A Comparison of Some Structural Testing Strategies

SIMEON C. NTAFOS, MEMBER, IEEE

Abstract—**In this paper we compare a number of structural testing strategies in terms of their relative coverage of the program's structure and also in terms of the number of test cases needed to satisfy each strategy. We also discuss some of the deficiencies of such comparisons.**

Index Terms—**Data flow, program testing, structural testing.**

I. Introduction

STRUCTURAL testing [7] is probably the most widely used class of program testing strategies. These strategies use the control structure of the program as the basis for developing test cases as opposed to alternative classes of strategies that emphasize the specifications (black-box testing), specific types of errors, or combinations thereof. The popularity of structural testing strategies is mainly due to their simplicity and the resulting availability of software tools to assist with them.

The main shortcomings of structural testing strategies result from their dependence on the control structure of the program. An obvious problem is that the control structure itself may be incorrect. This makes it difficult, if not impossible, to detect errors in the specifications that are not reflected in the program structure. Another problem is that most structural strategies do not provide any guidelines for selecting test data from within a path domain and many errors along a path can be detected only if the path is executed with values from a small subset of its domain.

Despite the shortcomings, interest in structural testing continues unabated with more and more strategies been proposed and studied. Lagging are evaluations of the effectiveness of these strategies and comparisons of their relative power. This is due to the lack of generally accepted models for measuring testing effectiveness and cost. An indication of the relative power of the strategies can be obtained by ordering strategies according to the relation "strategy A includes (subsumes) strategy B." Such comparison are reported in [2,] [13], [14], [16] for some strategies based on data flow analysis. A similar comparison that includes some strategies based on testing expressions is reported in [15]. The fact that a strategy A includes strategy B does not mean that strategy A is better than strategy B since cost is not considered. An estimate of the relative cost of the strategies can be obtained by determining the number of test cases needed to satisfy the requirements of each strategy. In this paper we compare a number of structural strategies in terms of inclusion and in terms of the worst case complexity of the test sets required by each strategy. We also discuss some of the limitations of such comparisons.

Manuscript received November 29, 1985. This work was supported in part by the National Science Foundation under Grant MCS-8202593.

The author is with the Computer Science Program, University of Texas at Dallas, Richardson, TX 75080.

IEEE Log Number 8820975.

II. The Testing Strategies

Best known among the structural testing strategies are segment (statement), branch, and path testing [7]. *Segment testing* requires each statement in the program to be executed by at least one test case. *Branch testing* asks that each transfer of control (branch) in the program is exercised by at least one test case and is usually considered to be a minimal testing requirement. *Path testing* requires that all execution parts in a program are tested but is impractical since even small programs can have a huge (possibly infinite) number of paths. Most of the other structural testing strategies fill the gap between branch and path testing. They include structured path testing [6], boundary-interior path testing [5], strategies based on LCSAJ's (linear code sequence and jump [17]) and strategies based on data flow analysis [4], [8]–[10], [12], [14].

Structured path testing and *boundary-interior path testing* are restricted versions of path testing in which the number of test cases is limited by grouping together paths that differ only in the number of times that they iterate loops and then testing a few representative paths from each such group. In boundary-interior testing we consider two classes of paths from each group of similar paths with respect to each loop. Paths in the first class enter the loop but do not iterate it (boundary tests) while paths in the second class iterate the loop at least once (interior tests). Among the boundary tests we perform those that follow different paths inside the loop. Among the interior tests we perform those that follow different paths through the first iteration of the loop. For example, consider a WHILE-DO loop that contains a single IF-THEN-ELSE. There are two boundary tests for this loop (one for each branch of the IF-THEN-ELSE) and both will exit the loop immediately. The number of interior tests is four and all of them will execute the body of the loop a second time. The four interior tests correspond to the four permutations of branches in the first two executions of the IF-THEN-ELSE (True-True, True-False, False-True, False-False). After the second execution of the body of the loop, each interior test path may exit the loop or iterate it any additional number of times taking either one of the branches in the IF-THEN-ELSE.

