Adequate Test Sets for Loop Testing

by

M.Z. Rupp, J.A. Makowsky and I. Pomeranz

CIS Report #9218
November 1992
Adequate Test Sets for Loop Testing

M.Z. Rupp*, J.A. Makowsky** and I. Pomeranz***.

* Department of Electrical Engineering
Technion, Technion City, Haifa 32000, Israel

**Dept. of Computer Science
Technion, Technion City, Haifa 32000, Israel

*** Department of Electrical Engineering
University of Iowa

Abstract

In this paper we combine results from different contexts of program testing. We first introduce a new data structure, the *PNI*, which describes the control flow between iterations of an innermost loop. This extends the method developed in [14] for loop-free intervals. To evaluate the merits of the *PNI* we discuss it in the formal framework of adequate test sets as developed in [7]. Essential in this framework is the notion of a suitable distance function between programs. In our case, a modification of the branch function introduced in [12] fits both the underlying intuition and the formal model's requirements. Our main result now shows how to use the *PNI* to construct adequate test sets for iterations of innermost loops. The same method allows us also to identify adequate test sets for testing the overall branching behavior of programs and to identify cases in which one can find an adequate test set for all branches. Implementations of our method can be used for acceptability testing and fault tolerant computing.

KEYWORDS Acceptability Testing, Control-Flow, Adequate Test Sets
Contents

1 Introduction .................................................. 2

2 Possible Next Iterations (PNI) .............................. 3

3 Adequate Test Set ............................................. 5
   3.1 Davis and Weyuker's Formal Model ................. 6
   3.2 Limitations of the Formal Model .................. 8

4 Branch Functions ............................................. 9
   4.1 Automatic Test Data Generation ................... 9
   4.2 Modified Branch Function .......................... 11
   4.3 A Distance Function for Branching .............. 11

5 Adequate Test Sets for Branching .......................... 14
   5.1 Adequate Test Set for Conditional Statements .... 14
   5.2 Adequate Test Set for ALL BRANCHES .......... 19
   5.3 Adequate Test Set for Loops ..................... 21
   5.4 Revealing More Errors by ALL BRANCHES ....... 22

6 Conclusions and Further Research ......................... 25
1 Introduction

Program testing is used in many contexts: test set selection criteria, functional testing, flow analysis, automatic test data generators, adequacy of test sets and coverage metrics, are just some. Many papers that present the different approaches taken within each context and compare between them, have appeared (e.g. [6]).

In the research reported here, we investigate the possible applicability of results that were achieved in one context, to other different contexts. As a focal point of our approach, we study the flow of control from one iteration to another inside a loop. Based on an approach that partitions the control flow graph into loop-free intervals [14], we define a matrix to represent the possible flow of control from one iteration to another inside an innermost loop. This matrix, called PNI, was first introduced in [13], where its properties are discussed.

To evaluate the usefulness of PNI, we study it in the context of adequate test sets, as introduced in [7]. There, a formal model of the quality of programs testing is developed. In order to be able to study PNI in their formal model, we have to identify the adequate test set for our case. Essential to this formal model is the definition of a distance function that suits the intuition of the tester of the program, about the "most likely to occur" errors in the program. However, the obvious distance function suggested by [7] does not capture the intuition behind PNI. In [12], a notion of branch function was introduced in the context of test data generators. A modification of this branch function allows us to use it as the missing link between our purpose – the analysis of PNI – and the formal model of [7].

The results of this research enabled us to identify an adequate test set to the particular case we were interested in. But, moreover, we also obtained new results that are applicable to the different contexts that were used. This includes the construction of an adequate test set for conditional statements; identification of the cases where it is feasible to have an adequate test set for the "ALL BRANCHES" testing criteria, and if the test set exists – its construction; and a method to reveal more errors by the "ALL BRANCHES" test set. We restrict our intention in this paper to conditional statements with one variable only.

Our main results are:

- The introduction of PNI as a data structure which captures the control flow behavior of a program during the execution of a loop;
- The formal specification and analysis of PNI and of branching behavior in general, using the approach of [7];
- A distance metric suitable for control flow testing;
- Adequate test sets for control flow in innermost loops, and for branching statements.
The paper is organized as follows: The next section introduces the background to control flow analysis, that is required for the definition of \textit{PNI}, gives \textit{PNI}'s definition and shows some of its properties. Section 3 gives the formal model that was introduced by Davis and Weyuker [7], for adequacy of test sets. It also discusses some drawbacks from this model, we propose to remedy. Section 4 deals with test data generators, focusing on a pathwise test data generator suggested by Karel [12]. We modify Karel's method to define a distance function that suits the formal model. Then we show how to put all these together. In Section 5 we present our main results and applications: We exhibit adequate test sets for conditional statements, the ALL BRANCHES testing criterion and for the execution of innermost loops, and use them for on-line testing. In section 6, we present our conclusions and suggestions for further research.

2 Possible Next Iterations (PNI)

We are interested in testing the flow of control in a loop from one iteration path to another. To set up our definition, we follow [14]. Given a program \( P \) which is a sequence of atomic statements, we use the following definitions:

(i) A \textit{block} of a program is a maximal set of ordered statements of the program that can only be executed as follows: its execution starts from the first statement, terminates at the last statement, and all its statements are executed in the given order.

(ii) The \textit{control flow graph} of a program is the graph in which each block is presented by a unique node, and for any two blocks \( x \) and \( y \), there is an arc \((x, y)\), if and only if control can potentially transfer from block \( x \) to block \( y \) at run-time. The notation \( G = (V, B) \) is used to describe the control flow graph. In this notion, \( V \) is the set of nodes in the graph and \( B \) is the set of directed arcs between these nodes.

(iii) An \textit{interval} is a collection of blocks with the property that the execution of these blocks always starts from the same block (called the header of the interval), and each of the other blocks in the interval is reachable only from blocks that are also in the same interval.

(iv) A \textit{loop-free interval} \( I_k = (V_k, B_k) \), \( k = 1, 2, \ldots, l \), of a program graph \( G = (V, B) \), is a maximal subgraph consisting of the subset \( V_k \) of \( V \) and the subset \( B_k \) of \( B \) of the program graph, such that the subgraph has a single entry and contains no loops (or no latches) [14].

In the loop-free intervals partitioning, arcs to the headers of the intervals, including latches, are not contained in any of the intervals. An algorithm for the identification of the loop-free intervals in a program, can be found in [14]. The limitation of this method is that it does not handle erroneous control flow on
the transition from one path through an interval, to another path of the same kind.

Yau & Chen [14] limited their discussion to paths of "loop-free" intervals, because the database holding the information required for handling longer paths, can result in a large storage overhead. However, the dependencies occurring between distinct paths of the same loop-free interval can be handled using a database of reasonable size. Formally, we define:

Definition 2.1 An iteration path is a path in a loop-free interval that begins at the header of the loop-free interval, terminates in a block from which an arc to the header of the loop (a latch) exits and contains no latches (the header of the interval is visited only once along the path).

