Data Flow Analysis Techniques for Test Data Selection

Sandra Rapps* and Elaine J. Weyuker

Department of Computer Science, Courant Institute of Mathematical Sciences,
New York University, 251 Mercer Street, N.Y., N.Y. 10012

*also, YOURDON inc., 1133 Ave. of the Americas, N.Y., N.Y. 10036

Abstract

This paper examines a family of program test data selection criteria derived from data flow analysis techniques similar to those used in compiler optimization. It is argued that currently used path selection criteria which examine only the control flow of a program are inadequate. Our procedure associates with each point in a program at which a variable is defined, those points at which the value is used. Several related path criteria, which differ in the number of these associations needed to adequately test the program, are defined and compared.

Introduction

Program testing is the most commonly used method for demonstrating that a program actually accomplishes its intended purpose. The testing procedure consists of selecting elements from the program's input domain, executing the program on these test cases, and comparing the actual output with the expected output (in this discussion, we assume the existence of an "oracle", that is, some method to correctly determine the expected output). While exhaustive testing of all possible input values would provide the most complete picture of a program's performance, the size of the input domain is usually too large for this to be feasible. Instead, the usual procedure is to select a relatively small subset of the input domain which is, in some sense, representative of the entire input domain. An evaluation of the performance of the program on this test data is then used to predict its performance in general. Ideally, the test data should be chosen so that executing the program on this set will uncover all errors, thus guaranteeing that any program which produces correct results for the test data will produce correct results for any data in the input domain. However, discovering such a perfect set of test data is a difficult, if not impossible task [1,2]. In practice, test data is selected to give the tester a feeling of confidence that most errors will be discovered, without actually guaranteeing that the tested and debugged program is correct. This feeling of confidence is generally based upon the tester's having chosen the test data according to some criterion; the degree of confidence depends on the tester's perception of how directly the criterion approximates correctness. Thus, if a tester has a "good" test data criterion, the problem of test data selection is reduced to finding data that meet the criterion.

One class of test data selection criteria is based on measures of code coverage. Examples of such criteria are statement coverage (every statement of a program must be executed at least once during testing) and branch coverage (every branch must be traversed). Other coverage measures include Cn coverage measures [3], TERn measures [4] and boundary-interior [5]. Obviously, once such a criterion has been chosen, test data must be selected to fulfill the criterion. One way to accomplish this is to select paths through the program whose elements fulfill the chosen criterion, and then to find the input data which would cause each of the chosen paths to be selected.

Using path selection criteria as test data selection criteria has a distinct weakness. Consider the strongest path selection criterion which requires that *all* program paths $p_1, p_2, \ldots$ be selected. This effectively partitions the input domain D into a set of classes $D = \cup D[j]$ such that for every $x \in D$, $x \in D[j]$ if and only if executing the program with input x causes path p_j to be traversed. Then a test $T = \{t_1, t_2, \ldots\}$, where $t_j \in D[j]$ would seem to be a reasonably rigorous test of the program. However, this still does not guarantee program correctness. If one of the D[j] is not revealing [2], that is for some $x_1 \in D[j]$ the program works correctly, but for some other $x_2 \in D[j]$ the program is incorrect, then if x_1 is selected as t_j the error will not be discovered. In figure 1 we see an example of this.

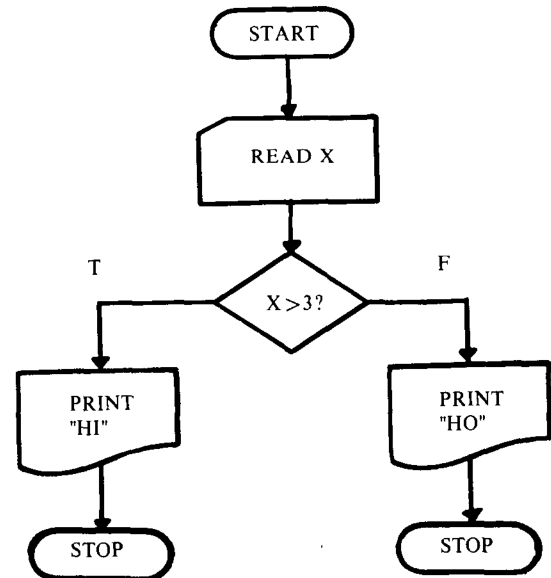

figure 1

Two test cases would be sufficient to execute all paths in this program. If the two test values chosen for x are 2 and 5, then we would not discover that the condition *if* $x>3$ should, in fact, have been *if* $x \geq 3$. This problem is compounded further by the fact that

0270-5257/82/0000/0272$00.75 © 1982 IEEE

many programs have a very large, or possibly infinite, number of paths and thus the criterion that all paths be selected must be replaced by a significantly weaker criterion that selects only a subset of the paths.

