MATRIX: Maintenance-Oriented Testing Requirements Identifier and Examiner

Taweesup Apiwattanapong,† Raul Santelices,† Pavan Kumar Chittimalli,‡
Alessandro Orso,† and Mary Jean Harrold†
†College of Computing, Georgia Institute of Technology
‡Tata Research Development & Design Centre, Tata Consultancy Services Limited
{term|raul|orso|harrold}@cc.gatech.edu, pavan.chittimalli@tcs.com

Abstract

This paper presents a new test-suite augmentation technique for use in regression testing of software. Our technique combines dependence analysis and symbolic evaluation and uses information about the changes between two versions of a program to (1) identify parts of the program affected by the changes, (2) compute the conditions under which the effects of the changes are propagated to such parts, and (3) create a set of testing requirements based on the computed information. Testers can use these requirements to assess the effectiveness of the regression testing performed so far and to guide the selection of new test cases. The paper also presents MATRIX*, a tool that partially implements our technique, and its integration into a regression-testing environment. Finally, the paper presents a preliminary empirical study performed on two small programs. The study provides initial evidence of both the effectiveness of our technique and the shortcomings of previous techniques in assessing the adequacy of a test suite with respect to exercising the effect of program changes.*

1 Introduction

Software engineers spend most of their time maintaining and evolving software systems to fix faults, improve performance, or add new features. A fundamental activity during software evolution is regression testing. *Regression testing* is the testing of modified software to gain confidence that the changed parts of the software behave as expected and to ensure that the changes have not introduced new faults into the rest of the software.

To date, researchers have mostly focused on three regression-testing activities: regression-test selection, test-suite reduction (or minimization), and test-suite prioritization. *Regression-test selection* identifies test cases in an existing test suite that need not be rerun on the new version of the software (e.g., [6, 22, 28]). *Test-suite reduction* eliminates redundant test cases in a test suite according to given criteria (e.g., [13, 23, 30]). *Test-suite prioritization* orders test cases in a test suite to help find defects earlier (e.g., [24, 26]). Researchers have shown that the use of these techniques can reduce considerably the regression testing time. Little attention, however, has been paid to a fundamental problem of testing modified software: determining whether an existing regression test suite adequately exercises the software with respect to the changes and, if not, providing suitable guidance for creating new test cases that specifically target the changed behavior of the software. We call this problem the *test-suite augmentation* problem.

The modified parts of a program can be tested by ensuring that this modified code is executed. The most problematic regression errors, however, are typically due to unforeseen interactions between the changed code and the rest of the program. Previous research has addressed this problem by creating criteria that focus on exercising control- or data-flow relationships related to program changes (e.g., [3, 11, 21]). Our studies show that considering the effects of changes on the control- and data-flow alone, as these techniques propose, is often not sufficient for exercising the effects of the changes on the software. Even when test cases exercise the data-flow relationships from the point of the change(s) to the output [9], the modified behavior of the software may not be exercised.

1.1 Motivating Example

To illustrate the inadequacy of these criteria, consider the example in Figure 1, which shows a class E and its method $simple$. For statements $s1$ and $s2$, alternative versions $c1$ and $c2$, respectively, are provided as comments. These alternative versions can be used to construct modified versions of E. Hereafter, we indicate with E_{c1} (E_{c2}) the version of E obtained by replacing $s1$ ($s2$) with $c1$ ($c2$). Consider program E_{c2}. The only difference between E and E_{c2} is the conditional statement $s2$, which is changed from `(x > 5)` to `(x >= 5)`. A technique that tests the changed program by rerunning all test cases that traverse the change would generally not reveal the regression fault introduced by the change (the division by zero at statement $s3$). Even a technique that exercises data-flow relationships from the point of the change to an output would be unlikely to reveal the problem. The only way to suitably exercise the change at $c2$ is to require that method $simple$ in E_{c2} be called with 5 as an argument.

The problem is that changes in the software affect, after their execution, the state and the control-flow of the software, but these effects often manifest themselves only under specific conditions. Therefore, criteria that simply re-

```
      public class E {
        void simple(int i) {
s1        int x=i;    // c1: int x=i+b;
s2        if (x>5) { // c2: if (x>=5) {
s3          x = (5/(x-5));
          }
s4        x--;
s5        if (x == 0) {
s6          System.out.print(x);
          } else {
s7          System.out.print(10/x);
          }
        }
        ...
      }
```

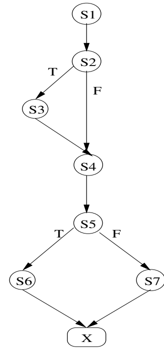

Figure 1. Example program E and control-flow graph of E's method $simple$.

quire the coverage of program entities (e.g., statements and data-flow relationships) are usually inadequate for exercising program changes. These criteria are satisfied when all required program entities are executed and do not consider the effects of changes on the state of the program caused by program modification or the propagation of these changes. To account for this limitation, one possibility is to use techniques that consider the state of the program, such as symbolic evaluation and model checking [7, 8]. However, these techniques are complex and expensive and, if applied to the whole program, would typically not scale.

1.2 Overview of the Approach

This paper presents a novel approach that addresses the shortcomings of existing techniques by combining data- and control-dependence information, gathered through program analyses, with state-related information, gathered through symbolic evaluation. Because of this combination, our criteria are more likely to propagate the effects of the changes to the output in a modified program than criteria based only on the program's structure or entities. To limit the cost of symbolic evaluation and make the approach practical, we limit the number of statements that we analyze based on their distance (in terms of data- and control-dependences) from the changes. As our studies show, considering statements that are only a few dependences away from the change lets us build testing requirements that are effective in revealing regression faults while keeping the cost of our technique contained and making our overall approach practical.

Our technique for test-suite augmentation involves several steps. First, the technique uses information about the differences between the old program (P) and the new program (P'), along with mappings between corresponding statements in them, to identify pairs of corresponding statements in P and P'. Second, the technique uses symbolic evaluation to compute, for statements that could be executed after the changed statements in P and P', a path condition and a symbolic state. Third, the technique compares path conditions and symbolic states of corresponding statements in P and P' and defines testing requirements based on the comparison. Fourth, the technique instruments P' to assess, during regression testing, the extent to which a test suite satisfies these testing requirements. Finally, the set of unsatisfied testing requirements provides guidance to the tester for the development of new test cases.

