Selecting Software Test Data Using Data Flow Information

SANDRA RAPPS AND ELAINE J. WEYUKER

Abstract—**This paper defines 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 test data selection criteria, differing in the type and number of these associations, are defined and compared.**

Index Terms—**Data flow, program testing, test data selection.**

INTRODUCTION

PROGRAM testing is the most commonly used method for demonstrating that a program accomplishes its intended purpose. It involves 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," i.e., some method to determine whether or not the output produced by a program is correct. See [11] for a discussion of this assumption.) While testing all possible input values would provide the most complete picture of a program's behavior, the input domain is usually too large for exhaustive testing to be practical. Instead, the usual procedure is to select a relatively small subset, which is in some sense representative of the entire input domain. An evaluation of the behavior of the program on this data is then used to predict its behavior 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 an ideal set of test data is in general an impossible task [3], [12].

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 [7], TERn measures [13], and boundary-interior testing [5]. 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.

Manuscript received August 3, 1983; revised June 4, 1984. This work was supported in part by the National Science Foundation under Grant MCS-82-01167.

The authors are with the Courant Institute of Mathematical Sciences, Department of Computer Science, New York University, New York, NY 10012.

Using such path selection criteria as the basis for test data selection criteria presents two distinct problems. Consider the strongest path selection criterion, *all-paths*, which requires that *all* program paths $p_1, p_2, \cdots$ 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 set $T = \{t_1, t_2, \cdots\}$, where $t_j \in D[j]$, would seem to be a reasonably rigorous test of the program. However, if one of the $D[j]$ is not *revealing* [12] (i.e., for some $x_1 \in D[j]$ the program outputs the correct value, 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. Thus, traversing all paths does not guarantee that all errors will be detected. The second problem is that programs with loops may have an infinite number of paths, and thus, the all-paths criterion must be replaced by a weaker one which selects only a subset of the paths.

We present a family of test data selection criteria for which the number of paths selected is always finite, and chosen in a systematic and intelligent manner in order to help us uncover errors. In addition, determining whether or not the criteria have been fulfilled can be checked for mechanically. That is, we can write a program which can determine for a given program, test set, and selection criterion, whether or not the paths that would be traversed by the test set satisfy the criterion.

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 [4], 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. Our criteria are constructed so that critical associations between the definition of a variable and its uses are examined during program testing. Just as one would not feel confident about the correctness of a portion of a program which has never been executed, we believe that if the result of some computation has never been used, one has no reason to believe that the correct computation has been performed.

In the next section we present a programming language and define some graph-theoretic terminology. We then introduce a family of path selection criteria based on both the control and data flow of a program, and discuss the relative strengths and weaknesses of these 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. Although most of the ideas of the paper are independent of the particular details of this language, it is nonetheless necessary to have an explicit syntax. Our language allows only simple variables and contains the following *legal statement* types:

Start statement: **start**
Input statement: **read** $x_1, \cdots, x_n$
where $x_1, \cdots, x_n$ are variables.
Assignment statement: $y \leftarrow f(x_1, \cdots, x_n)$
where f is an n-ary function ($n \geqslant 0$) and $y, x_1, \cdots, x_n$ are variables.
Output statement: **print** $e_1, \cdots, e_n$
where e_i, $i = 1, \cdots, n$, is either a literal or a variable.
Unconditional transfer statement: **goto** m
where m is a label.
Conditional transfer statement:
if $p(x_1, \cdots, x_n)$ **then goto** m
where p is an n-ary predicate ($n > 0$), $x_1, \cdots, x_n$ are variables, and m is a label. Note that 0-ary predicates, such as TRUE and FALSE, are prohibited.
Halt statement: **stop**

A *program* is a finite sequence of legal statements. Each statement may have a unique identifier, 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. If 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, then we say that statement s_1 *executionally precedes* statement s_2 (s_2 *executionally succeeds* s_1). A statement s is *syntactically reachable* if there is a sequence of statements $s_1, \cdots, s_n$ such that s_1 is the start statement, s_n is s, and for each $i = 1, \cdots, n - 1$, 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 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 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 executed in the given order. Furthermore, the first statement of the block is 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 $b = \langle s_1, \cdots, s_n \rangle$ such that if $n > 1$, for $i = 2, \cdots, n$, 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* G representing a program Q consists of one node i corresponding to each block b_i of Q 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 $(n_1, \cdots, n_k)$, $k \geqslant 2$, such that there is an edge from n_i to n_{i+1} for $i = 1, 2, \cdots, k - 1$. 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 a path is a purely syntactic notion. A path is *simple* if all nodes, except possibly the first and last, are distinct. A path is *loop-free* if all nodes are distinct.

A *complete path* is a path whose initial node is the start node and whose final node is an exit node. Note that it is possible 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 or not a given path is executable [10], we do not require that all complete paths be executable.