Although we must be aware that path selection criteria cannot insure that a set of test data capable of uncovering all errors will be chosen, we are not arguing that such criteria be abandoned. In the absence of feasible and reliable methods to formally prove correctness for all programs, we must continue to use testing strategies. Developing adequate path selection criteria will help bring us closer to establishing correctness. In [6] the reliability of path analysis is demonstrated. Furthermore, path selection criteria are used to determine correctness by symbolic execution of the code [7,8,9,10]. Our main goal for path selection criteria is that the number of paths selected be small enough so that all tests can be completed, yet large enough so that we can uncover many errors. In addition, we want criteria that can be mechanically checked. That is, we should be able to write a program that, given as input a program, a set of test data, and a path selection criterion, will tell us whether the program paths that would be executed using the test data are sufficient to satisfy the criterion. In addition, this program should also be able to give us some indication as to why a given set of test data is inadequate. Of course, we would also like to be able to use the path selection criteria to mechanically generate a set of paths that meet the criterion and/or a set of test data for a given program, but that is a difficult, and sometimes impossible, task.

Most path selection criteria are based on control flow analysis, which examines the branch and loop structure of a program. We believe that data flow analysis, which is widely used in code optimization [11], should be considered as well. Data flow analysis focuses on how variables are bound to values, and how these variables are to be used. Rather than selecting program paths based solely on the control structure of a program, the data flow criteria presented in this paper track input variables through a program, following them as they are modified, until they are ultimately used to produce output values. These criteria are constructed so that critical associations between the definition of a variable and its uses are examined during program testing. It is our belief that, just as one would not feel confident about a program without executing every statement in it as part of some test, one should not feel confident about a program without having seen the effect of using the value produced by each and every computation.

In the next section we present a programming language and define some graph-theoretic terminology. We then introduce a hierarchy of path selection criteria based on control and data flow analysis of a program. In the last section we discuss the relative strengths and weaknesses of the criteria.

The Programming Language

We now introduce our formal programming language. This may be thought of as either the intermediate level language produced by compilation from a high level language or the actual language in which the program was written. Our language allows only simple variables and contains the following **legal statement** types:

Start statement: start

Input statement: read $x_1,...,x_n$
where $x_1,...,x_n$ are variables.

Assignment statement: $y:=f(x_1,...,x_n)$
where f is an n-ary function ($n \geq 0$) and $y,x_1,...,x_n$ are variables.

Output statement: print $e_1,...,e_n$
where e_i, $i=1,...,n$, is either a literal or a variable.

Unconditional transfer statement: goto m
where m is an integer.

Conditional transfer statement: if $p(x_1,...,x_n)$ then goto m
where p is an n-ary predicate ($n>0$), $x_1,...,x_n$ are variables, and m is an integer. 0-ary predicates, such as TRUE and FALSE are prohibited.

Halt statement: stop.

A **program** is a finite sequence of legal statements, each statement prefixed by a unique integer, known as its **label.** We shall use the term "transfer statements" whenever we wish to include both conditional and unconditional transfers. For every transfer statement *goto m* or *if p then goto m,* m must be the label of some statement in the program. That statement is called the **target** of the transfer statement. Every program contains exactly one start statement which appears as the first statement of the sequence and may not be the target of a transfer statement. Every program contains at least one halt statement. The final statement of a program must be either a halt statement or an unconditional transfer.

If s_1 is the kth statement in a program and s_2 is the $(k+1)^{st}$ statement then we say that s_1 **physically precedes** s_2, and s_2 **physically succeeds** s_1. We say that statement s_1 **executionally precedes** statement s_2 (s_2 **executionally succeeds** s_1) if and only if either s_1 is a transfer statement (either conditional or unconditional) and s_2 is its target, or, s_1 is not an unconditional transfer or halt statement, and s_2 is its physical successor. A statement s is **syntactically reachable** if and only if there is a sequence of statements $s_1,...,s_n$ such that s_1 is the start statement, s_n is s, and for each $i=1,...,n-1$, s_i executionally precedes s_{i+1}.