0098-5589/88/0600-0868$01.00 © 1988 IEEE

In structured path testing, the representative paths that are selected from each group of similar paths are those that do not iterate a loop more than k times, where k is usually a small integer. Both structured path testing and boundary-interior path testing are based on the programming notion of a loop which may be open to interpretation in arbitrary control flow graphs. For the purposes of this discussion we will use the graph theoretic notion of a loop in defining structured path testing, i.e., *structured path testing* will test all paths P, where P does not contain any subpath p such that P consists of some subpath α, followed by more than k repetitions of p, followed by some subpath β. For boundary-interior path testing, we will keep the definition of the previous paragraph for well structured loops and assume that boundary-interior path testing is equivalent to structured path testing with $k = 1$ for all other loops. Note that, if a program does not contain any loops, both structured path testing and boundary-interior testing are equivalent to path testing.

LCSAJ's (linear code sequence and jump) are defined in terms of the program text. An LCSAJ is a sequence in consecutive statements in the program text, starting at an entry point or after a jump and terminating with a jump or at an exit point. In [17], a class of *test effectiveness ratios* (TER_n) is defined to each of which corresponds a testing strategy that asks that $\mathrm{TER}_n = 1$. TER_1, TER_2 are equal to 1 if segment and branch testing are achieved respectively. $\mathrm{TER}_{n+2} = 1$ if all subpaths containing up to n LCSAJ's are tested.

Strategies based on data flow analysis look at interactions involving definitions to program variables and subsequent references that are affected by these definitions and ask that certain such interactions be tested. A variable X is referenced in a segment if the first action involving X within the segment requires that a value for X be obtained. Variable X is defined in a segment if a value is assigned to it in the segment and that action is followed by zero or more references. A definition of X in segment i reaches a reference to X in segment j if there is a path from i to j along which the variable X is not redefined or undefined. For the purposes of defining data flow based strategies we assume that every segment contains a data flow action (e.g., constants are treated as variables defined at the start of the program).

The simplest type of data flow interaction involves a definition to a program variable and a reference reached by that definition. We call such an interaction a *2-dr interaction* [12]. The first strategy based on data flow was reported in [4] and amounts to testing each 2-dr interaction. The main shortcoming of this strategy is that it does not guarantee that branch testing is achieved. The *required pairs strategy* [10], [12] uses data flow analysis to construct a set of required pairs which are then covered by test cases. At least one required pair is produced for each 2-dr interaction. For each 2-dr interaction involving a reference in a branch predicate we produce one required pair for each outcome of the branch predicate. If the definition or the reference of the interaction occurs in a loop, two iteration counts for that loop are considered in producing the required elements, one specifying that the loop be exited at the first opportunity, while the other asks that the loop should be iterated some larger number of times.

In [14], a distinction is made between references to variables in a computation (c-use) and in a predicate (p-use). The p-uses are associated with branches corresponding to the outcomes of the predicate. Six strategies are proposed of which the "all uses" strategy is the central one. *All-uses* asks that all interactions between a c-use or a p-use and a definition that reaches it be tested. Four of the strategies are limited versions of "all uses" asking that: 1) each definition be tested with a path along which the definition reaches a c-use or p-use (*all-defs*), 2) each interaction between a p-use and a definition that reaches it be tested (*all-p-uses*), 3) each interaction between a c-use and a definition that reaches it be tested and each definition be tested by some path along which it reaches a use (*all-c-uses/some-p-uses*) and 4) each interaction between a p-use and a definition that reaches it be tested and each definition be tested by some path along which it reaches a use (*all-p-uses/some-c-uses*). The sixth strategy, *all-du-paths*, asks that each interaction between a p-use or a c-use and a definition that reaches it be tested along all cycle-free paths connecting the appropriate statements (including simple cycles from a definition to a reference that occurs in the same segment as the definition [14]).

