Correspondence

On the Relationships Among the All-Uses, All-DU-Paths, and All-Edges Testing Criteria

Allen S. Parrish and Stuart H. Zweben

Abstract—The all-du-paths data flow testing criterion was designed to be more demanding than the all-uses criterion, which itself was designed to be more demanding than the all-edges criterion. However, formal comparison metrics developed within the testing community have failed to validate these relationships, without requiring restrictive or undesirable assumptions regarding the universe of programs to which the criteria apply. In this correspondence, we show that the formal relationships among these criteria can be made consistent with their intended relative strengths, without making restrictive or undesirable assumptions.

Index Terms—Adequacy criteria, data flow testing, subsumes, covers.

I. INTRODUCTION

Data flow testing criteria were developed as alternatives to popular control flow criteria such as statement and branch coverage. The idea in data flow testing is to base coverage on aspects of the definition and usage patterns of the variables in the program, rather than simply on control characteristics such as coverage of all the edges in the program's flowgraph. The rationale for this approach is that it will lead to adequacy criteria that are both intuitively and theoretically stronger than the control flow criteria, yet not so demanding that they require the pragmatically impossible coverage of all paths in the program. Therefore, analyses of the relationships among the different criteria are of interest, since they may provide some insight into the relative adequacy of test sets that satisfy them. To make such analyses of practical value, it also is of interest to place as few restrictions as possible on the programs to which the criteria and their relationships apply.

The various data flow criteria are distinguished by the particular variable uses that are required to be covered for each definition in the program. For example, the all-uses criterion requires that, for every definition and every use associated with that definition (a use is associated with a definition if there is at least one path from the definition to the use along which the variable is not redefined; that is, there is at least one "definition-clear" path from the definition to the use), the test set must cause traversal of at least one path from that definition to the use. Uses may occur in predicates or computations. If a use occurs in a predicate, every branch directly emanating from that predicate must be traversed in order for that use to be covered. The all-du-paths criterion requires, for every definition-use association, that every simple path covering that association be traversed by the test set. A simple path is one in which every node traversed, except possibly the first and last, is distinct.

For each required definition-use association, there is a subdomain of the program's input space that causes the program to traverse that definition-use association. These subdomains can (and frequently do) overlap for a given program and criterion, and two or more definition-use associations may be covered by exactly the same subdomain.

Thus, the set of definition-use associations relevant to a criterion induces a multiset of subdomains from the input space [4], [5].

Coverage of some of the definition-use associations may not be feasible due to the program's logic. Undecidability results make it impossible, in general, to guarantee being able to tell if all the definition-use associations of interest are feasible, though for many programs one can tell if the feasible definition-use associations of interest have been covered. Here, we discuss only the feasible versions of the criteria, since there is no point discussing attempts to test code that never can be reached. Thus, by all-uses we mean that at least one (feasible) path must be traversed for each feasible definition-use association. All-du-paths requires every (feasible) simple path to be traversed for each feasible definition-use association. All-edges requires every feasible edge in the program's flowgraph to be traversed.

Three relations designed to formally compare testing criteria are the subsumes, covers, and properly covers relations. The subsumes relation can be defined as follows: Criterion A subsumes criterion B if every test set satisfying A also satisfies B (for every program and specification). The covers and properly covers relationships were developed in recent work [4], in part to address limitations of the subsumes relation in comparing the defect-revealing ability of criteria. Criterion A covers criterion B if, for every subdomain S of the multiset induced by B, there is a subdomain S of the multiset induced by A whose union is exactly S. A properly covers B if it covers B and, in addition, the number of times a subdomain S of A is used to characterize the various subdomains of B is at most the number of times S appears in the multiset induced by A.

The covers and properly covers relations are relative to a given program (and specification, if the criterion depends on the specification also). Thus, the terms universally covers and universally properly covers were introduced to indicate coverage (proper coverage) for all programs and specifications. Since we will not need to discuss these relationships in the context of a single program and specification, we simply use the terms covers and properly covers to mean universally covers and universally properly covers, respectively.