A *syntactically endless loop* is a path $(n_1, \cdots, n_k)$, $k > 1$, $n_1 = n_k$, such that none of the blocks represented by the nodes on the path contains 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 Definition 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. Each variable occurrence is classified as being a definitional, computation-use, or predicate-use occurrence. We refer to these as *def*, *c-use*, and *p-use*, respectively.

The assignment statement "$y \leftarrow f(x_1, \cdots, x_n)$" contains *c*-uses of $x_1, \cdots, x_n$ followed by a def of y.

The input statement "**read** $x_1, \cdots, x_n$" contains defs of $x_1, \cdots, x_n$.

The output statement "**print** $x_1, \cdots, x_n$" contains *c*-uses of $x_1, \cdots, x_n$.

The conditional transfer statement "**if** $p(x_1, \cdots, x_n)$ **then goto** m" contains *p*-uses of $x_1, \cdots, x_n$.

Work involving data flow analysis generally classifies each variable occurrence as being either a definition or a use. But among uses, we recognized two substantially different types of uses. The first type directly affects the computation being performed or allows one to see the result of some earlier definition. We have called such a use a *c*-use. Of course, a *c*-use may indirectly affect the flow of control through the program.

In contrast, the second type of use directly affects the flow of control through the program, and thereby may indirectly affect the computations performed. We have called such a use a *p*-use. The usefulness of this distinction will be made clear in a later section.

We shall 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, respectively, a *c*-use or a def of that variable. In Fig. 1, 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.

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. We thus make the following distinction: a *c*-use of a variable x is a *global c-use*, provided there is no def of x preceding the *c*-use within the block in which it occurs. That is, the value of x

```
    START
    READ X,Y
    IF Y<0 THEN GOTO A
    POW ← Y
    GOTO B
[A] POW ← -Y
[B] Z ← 1
[C] IF POW=0 THEN GOTO D
    Z ← Z*X
    POW ← POW-1
    GOTO C
[D] IF Y≥0 THEN GOTO E
    Z ← 1/Z
[E] ANSWER ← Z+1
    PRINT ANSWER
    STOP
```

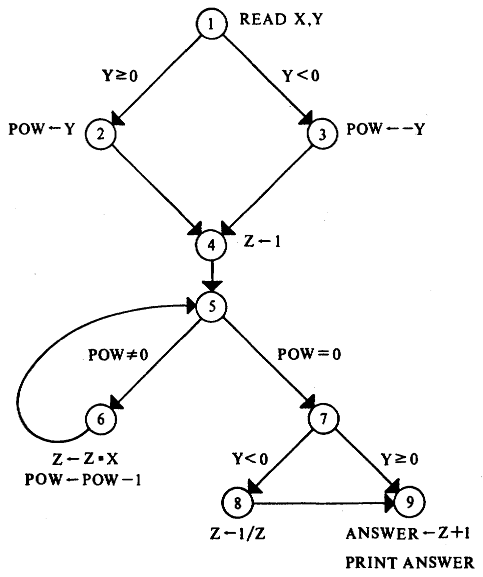

Fig. 1.

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 [4].

A conditional transfer statement is always the last statement of a block, and has two executional successors, which are in two different blocks. Since the value of the variable occurring in the predicate portion of the conditional transfer statement directly determines which of these two blocks is to be executed next, we associate *p*-uses with edges rather than with the node in which the predicate portion occurs. If the final statement of the block corresponding to node i is "**if** $p(x_1, \cdots, x_n)$ **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 $x_1, \cdots, x_n$. Edges (5, 6) and (5, 7) in Fig. 1 each contain a *p*-use of pow. Note that since *p*-uses are associated with edges, no distinction need be made between local and global *p*-uses.

Let x be a variable occurring in a program. A path $(i, n_1, \cdots, n_m, j)$, $m \geqslant 0$, containing no defs of x in nodes $n_1, \cdots, n_m$ is called a *def-clear path with respect to (w.r.t) x from node i to node j*. A path $(i, n_1, \cdots, n_m, j, k)$, $m \geqslant 0$, containing no defs of x in nodes $n_1, \cdots, n_m, j$ is called a *def-clear path w.r.t. x from node i to edge* (j, k). An edge (i, j) is a def-clear path w.r.t. x from node i to edge (i, j). A def of a variable x in node i is a *global def* if it is the last def of x occurring in the block associated with node i and there is a def-clear path w.r.t. 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 can 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 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 Fig. 1 is local. Any def which is neither global nor local can never be used and the program should be examined for possible error.