In [8], two testing strategies are proposed. The first strategy is the same as that proposed in [4], i.e., it requires that all 2-dr interactions are tested. The second strategy requires that each elementary data context of every instruction be tested at least once. An *elementary data context* of an instruction is a complete set of definitions for the variables referenced in the statement such that the definitions reach the statement. A more extensive strategy called *ordered data contexts* is also mentioned in [8]. It requires that the definitions in each elementary data context be visited in all possible orders.

A number of other strategies based on data flow are proposed in [9] and [12]. The *required k-tuples* strategies [12] are extensions of the required pairs strategy. They ask that all sequences of $k - 1$ (or less) related 2-dr interactions be tested. In a sequence $r_1, r_2, \cdots, r_{k-1}$ of 2-dr interactions, the ith 2-dr interaction is related to the $(i + 1)$st 2-dr interaction if the reference in the ith interaction is used directly in the definition associated with the $(i + 1)$st interaction, $0 < i < k - 1$. The first definition and the last reference in the sequence are treated the same way as in the required pairs strategy. By varying k we get a class of testing strategies with the property that required k-tuples subsumes required m-tuples for any $m < k$. This definition clarifies the definition of the required k-tuples strategy in [12]. There, it is stated erroneously that the k data flow actions must occur in distinct segments. It is clear from the rest of the discussion that the word "dis-

tinct'' should not appear in the definition (e.g., it contradicts the definition of required pairs, and it is a very unnatural restriction to place on the k-tuples).

A strategy called *definition-tree testing* is proposed in [9]. A subset of the program variables is selected and then the data flow for them is traced back from the output through a series of definitions until the beginning of the program or a cyclic use is reached. The required k-tuples strategies and definition-tree testing allow us to follow a sequence of computations related by data flow and, as pointed out in [9], [12], they may provide a connection to the program specification and other information so that more effective testing strategies can be formulated. Looking at required k-tuples as purely structural strategies it has been noted [11] that they quickly run into the law of diminishing returns as k increases. A similar phenomenon was reported for the $\text{TER}_n = 1$ strategies and as a result $\text{TER}_n = 1$ with $n > 4$ are rarely used.

III. Comparisons of the Strategies

Comparisons of testing strategies in terms of inclusion have been made since the early days of software development (e.g., branch testing subsumes segment testing). More recently, this type of comparison was used by Rapps and Weyuker to delineate the six data flow based strategies they introduced. Independently, in [2] and [13], these comparisons were extended to include other data flow based strategies. In this section we further extend the partial orderings of structural strategies reported in [2], [13], [14] to include structured and boundary-interior path testing and the $\text{TER}_n = 1$ strategies. The ordering is based on inclusion, i.e., a strategy X (strictly) includes strategy Y if any test set that satisfies X also satisfies Y and there is some test set that satisfies Y but not X [15]. Fig. 1 shows the resulting ordering for our set of structural strategies.

We start by pointing out three differences in the ordering of Fig. 1 from the orderings presented in [2], [15]. All of them have to do with changes to or different interpretations of the definitions of some data flow based strategies. First, in [15], the all-uses strategy is redefined by associating all the p-uses for a predicate with the segment in which the predicate occurs. The effect is that this version of all-uses does not include branch testing. The definition of p-uses in [14] is perfectly clear and no justification is given in [15] for changing it. The second difference has to do with the required k-tuples class of strategies which are extensions of the required pairs strategy. In [2], required k-tuples are defined so that they exclude testing of k-tuples that visit the same node more than once (based on the erroneous definition in [12]). The definition of required k-tuples was clarified in [13] and in the previous section and is much more natural than the version used in [2]. As far as Fig. 1 is concerned, the difference is that the proper version of required pairs includes all-uses while the version used in [2] does not. The third difference has to do with whether or not testing 2-dr interactions includes segment testing. This depends on how we handle data flow anomalies and segments in which

