \cdots, n_j, n_k)$ is a def-clear simple path with respect to x, or

2) (n_j, n_k) has a p-use of x and $(n_1, \cdots, n_j)$ is a def-clear loop-free path with respect to x.

An *association* is a definition–c-use association, a definition–p-use association, or a du-path.

V	=	the set of variables
N	=	the set of nodes
E	=	the set of edges
def(i)	=	{x ∈ V ∣ x has a global definition in block i}
c-use(i)	=	{x ∈ V ∣ x has a global c-use in block i}
p-use(i,j)	=	{x ∈ V ∣ x has a p-use in edge (i,j) }
dcu(x,i)	=	{j ∈ N ∣ x ∈ c-use(j) and there is a def-clear path wrt x from i to j}
dpu(x,i)	=	{(j,k) ∈ E ∣ x ∈ p-use(j,k) and there is a def-clear path wrt x from i to (j,k) }

Fig. 2. Definitions.

A *complete path* is a path from the entry node to the exit node of the flow graph. A complete path π *covers* a definition–c-use association (i, j, x) [respectively, a definition–p-use association $(i, (j, k), x)$] if it has a definition clear subpath with respect to x from i to j [respectively, from i to (j, k)]. π covers a du-path π' if π' is a subpath of π. A set Π of paths covers an association if some element of the set does. A test-set/driver-program pair (D, T) covers an association if, when input to D, the elements of T cause the execution of the set of paths Π, and Π covers the association.

Roughly speaking, the family of DF testing criteria is based on requiring that the test data execute definition clear paths from each node containing a global definition of a variable to specified nodes containing global c-uses and edges containing p-uses of that variable. For each variable definition, we can demand that $\begin{bmatrix} \textit{all} \\ \textit{some} \end{bmatrix}$ definition clear paths with respect to that variable from the node containing the definition to $\begin{bmatrix} \textit{all} \\ \textit{some} \end{bmatrix}$ of the $\begin{bmatrix} \textit{uses} \\ \textit{c-uses} \\ \textit{p-uses} \end{bmatrix}$ reachable by some such paths be executed. The criteria are defined precisely in Fig. 3.

If variable x has a global definition in node i, the all-defs criterion requires the test data to exercise *some* path which goes from node i to *some* node or edge at which the value assigned to x in node i is used. The all-uses criterion requires the test data to exercise *at least one* path to *each* such node and to *each* such edge. The all-du-paths criterion requires that *all* of the du-paths from i to *each* such node and *each* such edge be exercised. The criteria all-p-uses, all-c-uses, all-p-uses/some-c-uses, and all-c-uses/some-p-uses place emphasis on either c-uses or p-uses. Note that any subprogram has only finitely many definition–use associations, so none of the DF criteria requires an infinite amount of test data. Upper bounds on

THE DATA FLOW TESTING CRITERIA

A test-set/driver-program pair (T,D) satisfies criterion C for subprogram P if and only if for each node i in P's flow graph and each x ∈ def(i) the set Π of paths executed by T covers the following associations:

CRITERION	ASSOCIATIONS REQUIRED
All-defs	Some (i,j,x) s.t. j∈dcu(x,i) or some (i,(j,k),x) s.t. (j,k)∈dpu(x,i).
All-c-uses	All (i,j,x) s.t. j∈dcu(x,i).
All-p-uses	All (i,(j,k),x) s.t. (j,k)∈dpu(x,i).
All-p-uses/some-c-uses	All (i,(j,k),x) s.t. (j,k)∈dpu(x,i). In addition, if dpu(x,i)=ϕ then some (i,j,x) s.t. j∈dcu(x,i). Note that since i has a *global* definition of x, dpu(x,i)=ϕ ⇒ dcu(x,i)≠ϕ.
All-c-uses/some-p-uses	All (i,j,x) s.t. j∈dcu(x,i). In addition, if dcu(x,i)=ϕ then some (i,(j,k),x) s.t. (j,k)∈dcu(x,i). Note that since i has a *global* definition of x, dcu(x,i)=ϕ ⇒ dpu(x,i)≠ϕ.
All-uses	All (i,j,x) s.t. j ∈ dcu(x,i) and all (i,(j,k),x) s.t. (j,k)∈dpu(x,i).
All-du-paths	All du-paths from i to j with respect to x for each j∈dcu(x,i) and all du-paths from i to (j,k) with respect to x for each (j,k)∈dpu(x,i).

For comparison we also define the criteria all-nodes (all-edges, all-paths, respectively) which require that Π cover every node (every edge, every path, respectively) in the flow graph.

Fig. 3. Definitions of the DF testing criteria.

the amount of test data required by the DF criteria are established in [21].

Criterion C_1 *includes* criterion C_2 if and only if, for every subprogram, any test-set/driver-program pair which satisfies C_1 also satisfies C_2. Criterion C_1 *strictly includes* criterion C_2, denoted $C_1 \Rightarrow C_2$, if and only if C_1 *includes* C_2 and for some subprogram P there is a test-set/driver-program pair which satisfies C_2 but does not satisfy C_1. The notion of *subsumption* in [1] is similar to our notion of inclusion.

Rapps and Weyuker proved that for the simple language for which DF testing was originally defined, the relationship among the criteria is as shown in Fig. 4 [19]. Clarke *et al.* [1] have shown the relationship of the criteria defined by Laski and Korel [14] and Ntafos [16] to the DF criteria.

In extending the theory of DF testing to apply to programs written in Pascal, we have preserved the inclusion relations among the DF criteria. Doing so required the inclusion of definitions of all nonlocal variables in the entry node of the procedure and careful treatment of implicit uses of the variable *input*↑. For symmetry, we have also added the all-c-uses criterion, which was not defined in [19]. For more details of the proof that the relationship among the criteria is as shown in Fig. 4, see [5].