Techniques which detect program anomalies using data flow analysis, such as those discussed in [2] and [9], generally consider the presence of *any* def-clear path w.r.t. 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 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 a set with each edge and two sets with each node. *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 nonempty is called a *labeled edge*; if p-use$(i, j) = \phi$, then (i, j) is called an *unlabeled edge*. Because 0-ary predicates are not allowed, edges which are the sole out-edges of a node are always unlabeled, while those which are one of a pair of out-edges are always labeled. In Fig. 1 these sets are

node	c-use	def	edge	p-use
1	ϕ	$\{x, y\}$	$(1, 2)$	$\{y\}$
2	$\{y\}$	{pow}	$(1, 3)$	$\{y\}$
3	$\{y\}$	{pow}	$(5, 6)$	{pow}
4	ϕ	$\{z\}$	$(5, 7)$	{pow}
5	ϕ	ϕ	$(7, 8)$	$\{y\}$
6	$\{x, z, \text{pow}\}$	$\{z, \text{pow}\}$	$(7, 9)$	$\{y\}$
7	ϕ	ϕ		
8	$\{z\}$	$\{z\}$		
9	$\{z\}$	ϕ		

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 unlabeled.

We now define several sets needed in the construction of our 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 w.r.t. 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 w.r.t. x from i to j. The dcu and dpu sets for Fig. 1 are

$$\text{dcu}(x, 1) = \{6\}$$
$$\text{dpu}(x, 1) = \phi$$

$$\text{dcu}(y, 1) = \{2, 3\}$$
$$\text{dpu}(y, 1) = \{(1, 2), (1, 3), (7, 8), (7, 9)\}$$

$$\text{dcu}(\text{pow}, 2) = \{6\}$$
$$\text{dpu}(\text{pow}, 2) = \{(5, 6), (5, 7)\}$$

$$\text{dcu}(\text{pow}, 3) = \{6\}$$
$$\text{dpu}(\text{pow}, 3) = \{(5, 6), (5, 7)\}$$

$$\text{dcu}(x, 4) = \{6, 8, 9\}$$
$$\text{dpu}(z, 4) = \phi$$

$$\text{dcu}(z, 4) = \{6, 8, 9\}$$
$$\text{dpu}(z, 6) = \phi$$

$$\text{dcu}(\text{pow}, 6) = \{6\}$$
$$\text{dpu}(\text{pow}, 6) = \{(5, 6), (5, 7)\}$$

$$\text{dcu}(z, 8) = \{9\}$$
$$\text{dpu}(z, 8) = \phi$$

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 $(n_1, \cdots, n_m)$ 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 $(n_1, \cdots, n_m)$ such that $i_1 = n_j$ and $i_2 = n_{j+1}$ for some j, $1 \leq j \leq m - 1$. A *path $(i_1, \cdots, i_k)$ is included in P* if P contains a path $(n_1, \cdots, n_m)$ and $i_1 = n_j$, $i_2 = n_{j+1}, \cdots, i_k = n_{j+k-1}$ for some j, $1 \leq j \leq m - k + 1$. 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 data.

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 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.

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 w.r.t. x from i to some element of dcu(x, i) or dpu(x, i).
- P satisfies the *all-p-uses* criterion if for every node i and every $x \in$ def(i), P includes a def-clear path w.r.t. 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 w.r.t. x from i to every node in dcu(x, i); if dcu(x, i) is empty, then P must include a def-clear path w.r.t. 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 $x \in \text{def}(i)$, P includes a def-clear path w.r.t. x from i to all elements of $\text{dpu}(x, i)$; if $\text{dpu}(x, i)$ is empty, then P must include a def-clear path w.r.t. x from i to some node contained in $\text{dcu}(x, i)$. 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 $x \in \text{def}(i)$, P includes a def-clear path w.r.t. x from i to all elements of $\text{dcu}(x, i)$ and to all elements of $\text{dpu}(x, i)$.
- P satisfies the *all-du-paths* criterion if for every node i and every $x \in \text{def}(i)$, P includes every du-path with respect to x. Thus, if there are multiple du-paths from a global definition to a given use, they must all be included in paths of P.
- P satisfies the *all-paths* criterion if P includes every complete path of G. Note that programs which are represented by graphs containing loops may contain infinitely many complete paths.

When selecting a criterion, there is, of course, a tradeoff. The "stronger" the selected criterion, the more closely the program is scrutinized in an attempt to locate program faults. However, a "weaker" criterion can be fulfilled, in general, using fewer test cases. The decision of which criterion to use as a basis for test data selection would depend on several factors, including the size of the program, time and cost requirements, and criticality and consequence of failure.

The criteria all-nodes (statement coverage) and all-edges (branch coverage) are often used in program testing despite the fact that it is well known that they are weak criteria. We included them here for comparison purposes. Certainly they represent necessary conditions, for if some portion of the program has never been executed, one would not in general feel confident about its behavior. A similar intuition motivated the definition of our first criterion, all-defs. We reasoned that, even if every statement and branch had been executed, if the *result* of some computation had never been used, one would have little evidence that the intended computation had been performed.