All-du-paths was designed to be more demanding than all-uses, while all-uses was designed to be more demanding than all-edges. Formally, one therefore would expect all-du-paths to subsume (cover, properly cover) all-uses (cover, properly cover) all-du-paths. All-du-paths results indeed exist showing conditions under which all-du-paths subsumes (covers, properly covers) all-edges [3], [4], [5]. However, each of these results imposes a restriction on the class of programs under consideration, a restriction that the authors of these results themselves note is undesirable. The literature also shows that all-du-paths neither subsumes, covers, nor properly covers all-uses [3], [5]. One can, however, identify restrictions on the class of programs under consideration so that the relationships will hold.

In this correspondence, we examine these relationships and their associated restrictions. We show how the restrictions can be relaxed while preserving the intended relationships between all-uses and all-edges, and between all-uses and all-du-paths. In the former case, the discussion involves a property known as "no feasible anomalies," which has been assumed in much of the data flow testing literature. We will show how it is possible to relax this restriction under a certain model of language execution, while under other models, the no feasible anomalies property already is unrestricted. In the latter case, the problem is with the definition of all-du-paths. This problem also was noted in [7]. A revised definition of the criterion, along the lines
suggested in [7], is shown to solve the problem. The former case (all-uses versus all-edges) is the subject of the next section, while the latter case (all-uses versus all-du-paths) is the subject of Section III.

II. ALL-USES AND ALL-EDGES

Previous work comparing the all-uses and all-edges criteria makes varying assumptions about the set of programs under consideration. In particular, different assumptions are made about the presence of data flow anomalies [2], [10]. The anomalies issue centers around a property called no anomalies (NA), defined in [3], and its "feasible counterpan," no feasible anomalies (NFA), defined in [4]. A criterion satisfies the NA (NFA) property if, for every program, every path (feasible path) from the beginning of the program to a use of a variable \( x \) must pass through a definition of \( x \).

A. Anomaly Assumptions and Subsumes

It can easily be shown that, provided NA is assumed, all-uses subsumes all-edges [3]. However, NA is too restrictive. In particular, a program can be reasonable and still contain indefeasible paths from the beginning of the program to a particular use of a variable, along which there is no definition of that variable. Such perfectly good programs are eliminated by NA from the universe of programs subject to testing. Alternatively, it is observed in [3] that NA may be guaranteed by considering the entry node in every program to have a definition of each variable. This method of determining definitions makes NA nonrestrictive, although it is noted in [3] that it is not consistent with the actual data flow in some programs. This issue is discussed further in Section II.C below.

NFA is a more reasonable assumption, in that it only requires definitions to appear on all feasible paths leading to every use. Moreover, as is shown in [4], [5], NFA guarantees that all-uses subsumes all edges. However, although NFA is a theoretically acceptable restriction, it is undecidable and therefore pragmatically problematic, since it is impossible to guarantee (e.g., by a preprocessor) that it is satisfied by the universe of programs being tested. That it is undecidable to place undecidable restrictions on the class of programs to which criteria apply is observed in [3]. Under a weaker, decidable restriction (that, for every predicate use of a variable, there is at least one path from the start to that use containing a definition of the variable), [3] proves that all-uses does not subsume all-edges.

B. Anomaly Assumptions and (Properly) Covers

In [4], it is shown that all-uses covers all-edges. However, this result assumes the NFA property, making it subject to the same pragmatic difficulties as above. The proof of the result relies on the NFA property in the following sense. For a given use of a variable \( v \), let \( D_v \) be the subdomain consisting of the set of inputs that executes \( v \). By NFA, we know that every feasible path leading to \( u \) contains some definition of \( v \). By all-uses, each of these definition-use associations must be covered. Thus, the union of the subdomains induced by these definition-use associations must be \( D_v \).

The proof that all-uses properly covers all-edges is similar to the proof that all-uses covers all-edges, and is identical with respect to the use of NFA. Note also that it is possible to use NA rather than NFA in deriving any of these results (as was the case with the subsumption results of the previous section), as NFA is a trivial consequence of NA.

C. Anomaly Assumptions and Language Execution Models

The previous sections observe that NFA is sufficient to guarantee that all-uses subsumes, covers and properly covers all-edges; however, NFA is a pragmatically undesirable property because of its undecidability. NA is also sufficient to guarantee these relationships; however, NA either rules out interesting programs or, by assuming that the entry node has a definition of every variable, may not accurately reflect the data flow in every program [3].