The set of all iteration paths of a loop-free interval is the set of all the paths of the interval excluding paths that lead to the exit from the loop. To reflect the relations between the iteration paths, we define a matrix which we call PNI (Possible Next Iterations) in the following manner:

Definition 2.2 Let \( L = (l_1, l_2, \ldots, l_n) \) be an enumeration of the iteration paths of a loop-free interval. PNI (Possible Next Iterations) is the \( n \times n \) matrix, defined by: \( PNI(i, j) = \begin{cases} 1 & \text{if there is a possibility to execute iteration path } j \text{ immediately following the execution of iteration path } i, \\ 0 & \text{otherwise.} \end{cases} \)

\( PNI \) is in fact the adjacency matrix of the iteration paths of a loop-free interval. The process of deriving \( PNI \) can be automated using symbolic evaluation (e.g., SELECT [3], EFFIGY [11], ATTEST [5], DISSECT [8] and SMOTL [1]).

From the definition we see that the \( PNI \) matrix has the following two properties:

Property 2.1 A row in the \( PNI \) matrix has all elements equal zero, if and only if the appropriate path can be executed at most once during each execution of the loop, and only as the last iteration before an exit path from the loop is executed.

Property 2.2 A column in the \( PNI \) matrix has all elements equal zero, if and only if the appropriate path can be executed at most once during each execution of the loop, and only as the first iteration.

Other interesting properties of the loop can be obtained by considering powers of \( PNI \), when \( PNI \) is treated as a boolean matrix. Special properties of boolean matrices can be found in textbooks in that field (for example [10]). The application of powers of \( PNI \) to loops is discussed in [13], but is not needed in this paper.
Note that $PNI(i, j) = 1$ does not imply that if iteration path $i$ is executed then iteration path $j$ will be the next executed iteration path; it only implies that it might be iteration path $j$.

If in some actual execution of the program iteration paths are being executed in an order that contradicts $PNI$, we speak of a control flow error. That is:

**Definition 2.3** Let $i$ and $j$ be any two iteration paths for which $PNI(i, j) = 0$. Then, if at any time during the execution of the program, iteration path $j$ is being executed immediately following iteration path $i$, then, there exists a control flow error between the iterations of the loop.

If $PNI$ is derived from the specification of the program, than the occurrence of a control flow error between iteration paths indicates wrong implementation of the requirements. If $PNI$ is derived based on the program itself, than a control flow error between iteration paths indicates a problem in the environment in which the program is run. Such problems might occur, for example, as a result of a hardware malfunction, memory ionization in space, virus attack and such. This suggests the use of our results for acceptability testing and fault tolerant computing.

In the loop-free interval partitioning all iteration paths start from the same node. Hence, the point where the control flow error might occur, is a node in which iteration paths split from each other (where, for any two iteration paths in our model, such a node exists). $PNI$ can help in identifying these statements. That is:

- For any iteration path $i$, the zero elements in the $i$'s row of $PNI$ represent iteration paths that cannot be executed immediately following iteration path $i$. The non-zero elements in that row represent those iteration paths that can.

- For any iteration path $j$, the zero elements in the $j$'s column of $PNI$ represent iteration paths immediately after which iteration path $j$ cannot be executed. The non-zero elements in that column represent those iteration paths immediately after which it can.

Iteration paths split from each other at conditional statements. This leads us to look for an adequate test set for conditional statements. Specifically, we are interested in conditional statements in which iteration paths are separated from each other, and where $PNI$ contains meaningful data, as observed above.

## 3 Adequate Test Set

Davis and Weyuker [?] suggest a formal model for the notion of an adequate test set. Using this formal model we want to construct an adequate test set for the detection of control flow errors between iteration paths.
3.1 Davis and Weyuker’s Formal Model

The approach to the construction of adequate test sets as presented in [7], can be viewed as a generalization of the mutation analysis method [2], [4], [9]. Given a finite set \( \Phi(P) \) of programs, each of which is not equivalent to \( P \), a set of test data \( T \) is said to be \( \Phi \)-adequate for \( P \) if

\[
(\forall Q \in \Phi(P))(\exists t \in T)(P(t) \neq Q(t)).
\]

\( \Phi(P) \) serves as an approximation to the set of all programs inequivalent to \( P \). Davis and Weyuker introduce a method to the derivation of a set of inequivalent programs \( \Phi(P) \), which is appropriate to the intuition of the tester about errors that might reside in the program. The idea is to translate this intuition into terms of a distance between programs. The program having the most likely error will be the closest to \( P \). The less likely the error is, the bigger is the distance of a program having this error from \( P \). With this, the tester of the program can construct \( \Phi(P) \) by considering alternative programs to \( P \), which are at a distance appropriate to his intuition about the likeliness of errors in \( P \).

For this purpose, [7] define the following:

**Definition 3.1**

(i) A distance function is a real valued function \( \rho \) on pairs of programs, such that for all programs \( P, Q \) and \( R \):

1. \( \rho(P, Q) \geq 0 \);
2. \( \rho(P, Q) = 0 \) if and only if \( P = Q \) (\( P \) and \( Q \) are syntactically identical);
3. \( \rho(P, Q) = \rho(Q, P) \);
4. \( \rho(P, R) \leq \rho(P, Q) + \rho(Q, R) \).

(ii) If for each \( P \) and each \( d > 0 \) the number of programs \( Q \) for which \( \rho(P, Q) \leq d \) is finite, then \( \rho \) is called a finite distance function.

(iii) If \( \rho(P, Q) \) is always an integer, we say that \( \rho \) is a discrete distance function.

The distance was defined to be zero between syntactically identical programs, and not semantically equivalent ones, because the check for the latter might be an undecidable problem. Because of this, they base their model on the grammar of the program. In order to have an approximation to the set of all inequivalent programs, reduction rules are defined. Alternative programs are produced by activating the reduction rules on \( P \). In order to eliminate equivalent programs from the set of alternatives, they introduce the term embedded program:

**Definition 3.2**

(i) If a program \( N \) can be obtained from a program \( M \) by zero or more applications of the reduction rules, we say that \( M \) reduces to \( N \).
(ii) $M$ is embedded in $N$ if $N$ reduces to some program which is equivalent to $M$.

(iii) A mapping $\Phi$ is called protected for a program $P$ if $\Phi(P)$ contains no program in which $P$ is embedded.

With these definitions, they define in [7] the set of alternative programs and the test set to distinguish them from $P$ in terms of a distance. More precisely, they define:

**Definition 3.3** Let $\rho$ be a given finite distance function. Then for each $d \geq 0$, $\Phi_d(P)$ is the set of all programs $Q$ such that:

1. $\rho(P, Q) \leq d$;
2. $P$ is not embedded in $Q$.

If a fixed distance function is used, we shall say that a set $T$ is $d$-adequate, meaning that it is $\Phi_d$-adequate.

Now that we have the definition of $d$-adequacy, we can also talk about the points that must be included in the test set as a function of the distance. This is done by the definition of critical points.