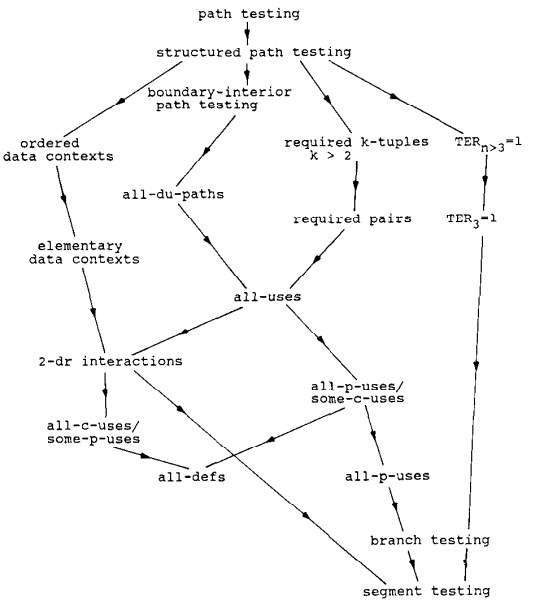


Fig. 1. Partial ordering of some structural testing strategies.

there is no data flow action (e.g., an output statement in which only constants are used). Both of these conditions can be easily detected by static data flow analysis and need not affect the definitions of data flow based strategies (e.g., in [15], many of the results are stated with the disclaimer ''except for constant references''). The approach we took in required pairs, was to assume that every segment contains a data flow action, that everything is defined at the start of the program and referenced at the end of the program. We feel that this is easily handled and fits better with the intent of the flow based strategies. Using versions that do not include segment testing begs the introduction of versions that do, and serves no purpose except the dubious one of increasing the number of data flow based strategies.

In the following lemmas we establish the inclusion relations for structured path testing, boundary-interior path testing and the $\text{TER}_n = 1$ class of strategies. These strategies were not considered in the comparisons reported in [2], [14], [15].

Lemma 1: Structured path testing with $k = 4$ includes boundary-interior, required pairs, $\text{TER}_3 = 1$, and ordered data contexts.

Proof: In structured path testing ($k = 4$), each loop will be iterated up to four times. Thus, the requirements of boundary-interior path testing will be satisfied while the converse is not true.

In the required pairs strategy, the treatment of loops depends on the patterns of definitions and references in the loop. The most extensive testing of a loop occurs in situations where a definition within the loop reaches a reference that precedes it within the body of the loop. Consider the partial flowchart shown in Fig. 2. The set of required pairs of this code will include the following:

	SEG	SEG	VAR	DEF	REF	
1.	1	2	X		Py-Le	Py: Predicate is true
2.	1	2	X		Pn-Le	Pn: Predicate is false
3.	1	2	X		Pn-Lr	Le: Exit Loop
4.	3	2	X	Lr	Pn-Le	Lr: Repeat Loop
5.	3	2	X	Lr	Py-Lr	

These require that the loop be iterated 0, 1, 2, 3, and 4 times, respectively. Since structured path testing with $k = 4$ will also iterate the loop up to four times, it follows that it includes required pairs testing. Required pairs does not include structured path testing for any k since it can be satisfied without testing all paths from some segment i to another segment j.

According to [3], to satisfy $TER_3 = 1$ each loop must be iterated at least once and also three or more times. TER_3 can be achieved by iterating each loop up to three times. Thus, structured path testing ($k = 4$) includes TER_3 while the converse is not true for any k.

The order data contexts strategy does not include structured path testing (for any k) since it does not always test all paths from a definition to a reference. Structured path testing includes the ordered data contexts strategy since the latter never needs to iterate a subpath. Note that in code like the one shown in Fig. 3, elementary data contexts requires that we iterate the loop (in the programming sense) but can be satisfied by taking different paths through the loop, i.e., we do not need to iterate a subpath. Q.E.D.