While universally assuming that the entry node has a definition of every variable is indeed inappropriate under some language execution models, we believe that there are other models where such an assumption is appropriate for every program. Specifically, this issue is impacted by how the language treats variables that have not been defined explicitly by the program. There are three approaches to the treatment of such variables [1], [9]:

1) The initial value is arbitrarily obtained from the environment (i.e., from leftover values in memory). Thus, the initial value for an undefined variable may vary depending on the context in which the program is executed. This often is the way dynamically allocated memory is handled.

2) There is a specific, defined initial value. In general, the initial value for a variable of a given type is some base value for that type.

3) References to undefined variables are defined to be illegal. This rule is enforced by checks at compile-time and at run-time.

All of these environments are legitimate. 11 is quite common, simply because it requires no special implementation. However, 12 is strongly supported in [11], while 13 is strongly supported in [8]. Clearly, both 12 and 13 provide additional support for software reliability. Although both involve additional implementation cost, for 12 this is typically more [6], and for both 12 and 13 there has been some recent work toward reducing that cost [11].

For 11 and 12, all variables can be assumed to be defined at the start of the program (either from the environment as in 11, or to a specific value as in 12). Thus, NA is guaranteed, as discussed above. Since NA holds for all programs under the 11 and 12 models, all-uses subsumes, covers, and properly covers all-edges under those models.

By definition of 13, undefined variables are not implicitly given values on entry. Thus, it is inconsistent with 13 to assume all variables are initialized on entry simply to guarantee that NA holds. However, the following property captures the 13 model exactly; that is, it is satisfied by all programs under the 13 model.

**DEFINITION. (Terminate on Anomalies (TA)) Program execution terminates at any statement where an undefined variable is used.**

The TA property can be used in place of NA and NFA to establish that all-uses subsumes, covers, and properly covers all-edges. We first consider subsumes. In the absence of NA, [3] showed that all-uses failed to subsume all-edges because of programs in which there is a branch (edge) that is feasible, but neither the code along the branch nor the predicate governing the branch contains variable uses that are part of feasible definition-use associations. By assuming TA, neither edge following the predicate governing the branch is executable, because the program will terminate with an error message when the predicate is executed. Thus, all-edges will not require any edge following the predicate to be executed at all, and therefore it is not necessary for all-uses to execute either edge following the predicate.

It also is possible to establish that all-uses covers (properly covers) all-edges under this assumption. Consider a use \( u \) of a variable \( v \) that is undefined on some feasible path \( p \) leading to \( u \). Taking path \( p \) will result in immediate termination before completing the execution of \( u \). Thus, any input that drives path \( p \) should be excluded from \( D_v \), the subdomain of inputs that cause the program to execute \( u \). Consequently, it is impossible to cover any edge emanating from \( u \) by executing \( p \). That is, although inputs that drive path \( p \) are not in any of the subdomains needed for satisfying all-uses, they also are not in the...
III. ALL-DU-PATHS AND ALL-EDGES

The argument that all-du-paths fails to subsume all-uses is illustrated by the program in Fig. 1 (adapted from [3]).

```plaintext
input x, y;  
i := 1;  
while i <= 2 do  
  write 'hello';  
i := i + 1  
end while;  
if y > 0 then  
  output x  
endif;
```

Fig. 1. All-du-paths/all-uses subsumption failure.

This program contains feasible definition-use associations (e.g., the ones involving x and y) that require two iterations of the loop to be traversed. Therefore, any path that covers these associations is not simple. This means that, to satisfy all-du-paths, it is possible to avoid exercising both branches of the if-statement. Yet to satisfy all-uses, some feasible path must be traversed for each feasible definition-use association. Therefore, both branches of the if-statement will be exercised by any test set satisfying all-uses.

The restriction to simple paths is made to prevent the number of paths that must be tested from becoming unbounded. It appears that the subsumption problem above is caused by the expectation that, whenever some definition-use association is feasible, there is at least one feasible simple path covering it. That is, the definition-use association does not have an inherently cyclic cover. Formally, we define the notion of an inherently cyclic cover as follows.