III. The Feasible DF Testing Criteria

Given a subprogram P and a DF criterion C, it may be the case that no test-set/driver-program pair for P satisfies C. This occurs when none of the paths which cover a particular association required by C is executable. In such a case, P cannot be adequately tested according to C. In this section, we introduce a new family of criteria, derived from the DF criteria, which circumvent this problem, and

$$\begin{tikzcd}
 & & \text{ALL-PATHS} \arrow[d, Rightarrow] & & \\
 & & \text{ALL-DU-PATHS} \arrow[d, Rightarrow] & & \\
 & & \text{ALL-USES} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \\
 & \text{ALL-C-USES/SOME-P-USES} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & & \text{ALL-P-USES/SOME-C-USES} \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
\text{ALL-C-USES} & & \text{ALL-DEFS} & & \text{ALL-P-USES} \arrow[d, Rightarrow] \\
 & & & & \text{ALL-EDGES} \arrow[d, Rightarrow] \\
 & & & & \text{ALL-NODES}
\end{tikzcd}$$

Fig. 4. The relationship among the DF testing criteria.

investigate some of its properties. We assume that all aliasing and side effects are known. We also assume that no element of the test set causes the program to crash; thus, if a test case causes the execution of the entry node of some subprogram, it will cause the execution of a path from the entry to the exit of that subprogram.

Recall that a *complete path* is a path from the entry node to the exit node of a subprogram's flow graph. We say that a complete path is *executable* or *feasible* if there exists some assignment of values to input variables, nonlocal variables, and parameters which causes the path to be executed. We say that a path is executable if it is a subpath of an executable complete path. Similarly, a node or edge is executable if it lies on some executable complete path. According to this definition, the question of whether or not a given path through a subprogram is executable is independent of the context in which that subprogram is called. However, it may depend on the effects of any procedures or functions which are called along the path. In Section IV, we will discuss the consequences of modifying this notion of executability to take into account information about the context in which the subprogram is called. Note that whether or not a particular path is executable depends on the actual subprogram, not just on its def–use graph.

We say that an association is *executable* if there is some executable complete path which covers it; otherwise, it is *unexecutable*. We define subsets $\text{fdcu}(x, i) \subseteq \text{dcu}(x, i)$ and $\text{fdpu}(x, i) \subseteq \text{dpu}(x, i)$, whose elements correspond to those associations which are executable as follows: *fdcu* $(x, i) = \{$nodes j such that $x \in$ c-use(j) and there is an *executable* definition clear path with respect to x from i to $j\}$. *fdpu* $(x, i) = \{$edges (j, k) such that $x \in p$-use(j, k) and there is an *executable* definition clear path with respect to x from i to $(j, k)\}$. Equivalently, $\text{fdcu}(x, i) = \{ j \in \text{dcu}(x, i) \mid$ the association (i, j, x) is executable$\}$ and $\text{fdpu}(x, i) = \{(j, k) \in \text{dpu}(x, i) \mid$ the association $(i, (j, k), x)$ is executable$\}$. For each DF criterion C, we define a new criterion C^* by selecting the required associations from $\text{fdcu}(x, i)$ and $\text{fdpu}(x, i)$ instead of from $\text{dcu}(x, i)$ and $\text{dpu}(x, i)$. Precise definitions of these

THE FEASIBLE DATA FLOW TESTING CRITERIA

fdcu(x,i) = {j∈ dcu(x,i)| the association (i,j,x) is executable}
fdpu(x,i) = {(j,k)∈ dpu(x,i)| the association (i,(j,k),x) is executable}

A test-set/driver-program pair (T,D) satisfies criterion C for subprogram P if and only if for each node i in P's flow graph and each x ∈ def(i) the set Π of paths executed by T covers the following associations:

CRITERION	REQUIRED ASSOCIATIONS
(all-defs)*	if fdcu(x,i) ∪ fdpu(x,i) ≠ ϕ then some (i,j,x) s.t j∈ fdcu(x,i) or some (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-c-uses)*	all (i,j,x) s.t. j∈ fdcu(x,i).
(all-p-uses)*	all (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-p-uses/some-c-uses)*	all (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i). In addition, if fdpu(x,i) = ϕ and fdcu(x,i) ≠ ϕ then some (i,j,x) s.t. j∈ fdcu(x,i).
(all-c-uses/some-p-uses)*	all (i,j,x) s.t. j∈ fdcu(x,i). In addition, if fdcu(x,i) = ϕ and fdpu(x,i) ≠ ϕ then some (i,(j,k),x) s.t. (j,k)∈ fdpu(x,i).
(all-uses)*	all (i,j,x) s.t. j ∈ fdcu(x,i) and all (i,(j,k),x) s.t. (j,k) ∈ fdpu(x,i).
(all-du-paths)*	all executable du-paths with respect to x from i to j s.t. j∈ dcu(x,i) and all executable du-paths with respect to x from i to (j,k) for each (j,k) ∈ dpu(x,i).

For comparison we also define the criteria (all-nodes)* [(all-edges)*, (all-paths)*, respectively] which require that Π cover each executable node [each executable edge, each executable path, respectively.]

Fig. 5. Definitions of the feasible DF testing criteria.

criteria are given in Fig. 5. We refer to the criteria {(all-du paths)*, (all-uses)*, (all-p-uses/some-c-uses)*, (all-c-uses/some-p-uses)*, (all-p-uses)*, (all-c-uses)*, and (all-defs)*} as *feasible DF testing criteria*, or FDF criteria for short.

The FDF criteria satisfy the applicability property: for any subprogram P and any FDF criterion C^*, there is some test set T which satisfies C^*. However, the question of whether a particular T satisfies C^* for subprogram P is undecidable. In going from the family DF to the family FDF, we have traded the undecidability of the existence question "Is there any test set which is C-adequate for P?" for the undecidability of the recognition problem "Is a given test set C^*-adequate for P?"

Observe that for any DF criterion C, $C \Rightarrow C^*$. We now investigate the inclusion relations along the FDF criteria.

Theorem 1: The family of FDF criteria is partially ordered by strict inclusion, as shown in Fig. 6. Furthermore, FDF criterion C_i^* includes FDF criterion C_j^* if and only if the inclusion is explicitly shown in the figure or follows from the transitivity of the relations.

Proof:

A) Strictness of the Inclusions: We first observe that if subprogram P has no unexecutable paths, then a test set is C-adequate for P if and only if it is C^*-adequate for P. This observation, along with the proofs of strictness of the inclusions in Theorem 1 of [19], none of which involves subprograms with unexecutable paths, shows that all of the inclusions in Fig. 6 are strict. It thus suffices to show that the inclusions in Fig. 6 hold.