The current version of the technique works on a single change at a time and shares symbolic execution's current limitations. In particular, it cannot handle some program constructs (e.g., unbounded dynamically allocated objects), but only when these constructs are affected by the changes.

The paper also presents a tool, MATRIX (Maintenance-oriented Testing Requirements Identifier and eXaminer), that partially implements our approach. The current version of MATRIX incorporates data- and control-dependence analyses and instrumentation capabilities, and is integrated into a regression-testing environment. The environment includes other tools that perform supporting tasks for the test-suite augmentation technique and regression testing in general. We are currently working on a version of MATRIX that incorporates symbolic evaluation.

Finally, the paper discusses two studies performed on nine versions of two small software subjects. In the studies we compare, in terms of effectiveness, our technique with two alternative test-suite augmentation approaches: one that requires test cases to traverse the changes in the software and the other that requires test cases to exercise all definition-use associations affected by the changes. The results of the study show that, for our subjects and versions, test suites that satisfy the testing requirements generated by our technique are more likely to reveal regression errors than test suites that satisfy the testing requirements generated by the alternative approaches but do not satisfy our requirements.

The main contributions of the paper are:

- A new test-suite augmentation technique that uses differencing, program analysis, and symbolic evaluation to compute a set of testing requirements that can be used to assess and augment a regression test suite.
- A tool, MATRIX, that partially implements the test-suite augmentation technique, and an environment, in which the tool is integrated, that provides comprehensive support for regression testing.
- Two preliminary empirical studies that compare our test-suite augmentation technique to existing techniques and show the potential effectiveness of our technique in revealing regression errors.

2 Computation of Testing Requirements

This section provides details of our test-suite augmentation approach. We first describe our change-based criteria for testing changed programs and illustrate them with an example. We then present our algorithm to compute testing requirements that meet this change-based criteria.

2.1 Change-based Criteria

Ideally, a criterion for adequately testing changes between P and P' should guarantee that the effects of the changes that can propagate to the output actually propagate, such that the effect of the changes will be revealed. Such an approach can be seen as an application of the PIE model [27] to the case of changed software: the criterion should ensure that the change is executed (E), that it infects the state (I), and that the infected state is propagated to the output (P). However, generating testing requirements for such a criterion entails analyzing the execution of P and P' (e.g., using symbolic evaluation) from the change until the program terminates, which is impractical.

To make the approach practical, while maintaining its effectiveness, we define a set of criteria, each of which ensures that P and P' are in different states after executing statements at a specific distance from the change (i.e., it ensures that the effects of the change have propagated to these statements). The distance, expressed in terms of data- and control-dependence chains, provides the tester a way to balance effectiveness and efficiency of the criterion. On the one hand, criteria involving longer distances are more expensive to compute and satisfy than those involving shorter distances. On the other hand, criteria involving longer distances ensure that states farther away from the change differ and, thus, that the effects of the change have propagated *at least* to those points in the program. Intuitively, requiring this propagation increases the likelihood that different (possibly erroneous) behaviors due to the changes will be exercised and revealed. Note that, in some cases, it may not be possible to propagate the effects of a change beyond a given distance d. For example, imagine a change in the way an array is sorted; after the sorting is done, the states should be exactly the same in P and P'. In these cases, there would be no testing requirements involving a distance d or higher because no different states in P and P' could be generated. We discuss this aspect in more detail in Section 2.2, where we present our algorithm for computing testing requirements.

As stated above, we define *distance* in terms of data- and control- dependences. A statement has *distance 1* from a change if it is data- or control- dependent on that change (i.e., a statement that contains a use of the variable defined at a modified statement or a statement that is control-dependent on a modified predicate). A statement has *distance n* from a change if it is data- or control- dependent on a statement that has distance $n - 1$ from the change. Note that the change itself is considered to have *distance 0*, that is, requirements defined for distance 0 refer to the state immediately after the changed statement is executed.

For each change and distance value, our algorithm generates a set of testing requirements that must be met to satisfy our criterion at that particular distance. The testing requirements are represented as boolean predicates, expressed in terms of the constants and values of the variables at the point immediately before the change in P'. For example, consider version E_{c1} of class E (Figure 1). The testing requirement generated for distance 0 is a predicate $i_0 \neq i_0 + b_0$ (i.e., $b_0 \neq 0$, when simplified), where i_0 and b_0 are the values of variables i and b, respectively, at the point immediately before executing $s1$ and $c1$. This predicate ensures that the states of E and E_{c1} differ after executing $s1$ and $c1$.

```
COMPUTEREQS()
Input:   P, P': original and modified versions of the program, respectively
         change: pair (c in P, c' in P') of changed statements
         requested_distance: dependence distance requested
Output:  reqs: set of testing requirements, initially empty
Use:     match(n') returns a statement in P that corresponds to n' in P'
         def(n) returns the variable defined at statement n if any, or null
         FDD(n, d, P) returns set of statements dependent on n
         PSE(c, n, P) returns program state at n
         TRI(S, S') returns set of testing requirements
Declare: affected, next_affected: sets of pairs of affected statements
         s, s' : statement in P and P', respectively
         n, n' : statement in P and P', respectively
         S, S' : program states

(1)  affected= {change} // Step 1: Identify affected parts of P'
(2)  while requested_distance-- > 0
(3)      next_affected = ∅
(4)      foreach (s, s') ∈ affected
(5)          foreach n' ∈ FDD(s', def(s'), P')
(6)              n = match(n')
(7)              next_affected = next_affected ∪(n, n')
(8)          endfor
(9)      endfor
(10)     affected = next_affected
(11) endwhile
(12) foreach (s, s') ∈ affected // Step 2: Compute testing requirements
(13)     S = PSE(c, s, P); S' = PSE(c', s', P')
(14)     reqs = reqs ∪ TRI(S, S')
(15) endfor
(16) return reqs
```

Figure 2. Algorithm to compute testing requirements.