**Definition 3.4** A point $c$ is called $\Phi$-critical for a program $P$ if there exists a program $Q \in \Phi(P)$ such that $P(t) = Q(t)$ for all $t \neq c$, but $P(c) \neq Q(c)$.

Beyond the critical points, there might be some other points, which might be required when coming to distinguish $P$ from its alternatives. Such points might rise when $P$ differs from an alternative program in more than one point. In this case, one has to select a representative from each such set of points. In order to minimize the testing activity, this selection should be made such that the resulted test set is minimal. In the terms of distance, their definition for this set is:

**Definition 3.5** A set $T$ is minimally $\Phi$-adequate for a program $P$ if:

1. $T$ is $\Phi$-adequate for $P$;
2. if $S \subseteq T$ and $S$ is $\Phi$-adequate for $P$, then $S = T$.

We say that a set $T$ is minimally $d$-adequate to a program $P$, if it is minimally $\Phi_d$-adequate to it.

Thus, the actions one has to take when coming to test a program, are:

1. Translate his intuition of possible errors in the program to terms of distance;
2. Determine the critical points for the distance up to which he considers alternative programs;

3. Derive the minimal adequate test set for this distance.

The results of the execution of the program on this test set, will indicate whether the type of errors he considered as possible, reside in the program, or not.

3.2 Limitations of the Formal Model

Before we come to define a distance function appropriate to our case, let us make some observations on the formal model. The model was suggested in order to enable the evaluation of a set of programs, each differing slightly from the tested program, where the tested program is supposed to be erroneous, while the correct program is assumed to be one of the programs in the set. As the evaluation of semantic equivalence is in general an undecidable problem, Davis and Weyuker base their definition of distance between programs on the program itself, and specifically, on the syntax of the program (and not on its semantics). The most crucial aspect of this attitude, which is also emphasized in [7], is that two programs that are semantically equivalent, but syntactically differ from each other, are said to have distance bigger than 0 between them.

The meaning of this is, that one program belongs to the set of the "close programs" to the other one (for the appropriate distance between the two programs in the selected distance function). If we consider the purpose of the evaluation of the set of "close programs", this also means that when coming to test the program, one should find an input to the program that will distinguish between these two programs... This of course is impossible! To eliminate this problem, it is required [7] that the set of alternative programs will not include equivalent programs. But here again we face the problem of evaluating semantic equivalence between programs. If for a given pair of programs this is undecidable, then we can not overcome the problem!

This disadvantage is also reflected in the grammar-based distance function introduced in [7]: with that distance function, $x < 5$ and $x < 1000$ are both said to be at distance one from $x < 6$ because there is one syntactic difference between them.

Another problem we face when coming to define a distance function according to [7] is also related to the above discussion. They require that distance 0 will be attached only to syntactically identical programs. Therefore, they do not support the use of their results for the evaluation of the minimal adequate test set for a given distance, in test criteria that are related only to parts of the program (such as only conditional statements in which we are interested).

However, what if we do wish to select test data only for conditional statements? If we want to use the formal model as it is, we cannot give distance 0
between programs that differ from each other in statements that do not interest us. Furthermore, if we define the distance between programs that differ in this type of statements to be some fixed non-zero value, say \( i \), then, if one will want to find all programs at distance up to, say \( i + 1 \), according to our distance function, he will have to consider all those undesired cases.

4 Branch Functions

In order to use the formal model, we look for a function that will suit our intuition of errors in a program that might cause erroneous control flow. We chose to base our function on a function that was introduced in the context of automatic test data generators. The following subsections describe that function and how it was modified for our purposes.

4.1 Automatic Test Data Generation

Korel [12] suggests an approach to pathwise test data generators. According to this approach, the test data is generated based on actual execution of the program under test. The program is given some arbitrary input, and the flow of control of the program when it is executed on this input, is monitored. If during execution, an undesirable execution flow, with respect to the required path, is observed, then a real valued function is associated with this branch. This function, called the branch function is positive (or, in some cases, 0) when the predicate is false, and negative when it is true. By using function minimization and dynamic data flow analysis, the input data that caused the undesirable flow is detected, and values for which the function becomes negative are located. A detailed description of the search process and some heuristic to assist it, are given in [12].

4.2 Modified Branch Function

The branch function as defined in [12] is not a distance function. The two major reasons for that are:

- The branch function can give a value zero for both the case that the branch predicate is false, and the case it is true, depending on the type of relational symbol in the predicate. In [7] only identical programs have distance 0 from each other.

- The branch function gives negative value to all inputs for which the predicate is true (except for some that get the value 0 as mentioned above). Although this definition serves its purpose – finding an input that will make the branch predicate true is all that is required – it does not suit
our case where we try to define distances between programs. A distance cannot assume negative values.

This leads to the definition of a modified branch function:

We make the assumption that the branch predicates are simple relational expressions of the form

\[ E_1 \ op \ E_2 \]

where \( E_1 \) and \( E_2 \) are either variables or constants where at most one of them is a variable and \( \text{op} \) is one of \{ \(<, \leq, >, \geq, =, \neq \) \}. We also assume that predicates do not contain boolean operators. The last assumption we make is that the variables in the expression take integer values. Note that this is more restrictive than [12]. Still it is true that every program in [12] can be transformed to an equivalent program in our setting by introducing new variables and assignment statements. However, the source of a possible error may be shifted from the branch statement to the assignment statement.

Let us define for each relational expression, the value of the modified branch function. In all cases, the leading intuition is that the boundary number that still fulfills the condition is considered to yield value 0 for the modified branch function, while the value of the function for any other number is its distance from that number. The table below summarizes the definition of the function:

<table>
<thead>
<tr>
<th>Branch Predicate</th>
<th>Branch Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>( E_1 &gt; E_2 )</td>
<td>( \text{abs}(E_1 - E_2 - 1) )</td>
</tr>
<tr>
<td>( E_1 \geq E_2 )</td>
<td>( \text{abs}(E_1 - E_2) )</td>
</tr>
<tr>
<td>( E_1 &lt; E_2 )</td>
<td>( \text{abs}(E_1 - E_2 + 1) )</td>
</tr>
<tr>
<td>( E_1 \leq E_2 )</td>
<td>( \text{abs}(E_1 - E_2) )</td>
</tr>
<tr>
<td>( E_1 = E_2 )</td>
<td>( \text{abs}(E_1 - E_2) )</td>
</tr>
<tr>
<td>( E_1 \neq E_2 )</td>
<td>( \text{abs}(\text{abs}(E_1 - E_2) - 1) )</td>
</tr>
</tbody>
</table>

Example

Consider the relational expression \( x \neq 5 \). In this case we have two boundary numbers that still fulfill the condition: 4 and 6. This is because 5 does not at all, and 3 and 7 (and of course other number more far from 5) do, but they are not boundary numbers of the expression. Therefore, according to our leading intuition, for 5, which is at distance 1 from 4 (and 6), the modified branch function returns 1. For numbers smaller than 4, or bigger than 6, the function returns the distance of the number from 4 or 6, respectively.