B1) (all-paths) ⇒ (all-uses)*:* Suppose that this does not hold. Then there are a subprogram P and a set T of

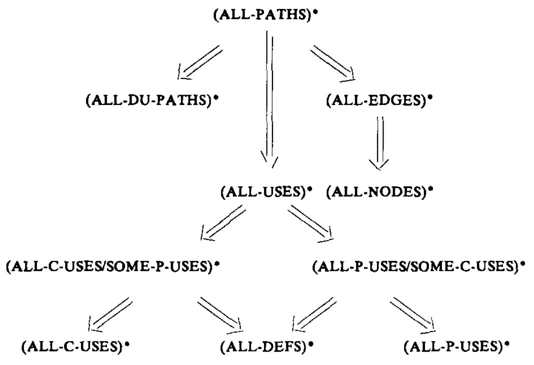


Fig. 6. Relationship among the FDF testing criteria.

test data which are (all-paths)*-adequate for P, but not (all-uses)*-adequate. Let Π be the set of paths through P which T executes. There exist a node i in P with a global definition of some variable x, a node j with a global c-use of x or edge (j, k) with a p-use of x, and an *executable* definition clear path with respect to x from i to j [respectively, from i to (j, k)] which is not covered by Π. This contradicts the fact that Π covers every executable path.

B2) (all-paths) ⇒ (all-du-paths)*:* Suppose that this does not hold. Then there are a subprogram P and a set T of test data which are (all-paths)*-adequate for P, but not (all-du-paths)*-adequate. Let Π be the set of paths through P which T executes. There exists an *executable* du-path which is not covered by Π. This contradicts the fact that Π covers every executable path.

B3) (all-paths) ⇒ (all-edges)*:* Suppose that this does not hold. Then there are a subprogram P and a set T of test data which are (all-paths)*-adequate for P, but not (all-edges)*-adequate. Let Π be the set of paths through P which T executes. There exists an executable edge (i, j) which is not covered by Π. This contradicts the fact that Π covers every executable path.

B4) (all-edges) ⇒ (all-nodes)*:* Let T be a test set which satisfies (all-edges)* for subprogram P, and let Π be the set of paths executed by T. Let n be any executable node in P. If n is the entry node, then n has a unique successor m, and (n, m) is executable. So Π covers (n, m) and hence covers n. If n is not the entry node, then since n is executable, some branch (i, n) is executable. So Π covers (i, n) and hence covers n.

B5) (all-uses) ⇒ (all-p-uses/some-c-uses)*, (all-p-uses/some-c-uses)* ⇒ (all-p-uses)*, (all-p-uses/some-c-uses)* ⇒ (all-defs)*, (all-uses)* ⇒ (all-c-uses/some-p-uses)*, (all-c-uses/some-p-uses)* ⇒ (all-defs)*:* These inclusions follow immediately from the definitions of the criteria given in Fig. 5. For example, any set Π of paths which covers all of the associations required by (all-uses)* will *a fortiori* cover all of the associations required by (all-p-uses/some-c-uses)*.

We next show that those relations not in the transitive closure of the diagram in Fig. 6 do not hold.

C1) (all-du-paths) ⇏ (all-p-uses)*, (all-du-paths)* ⇏ (all-p-uses/some-c-uses)*, (all-du-paths)* ⇏ (all-uses)*,* *(all-du-paths)* ⇏ (all-c-uses)*, (all-du-paths)* ⇏ (all-c-uses/some-p-uses)*, (all-du-paths)* ⇏ (all-defs)*, (all-du-paths)* ⇏ (all-edges)*, (all-du-paths)* ⇏ (all-nodes):* It suffices to show that (all-du-paths)* ⇏ (all-p-uses)*, (all-du-paths)* ⇏ (all-c-uses)*, (all-du-paths)* ⇏ (all-defs)*, and (all-du-paths)* ⇏ (all-nodes)*. The rest follows from the transitivity of ⇒. Consider the subprogram shown in Fig. 7(a). Its du-paths are shown in Fig. 7(b). Of these, only (1, 2), (2, 3, 4), (4, 3, 4), and (4, 3, 5) are executable. Let $T = \{(X, Y)\}$ where X is any integer and $Y < 0$. Since T executes $\Pi = \{(1, 2, 3, 4, 3, 4, 3, 5, 6, 7, 9, 10)\}$, T satisfies (all-du-paths)*. However, Π does not cover the associations (2, (6, 8), y) (2, 8, x), or node 8, all of which are covered by the executable path (1, 2, 3, 4, 3, 4, 3, 5, 6, 8, 9, 10), so T does not satisfy (all-p-uses)*, (all-c-uses)*, (all-defs)*, or (all-nodes).*

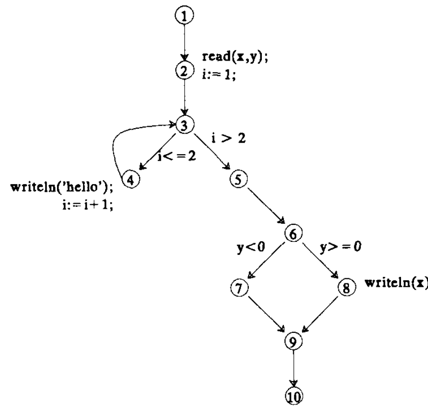
Intuitively, (all-du-paths)* fails to include these criteria because it is possible for a subprogram to have certain definition-use associations which can be executed only by paths which traverse some loop one or more times.

C2) (all-p-uses) ⇏ (all-edges)*, (all-p-uses/some-c-uses)* ⇏ (all-edges)*, (all-uses)* ⇏ (all-edges)*, (all-p-uses)* ⇏ (all-nodes), (all-p-uses/some-c-uses)* ⇏ (all-nodes), (all-uses)* ⇏ (all-nodes)*:* Consider the subprogram in Fig. 8, where y is a local variable (and hence *does not* have a definition in the entry node). Notice that since node 3 is unexecutable, y is always uninitialized when control reaches node 5. In the absence of any information about which (if either) edge leaving node 5 will be executed when the program is run on actual test data, we make the worst case assumption that edges (5, 6) and (5, 7) are both executable. This would be the case, for example, in an environment in which uninitialized variables receive arbitrary values. Since node 3 is unexecutable, the only executable definition-use associations are (1, 2, *input*↑), (2, (2, 4), x), and (2, 9, *input*↑). Let T be a test which executes $\Pi = \{(1, 2, 4, 5, 6, 8, 9)\}$ or $\Pi = \{(1, 2, 4, 5, 7, 8, 9)\}$. Then T satisfies (all-p-uses)*, (all-p-uses/some-c-uses)*, and (all-uses)*, but does not satisfy (all-edges)* or (all-nodes)*.

