Selectivity of Data-Flow and Control-Flow Path Criteria

Steven J. Zeil

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Abstract

A given path selection criterion is more selective than another such criterion with respect to some testing goal if it never requires more, and sometimes requires fewer, test paths to achieve that goal. This paper presents canonical forms of control-flow and data-flow path selection criteria and demonstrates that, for some simple testing goals, the data-flow criteria as a general class are more selective than the control-flow criteria. It is shown, however, that this result does not hold for general testing goals, a limitation that appears to stem directly from the practice of defining data-flow criteria upon the computation history contributing to a single result.

1. Introduction

Many of the best known testing methods are path selectors, in which the set of paths through the code under test are partitioned and testing is required to cause execution of at least one path from each class. The older and better known of these methods are control-flow based [1,7,9], including criteria such as statement, branch, and path coverage, or as they are termed in [14], all-nodes, all-edges, and all-paths, respectively.

More recently, attention has focused upon data-flow based methods [3,10,13,14]. The basic building block of these methods is a tuple consisting of a definition of (assignment of a value to) a variable in some statement and a use of (reference to the value of) that variable in some statement such that the definition reaches that use (i.e., there exists some path from the defining statement to the using statement with no intervening definitions/undefinitions of that variable). From this basic building block, one may obtain such criteria as all-defs: "every definition should be tested on at least one path where it reaches some use" [14] or all-uses: "every definition should be tested on every path where it reaches a use" [8,10,14].

In the search for more rigorous criteria, a number of different approaches have been taken to expand upon the basic def-use building block. Rapps and Weyuker have formed criteria from combinations of data-flow and control-flow information [14]. Control-flow information is also used, to a lesser degree, by Laaski and Korel [10] and by Ntafos [13] but the main extensions to the basic def-use model proposed by these authors are more purely data-flow oriented. Laaski and Korel base their criteria upon a statement's context, the set of all definitions of variables reaching uses in that statement. They propose requiring coverage of each such distinct set, thus requiring a variety of different combinations of def-use pairs. Ntafos, on the other hand, bases criteria upon du-chains, a sequence of alternating definitions and uses such that a starting definition reaches some statement where it is used in a computation to define some other variable, which definition reaches a use in some statement where it is used to define some other variable, and so on. Ntafos's criteria require that paths exhibiting each possible chain (up to some specified length) be executed during testing.

Comparisons of specific data-flow and control-flow criteria can be found, among other places, in [3,14], but to date there has been little discussion about the more general classes. Specifically, little has been said regarding the relative advantages and disadvantages of data-flow versus control-flow criteria or of the different approaches to extending the simple def-use basis of data-flow criteria. This paper presents a preliminary investigation of data-flow and control-flow criteria as general classes. In the subsequent sections, canonical forms for these classes of criteria will be developed, and the utility of these classes for some simple testing problems discussed.

2. Control-Flow and Data-Flow Criteria

2.1 Basic Definitions

A control-flow graph (CFG) for a program P is a graph $G^c = (N, E^c)$ where N is a set of nodes corresponding to code in P and $E^c$ is a set of edges denoting potential flow of control between nodes of P. Because of the statement-level orientation of most path selection criteria, we will restrict our attention to CFGs in which each node corresponds to single, non-compound statements or to the condition evaluation of a compound conditional statement. For convenience, however, we will permit the use of nodes corresponding to a null or empty statement where such nodes aid in the presentation of the "structure" of the program. Figures 1 and 2 give an example of a small code module and a CFG for that module.

A path is an ordered sequence of nodes. The edge sequence for a path $(n_1, n_2, \ldots, n_{k-1}, n_k)$ is the sequence $(((n_1, n_2), \ldots, (n_{k-1}, n_k))$. A def-use graph (DUG) for a program P is a graph $G^d = (N, E^d)$ where N is a set of nodes corresponding to single statements or condition evaluations in P and $E^d$ is a set of edges labeled by variable names such that $e = (n_i, n_j) \in E^d$ with $\lambda(e) = z$ if $n_i$ contains a definition of variable z, that definition reaches $n_j$, and $n_j$ contains a use of z. Figure 3 shows the DUG corresponding to the module in Figure 1.