**Definition.** (Inherently Cyclic Cover) A definition-use association du has an inherently cyclic cover iff all feasible paths covering du contain cycles.

We say that programs containing no definition-use associations with inherently cyclic covers satisfy the NICC (No Inherently Cyclic Covers) property.

From the definitions of the criteria, it is easy to see that, by restricting the applicability of all-du-paths to programs satisfying the NICC property, all-du-paths subsumes (covers, properly covers) all-uses. Assuming NICC to guarantee these relationships is analogous to assuming NA (NFA) and TA in the previous section to guarantee that all-uses subsumes (covers, properly covers) all-edges. However, restricting the applicability of all-du-paths to programs satisfying NICC is not a good solution. In particular, many very reasonable programs contain loops that must be traversed. For-loops are a good example of such loops. Horgan and London [7] observe that C programs often contain definition-use associations with inherently cyclic covers.

Although NICC is not a reasonable restriction, it is possible to modify the definition of all-du-paths such that the desired results hold. One possibility for such a modification is to eliminate the restriction to simple paths, since the total number of executable paths in any program is actually finite (provided an assumption is made that there are a finite number of values that any given program variable can ever assume). However, this is impractical for many programs where, for example, the number of iterations of a loop is dependent on the value of an input. A more pragmatic alternative is, if some feasible definition-use association has no feasible simple path covering it, then include in the test set required to satisfy all-du-paths at least one of the feasible non-simple paths covering it. Horgan and London [7] also suggest that non-simple paths count as having covered their corresponding simple sub-path.

Thus, we define the revised form of all-du-paths as follows: A test set T satisfies revised-all-du-paths for a program p iff:

1) T satisfies all-du-paths for p, and,
2) For every definition-use association in p having an inherently cyclic cover, some non-simple path covering that definition-use association is traversed by some element of T.

This definition is not considered in any of the theoretical work comparing the criteria [3], [4], [5]. Yet, as observed in [7], such a definition is more useful in practice than is the original one. As such, it should be considered in theoretical analyses as well.

To see that revised-all-du-paths subsumes all-uses, note that every feasible definition-use association either has a feasible simple path covering that association or it doesn't. In the former case, the first conjunct in the definition will guarantee coverage of the definition-use association, while in the latter case the second conjunct guarantees coverage. Thus, even if every feasible path covering a definition-use association requires traversing a cycle, that definition-use association is guaranteed to be covered.

To see that revised-all-du-paths properly covers all-uses, note that the subdomain induced for a definition-use association by all-uses is just the union of the subdomains induced for the same definition-use association by revised-all-du-paths. This guarantees that a subdomain induced by revised-all-du-paths cannot be used too many times in the coverage of the subdomains induced by all-uses. Since revised-all-du-paths properly covers all-uses, it also covers it.

IV. CONCLUSION

In this correspondence, we showed how it is possible to preserve the intended relationships between all-uses and all-edges, and between all-uses and all-du-paths, without severely restricting the class of programs to which the criteria can be applied. In the former case, we clarified the use of the NA and NFA properties in the analysis of the data flow testing criteria. There has been some uncertainty in the data flow testing literature regarding whether or not these properties should be assumed. In this correspondence, we observed that the answer to this question depends on how undefined variables are treated within the programming language. We showed that NA (and therefore NFA) is consistent with most language models, with the exception of the model that treats undefined variables as errors. For this model, we identified a property called TA that is satisfied by all programs and preserves the intended relationships among the data flow criteria. In the latter case, we showed that while a restriction to programs satisfying NICC is inappropriate, a redefinition of the all-du-paths criterion, along the lines suggested in previous research, will preserve the intended relationships. We feel that clarifications such as this one strengthen the foundation for future work in this area, and provide a better basis for the practical interpretation of existing analytical results.

ACKNOWLEDGMENTS

The authors thank the anonymous referees for their helpful comments, which improved both the presentation and accuracy of our remarks.