Our initial intuition, then, was that all-defs was a "stronger" criterion than all-edges and all-nodes. As we shall demonstrate in the theorem which follows, this is not necessarily the case. In fact, the criteria are "incomparable" in the sense that it is possible for a given def-use graph G and sets of complete paths P_1 and P_2, that P_1 satisfies all-edges, but not all-defs for G, while P_2 satisfies all-defs, but not all-edges. Similarly, all-nodes and all-defs are shown to be incomparable.

Once we realized that these criteria were incomparable, we tried to find a criterion which "included" the all-defs criterion, as well as all-nodes and all-edges. We were able to show that all-uses fulfilled this requirement, but might require that certain definitions be used repeatedly. We therefore wanted to find, if possible, a criterion which included both all-edges and all-defs, but did not require as many def-use pairs as all-uses.

It was at this point that we recognized the two fundamentally different types of uses a variable might have, and, as a result, understood why all-defs does not include all-edges. Choosing a c-use as a use of a definition to satisfy the all-defs criterion, when there is also a p-use of that definition, may well mean that an edge on which the unselected p-use appears remains untraversed by the test data. We then showed that all-p-uses includes all-edges, but not necessarily all-defs. This is because a definition might not have any p-uses and, thus, remain unused if only p-uses *must* be exercised. Thus, the all-p-uses/some-c-uses criterion provides a way to satisfy our goal of a criterion which includes all-defs and all-edges, but requires fewer test cases, in general, than the all-uses criterion.

The all-uses criterion requires that test data be included which cause *some* path to be traversed between every definition and each of its uses. An even stronger requirement is that test data be included which cause *every* path between a definition and its uses to be traversed. But if the program contains loops, there may well be infinitely many such paths. Thus, our "strongest" criterion, all-du-paths, requires that test data be included which traverse every du-path between a definition and each of its uses, thus avoiding this problem.

We have defined one other criterion, all-c-uses/some-p-uses. One reason it was included was for symmetry. A more important reason is that it provides a way of guaranteeing that each definition has been used, with emphasis on the flow of data, rather than the flow of control (as in the case of all-p-uses/some-c-uses.) Of course, all-uses includes both, and the chief advantage of the all-c-uses/some-p-uses criterion would again be one of cost.

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 $c_1 \Rightarrow c_2$, provided 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 a transitive relation. We say that criteria c_1 and c_2 are *incomparable* if neither $c_1 \Rightarrow c_2$ nor $c_2 \Rightarrow c_1$.

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 labeled edges. This follows immediately from our definition of block, and the requirement that all transfer statements be effective.

Theorem: The family of criteria is partially ordered by strict inclusion as shown in Fig. 2. Furthermore, criterion c_i strictly includes criterion c_j if and only if it is explicitly shown to be so in Fig. 2 or follows from the transitivity of the relationship.

Proof: In most cases the inclusion is immediate and will not be discussed. In such cases we restrict our proof to the demonstration that each inclusion shown in Fig. 2 is in fact a strict inclusion, and that pairs of criteria that are not shown in Fig. 2 to be related by strict inclusion are incomparable.

1) all-paths $\Rightarrow$ all-du-paths

Let G be any graph containing an infinite number of paths. G can contain only a finite number of different du-paths. Let P be the smallest set of complete paths containing all du-paths of G. By definition, P satisfies the all-du-paths criterion. Since P can contain at most one complete path for each du-path (otherwise there would be a smaller set of complete paths containing all du-paths), P must be finite and, hence, does not satisfy the all-paths criterion for G. Notice that this argument shows that the primary problem associated with the all-paths criterion, potentially infinitely many paths to be traversed, is

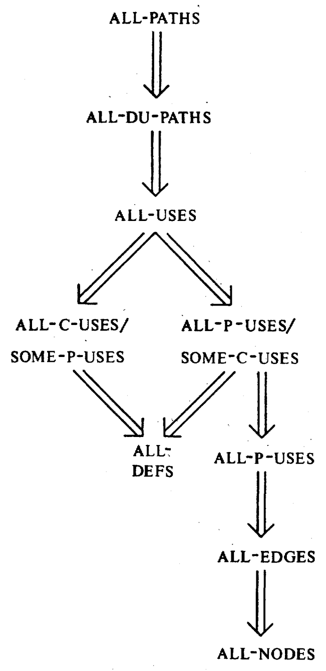

Fig. 2.