A path selection criterion is a procedure for deriving a set of path descriptions for a program. A set of paths $S$ satisfies a path selection criterion if, for each description generated by the criterion, some path in $S$ satisfies the description. What is meant by "satisfying a description" will vary from one form of criterion to another.
1: input x, y;
2: gcf := x;
3: while gcf ≠ y loop
4:   if gcf > y then
5:     gcf := gcf - y;
6:   else
7:     temp := gcf;
8:     gcf := y;
9:     y := temp;
end if;
end loop;
10: print gcf;

---

2.2 Control-Flow Criteria

We will say that control-flow path selection criteria are those that are based entirely upon information in the CFG. For such criteria, any two programs with identical CFGs could be tested using the same set of paths, no matter what differences might appear in the DUG. As a canonical form for control-flow criteria, we will say that a path description for a control-flow criterion consists of a regular expression over \( E^\ast \). A path satisfies a path description \( R \) if the edge sequence for that path is in \( R \).

This definition of control-flow criterion easily captures such common criteria as all-nodes, all-edges, or all-paths [14]. It can also express such less common criteria as LCSAJs [7] and minimal and non-minimal loop iteration requirements [9]. It does not, however, allow one to express a maximal loop iteration requirement, but such a requirement is not really a function of the CFG only, as it depends upon the semantics of the loop body and of the surrounding code.

---

2.3 Data-Flow Criteria

We will say that data-flow criteria are those path selection criteria based entirely upon information from the DUG. When we wish to relate the DUG to paths in the code under test (as we must in order to use the DUG as the basis for path selection), we will say that a graph \( H \) is a du-history for a path \( P \) if \( H = (N_H, E_H) \) with one node in \( N_H \) for each node occurrence in \( P \) and with \( E_H \) a set of edges labeled by variable names such that \( e = (m_i, m_j) \) is in \( E_H \) with \( \lambda(e) = z \) whenever a def of \( z \) in \( m_i \) reaches a use of \( z \) in \( m_j \) along \( P \) (i.e., whenever \( m_0 \) and \( m_j \) correspond to occurrences in \( P \) of CFG nodes \( n \) and \( n_j \), the DUG contains an edge \( e' \) from \( n_i \) to \( n_j \) with \( \lambda(e') = z \), and there does not exist a node \( m_k \) occurring in \( P \) between those occurrences of \( n_i \) and \( n_j \) for which the DUG contains an edge \( e'' \) from \( m_k \) to \( n_j \) with \( \lambda(e'') = z \)).

Figure 4 shows the du-history of the path traversed by all inputs to the module of Figure 1 where \( x > 0 \) and \( y = 2 \times x \).

A subgraph of some du-history of \( P \) is a du-scenario for some CFG node \( m \) if it contains a node \( m \) corresponding to an occurrence
of $n$ in $P$ such that $m$ has no successors within the subgraph and such that for each node $m'$ in the subgraph, there is a path from $m'$ to $m$.

A path description for a data-flow criterion is a du-scenario. A path satisfies such a path description if the scenario is a subgraph of that path's du-history. Figure 5 shows some du-scenarios that would be satisfied by the path whose history appears in Figure 4.

Du-scenarios are closely related to the data-flow constructors in [10,13]. A scenario constructed via breadth-first traversal (against the direction of the edges) of a du-history from some node $n$ yields the context of $n$ for that path, as in Figure 5b. Depth-first traversals yield the du-chains ending at $n$, as in Figure 5c. Of course, more general scenarios can also be constructed, as in Figure 5d.

3. Selectivity

Most discussions of the testing power of control and data-flow criteria have focused on specific criteria and on the ability of certain of those criteria to subsume others, (i.e., to be satisfied only by sets of paths that would also satisfy the others). While subsumption provides important information regarding upper limits on the power of a criterion, it does not address other important questions such as the kinds of faults and/or errors likely to be detected by such criteria or the cost of using various criteria to answer such questions as the kinds of faults and/or errors likely to be detected by such criteria or the cost of using various criteria to answer such questions.