In both the required n-tuples ($n > 2$) and $TER_n = 1$ ($n > 3$) strategies, some loop may be iterated more than 4 times depending on whether or not a 2-dr interaction or an LCSAJ is allowed to appear more than once in a sequence. Still, structured path testing with appropriate k will include both required n-tuples and $TER_n = 1$ while the converse is not true for any k (even with $k = 0$) since they will not test all paths in a program without loops.

Lemma 2: Boundary-interior path testing includes the "all-du-paths" strategy.

Proof: The all-du-paths strategy never requires that a loop be interated more than one time. Thus, boundary-interior path testing includes all-du-paths. The converse is not true since all-du-paths requires only that loop free paths from a definition to a reference be tested while boundary-interior path testing will also interate any intervening loops. Q.E.D.

Lemma 3: Boundary-interior path testing is incomparable with the data contexts strategies, the required k-tuples strategies and the $TER_k = 1$ strategies.

Proof: The data contexts, required k-tuples, and $TER_k = 1$ strategies do not include boundary-interior path testing since, in a program without loops, they do not need to test all paths connecting two segments. Conversely, boundary-interior path testing does not include required pairs and $TER_3 = 1$ because some loop may need to be iterated three or more times. Also, it does not include the elementary data contexts strategy as shown in Fig. 3 where elementary data contexts requires that a path like 1-2-3-4-5 be tested which is not necessary in order to satisfy boundary-interior path testing. Q.E.D.

Lemma 4: Required k-tuples, the data contexts strategies, all-defs and all-p-uses are incomparable with $TER_k = 1$ for any $k > 2$.

Proof: Consider the code segments shown in Figs. 4 and 5. In Fig. 4, there are two alternation structures forming a total of four paths. The set of LCSAJ's for this code is: {1-2 → 5, 1-2-3-4 → 6, 5-6 → 9, 5-6-7-8 → 10, 6 → 9, 6-7-8 → 10, 9-10}. To cover this set, we must use the path 1-2-5-6-9-10. However, note that there is no data flow interaction between statements 5 and 9 and therefore none of the data flow based strategies needs to select this path.

Consider the code segment shown in Fig. 5. This code contains the following set of LCSAJ's: {1-2 → 7, 1-2-3-4 → 7, 1-2-3-4-5-6 → 10, 7-8 → 10, 7-8-9-10}. These LCSAJ's can be covered with the paths 1-2-7-8-9-10, 1-2-3-4-7-8-10, and 1-2-3-4-5-6-10. Note that there is a reference to variable B in statement 9 and B is defined in statement 3. Thus, all-defs requires that a path through both 3 and 9 be used and it follows that $TER_3 = 1$ does not include all-defs. Consider again the code shown in Fig. 5 and replace $X3$ with the test $B > 0$. Then, the set of test paths that satisfy $TER_3 = 1$ will fail to test the interaction between the definition of B in statement 3 and the p-use associated with the "false" branch of the predicate $B > 0$ in statement 8. Thus, $TER_3 = 1$ does not include all-p-uses.

Note that $TER_4 = 1$ will require a path through statements 3 and 9 in the code segment of Fig. 5 in order to cover the pair consisting of the LCSAJ's 1-2-3-4 → 7 and 7-8-9-10. However, one can easily extend this example so that, given any n, $TER_n = 1$ can be achieved without satifying all-defs or all-p-uses. Q.E.D.

Proofs for the remaining relations shown in Fig. 1, appear in [2], [13], [14], or follow from the above lemmas and the definitions. The definition-tree strategy [9], is not included in Fig. 1 because the inclusion relations for it depend on which variables are traced back. Technically, it is incomparable to almost all the strategies in Fig. 1. It bears the most resemblance to the required k-tuples strategy with a sufficiently high k.