The rest of the noninclusions follow immediately from the incomparability and strictness proofs for the DF criteria, given in [19] and [5]. ■

It seems discouraging that (all-p-uses)* fails to include (all-edges)*. DF testing was developed in part in order to "bridge the gap" between branch testing and path testing. Since many "real-life" subprograms cannot be adequately tested using the unstarred versions of the DF criteria, one would hope that the FDF criteria would "bridge the gap" between (all-edges)* and (all-paths)*. We have seen that this is not the case. We next show that, for a certain class of "well-behaved" subprograms, any test which satisfies (all-p-uses)* satisfies (all-edges)*.

Definition: We will say that a subprogram P satisfies the No-Feasible-Undefined-P-uses property (NFUP) if and only if, for every executable edge (i, j) in P having a p-use of a variable x, there is some *executable* path from the start node to edge (i, j) which contains a global definition of x.



(a)

Path	With respect to	executable
(1,2)	input↑	yes
(2,3,4)	i	yes
(2,3,5)	i	no
(4,3,4)	i	yes
(4,3,5)	i	yes
(2,3,5,6,7,9,10)	input↑	no
(2,3,5,6,8,9,10)	input↑	no

(b)

Fig. 7. Program demonstrating that (all-du-paths)* fails to include the other criteria.

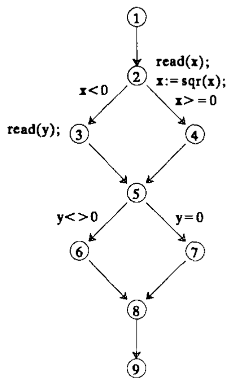


Fig. 8. Program demonstrating that (all-p-uses)* fails to include (all-edges)*.

We note that it is quite reasonable to expect subprograms to have property NFUP. If (i, j) is an edge which causes NFUP to fail, then any input which causes (i, j) to be executed will involve referencing an uninitialized variable.

Theorem 2: For the class of subprograms which satisfy NFUP, (all-p-uses)* ⇒ (all-edges)*.

Proof: Let P be a subprogram satisfying NFUP, let T be a test set which satisfies (all-p-uses)* for P, let Π be the set of paths executed by T, and let (i, j) be an executable edge in P. Suppose (i, j) has a p-use of a variable x. By hypothesis, there is an executable path π from the start node to (i, j) which includes a global definition of

x. Let n be the last node in π having a global definition of x. Then $(n, (i, j), x)$ is an executable definition–p-use association, so it is covered by Π. Hence, (i, j) is covered by Π.

If (i, j) has no p-uses, then the result follows by a straightforward modification of the corresponding part of the proof of (all-p-uses) $\Rightarrow$ (all-edges) [5]. ■

In [19], the class of subprograms to which DF testing applies was restricted to those subprograms satisfying the NSUP property, defined in Section II above. This restriction was necessary in order to ensure that all-p-uses $\Rightarrow$ all-edges. NFUP is a strengthening of NSUP. It takes into account the fact that even in subprograms satisfying NSUP, it may be the case that no *executable* path π from the entry node to some p-use of x has a definition of x.

It is tempting to restrict the class of programs to which the FDF criteria apply to those satisfying NFUP. It is our feeling, however, that while one can live with the undecidability of the adequate test *recognition* problem and perhaps (albeit very uncomfortably) with the undecidability of the adequate test *existence* problem, one should at least be able to decide algorithmically whether a given testing strategy *applies* to a given subprogram. Since it is undecidable whether a given subprogram satisfies NFUP, we refrain from requiring that this property hold for subprograms to be tested.

Another possible way to force (all-p-uses)* to include (all-edges)* would be to require subprograms to satisfy the No-Anomalies property (NA), which is as follows. Every path from the start node to a use of a variable x must contain a definition of x. Osterweil and Fosdick [17] consider any subprogram not satisfying this property to have a DF anomaly indicative of possible subprogram error. Since NA is a purely syntactic property and NA implies NFUP, we could restrict FDF testing to subprograms satisfying this property. We feel that this is overly restrictive since many perfectly good subprograms fail to satisfy NA.

One way to force NA to be satisfied is to give the entry node a definition of each variable. This would potentially increase the number of def-use associations and thus make the criteria more demanding. However, it would also make the model of the subprogram's DF reflect the actual DF less accurately.

Another approach is to perform FDF testing in conjunction with a check for DF anomalies. For any subprogram which satisfies NA and any test set T which satisfies (all-p-uses)*, the tester will be assured that T satisfies (all-edges)*. In case NA does not hold, the tester should explicitly check whether (all-edges)* is satisfied and, if necessary, add more test data or inspect the subprogram for references of uninitialized variables.

IV. A Generalized Notion of Executability

The definition of executability given in Section III fails to take into account any information about the context in which a subprogram is called. It may be the case that there are no input data to the program as a whole which cause the execution of a particular executable path through a subprogram. In order to test such a subprogram adequately with respect to a given FDF criterion, it may be necessary to write a driver program which assigns particular values to global variables and parameters and then calls the subprogram. Whether this extra effort is "worthwhile" depends on whether it is likely that the subprogram will ever be called in a context other than the one in which it currently appears in the program. In this section, we define a more general notion of executability which takes into account information about the context in which a subprogram is called. We then explore the effects of this generalization on the FDF criteria.

Consider the program

```
program main(input,output);
type CharString = array[1..10] of char;
var string1 : CharString;
     length : integer;

procedure WriteString(str: CharString; n: integer);
{Writes the first n characters of str to standard output.}
var i : integer;
begin
  for i := 1 to n do write(str[i])
end;

begin {statement part of main program}

  if length > 0 then WriteString(string1,length)
  else ...

end. {main }
```

Suppose that at every point in the program at which *WriteString* is called, the value of n is guaranteed to be strictly greater than zero. Then no input to the program can cause the execution of the path through the procedure which traverses the loop zero times.