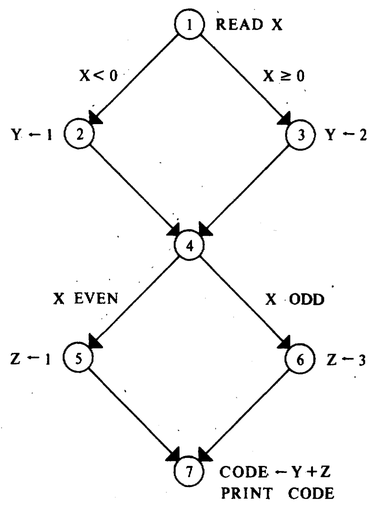

Fig. 3.

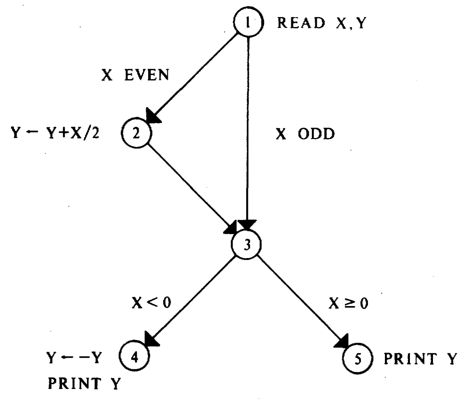

Fig. 4.

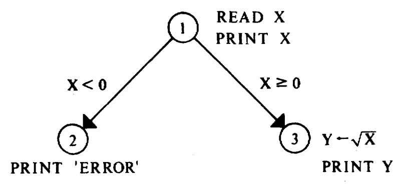

Fig. 5.

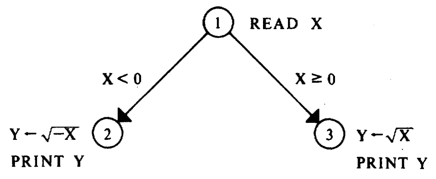

Fig. 6.

solved if all-du-paths is used. Thus, all-du-paths would be a usable criterion for programs containing loops when all-paths would not be.

2) all-du-paths ⇒ all-uses

Consider the graph of Fig. 3. {(1, 2, 4, 5, 7), (1, 3, 4, 6, 7)} satisfies all-uses, but not all-du-paths, since it does not include the path (1, 2, 4, 6, 7), which includes a def-clear path w.r.t. y from node 2 to node 7. This example also points out the primary advantage of all-du-paths over all-uses. All-du-paths requires that test data be included which cause certain combinations of path segments to be traversed, whereas all-uses would not require these combinations. This type of example is frequently used to argue for the use of the all-paths criterion, rather than all-edges.

3) all-uses ⇒ all-p-uses/some-c-uses

Consider the graph of Fig. 4. {(1, 2, 3, 4), (1, 3, 5)} satisfies all-p-uses/some-c-uses, but not all-uses, since there is no def-clear path w.r.t. y from the def of y in node 2 to the c-use of y in node 5. The example makes clear the advantage of the all-uses criterion over all-p-uses/some-c-uses. By not requiring that every c-use be exercised, we might well miss an error involving an incorrect treatment of positive even values for x.

4) all-uses ⇒ all-c-uses/some-p-uses

Consider the graph of Fig. 5. {(1, 3)} satisfies all-c-uses/some-p-uses, but does not satisfy all-uses since there is no path containing the p-use of x in edge (1, 2). This is also a classic example of the problem of not requiring that test data be included which cause every node and edge to be traversed. Any problems in node 2 or edge (1, 2) would go undetected.

5) all-c-uses/some-p-uses ⇒ all-defs
all-p-uses/some-c-uses ⇒ all-defs

Consider the graph of Fig. 6. {(1, 2)} satisfies all-defs, but does not satisfy all-c-uses/some-p-uses or all-p-uses/some-c-uses. Again, as in the previous case, problems which occur in unexecuted nodes or untraversed edges would go undetected.

6) all-p-uses/some-c-uses ⇒ all-p-uses

Consider the graph of Fig. 7. {(1, 2, 3, 5), (1, 3, 4)} satisfies all-p-uses, but not all-p-uses/some-c-uses, since dpu(y, 2) is empty, but there is no def-clear path w.r.t. y from the definition of y in node 2 to the c-use of y in node 4. This example illustrates the original problem we identified: it is possible to traverse all edges and still have a definition remain unused. In this case, an incorrect assignment to y in node 2 would presumably go undetected if the all-p-uses criterion was chosen.

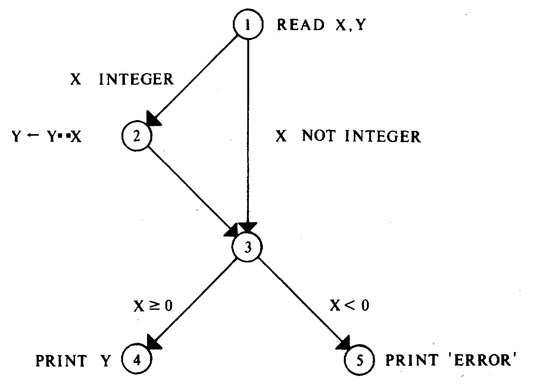

Fig. 7.