A serious shortcoming of comparisons in terms of inclusion is that the cost of the various strategies is not accounted for. Factors contributing to the cost of testing are the cost of generating a test set, the cost of running the test cases and the cost of checking the outputs. A common contributor to all these factors is the number of test cases needed to satisfy each strategy. We can use the number of test cases needed as a crude measure of cost. It would

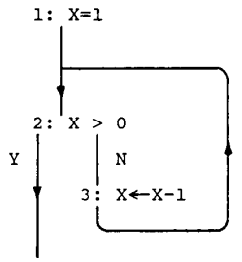


Fig. 2. Pattern resulting in the most extensive testing of a loop by required pairs testing.

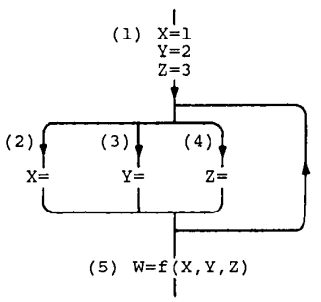


Fig. 3. Elementary data contexts requires that a path like 1-2-3-4-5 be tested. Boundary-interior path testing does not need to include such a path.

```
 1.        READ(X,Y)
 2.        IF X GOTO 10
 3.        Z=1
 4.        GOTO 20
 5.   10   W=1
 6.   20   IF Y GOTO 30
 7.        Z=W*Z
 8.        GOTO 40
 9.   30   Z=Z-2
10.   40   END
```

Fig. 4. This code segment has a total of four paths all of which are needed to satisfy $TER_3 = 1$. There is no data flow interaction along the path through statements 5 and 9 and this path is not needed to satisfy data flow based strategies.

```
 1.        A=1
 2.        IF X1 GOTO 40
 3.        B=1
 4.        IF X2 GOTO 40
 5.        A=2
 6.        GOTO 50
 7.   40   A=A+B
 8.        IF X3 GOTO 50          {IF B > 0 GOTO 50 }
 9.        B=B+1
10.   50   WRITE(A,B)
```

Fig. 5. This code contains five LCSAJ's that can be covered with three paths none of which covers the data flow interaction between statements 3 and 9 {or statements 3 and 8 with $B <= 0$}.

be more appropriate to make this comparison in terms of the "average" number of test cases needed. However, establishing what the average number of test cases needed for each strategy requires extensive statistical data on the control and data flow in real programs and no such data are available. Still, it is easy to determine the number of test cases needed in the worst case for each strategy and this does give us a good indication of the relative cost of the strategies.

Lemma 5: Let n be the number of segments in a program. Then in the worst case, we have that:

1) Path testing may require an infinite number of test paths.

2) Structured path testing (for any k), boundary-interior path testing, all-du-paths may require a number of test cases that is an exponential function of n.

3) Required pairs, $TER_3 = 1$, all-uses, the data contexts strategies, 2-dr interactions, all p-uses/some c-uses, all c-uses/some p-uses, all p-uses each may require $O(n^2)$ test paths.

4) All-defs, branch and segment testing may require $O(n)$ test paths.

Proof: Most of these bounds have been established or implied by the authors that proposed the various strategies. Worst cases for the data flow strategies proposed in [14] are reported in [16]. Claim 1) follows from the existence of programs that do not halt. For 2), consider a program consisting of a sequence of n IF-THEN-ELSE structures, each of which defines and references a variable X. Then, structured and boundary-interior path testing as well as all-du-paths may require 2^n test paths.

Required pairs involves the selection of up to 2 out of n segments in order to form a 2-dr interaction. Let m be the maximum number of variables that are referenced in any segment. Then, we can have $O(m * n^2)$ required pairs. Since it is common practice to assume that m is bounded by a constant, required pairs never needs more than $O(n^2)$ test paths. A program with a sequence of n IF-THEN-ELSE structures and a variable X that is defined and referenced in each one of them achieves this upper bound. Ordered data contexts will usually require more test paths in order to test the up to $m!$ orders in which m variables that are referenced in a segment can appear in the data context. However, if m is bounded by a constant (which is normally the case), the number of test paths needed is no worse than $O(n^2)$. $TER_3 = 1$ requires that each LCSAJ is tested. A program can have $O(n^2)$ LCSAJ's and $O(n^2)$ test paths may be required to cover them (e.g., consider a sequence of IF COND GOTO statements where each such statement is also a target of a GOTO).