The important property of the modified branch function is that for any input, it gives a value that is greater or equal zero. Nevertheless, it also captures our intuition of possible errors in a conditional statement. That is, it is more likely
that a programmer will write in his program $z < 5$ instead of $z < 6$, than having him write $z < 1000$ in this case. This intuition will later guide us when we will speak of the distance between different statements. That is, we will measure the distance between the two statements, by the difference between the first element that fulfills the condition in one, and the appropriate in the other. Therefore, what is actually left for us to do now, is to translate the modified branch function into the definition of distance between programs.

4.3 A Distance Function for Branching

Having the modified branch function, we can now put it in the context of the formal model, and thus construct the adequate test set for our case.

Because of the disagreements between our intuition concerning our problem we deal with space, and the requirements for a distance function as were defined in [7], our definition of a distance function will be a compromise between the semantic and syntactic approaches. We will use a fault model, in which the syntactic changes we suspect to reside between the tested program and the correct one are a sub-family of all the syntactic changes possible. We will consider programs having syntactic differences not of the restricted type, as impossible alternatives (that is, we will not say that the two programs are close, since we do not suspect "slight" programming mistakes to cause them). Specifically, we define the following:

**Definition 4.1** We will say that two programs have a minor syntactic difference from each other, if and only if the differences between them are of the following types:

1. An `< identifier >` is replaced by another `< identifier >`.
2. A `< constant >` is replaced by another `< constant >`.
3. An `< operator >` is replaced by another `< operator >`.
4. A `< relation >` is replaced by another `< relation >`.

If the two programs differ in any other type of syntactic differences, we will say that they are essentially syntactically different. If the syntactic difference between the two programs is of types (2) - (4) only, we will say that the two programs are slightly syntactically different.

The following two lemmas follow the definition:

**Lemma 4.1** Let $P, Q$ and $R$ be programs. Consider the case that $P$ differs slightly syntactically from $Q$ in one statement, but is semantically equivalent to $P$ in that statement, and $Q$ differs slightly syntactically from $R$ in the same statement that $P$ and $Q$ differ, but is semantically equivalent to $Q$ in that statement. Then, $P$ is semantically equivalent to $R$ and either $P$ and $R$ are syntactically identical, or, $P$ differs from $R$ slightly syntactically.
Lemma 4.2 A mapping that relates to a program $P$ programs that differ from it slightly syntactically only is protected for $P$ in the sense of definition 3.2.

Having this definition, we can talk about semantic equivalence between statements that have minor syntactic differences, and semantic differences between them. The only exception we have to make is for statements in which the minor syntactic difference is of type (1) above, i.e., where an identifier is replaced by another one. In this case, semantic equivalence might be undecidable even in the one statement level. The way we decide to treat this case is to neglect it as well. This is the reason for the definition of the notion slightly different.

Definition 4.2 We shall consider a real valued function $\rho$ on pairs of programs as a distance function, if for any three programs that have a minor syntactic difference $P, Q$ and $R$, it fulfills the following four axioms:

1. $\rho(P, Q) \geq 0$;
2. $\rho(P, Q) = 0$ if and only if each statement in $P$ is syntactically identical to the appropriate statement in $Q$ or differs from it slightly syntactically, yet is semantically equivalent to it;
3. $\rho(P, Q) = \rho(Q, P)$;
4. $\rho(P, R) \leq \rho(P, Q) + \rho(Q, R)$.

The way we suggest to solve the problem of the need to give non-zero distances to statements that are different in the compared programs, yet, do not belong to the statements that interest us, is to go round it. Roughly said, we will define the distance of the "problematic" cases, to be always bigger than the distance we decide to consider. This will be done by giving the distance for these cases an adaptive definition, rather than a definition of some fixed distance. That is, we will say that two programs that differ in this type of statements, are always at a distance that is bigger by one than the distance up to which the tester of the program looks for alternative programs. This way, the distance is not 0, yet, we will never have to consider these cases. Formally we define our distance function in the following manner:

Definition 4.3 Let $d$ be a natural number. We think of $d$ as the depth of testing chosen in advance. Our distance function, $\rho_d$, between programs that have a minor syntactic difference, is defined in the following way:

- For two programs $P$ and $Q$ that are syntactically identical, or, differ slightly syntactically from each other in one statement, but are semantically equivalent in that statement, $\rho_d(P, Q) = 0$. 

12
• Let $P$ and $Q$ be two programs that differ slightly syntactically, and semantically from each other, in a statement that in both programs is a conditional one. Let $\Psi_P$ and $\Psi_Q$ be the set of numbers fulfilling the conditional statement in $P$ and $Q$ respectively, and let $\Psi_d$ be $\Psi_P \setminus \Psi_Q$ (i.e., $\Psi_d$ is the set of all numbers that belong either to $\Psi_P$ or to $\Psi_Q$ but not to both). If $\Psi_d$ is finite then: Let $t_P$ and $t_Q$ be the value for which the modified branch function returns 0 in the compared statement in $P$ and in $Q$ respectively. Let $D$ be equal to $\text{abs}(t_P - t_Q)$. If $D \leq d$, then $\rho_d(P, Q) = D$.

• For two programs $P$ and $Q$ that differ in one statement in any other way, $\rho_d(P, Q) = d + 1$.

• The distance between programs that differ from each other in more than one statement is the sum of the distances appropriate to each difference.

This definition relies heavily on the assumptions we made on the modified branch function.

Theorem 4.1 $\rho_d$ is a discrete finite distance function in the sense of definition 4.2.

Proof: The theorem follows immediately from the definition of our distance function, from lemma 4.1 on and from the fact that the triangle inequality holds for the summation of absolute numbers. Q.E.D.

Examples

The definition of our distance function follows our simple intuition of common errors in conditional statements. In order to illustrate this, we find it appropriate to look at some examples before we proceed.

• Suppose that a program $P$ differs from an alternative program $Q$ only in one conditional statement, where in $P$ the conditional statement is $x \geq 13$, and in $Q$ it is $x > 12$. The two programs differ slightly syntactically, and are semantically equivalent. Therefore, $\rho_d(P, Q) = 0$.

• Suppose that the two programs $P$ and $Q$ differ from each other only in one conditional statement, where in $P$ the conditional statement is $x \geq 12$, and in $Q$ it is $x \geq 15$. According to the grammar based distance function of [7]'s, the two programs would have distance 1. With our distance function, if $d$ is chosen to be 3 or more, we have: $\Psi_d = \{12, 13, 14\}$. $\Psi_d$ is finite. The modified branch function returns 0 in the statement $x \geq 12$, for $x = 12$, and in the statement $x \geq 15$, for $x = 15$. $D = \text{abs}(15 - 12) = 3$, so, $\rho_d(P, Q) = 3$.

For $d = 1$ on the other hand, $\rho_d(P, Q) = d + 1 = 1 + 1 = 2$. 13
5 Adequate Test Sets for Branching

Once we have defined our modified distance function, we would like to adapt more of [7]'s definitions and results to our case. Specifically, we are interested in using their results in what they define as the minimal d-adequate test set.