7) all-p-uses ⇒ all-edges

In this case, the inclusion does not follow immediately from the definitions. Let P be any set of complete paths of a def/use graph G that satisfies all-p-uses. We will show that every edge (i, j) in G is included in P.

Case 1: (i, j) is labeled. Then (i, j) contains a p-use of some variable, say x. Since this p-use must be preceded by a def of x along some path from the start node to (i, j), P must include a def-clear path w.r.t. x from that def to (i, j). Since (i, j) is an edge along that path, P includes (i, j).

Case 2: (i, j) is not labeled.

Case 2a: i is the start node. Then edge (i, j) must be the first edge in every complete path of G. Since G has at least one labeled edge, P contains at least one complete path, and therefore P includes (i, j).

Case 2b: i has at least one labeled in-edge. This means that i has a predecessor k, and edge (k, i) is labeled. Since j is the unique successor of i, any complete path containing (k, i) must also contain (i, j). As (k, i) is a labeled edge, it must be included in P, and therefore, so must (i, j).

Case 2c: i has only unlabeled in-edges. The definition of "block" ensures that if any node has only one in-edge, that edge is labeled. We may therefore assume that i has more than one unlabeled in-edge. Thus, there must be at least two distinct paths from the start node to i. At least one of these paths must contain a labeled edge (otherwise the paths would be identical). Select any such path $p = (s, \cdots, n_1, n_2, \cdots, i)$, where s is the start node (s may be n_1), edge (n_1, n_2) is labeled, and $(n_2, \cdots, i)$ contains only unlabeled edges. Any complete path containing (n_1, n_2) must contain $(n_2, \cdots, i)$, and since i is the unique predecessor of j, edge (i, j) must be on that complete path as well. Because (n_1, n_2) is labeled, it must be included in P, and therefore, so is (i, j).

We now demonstrate that the inclusion is strict. Consider the graph of Fig. 8. $\{(1, 2, 3, 2, 4)\}$ satisfies all-edges but not all-p-uses, as it does not include a def-clear path w.r.t. x from the def of x in node 1 to edge $(2, 4)$.

8) all-edges ⇒ all-nodes

Consider the graph of Fig. 9. $\{(1, 2, 3)\}$ satisfies all-nodes but not all-edges.

9) all-c-uses/some-p-uses and all-p-uses/some-c-uses are incomparable

The graph of Fig. 4 demonstrates that all-p-uses/some-c-

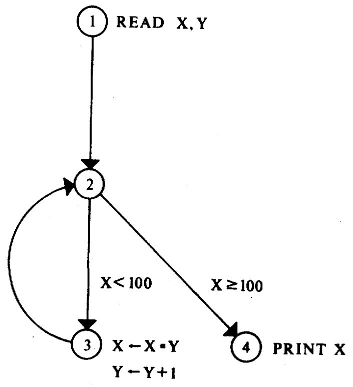

Fig. 8.

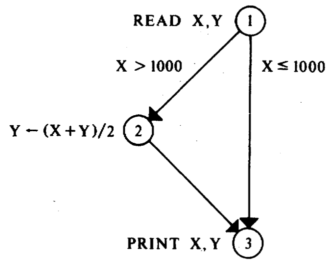

Fig. 9.

uses does not include all-c-uses/some-p-uses. $\{(1, 2, 3, 4), (1, 3, 5)\}$ satisfies all-p-uses/some-c-uses, but not all-c-uses/some-p-uses, since there is no def-clear path w.r.t. y from the def of y in node 2 to the c-use of y in node 5. Similarly, the graph of Fig. 5 demonstrates that all-c-uses/some-p-uses does not include all-p-uses/some-c-uses. $\{(1, 3)\}$ satisfies all-c-uses/some-p-uses but not all-p-uses/some-c-uses, since there is no path containing the p-use of x in edge $(1, 2)$.

10) all-defs and all-p-uses are incomparable
all-defs and all-edges are incomparable
all-defs and all-nodes are incomparable
all-c-uses/some-p-uses and all-p-uses are incomparable
all-c-uses/some-p-uses and all-edges are incomparable
all-c-uses/some-p-uses and all-nodes are incomparable

The graph of Fig. 5 demonstrates that all-c-uses/some-p-uses does not include all-nodes. $\{(1, 3)\}$ satisfies all-c-uses/some-p-uses, but does not include node 2. Because of the transitivity of inclusion, this also means that all-c-uses/some-p-uses does not include all-edges or all-p-uses, and that all-defs does not include all-p-uses, all-edges, or all-nodes. The graph of Fig. 8 demonstrates that all-p-uses does not include all-defs, since $\{(1, 2, 3, 2, 4), (1, 2, 4)\}$ satisfies all-p-uses, but the def of y in node 3 is not used along either path. Because of the transitivity of inclusion, this also means that neither all-edges nor all-nodes includes all-defs, and that all-p-uses, all-edges, and all-nodes do not include all-c-uses/some-p-uses.