All-defs and segment testing may require $O(n)$ test cases (e.g., consider a long sequence of nested IF-THEN-ELSE's). The bound for branch testing follows from the fact that the number of branches is $O(n)$ since the out-degree of each vertex is the control flow graph of a program is normally bounded by a small constant (one can construct control flow graphs with $O(n^2)$ branches but that would involve compound statements that should be treated as sequences of simpler statements). Q.E.D.

It should be noted that, while programs can be constructed that achieve these worst case bounds, in practice the number of test cases needed usually is considerably less than what is implied by the bounds.

IV. Conclusions

We presented a comparison of a number of structural testing strategies in terms of the relation "strategy A includes strategy B" and in terms of the number of test cases needed in the worst case by each strategy. It turns out that the comparison in terms of the number of test cases needed in the worst case provides a much more meaningful grouping of the various strategies than Fig. 1 does. The most interesting group is the one including the strategies that may require $O(n^2)$ test paths as most of them offer needed improvement over branch testing and their cost remains reasonable.

The comparison in terms of inclusion is useful but has a number of weaknesses. First, as can be seen from Fig. 1, many of the strategies are incomparable. Usually this is due to the details in the definitions of the strategies rather than a reflection of different approaches to structural testing. As a result, Fig. 1 tends to emphasize trivial differences rather than common approaches. Also, slight modifications in the definitions of the various strategies can alter the inclusion relations. As seen in the previous section, the required n-tuples and the $\text{TER}_n = 1$ strategies can easily be made incomparable with structured path testing (with $k = 4$). If we define boundary-interior testing so that it does not test all combinations of branches in the first two executions of the body of a loop, then we have that boundary-interior path testing is incomparable with all-du-paths and most of the other data flow based strategies. Also, if we use the programming notion of a loop in defining structured path testing, then it becomes incomparable with most of the data flow based strategies.

Some of the strategies are properly defined to allow various choices and the particular choices that are selected can alter the inclusion relations. As an example, consider the treatment of arrays in data flow based strategies. In our discussion we have assumed that all elements of an array are treated as occurrences of the same variable (since the data flow interactions between distinct elements can not be determined with static analysis). If we consider the elements of an array as distinct variables (using run-time instrumentation to verify coverage) then the data flow based strategies become incomparable with structured path testing (for any k) since they may require that a loop be iterated n times (where n is the size of the array).

Another problem with comparisons in terms of inclusion is that even when we can determine that one strategy includes another, we have no quantitative measure of the difference between the two strategies (i.e., how much better is path testing than structured path testing?). It is not at all clear that adopting a more extensive strategy is preferable to using a combination of simpler strategies or extending a simpler strategy. For example, consider testing an IF-THEN structure using branch and segment testing. To achieve branch testing we need to use two paths while segment testing can be achieved with just one path. Assuming that test data are selected in a similar fashion from the path domains, we can claim that branch testing will be more effective than segment testing. However, if we use two independent test cases for segment testing, it follows that segment testing will be more effective than branch testing in detecting errors within the body of the THEN branch. This points out a major deficiency that all structural testing strategies share. Many errors along a path can only be detected if the path is executed with values from some subset of its subdomain. Purely structural testing strategies provide no guidelines for selecting the actual values with which to execute a test path. Then, it may well be that effectiveness increases by testing a smaller set of test paths with more inputs as compared to testing a larger subset of paths with one input per test path.