Let $C$ be a predicate denoting a desired testing goal such that $C(S)$ is true for a set $S$ of paths if and only if that set meets the desired goal. We will say that one criterion is more selective than another with respect to $C$ if

1. the first criterion is satisfied only by sets of paths on which $C$ is true and there exists some program for which the second criterion is satisfied by a set of paths on which $C$ is false; or

2. both criteria are satisfied only by sets of paths for which $C$ is true but the first criterion selects sets of paths that are never larger than, and for some program is smaller than, the sets selected by the second criterion.

We will say that one class of criteria is more selective than another with respect to $C$ if no criterion in the second class is more selective than all criteria in the first class and if, for some criterion from the second class, there exists a more selective criterion in the first class.

In the remainder of Section 3, we will give examples of two useful testing goals for which the class of data-flow criteria is more selective than the class of control-flow criteria. We will also show, however, that subsumption relations between the two classes indicate that some predicate must exist for which the control-flow criteria must be more selective.

In this connection, it is useful to split the process of revealing a fault via testing into two steps: 1) the origination [15] or creation [11,12] of an erroneous internal execution state by the execution of a faulty statement and 2) the transference [16] or propagation [11,12] of that internal error from statement to statement, yielding a series of successive erroneous internal states, until an output statement (any statement whose operation is visible to an external oracle charged with determining the correctness of the code's response to test data) is executed that reveals the internal error as an actual error in the program behavior. Although the origination/transference model of [15] in general offers finer discriminations than does the creation/propagation model of [11,12], the difference lies primarily in how erroneous states within the sequence of operations constituting a single statement are treated. Since path selection criteria abstract away the details of individual statements, we will treat the two models as equivalent in this discussion.

3.1 Simple Transference

One natural error-detecting property that we would expect data-flow criteria to exhibit is good transference of errors in the computed values of variables. An erroneous assignment to some variable $x$ is transferred by any statement that does not reassign $x$ and may additionally be transferred to a variable $y$ (possibly the same variable) by any assignment $y := e$, with $e$ an expression containing a use of $x$. Whether or not transference actually occurs in the latter case depends upon the expression $e$, the value of $x$, and the values of other variables used in $e$.

Since most paths can be executed by more than one set of inputs, it is typical for a path to permit many of its variables to take on a number of distinct values. Consequently, the question of whether or not transference really occurs through an assignment statement is largely outside the purview of data-flow criteria and must normally be addressed during the selection of the actual inputs used to force execution of the desired paths. We will therefore concentrate upon the choice of paths along which, if appropriate inputs were chosen, transference to some output statement might occur.

We will say that $p$ is a simple transference path for variable $x$ if $p$ contains the sequence $(n_i, \ldots, n_j)$, $n_i$ contains a definition of $x$, $n_j$ corresponds to an output statement, and the du-history of $p$ contains a path from the occurrence of $n_i$ to the occurrence of $n_j$ corresponding to that sequence. We then have the following result:

**Theorem 1** The data-flow criteria are more selective than the control-flow criteria with respect to selection of simple transference paths.

Proof: For any program, any variable $x$, and for any CFG node $n$ from which a simple transference path exists, it is clear from...
Figure 5: Some Du-Scenarios Satisfied When $x > 0, y = 2 \cdot x$

Input $x_0$:

\[ z_i := f(x_0); \quad \text{-- } f \text{ is a faulty expression} \]

While $c(x_0)$ loop

Case $g(x_0)$ of

1: $S_i(\#)$;
2: $S_{i-1}(\#)$;
end case;
end loop;
print $z_n$;

Where the $S_j$ are code sequences of the form:

\[ z_{j+1} := z_j; \]

for $k := 1$ to $n$ loop

if $k \neq j+1$ then

$z_k := 0$;
end if;
end loop;

And where $i$ is a permutation of the integers from 1 to $n-1$.