2.2 Algorithm

Our algorithm for computing testing requirements for a change in a program, `ComputeReqs` (shown in Figure 2), takes three inputs: P and P', the original and modified versions of the program, respectively; $change$, a pair (c, c') of statements where c' is modified in, added to, or deleted from P'; and $requested_distance$, the dependence distance for which the testing requirements are generated. A new (deleted) statement is matched to a dummy statement in P (P'). If the new (deleted) statement contains a definition of a variable v, then the algorithm adds the dummy statement $v = v$ in P (P').[1] (This is needed to ensure the correct behavior of our algorithm.) Otherwise, the dummy statement is a simple *no-op*. Analogously, a new (deleted) branching statement is matched to a dummy branching statement in P (P') with the same control dependent region and predicate $true$. After the dummy statements have been introduced, new and deleted statements are simply treated as modified statements.

[1] Without loss of generality, we assume that a single statement can define only one variable. Any statement that defines more than one variable can be transformed into a number of statements with one definition each.

stmt	C	V	C'	V'
s1	C_1: True	$V_1 : \{i = i_0, x = i_0\}$	C'_1: True	$V'_1 : \{i = i_0, x = i_0 + b_0, b = b_0\}$
s2	$C_1 : i_0 > 5$	$V_1 : \{i = i_0, x = i_0\}$	$C'_1 : i_0 + b_0 > 5$	$V'_1 : \{i = i_0, x = i_0 + b_0, b = b_0\}$
	$C_2 : i_0 \leq 5$	$V_2 : \{i = i_0, x = i_0\}$	$C'_2 : i_0 + b_0 \leq 5$	$V'_2 : \{i = i_0, x = i_0 + b_0, b = b_0\}$
s3	$C_1 : i_0 > 5$	$V_1 : \{i = i_0, x = 5/(i_0 - 5)\}$	$C'_1 : i_0 + b_0 > 5$	$V'_1 : \{i = i_0, x = 5/(i_0 + b_0 - 5), b = b_0\}$
s4	$C_1 : i_0 > 5$	$V_1 : \{i = i_0, x = (5/(i_0 - 5)) - 1\}$	$C'_1 : i_0 + b_0 > 5$	$V'_1 : \{i = i_0, x = (5/(i_0 + b_0 - 5)) - 1, b = b_0\}$
	$C_2 : i_0 \leq 5$	$V_2 : \{i = i_0, x = i_0 - 1\}$	$C'_2 : i_0 + b_0 \leq 5$	$V'_2 : \{i = i_0, x = i_0 + b_0 - 1, b = b_0\}$

Table 1. Path conditions and symbolic states at s1-s4 of E and E_{c1}, the version of E with modification $c1$

`ComputeReqs` outputs $reqs$, a set of testing requirements that must be met to satisfy the criterion at the requested distance. `ComputeReqs` uses five external functions: $match(n')$ returns the statement in P that corresponds to n' in P'; $def(n)$ returns the variable defined at statement n or $null$ if n does not define a variable; $FDD(n, d, P)$, $PSE(c, n, P)$, and $TRI(S, S')$ are explained below.

`ComputeReqs` consists of two main steps: identification of the affected statements in P and P' at the requested distance and computation of the testing requirements, which correspond to the conditions under which the change induces different states, in P and P', after the execution of the affected statements.

In the first step, `ComputeReqs` initializes *affected*, a set of pairs of affected statements, to the pair of changed statements (line 1). For each iteration of the while loop, `ComputeReqs` computes the affected statements one more dependence distance away from the change by computing forward direct dependences for each affected statement at the current distance (line 5). *Forward Direct Dependence (FDD)* identifies the statements control-dependent on the input statement, s', or data-dependent on the definition of variable $def(s')$ at s'. For each affected statement n', `ComputeReqs` finds a matching statement n in P (line 6), forms a pair of affected statements (n, n'), and adds the pair to set *next_affected* (line 7). For example, to compute affected statements at distance 1 from $s1$ in E and version E_{c1} of E (see Figure 1), `ComputeReqs` calls $FDD(c1, x, E_{c1})$, which returns statements $s2$, $s3$, and $s4$. (FDD does not include statements $s5$, $s6$, and $s7$ because the analysis can establish that statement $s4$ kills the definition of x at $s1$.) `ComputeReqs` then finds a match for each statement that FDD returns. In our example, `ComputeReqs` would identify ($s2, s2'$), ($s3, s3'$), and ($s4, s4'$) as pairs of affected statements at distance 1 from the change, where $s2'$, $s3'$, and $s4'$ are statements in E_{c1} that correspond to statements $s2$, $s3$, and $s4$ in E, respectively. After `ComputeReqs` processes each pair of affected statements, it assigns *next_affected* to *affected*.

In the second step, `ComputeReqs` computes the testing requirements for the affected statements identified in the first step. To do this, the algorithm considers each pair of statements in *affected*. At each statement s in P (resp., s' in P'), `ComputeReqs` uses partial symbolic evaluation to identify the path conditions and symbolic states of s (resp., s'). *Partial Symbolic Evaluation (PSE)* is similar to global symbolic evaluation [8], except that the changed statement c is the starting point, all live variables at the changed statement are input variables, and s (resp., s') is the ending point. PSE differs from global symbolic evaluation in two respects. First, rather than considering all paths from program inputs to program outputs, PSE considers only finite subpaths from the change to s (resp., s') along dependence chains up to the desired distance. Second, instead of representing path conditions and symbolic states in terms of input variables, PSE expresses them in terms of constants and program variables representing the state of the program at the point immediately before the change. Analogous to global symbolic evaluation, PSE represents a program state with case expression. More formally, the program state at statement s, is defined as the set $\{(C_{s,i} : V_{s,i}) | i \geq 1\}$, where $C_{s,i}$ is the path condition to reach statement s from the change through path i, and $V_{s,i}$ is the symbolic state when $C_{s,i}$ holds. A symbolic state is represented as a set of variable assignments, $V_{s,i} = \{v_1 = e_1, v_2 = e_2, ...\}$, where e_i is the value of v_i expressed symbolically.