A transfer statement is called **ineffective** if it physically precedes its target. All other transfer statements are **effective.** We require that every statement in the program be syntactically reachable, and that all transfer statements be effective. Violations of these restrictions are at best the product of coding practices which tend to obscure program logic and should therefore be eliminated. More significantly, their presence may well be indicative of certain types of logical or typographical errors (e.g. incorrect or missing labels; missing statements). It seems unlikely that a programmer would intentionally write code which can never be executed or include a completely unnecessary transfer statement to the very same statement that would have been executed without the transfer. Although we are concerned mainly with testing as a means of uncovering program errors, it is, of course, highly desirable to find and correct as many errors as possible before testing begins. We propose that the procedure described in this paper include as part of its output some indication of potentially troublesome situations encountered in processing a program, similar in nature to 'syntax error' messages produced by a compiler. We will therefore continue to mention the types of program anomalies which may be discovered at each stage of the procedure.

Flow Graph Theoretic Concepts

A program can be uniquely decomposed into a set of disjoint blocks having the property that whenever the first statement of the block is executed, the other statements are then executed in the given order. Furthermore, the first statement of the block should be the only statement which may be executed directly after the execution of a statement in another block. Formally, a **block** is a maximal set of ordered statements $b=<s_1,...,s_n>$, such that if $n>1$, for $i=2,...,n$, s_i is the unique executional successor of s_{i-1}

and s_{i-1} is the unique executional predecessor of s_i. Thus the first statement of a block is the only one which may have an executional predecessor outside the block, and the last statement is the only one which may have an executional successor outside the block. Every conditional transfer must be the last statement of a block, since effective conditional transfers cannot have unique executional successors.

The **program graph** representing a program consists of one node i corresponding to each block b_i of the program and an edge from node j to node k, denoted (j,k), if and only if either the last statement of b_j is not an unconditional transfer and it physically precedes the first statement of b_k, or the last statement of b_j is a transfer whose target is the first statement of b_k. If there is an edge from node j to node k we say that node j is a **predecessor** of node k, and k is a **successor** of j. The node corresponding to the block whose first statement is the start statement of the program is known as the **start node.** Such a node has no predecessors. A node corresponding to a block whose final statement is a halt statement is known as an **exit node** and has no successors. In addition, a node has two successors if and only if the final statement of its corresponding block is a conditional transfer. The requirement that all transfer statements be effective guarantees that the two successors are different nodes. That is, for every pair of nodes i and j there is at most one edge from node i to node j.

A **path** is a finite sequence of nodes $(n_1,...,n_k)$, $k \geq 2$, such that there is an edge from n_i to n_{i+1} for $i=1,2,...,k-1$. Because all transfer statements must be effective, there is at most one edge between any pair of nodes, allowing us to represent a path as a sequence of nodes, rather than as a sequence of edges. Note that the definition of path is a purely syntactic one, that is, a path is any sequence of nodes connected by edges. A **complete path** is a path whose initial node is the start node and whose final node is an exit node. Note that it may be the case that there is no input which will cause the sequence of statements represented by a particular path to be executed. Since it is known that there can be no algorithm to decide whether a given path is executable [12], we do not require that all complete paths be executable.

A **syntactically endless loop** is a path $(n_1,...,n_k)$, $k>1$, $n_1=n_k$, such that none of the blocks represented by the nodes on the path contain a conditional transfer statement whose target is either in a block which is not on the path or is a halt statement. Such a loop contains no possible escape and can be detected algorithmically and eliminated from a program, or flagged as a possible error. We therefore assume that programs contain no syntactically endless loops. Since all statements in a program must be syntactically reachable and there may be no syntactically endless loops, we are guaranteed that every node appears on some complete path, although possibly an unexecutable one.

The Def/Use Graph

Our path selection criteria are based on an investigation of the ways in which values are associated with variables, and how these associations can affect the execution of the program. This analysis focuses on the occurrences of variables within the program; the actual functions and predicates to be computed play no role. Each variable occurrence is classified as being a definitional occurrence, computation-use occurence, or predicate-use occurrence. We shall refer to these as **def, c-use** and **p-use,** respectively.

> The assignment statement $y:=f(x_1,...,x_n)$ contains c-uses of $x_1,...,x_n$ followed by a def of y.
>
> The input statement *read* $x_1,...,x_n$ contains defs of $x_1,...,x_n$.
>
> The output statement *print* $x_1,...,x_n$ contains c-uses of $x_1,...,x_n$.
>
> The conditional transfer statement *if* $p(x_1,...,x_n)$ *then goto m* contains p-uses of $x_1,...,x_n$.