Figure 6: Programs With Single Simple Transference Paths

The definition of simple transference paths that one can trivially give a du-scenario satisfied only by a simple transference path. (Note also that the existence of simple transference paths can be determined via static data-flow analysis.) Consider the data-flow criterion consisting of that one du-scenario. Clearly this criterion will select a simple transference path for $x$ from $n$ while requiring the selection of only a single test path.

For control-flow criteria, it is clear that there exist criteria guaranteed to force selection of a simple transference path whenever one exists. This follows from the fact that path coverage (all-paths) is a control-flow criterion. In general, however, the control-flow criteria may require an inordinate number of paths before selection of a simple transference path is guaranteed. Consider, for example, the class of programs having the form shown in Figure 6. For this program, there exists exactly one simple transference path from the faulty assignment to $z_1$ to the output statement at the end. The path is different, however, for each possible permutation in $i$. Since the choice of $i$ does not affect the CFG, a control-flow criterion can guarantee selection of a simple transference path only if it forces selection of the simple transference paths for all $i$. This in turn means that the control-flow criterion would need to choose $(n-1)!$ paths for any program exhibiting the same CFG. It follows then that the data-flow criteria are more selective.

It is worth noting that the ability of data-flow criteria to selectively force execution of simple transference paths comes entirely from the du-chain component of the du-scenario. The du-context was irrelevant in this instance. As a result, it is true that an existing set of criteria, the Required $k$-tuples [13], would force execution of simple transference paths, although they would not be any more selective than would control-flow criteria. The next example presents a testing goal for which both the chaining and the context information are relevant.

3.2 Variable/Constant Differentiation

A number of testing criteria (not necessarily path selection criteria) seek to force each variable in the program to take on two or more distinct values during testing, in hopes of determining whether that variable could be replaced by a constant or of demonstrating that the subsequent uses of that variable are not merely correct for some selected, presumably common, special case [1,2,4,6,17,16]. There are two important cases to consider when trying to force a variable to take on two distinct values. First, there may exist a path along which the variable is assigned some function of the program inputs. (This need not occur in a single, direct assignment. It is possible that the variable in question will be assigned a function of some set of other program variables, some of which are themselves functions of the program inputs.) Second, the variable in question may always be assigned some constant value (again, possibly by assigning to it a function of some set of program variables that are all constants) along each path, but potentially a different constant value for different paths (loop counters would be a common example of such a variable). Based upon these two cases, we can state that the determination of whether a given variable $z_k$ can be replaced by a constant in some node $k$ requires:

1. The selection of a path through $n$ such that
(a) \( z \) is assigned a function of the program inputs along the path's prefix leading to \( n \),
(b) from the last definition of \( z \) prior to \( n \) in that path, a simple transference path is subsequently followed within the path, and
(c) there exists inputs in the domain of that path causing that function to yield two or more distinct values;

2. or the selection of a pair of paths passing through \( n \) such that
(a) \( z \) is assigned different functions of the program inputs along the two paths' prefixes leading to \( n \), and
(b) from the last definition of \( z \) prior to \( n \) in each path, a simple transference path is subsequently followed within each path, and
(c) there exist inputs in the domain of each path for which those two functions assigned to \( z \) are not equal and for which transference actually occurs along the simple transference paths.

Condition (c) in each major case is clearly outside the purview of path selection, although it represents a kind of problem that must always be solved when combining path selection with data selection. We will say that a path satisfying the first two conditions within either major case differentiates \( z \) from a constant at \( n \). We then have the following result:

