

Test Diversity*

Borislav Nikolik

Vidak Quality
9226 NW Bartholomew Dr.
Portland, OR 97229
(503) 201-7229
boris@vidakquality.com

October 1, 2004

Abstract

This paper describes a novel method for measuring the degree to which a set of test cases executes a given program in diverse ways with respect to the two fundamental programming concepts: control and data. Test Diversity is a method for measuring the variety of software control flow and data flow, comprising of four new measures: conditional diversity, data diversity, standard deviation of diversity, and test orthogonality. These closely-related measures could be used to evaluate the test effectiveness and the test-effort distribution of a test suite.

The Diversity Analyzer is a novel industrial-strength testing tool that can currently perform diversity analysis on software written in C/C++/C#/VB in Windows and .NET environments. The Diversity Analyzer is used to evaluate the fault-detection effectiveness of Test Diversity on various types of industrial projects.

Key Words: *testing tools, verification, theory, experimentation, conditional diversity, data diversity, standard deviation, diversity matrix, linear independence, orthogonality.*

1 Introduction

An important problem in software testing is generating test suites that maximize the probability of fault detection. Using different assumptions about the underlying fault distribution, many test adequacy criteria have been proposed in order to increase the chance of defect detection. For example, the mutation adequacy criteria are based on the competent programmer hypothesis, which assumes programmers develop almost perfect programs, containing a specific set of faults[5]. Control flow and data flow adequacy criteria are based on assumptions about faults occurring on interesting control and data paths through the program, such as c-use and def-use paths[22].

Rather than making some particular assumption about fault distribution that will not hold in general, the task of deciding in which control areas faults might be concentrated or dispersed is left to the tester, and Test Diversity could then be used as a tool for measuring if indeed the test cases

achieved the desired degree of control and data diversity. For instance, working under the assumption that the faults are dispersed in the control space of the program, test cases that yield concentrated control flows through the program have a lower probability of detecting all the faults in the program than test cases that yield dispersed control flows. Obviously, any fault-dispersion assumption, as any fault-distribution assumption, could never be verified simply because of course it could be true in some cases and false in others. However, even though a fault-distribution assumption may deny the very basic notion on which all testing criteria are based, which is fault concentration on specific paths, Test Diversity could be used as an additional tool for current coverage criteria to enhance test effectiveness. Furthermore, the flexible concept of Test Diversity is applicable in cases where a fault-dispersion assumption does not hold since in its limit diversity becomes convergence, where test cases converge on a particular point in the control and data space.

Test Diversity is based on the notion of test distribution, which is obtained by analyzing the frequency of executing program branches. The frequency of branch execution could be unevenly distributed for a particular test suite and a program. For example, a test suite could cause the *true* branches in the program to be more heavily exercised than the *false* ones. *Conditional Diversity* is a measure of the *distance* between the actual and the balanced frequency distribution, which gives the tester an insight into portions of code with high and low test concentration. A maximally skewed conditional diversity indicates a test suite that exercises all the branches in either *true* or *false* direction. A balanced test indicates a uniform distribution between the execution frequency of *true* and *false* branches in the code. This measure could be used per individual test case or per test suite on a cumulative basis. Conditional Diversity indicates the test-effort distribution of a test suite at the source code level. This measure is also an indication of control variation.

In general, for a given program, specification, and a test adequacy criterion there are a large number of test suites that satisfy the given adequacy criterion[10][7][4]. Typically, some of these suits detect faults while others do not, and there is no general procedure to decide *a priori* which ones do and which ones do not [18]. Since there are multiple sets of data that could cover a program with respect to the given criterion, ones

*©2004 Vidak Quality. All rights reserved.

that expose defects and others that do not, a program could be completely covered but, unfortunately, with data that does not expose defects[21][20].

Data Diversity is a measure of data variation among test cases, that is, a measure of the degree to which different data flows through the program on a set of test cases. Measuring data diversity would in general require collecting and analyzing huge portions of the memory at many points in the code, along the lines of what is currently attempted in some fault tolerant systems[1][14][9]. Control and data are tightly related with respect to test diversity. For example, branch selection in the code is governed by values of program variables, and vice versa, the values of program variables are governed by branch selection[16]. In particular, if two test cases result in different conditional diversities, then the internal data states involved in the test suites are different, in turn, resulting in different data flowing throughout the program. In effect, conditional diversity, which is simple to obtain, could be used to measure data diversity. However, since different conditional diversities is a sufficient condition for data diversity but not a necessary condition, different inputs or different intermediate data could result in same conditional diversities. Therefore, the term data diversity in this paper is limited in the sense that it refers to a sufficient condition for data variation. Conditional diversity is computed for each test case in a test suit, and these individual conditional diversities are used to compute data diversity based on the differences among the conditional diversities.