5.1 Adequate Test Set for Conditional Statements

By theorem 4.1 \( \rho_d \) is a finite distance function and by lemma 4.2 \( \Phi_d(P) = \{ Q : \rho_d(P, Q) = d \} \) is protected for \( P \). Therefore, the definition of d-critical points and of d-adequate test sets applies to our case. Recall that the d-critical points are included in any d-adequate test set of \( P \) and for each \( e > d \), they are also e-critical. Therefore, we first identify the 1-critical points.

**Definition 5.1** Let \( P \) be a program and \( b \) a particular conditional statement in \( P \). \( \Phi_{d,b} \) is the set of all programs \( Q \) such that \( \rho_d(P, Q) \leq d \) and \( Q \) differs from \( P \) only in \( b \).

For our distance function \( \rho_d \) we have:

**Lemma 5.1** \( \Phi_d(P) = \bigcup_{b \in P} \Phi_{d,b} \).

Therefore, it is enough to discuss critical points and adequate test sets for \( \Phi_{d,b} \). Symbolic evaluation should be used in order to compute the input data that will cause the execution of \( b \) with the required value.

**Definition 5.2** Let \( P \) be a program and \( b \) a particular conditional statement in \( P \). Then, \( t_{P,b} \) is the value for which the modified branch function returns 0 for \( b \) and \( f_{P,b} \) is the value for which the modified branch function returns 1 for \( b \), and \( b \) evaluates to false. (In other words, \( t_{P,b} \) is the boundary value for which the conditional statement is true, and \( f_{P,b} \) is the boundary for which it is false).

**Lemma 5.2** For each of the possible relation symbols, the elements for which the condition is true, are related to the relation symbol in the following way:

- when the relation is either \( < \) or \( \leq \): the condition is true for each of the elements that are smaller, or, smaller or equal, respectively, then \( t_{P,b} \), and only in these elements;

- when the relation is either \( > \) or \( \geq \): the condition is true for each of the elements that are bigger, or, bigger or equal, respectively, then \( t_{P,b} \), and only in these elements;

- when the relation is \( = \): the condition is true only for \( t_{P,b} \);
when the relation is $\neq$: there are two $t_{PB}$'s; one smaller than the only element for which the condition is false, and one bigger than it. In this case, the condition is true for each of the elements that are bigger or equal the bigger $t_{PB}$, and for each of the elements that are smaller or equal the smaller one.

Proof: Trivial.

Theorem 5.1 $t_{PB}$ and $f_{PB}$ are 1-critical points for our distance function when the relation symbol is one of $\{<,\leq,>,\geq\}$.

Proof: According to lemma 5.2 above, common to all of the relation symbols $<,\leq,>$ and $\geq$ is the fact that they have exactly one $t_{PB}$ and one $f_{PB}$.

Moreover, for each of the relation symbols $<,\leq,>$ and $\geq$, we have that the numbers that fulfill the condition are from the first one that fulfills it, and continuously from there on, in one direction (either all smaller or all bigger). If we combine this with the fact that alternative programs are evaluated for the case that $\Psi_d$ is finite, we get that the numbers that fulfill the condition in $P$ have to be in the same direction from the first that fulfills it, as in the alternative programs. Now, by the definition of our distance function, at distance 1 there are only programs for which the modified branch function returns 0 in one conditional statement, either on $t_{PB}$ - 1 or on $t_{PB}$ + 1. In both cases, as we have seen above, $f_{PB}$ is one of these two, and the second number for which the condition is true (one away from $t_{PB}$, with the condition true), is the other. Let $Q$ be an alternative program at distance one from $P$. Then, if in $Q t_{QB}$ is equal $f_{PB}$ (the condition is true in $Q$), all the numbers that fulfill the condition in $P$, fulfill it in $Q$ as well, and the two programs differ only on $t_{QB} = f_{PB}$. If, on the other hand, $t_{QB}$ is equal to the second number for which the condition is true in $P$, all the numbers that fulfill the condition in $Q$, fulfill it in $P$ as well, and only for $t_{PB}$, the condition is false in $Q$ and true in $P$. In this case, only $t_{PB}$ distinguishes between $P$ and $Q$. Therefore, the two programs differ either only in $t_{PB}$ or only in $f_{PB}$. Yet, we have, that for the relation symbols $<,\leq,>$ and $\geq$, there is only one $t_{PB}$ and one $f_{PB}$. This, by definition, makes $t_{PB}$ and $f_{PB}$ critical points. Q.E.D.

Lemma 5.3 With our distance function, no critical points arise from conditional statements with relation symbol either $=$ or $\neq$.

Proof: For both relation symbols $=$ and $\neq$, our distance function does not put programs with another relation symbol, in the set of close programs. This is because for such programs, based on lemma 5.2 above, $\Psi_d$ is infinite. For such cases, our distance function puts the two programs at a distance bigger than the distance up to which we test. Therefore, the case we have to consider is that the relation symbol is the same in $P$ and in the alternative program. But, in this case, if the relation symbol is $=$, the two programs differ in two
values: the only value for which which the condition in \( P \) is true, and the only value for which the condition in the alternative is true. These values are not the same, because if they were, the two programs were equivalent, contradicting the fact that our distance function does not put equivalent programs in the set of programs from which \( P \) has to be differed. As there is no unique value in which the two programs differ, there are no critical points for the relation symbol.

By the same argument, only by considering the only value for which the relation \( \neq \) is false, we get that the same holds for \( \neq \). Q.E.D.

Therefore, we conclude, that when the relation symbol is one of \( \{<, \leq, >, \geq\} \), \( t_{P,b} \) and \( f_{P,b} \) must be included in any adequate test set to \( P \).

By definition, the minimal \( d \)-adequate test set is the smallest set of inputs to the tested program, for which the output of the the tested program and the output of each of the programs that are at distance less or equal \( d \) from it, differ at least once. This definition is also meaningful for our distance function. We first examine the case \( d \) equals 1:

**Theorem 5.2** The minimal \( 1 \)-adequate test set for the case that the relation symbol is one of \( \{<, \leq, >, \geq\} \), is comprised from \( t_{P,b} \) and \( f_{P,b} \), and from these elements only.

**Proof**: As we have already seen, \( t_{P,b} \) and \( f_{P,b} \) have to be included in any adequate test set to \( P \), and therefore, also in the minimal \( 1 \)-adequate test set. Now, let us assume contradiction, that there is an error at distance one from the program, that \( t_{P,b} \) and \( f_{P,b} \) do not manage to reveal. Let \( Q \) be the program with this error, and \( t_{Q,b} \) and \( f_{Q,b} \) be in \( Q \), the appropriate to \( t_{P,b} \) and \( f_{P,b} \) in \( P \). For this distance, this means that either \( t_{Q,b} = t_{P,b} - 1 \), or, it is equal \( t_{P,b} + 1 \). But, for any such \( Q \), we have already seen in the proof of theorem 5.1 above, that \( P \) and \( Q \) differ only in one of \( t_{P,b} \) and \( f_{P,b} \). So, if they do not differ on these numbers, they do not differ at all. In this case, \( Q \not\in \Phi_1(P) \), and hence does not contribute elements to the test set. Otherwise, we get a contradiction to the assumption. Q.E.D.