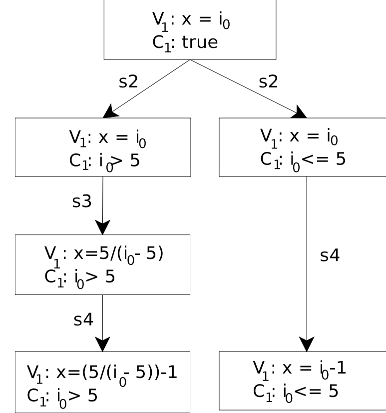

Figure 3. Symbolic execution tree for $s1$-$s4$ in E

For our example, $PSE(s1, s4, P)$ evaluates $s4$ in terms of i_0, the value of variable i at the point immediately before $s1$, on two paths: ($s1, s2, s3, s4$) and ($s1, s2, s4$). Figure 3 illustrates the symbolic execution tree from $s1$ to $s4$. Each rectangle represents a state in E, and each edge, labeled with a statement, represents the transformation from one state to another when that statement is executed. Note that, because $i = i_0$ at every point, we do not show i in the tree. From the tree, PSE returns $\{(i_0 > 5 : x = (5/(i_0 - 5)) - 1), (i_0 \leq 5 : x = i_0 - 1)\}$.

After each pair of affected statements at the requested distance is symbolically evaluated, `ComputeReqs` performs *Testing Requirement Identification (TRI)*, which compares each statement and its counterpart in terms of their path conditions and symbolic states to identify testing requirements. For each pair of corresponding statements (s, s') in P and P', TRI produces two testing requirements. These requirements guarantee that a test input satisfying either one of them would result in different states after executing s' (regardless of whether s is executed).

More precisely, the two testing requirements correspond to these *two conditions*: *(1)* the execution reaches statement s when running on P and statement s' when running on P', and the program states after executing those statements differ, or *(2)* the execution reaches statement s' when running on P' but does not reach statement s when running on P (i.e., it takes other subpaths that do not contain s). Using the representation of a program state, TRI can be described as follows. Let $S_s = \{(C_{s,i} : V_{s,i}) | 1 \leq i \leq m\}$ and $S_{s'} = \{(C_{s',j} : V_{s',j}) | 1 \leq j \leq n\}$ be the program states after executing statements s and s' in P and P', respectively, where $V_{s,i} = \{v_1 = e_1, v_2 = e_2, ...\}$ and $V_{s',j} = \{v'_1 = e'_1, v'_2 = e'_2, ...\}$. In the following, for simplicity, we abbreviate X_s with X and $X_{s'}$ with X' for any entity X. Condition (1) above can be expressed as $(C_i \wedge C'_j) \wedge R_{i,j}$ for all i, j, where clause $R_{i,j}$ evaluates to true if the program states differ after executing the statements on paths i and j. Because the program states differ if the value of at least one variable differs in the two states, $R_{i,j}$ can be expressed as the disjunction of the terms $(e_k \neq e'_k)$ for all k. Condition (2) can be expressed as $(C' \wedge \neg C)$, where $C' = \bigvee_{j=1}^{n} C'_j$ and $C = \bigvee_{i=1}^{m} C_i$.

Note that, to measure coverage of the generated requirements, the requirements need not be simplified or solved. Checking whether a test case satisfies any of these requirements can be performed by substituting each variable in a requirement with its value (obtained during the execution of the test case at the point immediately before the change) and evaluating the truth value of the requirement. Simplification and constraint solving are necessary only if we want to use the requirements to guide test-case generation or determine the requirements' feasibility.

As discussed above, there are changes whose effects do not propagate beyond a certain distance (see the array-sorting example provided in Section 2.1). In these cases, if the constraints corresponding to conditions (1) and (2) can be solved, they evaluate to *false*, which means that the corresponding requirements are unsatisfiable and testers do not need to further test the effects of that change.

For the example in Figure 1, the path conditions and symbolic states of $s1$-$s4$ in E and E_{c1} are shown in Table 1. When identifying testing requirements at distance 1, `ComputeReqs` computes the requirements necessary for revealing different states at $s4$ and $s4'$ by calling $TRI(S_{s4}, S'_{s4})$. The testing requirement that corresponds to condition (1) is:

$$((C_1 \wedge C'_1 \wedge R_{1,1}) \vee (C_1 \wedge C'_2 \wedge R_{1,2}) \vee (C_2 \wedge C'_1 \wedge R_{2,1}) \vee (C_2 \wedge C'_2 \wedge R_{2,2})), \text{ where}$$
$$R_{1,1} = (5/(i_0 - 5)) - 1 \neq (5/(i_0 + b_0 - 5)) - 1)$$
$$R_{1,2} = (5/(i_0 - 5)) - 1 \neq i_0 + b_0 - 1)$$
$$R_{2,1} = (i_0 - 1 \neq (5/(i_0 + b_0 - 5)) - 1)$$
$$R_{2,2} = (i_0 - 1 \neq i_0 + b_0 - 1).$$

The requirement can be simplified to:

$$(((i_0 > 5) \wedge (i_0 + b_0 > 5) \wedge (b_0 \neq 0)) \vee$$
$$((i_0 > 5) \wedge (i_0 + b_0 \leq 5) \wedge (((i_0 \neq 6) \vee (b_0 \neq -1)) \wedge ((i_0 \neq 10) \vee (b_0 \neq -9)))) \vee$$
$$((i_0 \leq 5) \wedge (i_0 + b_0 > 5) \wedge (((i_0 \neq 5) \vee (b_0 \neq 1)) \wedge ((i_0 \neq 1) \vee (b_0 \neq 9)))) \vee$$
$$((i_0 \leq 5) \wedge (i_0 + b_0 \leq 5) \wedge (b_0 \neq 0))).$$

The testing requirement that corresponds to condition (2) is $((C'_1 \vee C'_2) \wedge \neg(C_1 \vee C_2))$, which can be simplified to *false* (i.e., every execution reaching $s4'$ in P' reaches $s4$ in P). The testing requirements for changes at any distance can be generated using the same process described above.