**Theorem 2** Data-flow criteria are more selective than control-flow criteria at differentiation from constant values.

**Proof:** In part, this theorem follows from the first theorem regarding the selection of simple transference paths. We can directly conclude from that theorem that data-flow criteria exhibit superior selectivity with respect to part (b) of each major case. The superior selectivity of data-flow criteria in determining whether case 1 is possible and in selecting paths satisfying (1a) can be proven by a minor modification of the proof of Theorem 1 in which we seek a "transference" from an input statement to \( n \) instead of from \( n \) to an output statement. For part (2b), however, we need to show that data-flow criteria are more selective than control-flow criteria at selecting pairs of paths along which \( z \) is assigned two distinct functions of the program inputs. Since control-flow criteria have no information regarding where \( z \) is defined, what variables are used in the definition, where those variables are defined, etc., they can in general provide this property only by selection of all paths leading to \( n \). In contrast, for any path through \( n \), the largest du-scenario at the last definition of \( z \) is actually a graphical encoding of the function of the program inputs assigned to \( z \) along that path. Thus, if there exist programs for which the number of distinct largest du-scenarios at some node is smaller than the number of paths reaching that node, the higher selectivity of data-flow criteria will follow.

Such programs are easy to construct. For example, Figure 7 shows a program where differentiation of \( y \) and/or \( z \) at the print statement requires only two paths.

4. Subsumption versus Selectivity

In view of the preceding examples, it appears natural to ask whether the data-flow criteria are always more selective than the control-flow criteria. If so, then the current practice of mixing data-flow and control-flow aspects within a single criterion [13,14] would seem to be ill-advised. Actually, according to the definition of "more selective", this could not be true in a formal sense because there must necessarily exist predicates not satisfiable by any path selection criterion (e.g., the predicate \( C(S) = \text{false} \)). We will show however, that, this conjecture is false in a more substantive way, namely that there exist predicates for which control-flow criteria are more selective than data-flow criteria. As a first step, we will show that all-paths is not a data-flow criterion.

**Theorem 3** There exists no data-flow criterion forcing the selection of all paths through all programs. (i.e., All data-flow criteria are strictly subsumed by all-paths.)

**Proof:** Consider the code in Figure 8. Enumeration of the path histories for all four paths through this program and of the du-scenarios satisfied by those histories will show that any set of two paths satisfying all-edges will also satisfy all du-scenarios. Consequently, no data-flow criterion could require any additional paths beyond those two, and it follows that no data-flow criterion will require satisfaction of all-paths for this program. \( \square \)

It then follows almost immediately that:

**Corollary 1** There exist programs and predicates for which the control-flow criteria are more selective than data-flow criteria.

**Proof:** As proof, take the program from the previous proof and let the predicate be "all-paths is satisfied." \( \square \)
The proof of Theorem 3 and its Corollary is unsatisfying in two respects. First, it leaves open the possibility that this result is due to some limitation of our canonical form for data-flow criteria rather than being a fundamental limitation of data-flow criteria. Second, it fails to indicate whether there are any predicates representing practical testing goals for which control-flow criteria are more selective. Both of these objections are discussed in the remainder of this Section.

The fact that all-paths is not a data-flow criterion seems to be a fundamental property arising from a basic assumption regarding those criteria. To see this, suppose that we were to try and find a necessary and sufficient condition under which all-paths for a given program would in fact be realizable as a data-flow criterion. For this to be true, there would have to exist some data-flow path description for each path through the program's control-flow graph. Let us represent an arbitrary such path via the control-flow path description: \( E* e_e \), where \( E \) is a sequence of edge labels \( e_1 e_2 \cdots e_n \), and where \( e_i \) is a sequence of indices denoting each successive "branch" (i.e., each edge whose head node has out-degree of 2 or more) in the path.

We shall now attempt to inductively investigate the conditions under which a data-flow path description would exist for this path. As a base case, we begin with the path description \( E* e_e \). Define the disjoint node set of a branch edge \( e \) as the set of nodes reachable from the tail of \( e \) without passing (again) through the head node of \( e \) but not reachable from the tail of any sibling branch of \( e \) (i.e., any edge having the same head node as \( e \)) without passing through the head of \( e \).

Now, a data-flow path description capable of forcing execution of some path in \( E* e_e \) will exist if the disjoint node set of \( e_e \) contains a node with a definition that reaches some use or if the disjoint node set of \( e_e \) contains a node with a use reached by some definition. If we further assume that the start node(s) of any control-flow graph contain definitions of all variables, then a data-flow path description for this base case would also exist if the disjoint node set of each sibling of \( e_e \) contains a definition reaching a use outside the disjoint sets of \( e \) and its siblings.