**Theorem 5.3**

(i) If the relation symbol is \( = \), the minimal \( 1 \)-adequate test set is comprised from \( t_{P,b} \) and from it only.

(ii) If the relation symbol is \( \neq \), the minimal \( 1 \)-adequate test set is comprised from \( f_{P,b} \) and from it only.

**Proof**: As we have seen in the proof of lemma 5.3 above, for the relation symbol \( = \), \( t_{P,b} \) is the only element for which the condition is true. Moreover, each of the programs from which it has to be differed, differs from it in this point. Therefore, this point is sufficient to distinguish \( P \) from any of the alternative programs that are at distance 1 from it. On the other hand, for this relation symbol, there are two \( f_{P,b} \)'s. So, if we do not select \( t_{P,b} \) to the test set, both the \( f_{P,b} \)'s will have to be included in the test set, in order to distinguish \( P \) from
each of the possible two different alternative programs at distance one from it
(one with \( t_{Q_A} = t_{P_A} + 1 \) and the other with \( t_{Q_A} = t_{P_A} - 1 \)). Therefore, for the
1-minimal adequate test set, \( t_{P_A} \) is the number that has to be included in the set.

By the same argument, for the relation symbol \( \neq \), the test set should be
comprised either from \( f_{P_A} \) only, or from the two \( t_{P_A} \)'s. Therefore, for the 1-
minimal adequate test set, \( f_{P_A} \) is the number that has to be included in the set.
Q.E.D.

The proofs of the theorems also indicate when the 1-minimal adequate test
set is not adequate for bigger distances. That is, when \( P \) and \( Q \) give the same
result on \( t_{P_A} \) and \( t_{Q_A} \), or, on \( f_{P_A} \) and \( f_{Q_A} \), even though the conditions are
slightly syntactically different. This can happen only in the following cases:

1. The result of \( P \) and \( Q \) on \( t_{P_A} \) is the same even if in one the true branch
   from the condition is followed and in the other the false one, and same
   happens on \( f_{P_A} \);

2. When \( P \) and \( Q \) are run on any of \( t_{P_A} \) and \( f_{P_A} \) control does not flow
   through the conditional statement in which \( P \) and \( Q \) differ.

In both these cases, this phenomena implies that \( Q \notin \Phi_1(P) \). That is, either \( t_{P_A} \)
or \( f_{P_A} \) is not a critical point to \( P \). By the same argument as before, either the
point for which the condition is true, and the modified branch function returns
1 (one after \( t_{P_A} \) in the appropriate direction), or the point for which the the
condition is false, and the modified branch function returns two (one after \( f_{P_A} \)
in the appropriate direction), appropriately, is 2-critical. This is generalized in the
following theorem:

**Theorem 5.4** Let \( d \) be the least distance for which the \( d \)-minimal adequate test
set contains, for each conditional statement, at least one element on which the
condition is true, and one on which it is false. Then, if the relation symbol is
one of \( \{<, \leq, >, \geq\} \), for each \( e > d \), the \( e \)-minimal adequate test set is equal to
the \( d \)-minimal adequate test set.

**Proof:** Let \( S \) be the set of all the numbers that distinguish \( P \) from each of
its alternatives up to distance \( e \). Let \( x \) be the number for which the modified
branch function returns the value that is closest to \( t_{P_A} \) amongst the elements
of \( S \) on which the condition is true. In the same manner, let \( y \) be the number
for which the modified branch function returns the value that is closest to \( f_{P_A} \)
amongst the elements of \( S \) on which the condition is false. Based on lemma 5.2
above, with our distance function, in the case that the relation symbol is one
of \( \{<, \leq, >, \geq\} \), all the numbers that fulfill the condition are on the same side
of \( t_{P_A} \), and all those that don't, are on the other. Therefore, \( x \) distinguishes \( P \)
from all alternative programs in which the condition is false, and \( y \) distinguishes
\( P \) from all alternative programs in which the condition is true. Together, \( x \) and
y distinguish $P$ from all its alternatives. If we let $d$ be the maximum between the value that the modified branch function returns on $x$, and the value it returns on $y, z$ and $y$ are the $d$-minimal adequate test set. Moreover, they distinguish $P$ from all programs that are at distance $e$ from it, for all $e > d$. Therefore, the test set comprised from them, is also the $e$-minimal adequate test set. Q.E.D.

The meaning of this result is that when the relation symbol is one of \{<, \leq, >, \geq\}, there is an upper limit to the distance up till which one has to look for alternative programs to the tested one. This limit is the lowest distance in which there exists a program that returns a different result than the tested program on the true branch of the condition, and a program that returns a different value than the tested one on the false branch of the condition.

**Theorem 5.5** Assume

(i) the relation symbol in a conditional statement is $=$, and $t_{P\,b}$ does not distinguish $P$ from the programs that are at distance 1 from it,

or:

(ii) the relation symbol in a conditional statement is $\leq$, and $f_{P\,b}$ does not distinguish $P$ from the programs that are at distance 1 from it.

Then, for each $e \geq 1$, the $e$-minimal adequate test set includes at most $2e$ elements.

**Proof:** As we have seen in the proof of theorem 5.3 above, when the relation symbol in the conditional statement is $=$, either $t_{P\,b}$, or the two $f_{P\,b}$'s have to be included in the test set. Therefore, if $t_{P\,b}$ doesn't distinguish $P$ from the programs that are at distance 1 from it, the minimal 1-adequate test set is comprised from the two $f_{P\,b}$'s. For any bigger distance, the new programs encountered in the set of "close" programs to $P$, differ from $P$ at most only in $t_{P\,b}$ and in the number for which the modified branch function returns 0 in the alternative program. This is because, as we saw in that proof, the relation symbol in the alternative program has to be $=$ as well. Now that $t_{P\,b}$ does not distinguish, the other number is the only one that can. Therefore, it has to be included in the minimal adequate test set for this distance. Also, because of the symmetry around $t_{P\,b}$ of the numbers for which the condition is false each time that the considered distance is increased by one, two new alternative programs are introduced. This implies the theorem.

By the same arguments, but only by considering $f_{P\,b}$ instead of $t_{P\,b}$, and the elements for which the condition is true instead of those for which it is false, we get that the theorem holds for $\neq$ as well. Q.E.D.

It is interesting to note here, that this test set is the same as what pragmatic experience taught us to select. This can serve as a proof to the correctness of our intuition on the possible errors in conditional statements, as was reflected in our distance function. Looking at this result from another perspective, we find
this result encouraging because of the way we arrived to it. That is, we started by the definition of errors we suspect to reside, and accepted this adequate test set, as a result. This leads us to the belief that for other types of suspected errors, the same method would also appeal for the derivation of the adequate test set.