In order to test *WriteString* adequately with respect to the criterion (all-uses)*, it is necessary to include test data which cause the **for** loop to be traversed zero times. To do this, one must write a driver program which calls *WriteString* with the second parameter having a value less than or equal to zero. If we think that we might actually want to use the procedure *WriteString* in a less restricted context (for example, because of modifications of the calling program or reusing the procedure in a different program), then this is a reasonable thing to do. On the other hand, if we are fairly certain that the procedure will never be called in a context where n is less than or equal to zero, then writing a driver program could be construed as being a wasted effort. What is needed is a notion of test data adequacy which takes into account information about the context in which the subprogram being tested can be called.

We can achieve this by relativizing the definition of executability as follows. We associate with the subprogram to be tested a predicate $IC(V_1 \cdots, V_k)$ called the *input constraint* where $V_1, \cdots, V_k$ represent the subprogram's parameters and nonlocal variables. A path through the subprogram is then *executable relative to IC* if there exists some assignment of values to input variables, parameters, and nonlocal variables which satisfies IC and which causes the path to executed. A path is executable as defined in Section III if and only if it is executable relative to the input constraint $IC \equiv$ TRUE. The notion of executability of an association and the definitions of the FDF criteria can be relativized in a straightforward manner.

The relationship among the relativized FDF criteria is essentially the same as that among the nonrelativized criteria. The definitions must be modified to reflect the fact that the objects being tested now consist of pairs (P, IC) where P is a subprogram and IC is an input constraint. We say that the relativized criteria C_1 includes the relativized criterion C_2 if for every subprogram/input-constraint pair (P, IC) every test which satisfies C_1 for that pair also satisfies C_2. C_1 strictly includes C_2 if C_1 includes C_2 and for some pair (P, IC) there is a test which satisfies C_2 but does not satisfy C_1. It is easy to show that the relationship among the relativized criteria is as shown in Fig. 6.

One reasonable choice for the input constraint is the predicate IC^{spec} obtained by taking the constraints on the input to the program as a whole (drawn from the program's specification), conjoining them, and "pushing them through" the program to all points at which the subprogram being tested is called. In practice, one might want to use a weaker predicate than IC^{spec}, which can be built up during the testing process as follows. At some point in the testing process, the tester notices that a particular executable association has still not been exercised. Upon examining the program to see what values of input data, nonlocal variables, and parameters would cause the execution of that association, the tester sees that the needed values of nonlocal variables and parameters cannot arise in the context of the program as a whole. One can then formulate a constraint which reflects this fact and can conjoin it to the previous constraint.

If the calling program is modified some time after the subprogram has been certified to be adequately tested, the predicate IC will provide useful documentation which will help in selecting additional test data for the subprogram. If IC is still satisfied whenever the subprogram is called, then no further testing of the subprogram will be needed. If IC no longer holds at the points of call, however, it will be necessary to update IC, determine which def-use associations become executable relative to the new constraint, and add test data to exercise those associations.

V. Conclusions

We have introduced a new family of path selection criteria derived from the DF testing criteria and explored the relationships among them. These criteria, the feasible data flow (FDF) testing criteria, are obtained from the corresponding DF testing criteria by eliminating unexecutable associations from consideration.

For a large class of "well-behaved programs, the FDF criteria (all-p-uses)*, (all-p-uses/some-c-uses)*, and (all-uses)* "bridge the gap" between (all-edges)* and (all-paths)* in the same way that the corresponding DF criteria do. For certain programs with anomalies, however, there are tests which satisfy (all-p-uses)* without satisfying (all-edges)*. Furthermore, although (all-du-paths) $\Rightarrow$ (all-uses), (all-du-paths)* does not even include (all-nodes)*.

The advantage of the FDF criteria over the DF criteria is that they satisfy the applicability property: for every subprogram P and every FDF criterion C, there is some set of paths which is C-adequate for P. The DF criteria do not satisfy this property. The disadvantage of the FDF criteria is that it is undecidable whether a particular set of paths is C-adequate for P. Thus, in deciding whether to use the DF criteria or the FDF criteria, one is faced with a tradeoff between applicability and automatability.

Although it is in general undecidable whether a given association is executable, it is often easy for a person looking at a subprogram to determine whether or not a particular association is executable. Sometimes this requires very little semantic information. For example, any program with a **for** loop in which the upper bound is always greater than or equal to the lower bound has an unexecutable definition–p-use association. In other cases, determining whether a given association is executable seems to require a "high-level" understanding of how the subprogram and other subprograms which it calls operate.

We have developed a heuristic method which uses a combination of symbolic evaluation and DF analysis to attempt to identify unexecutable definition–use associations [5]. When the heuristic cannot determine whether or not a particular association is executable, the person using the tool will have to intervene. We hope that this approach will prove to be a practical way to preserve the applicability property enjoyed by the FDF criteria, while sacrificing automatability to only a small extent.

Appendix
Example of an Asset Session

In this Appendix, we present an annotated example of an ASSET session. To distinguish between text written by the system and that written by the user, we display text entered by the user in **boldface** type. Comments are written in *italics*. For further information, see the *ASSET USER MANUAL* [4].