Research supported in part by the National Science Foundation under grants CCR-8810388, CCR-9111892, and CCR-9311702.
REFERENCES


On “A Framework for Source Code Search Using Program Patterns”

Prem Devanbu

Abstract—The need to query and understand source code is an important practical problem for software engineers in large development projects. A recent paper by S. Paul and A. Prakash in this journal [1] proposes a workable solution to this problem. However, there are several previously reported systems [2], [3], [4], [5], [6], [7], [8], [9], [10] not cited by [1], that can also address this problem. The relationship of [1] to the body of existing work is the subject of this short paper.

I. INTRODUCTION

A recent paper in this journal addresses an important problem: Software engineers who are maintaining code are needed to search through large amounts of source code to locate needed information” [1]. As the authors point out, there is a need to provide programmers with a flexible, customizable tool that can conduct efficient searches through large bodies of code. They correctly point out many of the difficulties in applying existing tools to this problem. Thus, string-based pattern-matching systems such as grep [11] cannot handle complex multilime program constructs such as Switch statements in C. Cross-referencing tools such as CIA [12] extract predetermined information about global program references, and cannot be customized by programmers to extract specific information from syntactic program structures. To provide flexible querying of source code, one needs a tool framework that incorporates a front-end (performing lexical analysis, parsing, type checking, etc.) that builds a complete internal representation of the program, and a customizable back end which can be programmed to execute to specific traversals of this internal representation to extract the desired information. Thus, transformational systems such as REFINEd [13] have a complete parser that builds an internal representation, and a pattern language that can be used to create particular traversals thereof. REFINEd can thus address the issues raised by [1]. Paul and Prakash describe an improved pattern language and an implementation technique that is built upon and enhances REFINEd.

However, Paul and Prakash [1] do not cite, (nor discuss the relationship of their work to) some existing work that addresses the above issues; this relationship is the subject of this short paper.

II. DISCUSSION

To begin with, several modern compiler construction tools, such as the Pan system [2], CENTAUR [3], METATOOL [4], Gaindalf [6], POPART [8], [9], and the Cornell Synthesizer Generator [10] incorporate both a parser-generator system and enhanced methods for performing various kinds of semantic analysis by tree traversal. The problem of source code search raised by Paul and Prakash can be fully addressed in these systems. First, one implements a parser in these systems and then implements the source code search task in the respective tree traversal language (for example, in the Gaindalf [6], AWK [11] scripts in A* [7] or VTP in CENTAUR [3]). Indeed, Paul and Prakash use a similar technique, relying on the parse trees constructed by the REFINEd environment. In this context, their contribution is mainly the syntax of the pattern language that they propose, and their automaton-based implementation technique. Paul and Prakash describe a query language based on surface syntax, similar in style to that used by REFINEd. In addition to the types of patterns in the latter, their language allows pattern variables that can be bound to different elements of the subtree which matches a pattern. For example, a template like (IF (?C THEN ?T ELSE ?F) can be used to bind the variables ?C, ?T, and ?F to the condition, the true branch, and the false branch respectively of an IF statement. This allows the expression of more powerful queries than are possible with the base REFINEd pattern language, while retaining the surface-syntax based querying paradigm. The language described in the paper allows some types of condition expressions over pattern variables, such as equalities and set membership. It is not clear whether negated conditionals such as inequalities or set exclusions (universals) can be expressed. In fact, the expressive power of their pattern language is not explicit in the paper; however, it is apparent that SCRUPLE [1] is not suitable for applications involving the gathering and analysis of non-local information. Thus, tracing of control and data flow (such as the derivation of control flow and control dependence, or static slicing) would appear to be difficult to encode in a surface-syntax based pattern language. SCRUPLE does, however, make a useful distinction between sets and ordered lists (e.g., the set of local variables in a procedure, vs. its list of arguments).

Paul and Prakash report performance comparisons of their SCRUPLE system to the grep tool. However, the paper does not present performance comparisons with other tools which address the source code querying problem. In this sense, comparison with tools such as

The author is with the Software and Systems Research Laboratory, AT&T Bell Laboratories, Murray Hill, NJ 07974; e-mail: prem@research.att.com.

To order reprints of this article, e-mail: transactions@computer.org, and reference IEECS Log Number S95053.

0988-5589/95$04.00 © 1995 IEEE