5.2 Adequate Test Set for ALL BRANCHES

ALL BRANCHES is a testing criteria that executes each branching direction in each conditional statement in the program, without relating to alternative programs. A severe criticism against this type of criteria was expressed in [7]:

This way of thinking about test-data adequacy criteria helps elucidate a fundamental weakness of using statement or branch coverage as such a measure. A Φ-adequate criterion requires that test data be included which distinguish a given program from a fixed set of other inequivalent programs. Viewed another way, such a criterion requires test data which guarantee that certain predetermined errors are not present. Statement and branch testing, in contrast, do not seek to distinguish a given program from other programs (or to detect particular errors). They merely require that various parts of the program be executed.

Our distance function gives a solution if one wants to test only parts of the program, yet do it by considering alternative programs that suit one's intuition. We find it interesting to compare the adequate test set we got, with the one that is required by the "ALL BRANCHES" testing criteria, as it is also related to conditional statements. What we get is the following:

Theorem 5.6 For each conditional statement in the program;

If the relation symbol is one of \{<, ≤, >, ≥\} : the minimal adequate test set of our distance function is adequate for "ALL BRANCHES";

If the relation symbol is either = or ≠ : then:

- If \(t_P, P\) or \(f_P, P\) respectively, distinguish \(P\) from programs at distance 1 from it : then, the minimal adequate test set of our distance function is not adequate for "ALL BRANCHES";
- Otherwise : the minimal adequate test set for our distance function is not adequate for "ALL BRANCHES", and one of the following two possibilities holds:
  1. There is no adequate test set for "ALL BRANCHES";
  2. The adequate test set for "ALL BRANCHES" is not adequate for our distance function.
Proof: For each of the relation symbols $<$, $\leq$, $>$, and $\geq$, the adequate test set for our distance function, as we got in theorems 5.2 and 5.4 above, includes one element for which the condition is true, and one for which it is false. This is exactly what "ALL BRANCHES" requires!

For the relation symbols $=$ and $\neq$, for our distance function, only one element is required (the only one for which the condition is true, or, the only one for which it is false, respectively), if this element distinguishes $P$ from programs at distance 1 from $P$. Therefore, in this case, only one direction of branching is tested, which is not adequate to "ALL BRANCHES".

If, on the other hand, this only element does not distinguish the tested program from its alternatives, it does not belong to our minimal adequate test set. Hence, our minimal adequate test set does not exercise one of the directions of branching (the one taken only for this element). Now, if this only element doesn’t distinguish because the tested conditional statement is not reachable on this value, there is no way to perform "ALL BRANCHES" to the statement, because one direction of the branching can not be executed. That is, there is no adequate test set to "ALL BRANCHES". If, on the other hand, this only element doesn’t distinguish because the final result of both $P$ and its alternatives is the same even though the tested conditional statement is executed, then two elements: one for which the condition is true, and one for which it is false (two directions of the branching are exercised), are adequate for "ALL BRANCHES".

With our distance function, according to theorem 5.5, each time the distance is increased, more elements are required. Therefore, the adequate test set for "ALL BRANCHES", is not adequate for our distance function.

These are all the cases of the theorem. Q.E.D.

As the theorem reflects, there is much in common between our adequate test set, and the one required for "ALL BRANCHES". This leads us to conclude that our results can be used to define the adequate test set for "ALL BRANCHES" when the relation symbol is one of $<$, $\leq$, $>$, and $\geq$. Moreover, in the spirit of our diagnosis, we can suggest, that for the relation symbols $= \neq$, the only element for which the condition is true or false respectively will be used for practicing one direction of branching, and one of the two elements beside it (the first for which the condition is false or true respectively) will be the other. This way, not only that the test set we get, is adequate for "ALL BRANCHES", it also captures our intuition of errors that reside in conditional statements.

The differences that still reside between our adequate test set, and this suggested adequate test set for "ALL BRANCHES", are summarized in the following theorem:

**Theorem 5.7** When the relation symbol is one of $\{<, \leq, >, \geq\}$, the adequate test set to "ALL BRANCHES" is the same as the one of our distance function. When the relation symbol in a conditional statement is either $=$ or $\neq$, let $e_o$ be the only element for which the condition is true, or false, respectively. Then:
if control flows through the conditional statement on \( e_0 \): then, the test
set that is comprised from \( e_0 \) and either of its two adjacent numbers (\( e_0 + 1 \)
or \( e_0 - 1 \)) is adequate for "ALL BRANCHES".

else: There is no adequate test set for "ALL BRANCHES".

Proof: This theorem is a corollary from theorem 5.6 above. Q.E.D.

5.3 Adequate Test Set for Loops

As we saw, the set of statements that we find necessary to test based on \( PNI \),
are a subset of the set of all the conditional statements in the program. That
is, if we let \( S_{ACS} \) be the set of all conditional statements in the program, and
let \( S_{PNI} \) be the set of all those that were identified as necessary to be tested
based on \( PNI \), we have: \( S_{PNI} \subseteq S_{ACS} \). In the same manner:

Notation 5.1 (i) Let the \( d_{ACS} \)-adequate test set be the adequate test set
for all programs up to distance \( d \) from the tested one, with our distance
function, when all conditional statements in the program are taken into
consideration. Also, let the \( d_{PNI} \)-adequate test set be the appropriate set,
when only the conditional statements that are identified based on \( PNI \) are
considered.

(ii) Let the \( d_{ACS} \)-critical and the \( d_{PNI} \)-critical sets, be the sets of critical
points up to distance \( d \), when the appropriate set of conditional statements
is taken into consideration.

(iii) Let the \( d_{ACS} \)-minimal adequate test set and the \( d_{PNI} \)-minimal ade-
quate test set, be the sets of critical points up to distance \( d \), when the
appropriate set of conditional statements is taken into consideration.

Then, we have the following:

Theorem 5.8 For any distance \( d \), with our distance function

(i) If an element belongs to the \( d_{PNI} \)-adequate test set, it also belongs to
the \( d_{ACS} \)-adequate test set.

(ii) If an element belongs to the \( d_{PNI} \)-critical set, it also belongs to the
\( d_{ACS} \)-critical set.

(iii) If an element belongs to the \( d_{PNI} \)-minimal adequate test set, it also
belongs to the \( d_{ACS} \)-minimal adequate test set.

Proof: This theorem is an immediate outcome from our fault model, which
assumes minor syntactical changes in the "one statement"'s level only. Q.E.D.

Corollary 5.1 For any distance \( d \), with our distance function

21
(i) (The $d_{PNI}$-adequate test set) $\subseteq$ (the $d_{ACS}$-adequate test set).

(ii) (The $d_{PNI}$-critical set) $\subseteq$ (the $d_{ACS}$-critical set).

(iii) (The $d_{PNI}$-minimal adequate test set) $\subseteq$ (the $d_{ACS}$-minimal adequate test set).

That is, the test set for testing a program based on $PNI$, is a subset of what we found to be the adequate test set with our distance function. This result can be viewed in two ways: First, that the test set for testing based on $PNI$ can be achieved by eliminating from the adequate test set of our function, all the elements that distinguish the tested program from programs that differ from it in conditional statements other than those identified based on $PNI$. The other way is, that this test set can be derived directly, if in our distance function, in each place the phrase "a conditional statement" is used, it is replaced by "a conditional statement that was identified to be tested based on $PNI$".