Another way in which the various strategies can be evaluated and compared is to determine what types of errors they are effective in detecting and types of errors for which they are ineffective. Some results along these lines are reported in [3], [8], [11]. Experiments with the portable mutation system [1] reported in [11] show that the main weakness of required pairs is in detecting errors having to do with small shifts in domain boundaries and the handling of special values. These are types of errors for which all the structural strategies are relatively ineffective. It should be noted that strategies that totally disregard the program structure can be equally ineffective for other types of errors (e.g., consider errors in code that is entered under conditions specific to a particular implementation of an algorithm but do not appear in the specifications). Thus, it is important that strategies that combine structural testing with other approaches to testing be used. This has been noted by many of the researchers in the field. For example, data flow strategies were proposed in [9], [12] as a basis for combining structural with black box and error driven strategies and in [3], it is reported that the $\text{TER}_3 = 1$ strategy is used after applying functional testing.

References

[1] T. A. Budd, "The portable mutation testing suite," Univ. Arizona, Tech. Rep. TR83-8, Mar. 1983.

[2] L. A. Clarke, A. Podgurski, D. Richardson, and S. Zeil, "A comparison of data flow path selection criteria," in *Proc. 8th ICSE*, Aug. 1985, pp. 244-251.

[3] M. Hennel, D. Hedley, and I. J. Riddell, "Assessing a class of software tools," in *Proc. 7th Int. Conf. Software Engineering*, Mar. 1984, pp. 166-277.

[4] P. M. Herman, "A data flow analysis approach to program testing," *Australian Comput. J.*, vol. 8, no. 3, pp. 92-96, Nov. 1976.

[5] W. E. Howden, "Methodology for the generation of program test data," *IEEE Trans. Comput.*, vol. C-24, no. 5, pp. 554-559, May 1975.

[6] ——, "Symbolic testing-design techniques, costs and effectiveness," NTIS PB-268518, May 1977.

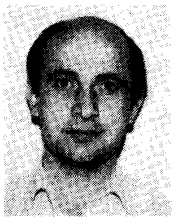
[7] J. C. Huang, "An approach to program testing," *ACM Comput. Surveys*, vol. 7, no. 3, pp. 114-128, Sept. 1975.

[8] J. Laski and B. Korel, "A data flow oriented program testing strategy," *IEEE Trans. Software Eng.*, vol. SE-9, no. 3, pp. 347-354, May 1983.

[9] J. Laski, "On data flow guided program testing," *SIGPLAN Notices*, vol. 17, pp. 62-71, Sept. 1982.

[10] S. Ntafos, "On testing with required elements," in *Proc. COMPSAC-81*, Nov. 1981, pp. 142-149.

[11] ——, "An evaluation of required element testing strategies," in *Proc. 7th Int. Conf. Software Engineering*, Mar. 1984, pp. 250–256.
[12] ——, "On required element testing," *IEEE Trans. Software Eng.*, vol. 10, no. 6, pp. 795–803, Nov. 1984.
[13] ——, "A comparison of some structural testing strategies," in *Proc. 19th Hawaii Int. Conf. System Sciences*, Jan. 1986, pp. 803–811.
[14] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Trans. Software Eng.*, vol. SE-11, no. 4, pp. 367–375, Apr. 1985.
[15] M. D. Weiser, J. D. Gannon, and P. R. McMullin, "Comparison of structured test coverage metrics," *IEEE Software*, vol. 2, no. 2, pp. 80–85, Mar. 1985.
[16] E. J. Weyuker, "The complexity of data flow criteria for test data selection," *Information Processing Lett.*, vol. 19, pp. 103–109, Aug. 1984.
[17] 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.



Simeon C. Ntafos (S'74–M'78) received the B.S. degree in electrical engineering from Wilkes College, Wilkes-Barre, PA, in 1974, and the M.S. degree in electrical engineering and the Ph.D. degree in computer science from Northwestern University, Evanston, IL, in 1976 and 1979, respectively.

He was a Visiting Assistant Professor in the Department of Electrical Engineering and Computer Science at Northwestern University during 1978–1979. He joined the faculty of the University of Texas at Dallas in 1979, where he is now an Associate Professor. He was Head of the Computer Science Program at UTD during 1985–1987. His current research interests include software reliability, computational geometry, and parallel algorithms.