In the following discussion we will say that a node of a program graph contains a c-use or a def of a variable if there is a statement in the corresponding block containing a c-use or a def of that variable. Because the value of a variable occurring in the predicate portion of a conditional transfer statement will affect the execution sequence of the program, we associate p-uses with edges rather than with nodes. If the final statement of the block corresponding to node i is *if* $p(x_1,...,x_n)$ *then goto m,* and the two successors of node i are nodes j and k then we will say that edges (i,j) and (i,k) contain p-uses of $x_1,...,x_n$. In figure 2, node 6 contains c-uses of z and x, followed by a def of z, followed by a c-use and a def of pow. Edges (5,6) and (5,7) each contain a p-use of pow.

```
1. start
2. read x,y
3. if y<0 then goto 6
4. pow := y
5. goto 7
6. pow := -y
7. z := 1
8. if pow=0 then goto 12
9. z := z×x
10. pow := pow-1
11. goto 8
12. if y≥0 then goto 14
13. z := 1/z
14. answer := z+1
15. print answer
16. stop
```

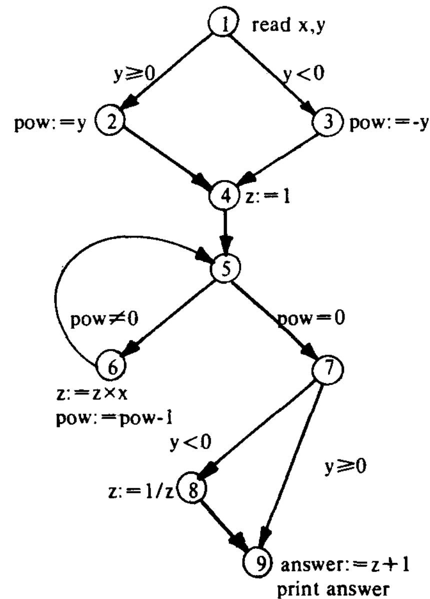

figure 2

Since we are interested in tracing the flow of data *between* nodes, any definition which is used *only* within the node in which that definition occurs is of little importance to us. Thus we categorize defs and uses as being either global or local. A c-use of a variable x is a **global c-use** if and only if there is no def of x preceding the c-use within the block in which it occurs. That is, the value of x must have been assigned in some block other than the one in which it is being used. Otherwise it is a **local c-use.** Global c-uses are often called locally exposed uses in the data flow analysis literature ([11]).

Let x be a variable occurring in a program. We say that a path $(i,n_1,...,n_m,j)$, $m \geq 0$, containing no defs of x in nodes $n_1,...,n_m$ is called a **def-clear path with respect to (wrt) x from node i to node j.** A path $(i,n_1,...,n_m,j,k)$, $m \geq 0$, containing no defs of x in nodes $n_1,...,n_m,j$ is called a **def-clear path wrt x from node i to edge (j,k).** An edge (i,j) is a def-clear path wrt x from node i to edge (i,j). A def of a variable x in node i is a **global def** if and only if it is the last def of x occurring in the block associated with node i and there is a def-clear path wrt x from i to either a node containing a global c-use of x or to an edge containing a p-use of x. Thus, a global def defines a variable which will be used outside the node in which the definition occurs. A def of a variable x in node i which is not a global def is a **local def** if and only if there is a local c-use of x in node i which follows this def, and no other def of x appears between the def and the local c-use. The def of *answer* in node 9 of figure 2 is local. Any def which is neither global nor local will never be used and the program should be examined for possible error.

Methodologies which detect program anomalies using data flow analysis [13,14] consider the presence of *any* def-clear path wrt a variable x from the start node to a use of x to be a possible error. Since some of these paths may not be executable, there may well be no error. If, however, *none* of these paths contains a definition of x, and at least one is executable, then there is indeed an error. Thus we assume that there is *some* path from the start node to every global c-use or p-use of a variable which contains a def of that variable. Programs which violate this assumption should be flagged as having a possible error.