AN EXAMPLE

To illustrate our thesis that a test data selection criterion should include both the all-defs and all-edges criteria, we present an example containing a simple error. The program, a translation into our programming language of the Wensley-*sq*-root program presented in [1], is shown in Fig. 10. It computes $\sqrt{p}$, $0 \leq p < 1$, to accuracy e, $0 < e \leq 1$. The program contains an error; the statements of node 5 should be interchanged.

```
     START
     READ P,E
     D ← 1
     X ← 0
     C ← 2*P
     IF C≥2 THEN GOTO D
[A]  IF D≤E THEN GOTO C
     D ← D/2
     T ← C-(2*X+D)
     IF T<0 THEN GOTO B
     X ← X+D
     C ← 2*(C-(2*X+D))
     GOTO A
[B]  C ← 2*C
     GOTO A
[C]  PRINT X
     STOP
[D]  PRINT 'ERROR'
     STOP
```

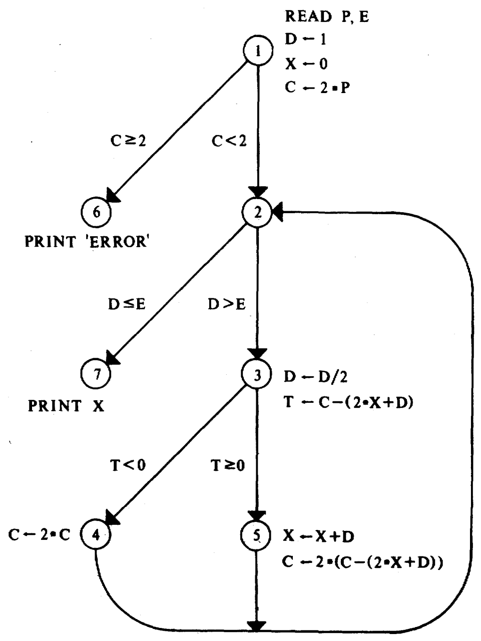

Fig. 10.

The set of paths {(1, 6), (1, 2, 3, 4, 2, 3, 5, 2, 7)} satisfies all-edges, but would not detect the error. The problem is that the definition of c in node 5 is never used unless the set of paths includes a definition-clear path w.r.t. c from node 5 to node 3. Thus, we could not expect to detect the error, in general, unless the path (5, 2, 3) is included. All-p-uses is not strong enough 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).

All-defs requires that every definition 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)}, which satisfies all-defs, should detect the error; however, node 6 and edge (1, 6) would not be exercised. Thus, any problems in this node or edge would go undetected. 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 well be an error that it does not explicitly check for $e \leqslant 1$.

All-c-uses/some-p-uses requires paths between every definition and every possible c-use of that definition. For Fig. 10, 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) or the path (1, 2, 7), and thus would not detect problems in these areas.

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. Thus, the error should be detected. Furthermore, edge (1, 6) must be included, and hence, problems in this area should be detected. In addition, the path (1, 2, 7) must be included, so the potential error described above should also be detected. However, some combinations of predicate outcomes need not be included, such as (5, 2, 3, 4) and (4, 2, 3, 5), and hence, problems reflected on such paths might not be detected.

The criterion all-uses includes both all-p-uses/some-c-uses and all-c-uses/some-p-uses, and should be able to detect all the errors described above. This criterion is similar to strategy 1 of [6] and required element testing [8]. One set of paths which satisfies the all-uses criterion for Fig. 10 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, as we saw in Fig. 3, for some programs this criterion may not test all possible combinations of predicate outcomes. This would be accomplished by all-du-paths.

Conclusions and Future Work

The data flow criteria which we have defined can be used to bridge the gap between the requirement that every branch be traversed and the frequently impossible 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. Thus, not only do our criteria fall in between branch testing and path testing in terms of difficulty of fulfillment, they also guide us in the intelligent selection of paths for testing.

A software tool based on these data flow criteria is currently being designed, and implementation should begin shortly. The user will be able to provide a program, test data, and criterion, and the tool should return a set of def-use pairs or paths required by the criterion, but not exercised by the submitted test data. Of course, some of these pairs may be within parts of the program which are not executable, or some of the required paths may be untraversable. Since these questions are, in general, undecidable [10], we cannot expect the tool to be able to select only executable ones. However, it should give the user an indication of how well the program has been tested, and which parts have been neglected.