Example 1: This example shows an ASSET session in which a brute-force string matching procedure is analyzed. The program reads a string and a pattern. It is supposed to print the position in the string at which the pattern first appears and print 0 if the pattern never appears in the string. The current working directory has a subdirectory called "StrMtch." The file "StrMtch/subject.p" contains the following program:

```
program TestSringMatch (input,output);
const MAX = 80;
      LENGTH = 10;
```

```
type Source = array[1..MAX] of char;
     String = array[1..LENGTH] of char;
var Pat : String;
   Txt : Source;
   i,result,TxtLen  : integer;

   function StringMatch(Pattern:String;
     SorText: Source;
     PatLen, SorLen : integer): integer;
   {Brute force pattern-mathcer. Returns 0 for no
     match}
   var PatPos, SorPos : integer;
   begin
     PatPos := 1;
     SorPos := 1;
     repeat
       if Pattern[PatPos] = SorText[SorPos] then
       begin
         SorPos := SorPos + 1;
         PatPos := PatPos + 1
       end {then}
       else
       begin
         SorPos := (SorPos - PatPos) + 2;
         PatPos := 1
       end; {else}
     until (PatPos > PatLen) or (SorPos > SorLen);
     if PatPos > PatLen
      then StringMatch := SorPos - PatLen
      else StringMatch := 0
   end; {StringMatch}

begin {main program}
   {Read Input}
   writeln('ENTER THE TEXT');
   i := 1;
   while (not(eoln) and (i <= MAX)) do
   begin
     read(Txt[i]);
     i := i + 1
   end;
   TxtLen := i - 1;
   readln;
   writeln('ENTER THE PATTERN');
   i := 1;
   while (not(eoln) and (i <= LENGTH)) do
   begin
     Read(Pat[i]);
     i := i + 1
   end;
   readln;
   result := StringMatch(Pat,Txt,i - 1, TxtLen);
   writeln('The pattern first appears at position ',
                     result:3, ' in the text.');
end.
```

```
Script started on Sun May 31 13:09:59 1987
csd27< asset
```

```
Welcome to ASSET. For help type "help."

Enter relative pathname of initial default director.
>>: StrMtch
>>>: begin
Enter name of subject procedure file.
>>: subject.p
Separate Compilation? (Y/N) [N]
```
The "N" in square brackets indicates that the default answer is "no."
```
>>: n
Enter the name of the procedure to be instrumented.
If you would like to be prompted with the names of
the procedures in the subject program, just hit carriage
return.
>>:

==> Should TestStringMatch
be instrumented for testing? (Y/N)
>>: n
==> Should StringMatch
be instrumented for testing? (Y/N)
>>: y
>>>: select
  SELECT A CRITERION
A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses
G. All-du-paths
Enter letter representing the selected criterion
>>: a
Criterion is All-defs.
>>>: find
```
We next check whether the criterion has been satisified with no test data. This is not necessary, but by doing this, we get a list of all of the def-c-use and def-p-use associations in the program.
```
>>>: check
ALL-DEFS:

Still must exercise at least one of the following def-clear
paths:

with respect to      from    to
Pattern              1       ( 3, 5)
Pattern              1       ( 3, 4)

AND
```

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorText	1	(3, 5)
SorText	1	(3, 4)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatLen	1	8
PatLen	1	(6, 3)
PatLen	1	(6, 7)
PatLen	1	(7, 9)
PatLen	1	(7, 8)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorLen	1	(6, 3)
SorLen	1	(6, 7)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	2	4
PostPos	2	5
PatPos	2	(3, 5)
PatPos	2	(3, 4)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	2	4
SorPos	2	5
SorPos	2	(3, 5)
SorPos	2	(3, 4)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	2	4
PatPos	4	5
PatPos	4	(3, 5)
PatPos	4	(3, 4)
PatPos	4	(6, 3)
PatPos	4	(6, 7)
PatPos	4	(7, 9)
PatPos	4	(7, 8)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	4	4
SorPos	4	5
SorPos	4	8
SorPos	4	(3, 5)
SorPos	4	(3, 4)
SorPos	4	(6, 3)
SorPos	4	(6, 7)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
PatPos	5	4
PatPos	5	5
PatPos	5	(3, 5)
PatPos	5	(3, 4)
PatPos	5	(6, 3)
PatPos	5	(6, 7)
PatPos	5	(7, 9)
PatPos	5	(7, 8)

AND

Still must exercise at least one of the following def-clear paths:

with respect to	from	to
SorPos	5	4
SorPos	5	5
SorPos	5	8

```
SorPos          5       ( 3, 5)
SorPos          5       ( 3, 4)
SorPos          5       ( 6, 3)
SorPos          5       ( 6, 7)

To look at these again use the command 'view results'.
```

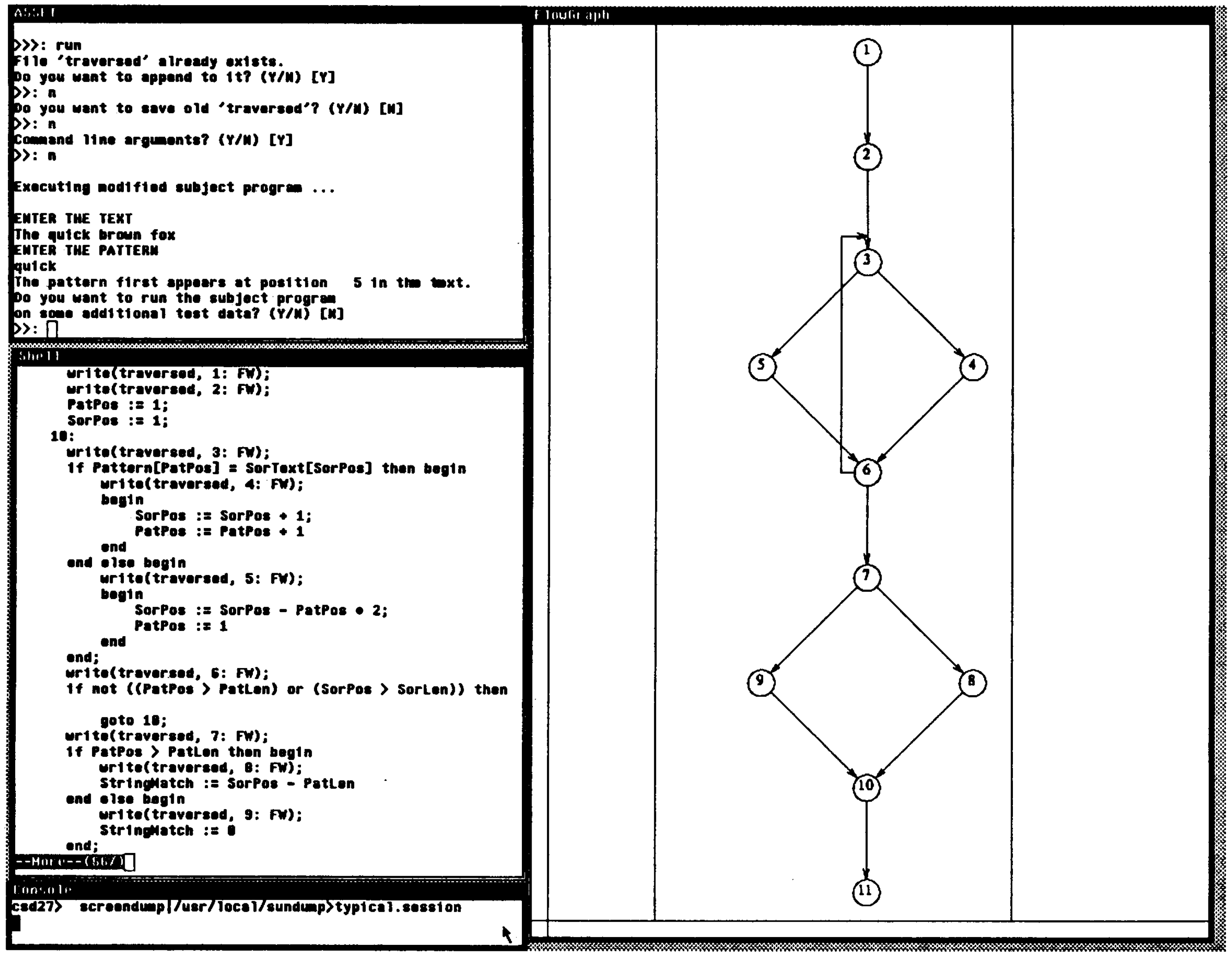
Next we will compile the program and start running it on some test data. As the initial test data set, we select one element in which the pattern appears in the string and one element in which the pattern does not appear in the string.

```
>>>: compile

Compilation begins ...
Done, and successful.

>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: n
Do you want to save old 'traversed'? (Y/N) [N]
>>: n
Command line arguments? (Y/N) [Y]
>>: n
Executing modified subject program ...
ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quick
The pattern first appears at position 5 in the text.
Do you want to run the subject program on some additional test data? (Y/N) [N]
>>: y
Command line arguments? (Y/N) [Y]
>>: n
Executing modified subject program ...
ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quack
The pattern first appears at position 0 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: check
ALL-DEFS:
CRITERION SATISFIED

To look at these again use the command 'view results'.
```