5.4 Revealing More Errors by ALL BRANCHES

As we mentioned above, beyond the fact that $PNI$ detects errors which are related to conditional statements, $PNI$ also captures data about paths. That is, $PNI$ reflects which iteration paths can not be executed immediately following others. This fact can be used to reveal errors which are not due to errors in the conditional statements themselves, but are of the type that is detectable using paths testing. In other words, our attitude when we evaluated the adequate test set for our distance function was, that the errors in the program are of a specific type: slight syntactical changes in conditional statements. Yet, we claim the following:

Property 5.1 If, while the program is tested with the adequate test set of our distance function, it is verified that the flow of control does not contradict $PNI$, there are errors in statements of a loop which are not conditional, which can also be detected.

Proof: Because of $PNI$'s property of capturing data about paths, if there is an error in a statement in a loop that is not a conditional statement, and this error causes control to flow from an iteration path to another that is not expected to be possible to be executed immediately after it (by changing incorrectly the value of a variable upon which there is a condition), this will contradict $PNI$. For this type of error, the property holds. Q.E.D.

Because of the similarities between the adequate test set of our distance function and of "ALL BRANCHES", as were reflected in theorem 5.7 above, property 5.1 can also be stated in terms of the adequate test set for "ALL BRANCHES". That is, we have:
Property 5.2 If, while the program is tested with the adequate test set of "ALL BRANCHES", it is verified that the flow of control does not contradict PNI, there are errors in statements of a loop-which are not conditional, which can also be detected.

Proof: The property is a corollary from theorem 5.7 and property 5.1 above. Q.E.D.

To illustrate this property, let us look at the following example:

Example

Consider the program P below:

![Diagram of program P]

The points that might distinguish P from same programs, only with the conditional statement $j < 1$ replaced by a statement at distance 1 from it are $j = 0$ and $j = 1$. The statement is reachable on both these values. If at the beginning of the execution, $x = x_0$, then, for $j = 0$ the program gives the result $x_0 + 2$. For $j = 1$ it gives $x_0 + 3$. The same program but with the conditional statement $j < 1$ replaced by $j < 0$, is at distance 1 from P. For $j = 0$ the alternative program returns $x_0 + 1$, which is not what $P$ returns. Therefore, 0 is a 1-critical point. In the same manner, the same program as P but with the conditional statement $j < 1$ replaced by $j \leq 1$, is at distance 1 from $P$. For $j = 1$ this alternative program returns $x_0$, which is not what $P$ returns. Therefore, 1 is also a 1-critical point. Now that the 1-minimal adequate test set includes both an element for which the condition is true, and an element for
which it is false, the test set \( \{ 0, 1 \} \) is the adequate test set for \( P \) for our distance function. Note, that since the relation symbol in the conditional statement is \(<\), then, according to theorem 5.7 above, this is also the adequate test set when performing "ALL BRANCHES", and thus, the example is relevant also for this testing criteria.

Now suppose that the error that really exists in the program is that instead of the assignment statement \( j := j - 2 \), the assignment statement \( j := j + 2 \) appears in the program. Now if we run the program with input \( j = 0 \), we get the result \( x_0 + 2 \), which is the same as in the correct program. Also, if we run the program with input \( j = 1 \), we get the result \( x_0 + 3 \), which is the same as in the correct program. That is, the adequate test for our distance function (and for "ALL BRANCHES") did not reveal the error! This happened because of the disadvantage of our distance function (and of "ALL BRANCHES"), that it does not take into account errors of "paths" characteristic.

Now, let us see how \( PNI \) can help. In this program, when it has no errors, once an iteration path has been selected, the other iteration path can not be executed immediately following it. This is because the assignment statements along the iteration path keep the iteration path's conditions valid. Therefore, we have the following \( PNI \) for program \( P \):

\[
PNI = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]

When the program containing the unrevealed error is being run on input \( j = 1 \), not only that the final result is the same as for the correct program, but also, no contradiction to \( PNI \) appears. Yet, when the erroneous program is being run on input \( j = 0 \), in the first iteration of the loop, the iteration path with the condition \( j < 1 \) true is executed. But, on the next iteration the iteration path with this condition false is the one that is executed. \textit{This contradicts PNI.}

Therefore, even though the result of the program for this input is the same as in the correct program, the comparison of the control flow against \( PNI \), revealed that there is an error in the program, and it did this with the very same test data!

We find this result very important. If we take into consideration the fact that the "ALL PATHS" testing criteria is infeasible, mainly because of its problem to deal with loops, and the fact that "ALL BRANCHES" is considered to be the most commonly used testing criteria, the meaning of our result is, that with the very same test data as required for "ALL BRANCHES", one can get a better approximation to "ALL PATHS" than "ALL BRANCHES" by itself gives! This, if he, or she, verifies during the execution of the program on the "ALL BRANCHES" adequate test set, that the flow of control does not contradict \( PNI \).
6 Conclusions and Further Research

In this research we obtained results by combining different approaches related to the reliability of programs:

- The control flow was analyzed using the program flow graph;
- \(PNI\) is derived using symbolic evaluation;
- A distance function was defined, using the intuition of a function used by pathwise test data generators;
- An adequate test set was derived based on a formal model for this purpose.

The use of the above methods in relation to each other, led us to identify how results from one field can be used to improve others. Specifically, we got the following results:

- A way to overcome the limitation of the formal model in handling semantically equivalent programs;
- A method to derive an adequate test set for conditional statements;
- A method to derive an adequate test set based on \(PNI\);
- A method to derive a test set that can be regarded adequate for the "ALL BRANCHES" testing criteria;
- A way to reveal more errors by the test set of "ALL BRANCHES" (using \(PNI\), than what this test set reveals without using this method;

The applicability of the results to other fields of program testing, such as acceptability testing and fault tolerant computing is a subject for further research. Nevertheless, we find our results motivating further research in the direction of combining results that were achieved in different branches in the field of program testing.
References


LIST OF CIS REPORTS

CIS 9001
Guy Foux, Michael Heymann, and Alfred M. Bruckstein: “Two-Dimensional Polygon Navigation Among Unknown Stationary Polygonal Obstacles”. 

CIS 9002

CIS 9003

CIS 9004

CIS 9005

CIS 9006

CIS 9007

CIS 9008

CIS 9009

CIS 9010

ANNUAL REPORT, 1990.

CIS 9011
CIS 9012

CIS 9101

CIS 9102

CIS 9103

CIS 9104

CIS 9105

CIS 9106

CIS 9107

CIS 9108

CIS 9109

CIS 9110

CIS 9111
CIS 9201

CIS 9202

CIS 9202 (Revised, June 1992)

CIS 9203

CIS 9204

CIS 9205

CIS 9206

CIS 9207

CIS 9208

CIS 9209

CIS 9210

CIS 9211

CIS 9212
CIS 9213

CIS 9214

CIS 9215

CIS 9216

CIS 9217

CIS 9218