We create the **def/use graph** from a program graph by associating each node i with two sets, def and c-use, and each edge (i,j) with the set p-use. **def(i)** is the set of variables for which node i contains a global def; **c-use(i)** is the set of variables for which node i contains a global c-use; **p-use(i,j)** is the set of variables for which edge (i,j) contains a p-use. An edge (i,j) for which p-use(i,j) is non-empty is called a **labelled edge;** if p-use(i,j)=$\emptyset$ then (i,j) is called an **unlabelled edge.** Because 0-ary predicates are not allowed, edges which are the sole out-edges of a node are always unlabelled, while those which are one of a pair of out-edges are always labelled.

In figure 2 these sets are:

node	**c-use**	**def**
1	$\emptyset$	{x,y}
2	{y}	{pow}
3	{y}	{pow}
4	$\emptyset$	{z}
5	$\emptyset$	$\emptyset$
6	{x,z,pow}	{z,pow}
7	$\emptyset$	$\emptyset$
8	{z}	{z}
9	{z}	$\emptyset$

edge	**p-use**
(1,2)	{y}
(1,3)	{y}
(5,6)	{pow}
(5,7)	{pow}
(7,8)	{y}
(7,9)	{y}

Note that *answer* which has only a local def and a local c-use, does not appear in these sets. Edges (2,4),(3,4),(4,5),(6,5), and (8,9) are unlabelled.

We now define several sets needed in the construction of our def/use criteria. Let i be any node, and x any variable such that $x \in$def(i). Then **dcu(x,i)** is the set of all nodes j such that $x \in$c-use(j) and for which there is a def-clear path wrt x from i to j; **dpu(x,i)** is the set of all edges (j,k) such that $x \in$p-use(j,k) and for which there is a def-clear path wrt x from i to (j,k). The dcu and dpu sets for figure 2 are:

variable	**node**	**dcu**	**dpu**
x	1	{6}	$\emptyset$
y	1	{2,3}	{(1,2),(1,3),(7,8),(7,9)}
pow	2	{6}	{(5,6),(5,7)}
pow	3	{6}	{(5,6),(5,7)}
z	4	{6,8,9}	$\emptyset$
z	6	{6,8,9}	$\emptyset$
pow	6	{6}	{(5,6),(5,7)}
z	8	{9}	$\emptyset$

Let P be a set of complete paths for a def/use graph of a given program. We say that a **node i is included in P** if P contains a path $(n_1,...,n_m)$ such that $i=n_j$ for some j, $1 \leq j \leq m$. Similarly, an **edge** (i_1,i_2) **is included in P** if P contains a path $(n_1,...,n_m)$ such that $i_1=n_j$ and $i_2=n_{j+1}$ for some j, $1 \leq j \leq m-1$. A **path** $(i_1,...,i_k)$ **is included in P** if P contains a path $(n_1,...,n_m)$ and $i_1=n_j$, $i_2=n_{j+1},...,i_k=n_{j+k-1}$ for some j, $1 \leq j \leq m-k+1$. In addition, we say that P is **executed** if every path contained in P is traversed during the course of executing the program on a set of test input data.

A Family of Path Selection Criteria

We now introduce a family of path selection criteria. Let G be a def/use graph, and P be a set of complete paths of G. Then

P satisfies the **all-nodes** criterion if every node of G is included in P.

P satisfies the **all-edges** criterion if every edge of G is included in P.

P satisfies the **all-defs** criterion if for every node i of G and every $x \in$def(i), P includes a def-clear path wrt x from i to some element of dcu(i,x) or dpu(i,x).

P satisfies the **all-p-uses** criterion if for every node i and every $x \in$def(i), P includes a def-clear path wrt x from i to all elements of dpu(x,i).

P satisfies the **all-c-uses/some-p-uses** criterion if for every node i and every $x \in$def(i), P includes some def-clear path wrt x from i to every node in dcu(x,i); if dcu(x,i) is empty, then P must include a def-clear path wrt x from i to some edge contained in dpu(x,i). This criterion requires that every c-use of a variable x defined in node i must be included in some path of P. If there is no such c-use, then some p-use

of the definition of x in i must be included. Thus to fulfill this criterion, every definition which is ever used must have some use included in the paths of P, with the c-uses particularly emphasized.

P satisfies the **all-p-uses/some-c-uses** criterion if for every node i and every $x \in def(i)$, P includes a def-clear path wrt x from i to all elements of dpu(x,i); if dpu(x,i) is empty, then P must include a def-clear path wrt x from i to some node in dcu(x,i). As in the case of all-c-uses/some-p-uses, this criterion requires every definition which is ever used to be used in some path of P. In this case, however, the emphasis is on p-uses.