Given a set of test cases and a program under test, the test cases could result in control and data highly concentrated in specific regions of the control and data space. In the worst case, all of the test cases would be concentrated on the same control and data point, effectively representing the same test cases. *Standard Deviation* of the conditional diversities could be used as a measure of control dispersion given by a test suite. Conditional diversities are computed for each test cases in the test suite, and the standard deviation is computed on the set of these individual conditional diversities.

Test cases could give rise to specific control and data relations in the program under test, such as linearly dependent program control. Such dependencies in control and data might not be desirable when attempting to exercise the program in unrelated ways. There could however be many ways in which the control and data resulting from different test cases are be related. Test Diversity might be a tool for analyzing dependencies in control and data. To demonstrate this potential, the concept of *orthogonality* is introduced as a measure of linear independence of program control in testing.

The Diversity Analyzer is a novel, industrial-strength tool for measuring diversity. The tool calculates conditional diversity, data diversity, and standard deviation in order to aid testers in adjusting and balancing their testing. The tool gives a summary of diversity metrics per source files, as well as hot-spots view at the source-code level. The Diversity Analyzer can currently analyze C,C++,C# and Visual Basic source code in the Windows and .NET environment. The tool instruments the source code with branch-distribution collecting functions, which essentially count the number of times branches are taken and not taken. This distribution is used to calculate conditional diversity, data diversity, and standard

```

int index=0;
bool found=false;
1 while(c(index<length&&!found,0)) //e1
2   if(c(item==list[index++],1)) //e2
3     found=true;

```

Figure 1: Instrumented Sequential Search

deviation of conditional diversity.

Three different types of industrial projects have been analyzed in order to determine the fault-detection effectiveness of test diversity. Our initial experimental results show a relationship between the magnitudes of conditional diversity, data diversity, standard deviation for a set of test cases, and fault-detection effectiveness of the test cases.

2 Overall Example

A simple and well-known code fragment is used to demonstrate the central notions of this paper: test distribution, conditional diversity, data diversity, standard deviation of diversity, and linear independence of tests. The instrumented C++ code fragment in Figure 1 performs a sequential search of the *list* array on the element *item*. The uninstrumented version of this code fragment appears frequently in textbooks on algorithms [3]. The code fragment consists of two conditional expressions, one controlling the *while* conditional statement (denoted by e_1), and the other controlling the *if* conditional statement (denoted by e_2). The Diversity Analyzer parses the code fragment, locates the two conditional expressions e_1 and e_2 , and places a c function around e_1 and e_2 to count the number of times e_1 and e_2 evaluate to *true* and *false*, as shown in Figure 1. The second parameter of c denotes the sequential position of e_1 and e_2 in the code. Test Diversity applies to multi-branching expressions as well, such as a *switch* statement in C, or *select* statement in Basic. The counting function $c(true, i)$ is placed at the beginning of each branch in a multi-branch to count the number of times the branch got executed. The *false* hits for the branch are obtained by subtracting the hits for that branch from the total hits for the whole multi-branch.

Suppose the code fragment of Figure 1 was executed on the four test cases t_1, \dots, t_4 from Figure 3. The *true* execution count T and *false* execution count F for e_1 and e_2 and test cases t_1, \dots, t_4 are given in Figure 3. Conditional diversity for a conditional expression, such as e_1 or e_2 above, is calculated as a *distance* between the actual distribution for that expression from the uniform distribution for that expression, as shown in Figure 2. The average of *true* hits and *false* hits is denoted by A. The conditional diversity is denoted by D. The value of the conditional diversity is negated if the number of *false* hits F is greater than the number of *true* hits T. D is undefined when the conditional expression is never executed. However, in practice D is assigned a specific value outside of the $[-1, 1]$ range so that vectors containing elements reflecting un-exercised branches can be used in the usual vector arithmetic operations. The formulas of Figure 2 simplify to $D = (T - F)/(T + F)$ if $T + F > 0$ otherwise \perp . Note that

$$A = \frac{T + F}{2}$$

$$D = \begin{cases} \frac{|A-T|+|A-F|}{T+F} & \text{if } T \geq F \\ -\frac{|A-T|+|A-F|}{T+F} & \text{if } F > T \\ \perp & \text{if } T + F = 0 \end{cases}$$