Note that our algorithm, as currently defined, handles one isolated change at a time. Although the case of multiple changes (or a change involving multiple statements) could theoretically be handled by computing requirements for each change in isolation, we might need to adapt the algorithm to handle cases of interacting changes. Also, we do not explicitly consider multi-threading in our algorithm. We believe that the presence of multi-threading will not affect the generation of requirements directly, but it could complicate the symbolic-evaluation part of the approach.

3 Implementation

To evaluate our test-suite augmentation technique and conduct empirical studies, we developed a tool that partially implements our algorithm. In this section, we first discuss the regression-testing environment in which our implementation is integrated and then present the details of our tool.

3.1 Regression-testing Environment

Our regression-testing system is written in Java, operates on Java programs, and assists users in several regression-testing tasks. The tasks supported by the system are depicted in Figure 4. The system takes three inputs: P, the original version of a program; P', a modified version of P; and T, an existing test suite for P. The system outputs two test suites, T' and T'', that can be used to exercise the parts of *P'* affected by the changes.

We provide a high-level overview of how the different tasks inter-operate. Given P and P', a differencing technique, JDIFF [1], computes *change information*, summarized as the set of new, deleted, and modified statements. Our testing-requirements identification technique, implemented in the MATRIX IDENTIFIER, uses the *change information* to produce a set of testing requirements for the portions of the program affected by the changes. These testing requirements then guide the instrumentation of P', performed by the MATRIX INSTRUMENTER. The instrumented P' is executed with T', a subset of T selected by

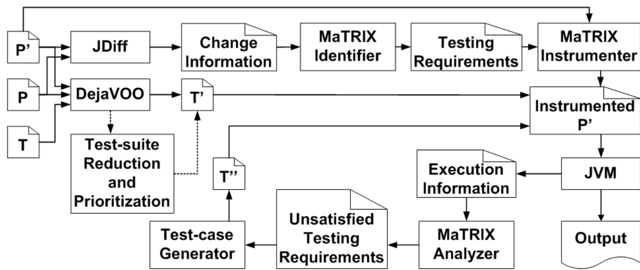

Figure 4. Our regression-testing environment.

DEJAVOO [19], a safe regression-test selection tool that also provides test-suite reduction and prioritization. When the instrumented P' is executed against T', it generates the necessary *execution information* for change coverage analysis, which is performed by the MATRIX ANALYZER. The result of this analysis is the set of testing requirements that have not been satisfied by T'.

A *test-case generator* module takes the set of *unsatisfied testing requirements* and produces a new set of test cases T'' aimed at satisfying them. The test-case generation, execution, and coverage analysis cycle continues until all testing requirements (or a user-specified percentage thereof) are satisfied. All parts of the regression-testing environment are automated except for the test-case generation, which is now performed manually. As discussed later, in Section 6, we are currently investigating how to use the testing requirements to automate, completely or in part, test-case generation.

3.2 MATRIX Toolset

The MATRIX toolset implements the three main components of our test-suite augmentation technique: the MATRIX IDENTIFIER identifies testing requirements related to the change from P to P'; the MATRIX INSTRUMENTER instruments P' so that, when it executes, it will record which testing requirements are satisfied; and the MATRIX ANALYZER examines the recorded information to determine which testing requirements have been satisfied.

A complete implementation of the MATRIX IDENTIFIER would require a full-fledged symbolic-evaluation engine for Java programs. To perform an early evaluation of our algorithm, we implemented a partial MATRIX IDENTIFIER, with no symbolic-evaluation capabilities, that identifies the subset of the change-related requirements corresponding to condition (2) (see Section 2.2). In other words, the MATRIX IDENTIFIER currently generates testing requirements that are satisfied when the execution follows different paths in P and P', but it does not generate requirements that require the state to be different after the execution of corresponding statements in P and P' (i.e., condition (1) in Section 2.2). Because only a subset of requirements is generated, the results obtained with this implementation of the MATRIX IDENTIFIER constitutes a lower bound for the actual performance of our algorithm. The MATRIX IDENTIFIER is implemented on top of JABA (Java Architecture for Bytecode Analysis[2]) and uses a JABA-based program slicer to compute data and control dependences. It inputs P, P', a change c, and the distance d for which requirements are to be generated and computes the subset of testing requirements discussed above for the set of statements at distance d from c.

The generated requirements are used by two components: the MATRIX INSTRUMENTER, which instruments the code to collect coverage of our requirements, and the MATRIX ANALYZER, which analyzes the coverage information produced by the instrumented program. Both components are implemented as plug-ins for INSECTJ [25], an instrumentation framework for Java programs.

4 Empirical Studies

We performed two empirical studies to evaluate effectiveness and cost of existing test-adequacy criteria and of our change-based criteria. To evaluate existing criteria, we extended MATRIX to compute and measure coverage of each criterion's testing requirements. In these studies, we used as subjects nine versions of two of the Siemens programs [15]: *Tcas* and *Schedule*. We chose *Tcas* and *Schedule*, two arguably small programs, because we wanted to have complete understanding of the subjects' internals to be able to thoroughly inspect and check the results of the studies. Moreover, selecting two small subjects let us use random test case generation to create suitable test suites for the studies. Because *Tcas* and *Schedule* were originally written in C, and our tool works on Java programs, we converted all versions of *Tcas* and *Schedule* to Java.

The Java versions of *Tcas* have two classes, 10 methods, and 134 non-comment LOC. The Java versions of *Schedule* have one class, 18 methods, and 268 non-comment LOC. *Schedule* requires some of the C standard library, which results in 102 additional LOC when converted to Java. In the studies, we use one base version (v0) and four modified versions (v1-v4) of *Tcas* and one base version (v0) and five modified versions (v1-v5) of *Schedule*. The changes in the modified versions are faults seeded by Siemens researchers, who deemed the faults realistic based on their experience.

In both studies, we measure the effectiveness of a criterion as the ability of test suites that satisfy the criterion to reveal different behaviors in the old and new versions of a program. To obtain this measure, we first pair a modified version (P') with its base version (P). We then identify the types and locations of changes between P and P' using JDIFF [1] and feed the change information to MATRIX IDENTIFIER to generate a set of testing requirements. We next use MATRIX INSTRUMENTER to instrument P' based on the generated requirements. Executing the instrumented P' against a test suite generates the information that is used by MATRIX ANALYZER to determine which testing requirements are satisfied by that test suite.