Although non-trivial, the conditions for this base case do not appear to be terribly restrictive. In fact, most treatments of data-flow criteria have imposed restrictions upon the control and data uses that would satisfy or nearly satisfy these conditions. Examples are assumptions that the program contains no data-flow anomalies, that every branch predicate contains at least one use, that predicate uses are associated with the branch edges rather than with the nodes, etc. [3,13,14,16].

Next, assume that a data-flow path description exists that is satisfied only by paths in \( E* e_e, E* e_e, \cdots E* e_e \). Under what conditions would there exist a data-flow description satisfied only by paths in \( E* e_e, E* e_e, \cdots E* e_e \)? The new path description would exist only if the DUG contains a path from some node in the disjoint node set of \( e_e \) to some node occurring in the old path description, or if, for every sibling of \( e_e \), the DUG contains a path from some node in that sibling's disjoint node set to some node occurring in the old path description.

It is this condition that the program in Figure 8 fails to satisfy. The fundamental difficulty in meeting this condition is that programs can contain sequences of computations that are, in effect, decoupled from one another, in that each sequence contributes to the eventual computation of a different result. If the control-flow structure of the program is decoupled in a similar fashion, we are left with branches that do not interact in any data-flow significant manner. There are many assumptions that have been made by various authors that work to increase the coupling of different branches, including the assumptions listed above in the base case discussion. There are others that could be reasonably imposed (e.g., treating "print" and other I/O statements as definitions/uses of a shared file variable), but these do not appear to be sufficient conditions for this step in the induction.

It appears, then, that the failure of data-flow criteria to be selective enough to force execution of an arbitrary control path is directly due to the assumption that data-flow path descriptions should be based upon the history of computations contributing to a single result. One could certainly conceive of a new definition of data-flow criteria where this would not be the case, but such a relaxation would be a fundamental change in the character of what most observers would regard as a "data-flow criterion". From this standpoint, the mixture of data-flow and control-flow elements within the same criterion begins to appear reasonable, although other approaches may also be of interest.

Finally, we turn to the question of the practical significance of this limitation on data-flow criteria. Certainly, there are practical testing goals that should be accessible to path selection criteria (and that therefore can be satisfied by all-paths), but that are not in general satisfiable by data-flow criteria as defined here. For example, in the previous Section, we discussed the selection of paths allowing variable/constant discrimination. A related, and equally popular testing goal might be termed variable/variable discrimination, which by analogy with Theorem 2 will require the selection of paths on which a given pair of variables are assigned distinct functions of the program inputs, a necessary condition to determining whether one variable name has been substituted for another [2,4,5,16]. Data-flow criteria cannot guarantee this property, because if the variables in question are not actually both used within the node where discrimination is desired, then the computation of the variables in question might be too "de-coupled" from one another to force the desired combinations of branches. On the other hand, this particular goal could be satisfied by replacing the DUG in the path description definition by a similar graph where edges exist from each definition to each node reached by that definition, whether or not that node contains a use of the defined variable. In essence, such a substitution introduces "might have been used" to the realm of data-flow criteria. Perhaps surprisingly, this much more tightly connected data-flow graph would still not, in general, allow a data-flow criterion equivalent to all-paths. With or without this change, it remains unclear whether there are testing goals of practical significance for which control-flow criteria other than all-paths would be more selective than data-flow criteria. In particular, if certain control-flow constructs are decoupled from one another with respect to data-flow, there may be little practical advantage to requiring them to be tested in varying combinations relative to one another.

5. Conclusions

As general classes, the data-flow criteria appear to be more selective than control-flow criteria for some practical testing goals,
although current data-flow criteria do not exhibit anywhere near the theoretically possible degree of selectivity for those goals. The higher selectivity of data-flow criteria does not hold for general testing goals, but the practical significance of this limitation is open to question. Nonetheless, this limitation does appear to offer support for the practice of mixing data-flow and control-flow elements together within a single path selection criterion.

REFERENCES