P satisfies the **all-uses** criterion if for every node i and every $x \in def(i)$, P includes a def-clear path wrt x from i to all elements of dcu(x,i) and to all elements of dpu(x,i).

A path $(n_1,...,n_k)$ is **loop-free** if and only if $n_i \neq n_j$ for $i \neq j$. P satisfies the **all-du-paths** criterion if for every node i and every $x \in def(i)$, P includes every loop-free def-clear path wrt x from i to all elements of dpu(x,i) and to all elements of dcu(x,i). Note that the *complete* paths contained in P need not be loop-free.

P satisfies the **all-paths** criterion if P includes *every* complete path of G. Note that, due to loops, many graphs have infinitely many complete paths.

Criterion c_1 **includes** criterion c_2 if for every def/use graph G, any set of complete paths of G that satisfies c_1 also satisfies c_2. Criterion c_1 **strictly includes** criterion c_2, denoted $c_1 \twoheadrightarrow c_2$, if and only if c_1 includes c_2, and for some def/use graph G there is a set of complete paths of G that satisfies c_2 but not c_1. Note that this is clearly a transitive relation. We say that criteria c_1 and c_2 are **incomparable** if neither c_1 includes c_2 nor c_2 includes c_1.

We can assume that all def/use graphs contain more than one node. Single-node graphs have only one path and thus any of the criteria would select that path. Furthermore we may assume that all def/use graphs have more than two nodes and at least two labelled edges. This follows immediately from our definition of block, and the requirement that all transfer statements be effective.

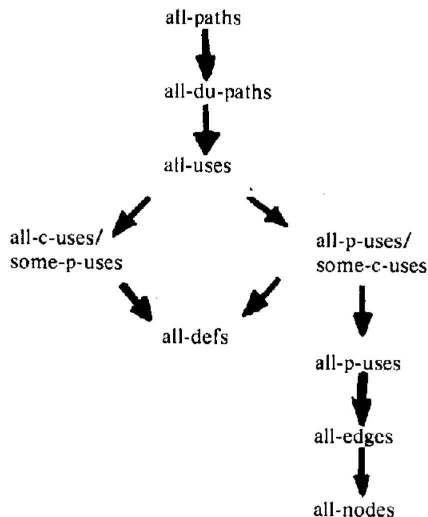

figure 3

Theorem:

The family of criteria is partially ordered by strict inclusion as shown in figure 3. Furthermore, criterion c_i strictly includes criterion c_j if and only if it is explicitly shown to be so in figure 3 or follows from the transitivity of the relationship.

A proof of this theorem is contained in [17].

Analysis of the Criteria

The criteria all-nodes (statement coverage) and all-edges (branch coverage) are often used in program testing despite the fact that they are extremely weak criteria. Our search for stronger criteria that make use of data flow information led us at first to all-defs. Our assumption is that every definition in a program was included by the programmer because it would eventually be used somewhere, and, thus, a well-tested program should test all definitions. However, we rejected all-defs as an adequate criterion since it does not include all-edges. In [15] errors are separated into domain errors, which occur when an incorrect path is chosen due to a control flow error, and computation errors, which occur when a correct path is chosen but an assignment statement contains an erroneous computation. All-defs can detect computation errors but not necessarily domain errors, while all-edges can detect domain errors but not necessarily computation errors. In looking for criteria that can detect both types of errors, we separated uses of variables into p-uses and c-uses. All-p-uses is our first data flow analysis criterion which includes all-edges, but it, too, primarily detects domain errors. It is stronger than all-edges since it requires a path from *every* definition of a variable to every possible p-use of that variable, while all-edges merely requires that there be *some* path that includes that p-use. Since the value of a variable contained in a predicate may have been defined in one of several places, it is clear that all-p-uses can uncover more errors than all-edges. The criterion all-p-uses/some-c-uses is the weakest of our criteria that includes both all-defs and all-edges. We are guaranteed that testing according to this criterion exercises every edge, and every computation.