The test set satisfies the all-defs criterion. We next check whether the same test set satisfies a stronger criterion, all-uses.

```
>>>: select

 SELECT A CRITERION
A. All-defs
B. All-c-uses
C. All-p-uses
D. All-c-uses/some-p-uses
E. All-p-uses/some-c-uses
F. All-uses
G. All-du-paths

Enter letter representing the selected criterion
>>: f
Criterion is All-uses.

>>>: check
ALL-USES

Still need to exercise all of the following def-clear paths:

with respect to    from    to
PatPos             2       4
SorPos             2       4
SorPos             5       8
PatPos             2       ( 3, 4)
SorPos             2       ( 3, 4)
PatPos             4       ( 7, 9)
PatPos             5       ( 5, 8)

To look at these again use the command 'view results'.
```

To aid in the selection of the test data which cover the remaining def-use associations, the user can draw the flow graph (see Fig. 9) and use "copy.p" to aid in labeling each node with the corresponding code. Notice that for each i, block i begins with the statement "write(traversed,i:FW);".



```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: n
Do you want to save old 'traversed'? (Y/N) [N]
>>: n
Command line arguments? (Y/N) [Y]
>>: n

Executing modified subject program ...

ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
quick
The pattern first appears at position   5 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>:
```

```
        write(traversed, 1: FW);
        write(traversed, 2: FW);
        PatPos := 1;
        SorPos := 1;
     10:
        write(traversed, 3: FW);
        if Pattern[PatPos] = SorText[SorPos] then begin
            write(traversed, 4: FW);
            begin
                SorPos := SorPos + 1;
                PatPos := PatPos + 1
            end
        end else begin
            write(traversed, 5: FW);
            begin
                SorPos := SorPos - PatPos + 2;
                PatPos := 1
            end
        end;
        write(traversed, 6: FW);
        if not ((PatPos > PatLen) or (SorPos > SorLen)) then

            goto 10;
        write(traversed, 7: FW);
        if PatPos > PatLen then begin
            write(traversed, 8: FW);
            StringMatch := SorPos - PatLen
        end else begin
            write(traversed, 9: FW);
            StringMatch := 0
        end;
--More--(66%)
```

```
csd27>  screendump|/usr/local/sundump>typical.session
```

Fig. 9. Screen dump of an ASSET session.

```
>>>: view copy.p
program TestStringMatch(traversed, input, output);
var
  traversed: text;
const
  MAX = 80;
  LENGTH = 10;
type
  Source = array [1..MAX] of char;
  String = array [1..LENGTH] of char;
var
  Pat: String;
  Txt: Source;
  i, result, TxtLen: integer;
  function StringMatch(Pattern: String; SorText: Source;
PatLen, SorLen: integer): inter;
label
    10;
  const
    FW = 4;
  var
    PatPos, SorPos: integer;
  begin
    write(traversed, 1: FW);
    write(traversed, 2: FW);
    PatPos := 1;
    SorPos := 1;
  10:
    write(traversed, 3: FW);
    if Pattern[PatPos] = SorText[SorPos] then begin
      write(traversed, 4: FW);
      begin
        SorPos := SorPos + 1;
        PatPos := PatPos + 1
      end
    end else begin
      write(traversed, 5: FW);
      begin
        SorPos := SorPos - PatPos + 2;
        PatPos := 1
      end
    end;
    write(traversed, 6: FW);
    if not ((PatPos > PatLen) or (SorPos > SorLen))
then
      goto 10;
    write(traversed, 7: FW);
    if PatPos > PatLen then begin
      write(traversed, 8: FW);
      StringMatch := SorPos - PatLen
    end else begin
    write(travsered, 9: FW);
      StringMatch := 0
    end;
    write(traversed, 10: FW);
    write(traversed, 11: FW)
  end; { StringMatch }

begin
  rewrite(traversed);
  writeln('ENTER THE TEXT');
  i := 1;
  while not eoln and (i <= MAX) do begin
    read(Txt[i]);
    i := i + 1
  end
  TxtLen := i - 1;
  readln;
  writeln('ENTER THE PATTERN');
  i := 1;
  while not eoln and (i <= LENGTH) do begin
    read(Pat[i]);
    i := i + 1
  end;
  readln;
  result := StringMatch(Pat, Txt, i − 1, TxtLen);
  writeln('The pattern first appears at position', result: 3,
'in the text.')
end. { TestStringMatch }
```

Examining the annotated flow graph, we see that in order to execute a path from 2 to 4 which is definition clear with respect to PatPos, a test case in which the first character of the string matches the first character of the pattern is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.

```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: y
Command line arguments? (Y/N) [Y]
>>: n
Executing modified subject program ...
ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN
The
The pattern first appears at position 1 in the text.
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n

>>>: check
Still need to exercise all of the following of def-clear
paths:
with respect to     from    to
SorPos              5       8
PatPos              4       ( 7, 9)
PatPos              5       ( 7, 8)
To look at these again use the command 'view results'.
```