Figure 2: Conditional Diversity

| | list, item | e_1T | e_1F | e_2T | e_2F |
|-------|---------------------------------|--------|--------|--------|--------|
| t_1 | $\langle 3, 1 \rangle, 0$ | 2 | 1 | 0 | 2 |
| t_2 | $\langle 3, 2, 7, 5 \rangle, 9$ | 4 | 1 | 0 | 4 |
| t_3 | $\langle 4, 3, 0 \rangle, 0$ | 3 | 1 | 1 | 2 |
| t_4 | $\langle 8, 3 \rangle, 8$ | 1 | 1 | 1 | 1 |

Figure 3: Execution Frequency Counts

conditional diversity gets higher as D approaches zero, and it gets lower as D approaches the extreme points 1 or -1 because $D = 0$ indicates balanced distribution and $D = 1 \vee D = -1$ indicates maximally skewed distribution. The Diversity Analyzer uses the frequency counts produced by the instrumented version of the code fragment to calculate the conditional diversities for e_1 and e_2 for t_1, \dots, t_4 using the formulas given in Figure 2.

Data diversity for a conditional expression e_p is the ratio d/n , where d is the number of unique conditional diversities given by t_1, \dots, t_n for e_p . If two test cases t_i and t_j give rise to different conditional diversities for e_p , then *different* data flows through e_p on t_i and t_j .

Standard deviation, when applied to diversity, is a measure of dispersion of conditional diversities around the conditional diversity mean. Higher standard deviation means more dispersed coverage of the control space. The conditional diversities, data diversities, and standard deviations for the code fragment of Figure 1, and test cases t_1, \dots, t_4 are given in Figure 4. The results from Figure 4 are interpreted as follows. The most balanced testing¹ is produced by t_4 and the most skewed testing is produced by t_2 . Since the standard deviations for both e_1 and e_2 are relatively low (when compared to the experimental range of $[.6, .8]$ of Section 6), the control space is covered in a concentrated rather than in a dispersed fashion. The data diversity for e_1 is $d/n = 4/4$ because there are $d = 4$ unique conditional diversities. The data diversity for e_2 is $d/n = 2/4$. The expression e_1 is tested with different data on each test case since the conditional diversities are different. The expression e_2 happens to also be exercised with different data, but there is a linear control pattern for e_2 on t_1 and t_2 ; the *true* and *false* hits for t_2 are two times the *true* and *false* hits for t_1 . That is, the vectors $[0 \ 2]^T$ and $[0 \ 4]^T$ lie on the same line.

In this particular case, all the lists of length four, such that the element searched for is at position two in the list, result in identical conditional diversities. Identical conditional di-

¹Balanced branch execution does *not* imply uniform path coverage since in general paths are not uniformly distributed along branches.

| | t_1 | t_2 | t_3 | t_4 | dd | std |
|-------|-------|-------|-------|-------|----|-----|
| e_1 | .3 | .6 | .5 | 0 | 1 | .23 |
| e_2 | -1 | -1 | -.3 | 0 | .5 | .19 |

Figure 4: Diversity Statistics

versities might indeed mean variety in path execution since the frequency count are cumulative and the order of execution is lost; however, identical conditional diversity should be avoided when attempting to test the program in unrelated ways.

The test cases t_1, \dots, t_4 are control dependent since the conditional diversities for e_1 and e_2 form a set of linearly dependent vectors, and as such are not free of control patterns.

3 Definitions

A *conditional expression* is an expression that occurs in conditional statements, such as *exp* in *if(exp)*, *while(exp)*, *for(;exp;)*, etc. An *expression vector* \vec{e} is a vector of conditional expressions $[e_1 \dots e_n]$, sequentially ordered in \vec{e} by their appearance in the source code. A *test case vector* \vec{t} is a vector of test cases $[t_1 \dots t_k]$. *Conditional diversity* for a conditional expression is a real value in the range $[-1, 1]$, computed using the formulas in Figure 2. As conditional-diversity values approach zero more diversity is obtained since such executions have more balance of *true* and *false* branches. *Conditional diversity vector* \vec{v} is a vector containing conditional diversities $[c_1 \dots c_n]$, where $c_i, 1 \leq i \leq n$, corresponds to e_i in \vec{e} .