Acknowledgment

We are grateful to an anonymous referee who made several very helpful suggestions and detailed comments which improved the organization of the paper. We are also grateful to S. Weiss for graciously doing the figures and P. Frankl for pointing out a problem in an earlier version.

References

[1] R. S. Boyer, B. E. Elspas, and K. N. Levitt, "SELECT–A formal system for testing and debugging," in *Proc. Int. Conf. Reliable Software*, Los Angeles, CA, Apr. 1975, pp. 234-245.

[2] L. D. Fosdick and L. J. Osterweil, "Data flow analysis in software reliability," *Comput. Surveys*, vol. 8, pp. 305-330, Sept. 1976.

[3] J. B. Goodenough and S. L. Gerhart, "Toward a theory of testing: Data selection criteria," in *Current Trends in Programming Methodology*, vol. 2, R. T. Yeh, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1977, pp. 44-79.

[4] M. S. Hecht, *Flow Analysis of Computer Programs*. New York: North-Holland, 1977.

[5] W. E. Howden, "Methodology for the generation of test data," *IEEE Trans. Comput.*, vol. C-24, pp. 554-560, May 1975.

[6] 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.

[7] E. Miller, "Coverage measure definitions reviewed," *Testing Tech. Newslett.*, vol. 3, p. 6, Nov. 1980.

[8] S. Ntafos, "On testing with required elements," in *Proc. IEEE COMPSAC 81*, Chicago, IL, Nov. 1981, pp. 132-139.

[9] L. J. Osterweil, "The detection of unexecutable program paths through static data flow analysis," in *Proc. IEEE COMPSAC 77*, Chicago, IL, Dec. 1977, pp. 406-413.

[10] E. J. Weyuker, "The applicability of program schema results to programs," *Int. J. Comput. Inform. Sci.*, vol. 8, pp. 387-403, Nov. 1979.

[11] —, "On testing non-testable programs," *Comput. J.*, vol. 15, no. 4, pp. 465-470, 1982.

[12] E. J. Weyuker and T. J. Ostrand, "Theories of program testing and the application of revealing subdomains," *IEEE Trans. Software Eng.*, vol. SE-6, pp. 236-246, May 1980.

[13] 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.

Sandra Rapps received the B.S. degree in mathematics from the City College of New York, New York, NY, and the M.S. degree in computer science from the Courant Institute of Mathematical Sciences, New York University, New York.

She was an Assistant Professor of Computer and Information Sciences at New Jersey Institute of Technology, Newark, and a Senior Staff Consultant at Yourdon, Inc. She is currently an independent consultant based in New York.

Elaine J. Weyuker received the A.B. degree in mathematics from the State University of New York at Binghamton, the M.S.E. degree in computer and information sciences from the Moore School of Electrical Engineering, University of Pennsylvania, Philadelphia, and the Ph.D. degree in computer science from Rutgers University, New Brunswick, NJ.

She is currently an Associate Professor in the Department of Computer Science at the Courant Institute of Mathematical Sciences, New York University, New York, NY. Prior to coming to Courant, she was a member of the faculty at Richmond College, City University of New York, and worked for IBM as a Systems Engineer. Her research interests are in software engineering, particularly software validation and verification, software reliability, and the measurement of software complexity. She is also interested in the theory of computation, and is the coauthor (with M. Davis) of the recent book *Computability, Complexity, and Languages* (New York: Academic, 1984).

Prof. Weyuker is a member of the IEEE Computer Society and the Association for Computing Machinery.

An Approach to the Modeling of Software Testing with Some Applications

THOMAS DOWNS, MEMBER, IEEE

***Abstract*–In this paper, an approach to the modeling of software testing is described. A major aim of this approach is to allow the assessment of the effects of different testing (and debugging) strategies in different situations. It is shown how the techniques developed can be used to estimate, prior to the commencement of testing, the optimum allocation of test effort for software which is to be nonuniformly executed in its operational phase. In addition, the question of application of statistical models in cases where the data environment undergoes changes is discussed. Finally, two models are presented for the assessment of the effects of imperfections in the debugging process.**

Index Terms–**Computer performance modeling, reliability growth, software reliability, software testing, stochastic models.**

Manuscript received March 8, 1983; revised October 5, 1984. This work was supported by Telecom Australia under Contract 57389 and by the Australian Computer Research Board.

The author is with the Department of Electrical Engineering, University of Queensland, St. Lucia, Brisbane, Queensland, 4067, Australia.

I. Introduction

In this paper an attempt is made to develop an approach to modeling which will allow assessment to be made of the effects of various testing strategies. The paper commences with the derivation of an expression for the failure rate of a software system; this expression is employed in the derivation of some of the later results. In order to use this expression, some knowledge concerning the distribution of faults over a software system is required. Accordingly, the problem of modeling the distribution of faults over software is addressed next