Examining the annotated flow graph, we see that in order to execute a path from 5 to 8 which is definition clear with respect to SorPos, a test case in which the pattern is the null string is needed. We run the program on such a test case, adding its trace to those produced by the test cases run previously.

```
>>>: run
File 'traversed' already exists.
Do you want to append to it? (Y/N) [Y]
>>: y
Command line arguments? (Y/N) [Y]
>>: n
Executing modified subject program ...
ENTER THE TEXT
The quick brown fox
ENTER THE PATTERN

The pattern first appears at position 2 in the text.
```

Examining the program's output, we see that the program has reported that the null string first appears in position 2 of the string. This is an error! The reader should note that this bug is guaranteed to be detected by any test set which is adequate according to the all-uses criterion. Having discovered a bug, we save the ASSET session and prepare to report the error.

```
Do you want to run the subject program
on some additional test data? (Y/N) [N]
>>: n
>>>: save
```

Note that to cover the one remaining association (4,(7,9),PatPos) we would have to include a test case in which the first k characters of the pattern matched the last k characters of the text, for some k such that 0 < k < the length of the pattern.

References

[1] L. A. Clarke, A. Podgurski, D. J. Richardson, and S. J. Zeil, "A comparison of data flow path selection criteria," in *Proc. 8th IEEE Int. Conf. Software Eng.*, London, UK, Aug. 1985, pp. 244-251.
[2] P. G. Frankl, S. N. Weiss, and E. J. Weyuker, "ASSET: A system to select and evaluate tests," in *Proc. IEEE Conf. Software Tools*, New York, Apr. 1985.
[3] P. G. Frankl and E. J. Weyuker, "A data flow testing tool," in *Proc. IEEE Softfair II*, San Francisco, CA, Dec. 1985.
[4] P. G. Frankl, *ASSET User Manual*, Dep. Comput. Sci. Courant Inst. Math. Sci., New York Univ., New York, Tech. Rep. #318, Sept. 1987.
[5] ——, "The use of data flow information for the selection and evaluation of software test data," Ph.D. dissertation, New York Univ., New York, Oct. 1987.
[6] M. R. Girgis and M. R. Woodward, "An integrated system for program testing using weak mutation and data flow analysis," in *Proc. 8th IEEE Int. Conf. Software Eng.*, London, UK, Aug. 1985, pp. 313-319.
[7] J. B. Goodenough and S. L. Gerhart, "Toward a theory of test data selection," *IEEE Trans. Software Eng.*, vol. SE-1, pp. 156-173, June 1975.
[8] M. S. Hecht, *Flow Analysis of Computer Programs.* Amsterdam, The Netherlands: North-Holland, 1977.
[9] P. M. Herman, "A data flow analysis approach to program testing," *Australian Comput. J.*, vol. 8, no. 3, Nov. 1976.
[10] W. E. Howden, "Reliability of the path analysis testing strategy," *IEEE Trans. Software Eng.*, vol. SE-2, no. 3, pp. 208-215, 1976.

[11] ——, "A survey of dynamic analysis methods," in *Tutorial: Software Testing and Validation Techniques*, E. Miller and W. E. Howden, Eds. Los Alamitos, CA: IEEE Computer Society, 1978.

[12] J. C. Huang, "An approach to program testing," *ACM Comput. Surv.*, vol. 7, no. 3, pp. 113-128, Sept. 1975.

[13] B. Korel and J. Laski, "A tool for data flow oriented program testing," in *Proc. IEEE Softfair II*, San Francisco, CA, Dec. 1985.

[14] J. W. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9, pp. 347-354, May 1983.

[15] E. F. Miller, Jr., M. R. Paige, J. P. Benson, and W. R. Wisehart, "Structural techniques of program validation," in *Dig. Compcon*, San Francisco, CA, Spring 1974, pp. 161-164.

[16] S. Ntafos, "On required element testing," *IEEE Trans. Software Eng.*, vol. SE-10, pp. 795-803, Nov. 1984.

[17] L. J. Osterweil and L. D. Fosdick, "DAVE—A validation error detection and documentation system for Fortran programs," *Software Practice Experience*, pp. 473-486, Oct.-Dec. 1976.

[18] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for program test data selection," in *Proc. Sixth Int. Conf. Software Eng.*, Tokyo, Japan, Sept. 1982, pp. 272-278.

[19] ——, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11, pp. 367-375, Apr. 1985.

[20] M. Schaeffer, *A Mathematical Theory of Global Program Optimization*. Englewood Cliffs, NJ: Prentice-Hall, 1973.

[21] E. J. Weyuker, "The complexity of data flow criteria for test data selection," *Inform. Processing Lett.*, vol. 19, no. 2, pp. 103-109, Aug. 1984.

[22] ——, "Axiomatizing software test data adequacy," *IEEE Trans. Software Eng.*, vol. SE-12, pp. 1128-1138, Dec. 1986.

[23] M. R. Woodward, D. Hedley, and M. A. Hennell, "Experience with path analysis and testing of programs," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 278-286, May 1980.

Phyllis G. Frankl received the B.A. degree in mathematics and physics from Brandeis University, Waltham, MA, in 1979, the M.A. degree in mathematics from Columbia University, New York, NY, in 1981, and the M.S. and Ph.D. degrees in computer science from New York University, New York, in 1985 and 1987, respectively.

She is currently an Assistant Professor of Computer Science at Polytechnic University, Brooklyn, NY. Her interests include software testing and theoretical computer science.

Dr. Frankl is a member of the IEEE Computer Society.

Elaine J. Weyuker received the Ph.D. degree in computer science from Rutgers University, New Brunswick, NJ, in 1977.

She has been on the faculty of the Courant Institute of Mathematical Sciences, New York University (NYU), New York, since 1977 and is currently an Associate Professor of Computer Science. Before coming to NYU, she was a Systems Engineer for IBM and was on the faculty of the City University of New York, New York, from 1969 to 1975. Her research interests are in software engineering, particularly software testing and reliability and software complexity measures, and in the theory of computation.

Dr. Weyuker is the author of a book (with Martin Davis), *Computability, Complexity, and Languages*, (Academic). She has been an ACM National Lecturer and was formerly a member of the Executive Committee of the IEEE Computer Society Technical Committee on Software Engineering. She is a member of the IEEE Computer Society and the Association for Computing Machinery.