Consider the program of figure 4, which is a translation into our programming language of the Wensleysqroot program presented in [7] to compute $\sqrt{p}$, $0 \leq p < 1$, to accuracy e, $0 < e \leq 1$. The program contains an error; statements 11 and 12 should be interchanged. The set of paths {(1,6), (1,2,3,4,2,3,5,2,7)} satisfies all-edges, but would not detect the error. As stated in [7], "a looping factor of two is required to derive test data that reveals the bug," that is, the tester must specify that some path containing at least two executions of the loop be tested. In fact, a looping factor of two or more may not suffice. The problem is that the definition of c in node 5 is never used unless the set of paths includes a definition-clear path wrt c from node 5 to node 3, and thus the error cannot be detected unless the path (5,2,3) is included. The all-defs criterion, however, does require that all definitions be used, and thus any set of paths selected according to this criterion would have to include (5,2,3). The set of paths {(1,2,3,5,2,3,5,2,7), (1,2,3,4,2,3,4,2,7)} would detect the error; however node 6 and edge (1,6) would not be tested. All-p-uses is not adequate to find the error either. {(1,6), (1,2,7), (1,2,3,5,2,7), (1,2,3,4,2,3,4,2,7)} satisfies all-p-uses without including (5,2,3). However, since the program contains no p-uses of the definitions of c in nodes 4 and 5, all-p-uses/some-c-uses does require that the paths (5,2,3) and (4,2,3) be included.

```
 1. start
 2. read p,e
 3. d:=1
 4. x:=0
 5. c:=2×p
 6. if c≥2 then goto 18
 7. if d≤e then goto 16
 8. d:=d/2
 9. t:=c-(2×x+d)
10. if t<0 then goto 14
11. x:=x+d
12. c:=2×(c-(2×x+d))
13. goto 7
14. c:=2×c
15. goto 7
16. print x
17. stop
18. print 'error'
19. stop
```

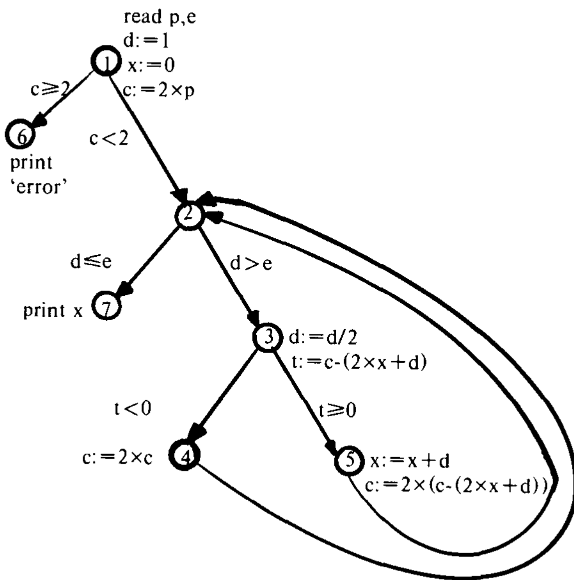

figure 4

Since the value of a variable used in a c-use may have been defined in one of several places, all-c-uses/some-p-uses is more likely to find computation errors than all-p-uses/some-c-uses. In particular, all-c-uses/some p-uses requires paths between every definition and every possible c-use of that definition. For figure 4, this means that any set of paths chosen according to all-c-uses/some-p-uses must include the paths (4,2,3,4), (4,2,3,5), (5,2,3,4) and (5,2,3,5). However, this criterion does not include all-edges either. For example, {(1,2,3,5,2,3,5,2,7), (1,2,3,4,2,3,5,2,7,), (1,2,3,4,2,3,4,2,7), (1,2,3,5,2,3,4,2,7)} satisfies all-c-uses/some-p-uses, but does not include edge (1,6). Furthermore, it does not include path (1,2,7), which would be executed if the input data were incorrect and e>1. Since the program does check for p<1, it may very well be an error that it does not explicitly check for e≤1. This error would be detected by all-p-uses/some-c-uses, however, since it does require that the path (1,2,7) be included.

The criterion all-uses, which includes both all-p-uses/some-c-uses and all-c-uses/some-p-uses, can detect both types of errors. This criterion is similar to required element testing [16]. One set of paths which satisfies the all-uses criterion for figure 4 is {(1,6), (1,2,3,5,2,3,5,2,7), (1,2,7), (1,2,3,4,2,3,5,2,7), (1,2,3,4,2,3,4,2,7), (1,2,3,5,2,3,4,2,7)}. Notice that any set of paths which satisfies this criterion must contain the paths (1,6), (1,2,7), and all of the combinations of predicates represented by the paths (4,2,3,4), (4,2,3,5), (5,2,3,4) and (5,2,3,5). However, for some programs, this criterion may not test all possible combinations of predicates, as can be seen in figure 5.