[2] http://www.cc.gatech.edu/aristotle/Tools/jaba.html

To create coverage adequate test suites for the different criteria considered, we proceeded as follows. For each modified version of the subject programs and each criterion, we built 50 coverage-adequate test suites by generating random test cases and selecting only test cases that provided additional coverage over those already added. We used a 30-minute time limit for the random generation: if the generator did not create a test input that covered additional testing requirements for 30 minutes, we stopped the process and recorded only the test cases generated thus far. To be able to generate randomly a sufficiently large number of coverage-adequate test suites, we limited the maximum distance to two (i.e., we created test suites for distances zero, one, and two). We measured the effectiveness of a criterion by counting the number of test suites for that criterion that contained at least one test case showing different behaviors in P and P'. As a rough approximation of the cost of a criterion, we used the number of test inputs in the test suites satisfying that criterion.

Threats to validity. The main threat to external validity is that our studies are limited to two small subjects. Moreover, these subjects were originally written in C, so they do not use object-oriented features such as inheritance and polymorphism. Therefore, the results may not generalize. Another threat to external validity is that the test suites used in the studies may not be a representative subset of all possible test suites. Threats to internal validity concern possible errors in our implementations that could affect outcomes. Nevertheless, we carefully checked most of our results, thus reducing these threats considerably.

4.1 Study 1

The goal of this study is to evaluate the effectiveness and cost of existing criteria for testing changes. The test-adequacy criteria we consider are statement and all-uses data-flow criteria. As discussed in Section 2, we define these criteria for modified software: the statement adequacy criterion is satisfied if all modified statements are exercised. For the all-uses data-flow adequacy criterion, we expand the criterion into a set of criteria, each of which requires du-pairs up to a specific dependence distance from the changes to be exercised. More precisely, the *all-uses distance-0* criterion requires all du-pairs containing modified definitions to be exercised; and the *all-uses distance-n* criterion requires the du-pairs whose definitions are control- or data-dependent on the uses of du-pairs at distance $n-1$ to be exercised.

To measure the effectiveness and cost of each criterion, we followed the process described earlier. Note that, for each of the all-uses distance-i criterion, where $i \geq 1$, we built the 50 test suites starting from the test suite satisfying the all-uses distance-$(i-1)$ criterion, rather than generating them from scratch.

Tables 2 and 3 show the percentage of test suites revealing different behaviors over all test suites satisfying statement and all-uses data-flow adequacy criteria, respectively (e.g., Table 3 shows that, for *Schedule* v2, only 16% of test suites satisfying all-uses distance-0 criterion reveal different behaviors). The data in the tables shows that, in all but one case, 22% or less of the test suites satisfying the statement adequacy criterion will reveal different behaviors. In the case of the all-uses distance-i adequacy criterion, $0 \leq i \leq 2$, the data also shows that the all-uses distance-2 adequacy criterion is adequate for *Tcas* v2. However, none of the all-uses distance-i adequacy criteria, $0 \leq i \leq 2$, is adequate for *Schedule* because the average percentage of test suites revealing different behaviors is only 16.8%. The results confirm our intuition that all-uses adequate test suites are more effective in revealing different behaviors than statement adequate test suites, and that the longer the dependence distances considered, the more effective the criteria become. However, the results also show that, in many cases, these test-adequacy criteria do not effectively exercise changes.

To measure the cost of generating a test suite satisfying the existing test-adequacy criteria, we measure the average size of the test suites we created. The size of all test suites satisfying the statement-adequacy criterion for any changes is 1. (Therefore, we do not show this result in the tables.) Table 4 shows the average number of test cases in test suites that satisfy an all-uses distance-i criterion for $0 \leq i \leq 2$. For example, the average size of the test suites satisfying all-uses distance-1 adequacy for the changes in *Tcas* v1 is 1.24. The data shows that the average size of the test suite satisfying any of the all-uses adequacy criteria is 3.00 or below in most cases, with the exception of the changes in *Tcas* v3 at distances 1 and 2, which is 4.22. Overall, the results show that the cost of generating test suites satisfying data-flow adequacy criteria considering only du-pairs that

version	v1	v2	v3	v4
% diff-revealing suites	2	14	22	40

Tcas

version	v1	v2	v3	v4	v5
% diff-revealing suites	0	14	20	10	0

Schedule

Table 2. Percentage of test suites revealing different behaviors over 50 test suites that satisfy the statement adequacy criterion for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	0	4	12
v2	6	6	100
v3	18	68	68
v4	80	94	94

Tcas

ver	distance		
	0	1	2
v1	0	0	0
v2	16	30	50
v3	14	30	32
v4	12	30	38
v5	0	0	0

Schedule

Table 3. Percentage of test suites revealing different behaviors over 50 test suites that satisfy all-uses distance-i adequacy criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	1.00	1.24	2.22
v2	1.00	1.00	3.00
v3	1.14	1.80	1.80
v4	2.74	4.22	4.22

Tcas

ver	distance		
	0	1	2
v1	1.00	1.54	1.78
v2	1.00	1.68	2.56
v3	1.00	1.68	2.10
v4	1.00	2.08	2.38
v5	1.34	1.44	1.68

Schedule

Table 4. Average number of test cases in test suites that satisfy all-uses distance-i adequacy criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

are only a few dependences away from the changes is not much higher than the cost of generating test suites satisfying the statement adequacy criterion.

We can also use these data to compute a measure of cost-effectiveness of the criteria, by computing the ratio of the percentage of test suites revealing different behaviors to the average size of the test suites. For example, for the all-uses distance-0 and distance-1 adequacy criteria for *Tcas* v3, the ratios are 15.79 (18/1.14) and 37.78 (68/1.8), respectively. The results show that, for the subjects and versions considered, the cost-effectiveness for the all-uses-based criteria tends to increase with the distance.

4.2 Study 2

The goal of this study is to evaluate the effectiveness and the cost of our change-based criteria. We use the same effectiveness and cost measures as in Study 1 and also follow the same process.

Table 5 shows the percentage of test suites revealing different behaviors for each of our distance-i criteria and for each version of our subjects. As the data shows, our change-based criteria are more effective than the corresponding all-uses criteria—and much more effective than the statement adequacy criterion—for distances greater than zero. (They are more effective in most cases also for distance 0.) In particular, for *Tcas*, between 90% and 100% of the test suites that satisfy the distance-2 requirements reveal different behaviors between old and modified versions of the program. The results for *Schedule* are not as good from an absolute standpoint, but are still considerably better than the results for the corresponding all-uses criteria.

Note that, for changes in Schedule v1 and v5, none of the test suites that satisfy our criteria reveals different behaviors. After inspecting the subjects, we discovered that the changes in these versions affect the program state but not the control- and data-flow of the program. Criteria based on control- or data-flow are therefore unlikely to reveal these changes, as the results for the statement- and all-uses-based criteria show (see Tables 2 and 3). The reason why our technique does not reveal the difference either is that its current implementation does not generate requirements to exercise differences in the program state, as discussed in Section 3.2.

ver	distance		
	0	1	2
v1	30	30	90
v2	4	100	100
v3	100	100	100
v4	100	100	100

Tcas

ver	distance		
	0	1	2
v1	0	0	0
v2	10	48	94
v3	16	64	82
v4	36	56	60
v5	0	0	0

Schedule

Table 5. Percentage of test suites revealing different behaviors over 50 test suites that satisfy our distance-i criteria ($0 \leq i \leq 2$) for *Tcas* and *Schedule*.

ver	distance		
	0	1	2
v1	1.00	1.00	1.80
v2	1.00	1.96	1.96
v3	1.70	1.70	1.70
v4	3.76	3.94	4.88

Tcas

ver	distance		
	0	1	2
v1	1.88	3.44	3.44
v2	1.00	1.84	4.50
v3	1.00	2.08	3.42
v4	1.50	2.38	3.20
v5	1.58	2.44	2.64

Schedule

Table 6. Average number of test cases in test suites that satisfy our distance-i criteria for $0 \leq i \leq 2$ and for modified versions of *Tcas* and *Schedule*.

Table 6 shows the average number of test cases in test suites that satisfy each of our distance-i criteria for each subject version. The results show that our set of criteria needs at most (for *Schedule* v1 and distance 1) about twice as many test cases as the all-uses adequacy criterion at the same distance. Note that, because the test suites for longer distances are built on those for lower distances, and they are not reduced, the number of test cases per test suite for longer distances (for both our change-based criteria and the all-uses criteria) may not accurately reflect the actual test-suite generation costs. This explains why, in some cases, all-uses adequacy criteria require more test cases than our change-based criteria for the same distance and the same subject (e.g., for *Tcas* v1 and distance 2).

In terms of cost-effectiveness, our criteria are more cost-effective than both statement-based and all-uses-based criteria in most cases. (In the following, we do not consider v1 and v5 of *Schedule*, for which none of the criteria generate test cases that can reveal changes in behavior.) For distances greater than zero, our criteria are more cost-effective than the alternative criteria in all but one case (*Tcas* v4). For distance 0, our criteria are more cost-effective in eight out of 14 cases.

The results of our preliminary studies are encouraging and promising, especially if we consider that we obtained them with a partial implementation of the approach that computes only a subset of requirements. Overall, the results show that our approach can be quite effective in computing requirements for changed software and seems to outperform, in most cases, alternative existing techniques.

5 Related Work

Many existing techniques are related to our test-suite augmentation approach. A first class of related techniques shares our goal of creating testing requirements based on program changes. Binkley [4] and Rothermel and Harrold [21] use System Dependence Graph (SDG) based slicing [14] to select testing requirements based on data- and control-flow relations involving a change. SDG-based techniques typically do not scale due to the memory and processing costs of computing summary edges [2]. Gupta and colleagues [11] propose a technique that is still based on slicing, but uses an on-demand version of Weiser's slicing algorithm [29] and avoids the costs associated with building SDGs. Their technique computes chains of control and data dependences from the change to output statements, which may include a considerable part of the program and are likely to be difficult to satisfy. Our technique differs from these previous efforts in both efficiency and effectiveness. In terms of efficiency, our technique computes dependences only up to a given distance from the change, which considerably constrains its costs—our initial results suggest that a short distance may be enough to generate effective requirements. In terms of effectiveness, in addition to considering control- and data-flow related to a change, our technique incorporates path conditions and symbolic states, related to the impact of the change, into our testing requirements. Our empirical studies show that, for the subjects considered, test suites satisfying these testing requirements have a higher likelihood of revealing errors than those that are based only on control- and data-flow (even when output-influencing data-flow chains [9, 20, 17] are considered).

A second class of techniques computes testing requirements for changed software in terms of program entities: control-flow entities or data-flow entities (e.g. [10, 16, 18]). Among these techniques, the most closely related to ours is Ntafos's [18] required-elements (k-tuples of def-use associations). This technique relies on additional tester-provided specifications for inputs and outputs to detect errors unlikely to be discovered by definition-use-association coverage alone. Our technique imposes stronger conditions than definition-use-tuple coverage, thus resulting in more thorough testing criteria. Furthermore, our technique automates the computation of the conditions and, thus, does not rely on possibly error-prone and incomplete specifications from the tester.

A third class of related techniques incorporates propagation conditions into their testing requirements. In their RELAY framework, Richardson and Thompson [20] describe a precise set of conditions for the propagation of faults to the output, using control- and data-flow. Morell [17] also builds a theory of fault-based testing by using symbolic evaluation to determine fault-propagation equations. These techniques do not target changed software and, moreover, rely on symbolic evaluation of an entire program, which is impractical for large software. Our technique, in contrast, constrains complexity by limiting the generation of testing requirements to the selected distance and incorporating conditions that guarantee propagation up to that distance. Furthermore, to compute the adequacy of a test suite, our technique does not need to solve such conditions but just check whether they are satisfied at runtime.