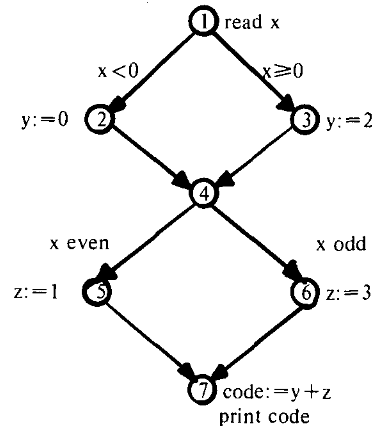

figure 5

{(1,2,4,5,7), (1,3,4,6,7)} satisfies all-uses, but it does not include the paths (1,2,4,6,7) and (1,3,4,5,7). We therefore defined the all-du-paths criterion.

Conclusion and Future Work

The data flow criteria that we have defined can be used to bridge the gap between the requirement that every branch be traversed and the requirement that every path be traversed. Our criteria focus on the interaction of portions of the program linked by the flow of data rather than solely by the flow of control. Research is currently underway to determine the relative costs of the criteria in terms of the number of test cases required to satisfy them, and to more precisely characterize the types of errors detectable by each. We envision a tester being able to select a particular criterion by determining whether the likely payoff in terms of errors detectable is worth the added cost in terms of additional tests necessary.

References

[1] J.B. Goodenough and S.L. Gerhart, "Toward a Theory of Testing: Data Selection Criteria," in **Current Trends in Programming Methodology,** Vol.2, ed. R.T. Yeh, Prentice-Hall, 1977, pp. 44-79.

[2] E.J. Weyuker and T.J. Ostrand, "Theories of Program Testing and the Application of Revealing Subdomains," **IEEE Trans. Software Eng.,** Vol.SE-6, May 1980, pp.236-246.

[3] E. Miller, "Coverage Measure Definitions Reviewed," **Testing Techniques Newsletter,** Vol.3, No.4, Nov 1980, p.6.

[4] M.R. Woodward, D.Hedley, and M.A. Hennell, "Experience With Path Analysis and Testing of Programs," **IEEE Trans. Software Eng.,** Vol.SE-6, May 1980, pp.278-286.

[5] W.E. Howden, "Methodology for the Generation of Test Data," **IEEE Trans. Computers,** Vol.TC-24, May 1975.

[6] W.E. Howden, "Reliability of the Path Analysis Testing Strategy," **IEEE Trans. Software Eng.,** Vol.SE-2, Sept 1976, pp.208-215.

[7] R.S. Boyer, B.E. Elspas, and K.N. Levitt, "SELECT--A Formal System for Testing and Debugging," **Proc. of the Int. Conf. on Reliable Software,** Los Angeles, April 1975, pp.234-245.

[8] L.A. Clarke, "A System to Generate Data and Symbolically Execute Programs," **IEEE Trans. Software Eng.,** Vol.SE-2, Sept 1976, pp.215-222.

[9] W.E. Howden, "Symbolic Testing and the DISSECT Symbolic Evaluation System," **IEEE Trans. Software Eng.,** Vol.SE-3, July 1977, pp.266-278.

[10] J.C. King, "Symbolic Execution and Program Testing," **Commun. ACM,** 19, July 1976, pp.385-394.

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

[12] E.J. Weyuker, "The Applicability of Program Schema Results to Programs," **Int. J. Comput. Inf. Sci.,** Vol.8, No.5, Nov 1979.

[13] L.D.Fosdick and L.J. Osterweil, "Data Flow Analysis in Software Reliability," **Comput. Surveys,** Vol.8, No.3, Sept 1976, pp.305-330.

[14] L.J. Osterweil, "The Detection of Unexecutable Program Paths Through Static Data Flow Analysis," **Proc. IEEE COMSAC,** Chicago, Ill., Dec 1977.

[15] L.J. White and E.I. Cohen, "A Domain Strategy for Computer Program Testing," **IEEE Trans. Software Eng.,** Vol. SE-6, May 1980, pp.247-257.

[16] S. Ntafos, "On Testing With Required Elements," U. Texas at Dallas Technical Report 90, July 1981.

[17] S. Rapps and E.J. Weyuker, "Data Flow Analysis Techniques for Test Data Selection," Dept of Computer Science Technical Report 023, Courant Institute of Mathematical Sciences, New York University, Aug 1980 (revised Dec 1981).