A fourth class of related techniques augments existing test suites to strengthen their fault-revealing capability. Harder and colleagues [12] introduce operational coverage, a technique based on a model of the behavior of methods. Whenever a candidate test case refines an operational abstraction (i.e., invariant) of a method, the candidate is added to the test suite. Bowring and colleagues [5] present another behavior-based technique that builds a classifier for test cases using stochastic models describing normal executions. These techniques do not provide an adequacy criterion for evaluation of test suites, but rather a means to classify and group test cases. Moreover, they do not perform any kind of change-impact propagation. Overall, these techniques are mostly complementary to our criteria.

6 Conclusions and Future Work

In this paper, we presented our technique for test-suite augmentation. The technique can be used to determine whether an existing regression test suite adequately exercises the changes between two versions of a program and to provide guidance for generating new test cases that effectively exercise such changes. We also presented MATRIX, a prototype tool that partially implements our technique, along with a description of the regression-testing environment in which MATRIX is integrated.

Although our empirical evaluation is preliminary in nature, and its results may not generalize, it provides initial evidence that our technique is more effective (and often cost-effective) than existing test-adequacy criteria in exercising the effects of software changes. Encouraged by these results, we are currently investigating several research directions.

First, as we described in Section 3, our current tool is a partial implementation of the `ComputeReqs` algorithm. This implementation lets us perform initial studies on the effectiveness of our technique, but we require a complete implementation to perform a more extensive evaluation and further studies. We are extending our implementation by integrating a customized version of the symbolic-evaluation engine provided by Java PathFinder (http://javapathfinder.sourceforge.net/) into MATRIX.

Second, our empirical evaluation shows the potential effectiveness of our technique only on two subjects of limited size and with no object-oriented features such as inheritance, polymorphism, and exception handling. Also, the evaluation does not measure the expense of the symbolic-evaluation component of our algorithm. Although our tech-

nique requires symbolic evaluation on a small part of the program (the part close to the changes), and we expect that it will scale to large programs, evaluation on more and larger subjects is needed to assess the actual applicability of the approach. With the new implementation of MATRIX, we will conduct studies on larger subjects that have object-oriented features and evaluate the efficiency of our approach. We will also empirically investigate the efficiency and effectiveness tradeoffs that arise when different thresholds for dependence distances are used.

Third, the technique presented in this paper is defined to operate on one change at a time. We are currently investigating the effectiveness of our technique in the presence of multiple, interacting changes. We will suitably extend and adapt our approach based on the results of this investigation.

Finally, we believe that the information gathered during symbolic evaluation can be leveraged to support developers in generating test-cases that satisfy the change-related requirements. We will investigate how to extend the current technique and tool so that it can semi-automatically generate test cases based on our testing requirements. This component, together with a full implementation of MATRIX, will complete our regression-testing system.

Acknowledgements

This work was supported by Tata Consultancy Services and partly by NSF awards CCR-0205422, CCR-0306372, and SBE-0123532 to Georgia Tech.

References

[1] T. Apiwattanapong, A. Orso, and M. J. Harrold. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE 2004)*, pages 2–13, September 2004.

[2] D. C. Atkinson and W. G. Griswold. Implementation techniques for efficient data-flow analysis of large programs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, pages 52–61, November 2001.

[3] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[4] D. Binkley. Semantics guided regression test cost reduction. *IEEE Transcations on Software Engineering*, 23(8):498–516, August 1997.

[5] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 195–205, July 2004.

[6] Y. F. Chen, D. S. Rosenblum, and K. P. Vo. Testtube: A system for selective regression testing. In *Proceedings of the 16th International Conference on Software Engineering*, pages 211–222, May 1994.

[7] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, January 2000.

[8] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *Journal of Systems and Software*, 5(1):15–35, February 1985.

[9] E. Duesterwald, R. Gupta, and M. L. Soffa. Rigorous data flow testing through output influences. In *Second Irvine Software Symposium* , pages 131–145, March 1992.

[10] P. Frankl and E. J. Weyuker. An applicable family of data flow criteria. *IEEE Transactionson Software Engineering*, 14(10):1483–1498, October 1988.

[11] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *Journal of Software Testing, Verification, and Reliability*, 6(2):83–111, June 1996.

[12] M. Harder, J. Mallen, and M. D. Ernst. Improving test suites via operational abstraction. In *Proceedings of the 25th IEEE and ACM SIGSOFT International Conference on Software Engineering (ICSE 2003)*, pages 60–71, May 2003.

[13] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Transactions on Software Engineering and Methodology*, 2(3):270–285, July 1993.

[14] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(1):26–60, January 1990.

[15] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th international conference on Software engineering (ICSE 94)*, pages 191–200, 1994.

[16] J. W. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, 9(3):347–54, May 1983.

[17] L. Morell. A Theory of Fault-Based Testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[18] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, SE-10:795–803, November 1984.

[19] A. Orso, N. Shi, and M. J. Harrold. Scaling regression testing to large software systems. In *Proceedings of the 12th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE 2004)*, pages 241–252, November 2004.

[20] D. Richardson and M. C. Thompson. The RELAY model of error detection and it application. In *ACM SIGSOFT Second Workshop on Software Testing, Analysis and Verification*, pages 223–230, July 1988.

[21] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*, pages 169–184, August 1994.

[22] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology*, 6(2):173–210, April 1997.

[23] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault-detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43, November 1998.

[24] G. Rothermel, R. Untch, C. Chu, and M. Harrold. Test Case Prioritization. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[25] A. Seesing and A. Orso. InsECTJ: A Generic Instrumentation Framework for Collecting Dynamic Information within Eclipse. In *Proceedings of the eclipse Technology eXchange (eTX) Workshop at OOPSLA 2005*, pages 49–53, San Diego, USA, october 2005.

[26] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 97–106, July 2002.

[27] J. Voas. PIE:A Dynamic Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[28] F. Vokolos and P. Frankl. Pythia: A regression test selection tool based on text differencing. In *IFIP TC5 WG5.4 3rd internatinal conference on on Reliability, quality and safety of software-intensive systems*, pages 3–21, May 1997.

[29] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[30] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Proceedings of the 17th International Conference on Software Engineering*, pages 41–50, April 1995.