Conditional diversity matrix C is a matrix whose k columns are conditional diversity vectors $[\vec{v}_1 \dots \vec{v}_k]$, where $\vec{v}_i, 1 \leq i \leq k$, corresponds to an execution of a test case t_i from the test case vector \vec{t} . *Conditional diversity mean vector* \vec{m} is a vector of real values $[m_1 \dots m_n]$, where $m_i, 1 \leq i \leq n$, is the arithmetic average of the corresponding c_i values in the columns of C . *Standard deviation vector* \vec{st} is a vector of real values $[s_1 \dots s_n]$, where

$$st_i = \sqrt{\frac{\sum_{j=1}^k c_{ij}^2}{k} - m_i^2}.$$

This is an iterative version of the usual standard deviation formula, given in statistics textbooks as $\sigma = E[(X - \mu)^2]$, where X is a random variable of the discrete type, and $\mu = E(X)$ is the arithmetic mean of the values of X .

Data diversity vector \vec{d} is a vector of real values, in the range $[0, 1]$, $[d_1 \dots d_n]$, where d_i is the ratio of unique to total corresponding c_i values in the columns of C .

The code fragment of Figure 1 and the test cases of Figure 3 give the results in Figure 5.

If only the trivial combination gives zero, so that $p_1\vec{u}_1 + \dots + p_k\vec{u}_k = 0$ only happens when $p_1 = p_2 = \dots = p_k = 0$, then the vectors $\vec{u}_1, \dots, \vec{u}_k$ are *linearly independent*. Otherwise they are linearly dependent, and one of them is a linear combination of the others. For example, two vectors are dependent if they lie on the same line. Three vectors are dependent if they lie in the same plane. A random choice of three vectors should produce

$$\vec{e} = \left[\begin{array}{l} \text{index} < \text{length} \&\& \text{found} \\ \text{item} == \text{list}[\text{index} + +] \end{array} \right]$$

$$\vec{t} = \left[\begin{array}{ll} \langle 3, 1 \rangle & 0 \\ \langle 3, 2, 7, 5 \rangle & 9 \\ \langle 4, 3, 0 \rangle & 0 \\ \langle 8, 3 \rangle & 8 \end{array} \right]$$

$$\vec{v}_1 = \begin{bmatrix} .33 \\ -1 \end{bmatrix} \quad \vec{v}_2 = \begin{bmatrix} .6 \\ -1 \end{bmatrix} \quad \vec{v}_3 = \begin{bmatrix} .5 \\ -.33 \end{bmatrix}$$

$$\vec{v}_4 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \vec{st} = \begin{bmatrix} .23 \\ .19 \end{bmatrix} \quad \vec{d} = \begin{bmatrix} 1 \\ .5 \end{bmatrix}$$

$$C = \begin{bmatrix} .3 & .6 & .5 & 0 \\ -1 & -1 & -.3 & 0 \end{bmatrix}$$

Figure 5: Definitions Example

linear independence. Two vectors \vec{x} and \vec{y} are *orthogonal* if $x_1y_1 + \dots + x_ny_n = 0$. The vectors $\vec{v}_1, \dots, \vec{v}_4$ from Figure 5 are linearly dependent and are not orthogonal.

The *program state* is a pair $\langle \vec{v}, \vec{s} \rangle$ of vectors in which each variable v_i of vector \vec{v} is associated with the corresponding value s_i of \vec{s} .² For example, in Figure 1, the program state before the first execution of line 2 on test case t_1 is

$$\langle [\text{list item index found}][\langle 3, 1 \rangle 0 0 \text{ false}] \rangle$$

The *total* program state is a vector \vec{S} of program states, that is, a vector containing all the states as they occur in the execution of the program. $\vec{S}_{t_i}^P$ denotes the total state resulting from executing the test t_i on a program P . We do *not* attempt the infeasible task of storing and analyzing program states. We only use this definition for demonstration purposes.

A control space of a program is the set of all complete feasible control paths in the program. A control region is a set of complete feasible control paths. A data space is the set of all program states. A data region is a set of program states.

4 Diversity Analysis

Diversity analysis involves measuring the degree of control and data variation among test cases at the source code level. Under a fault-distribution assumption, the quality of a test suite is considered to be better if its test cases cause more control and data diversity. Test Diversity is comprised of four individual measures, whose magnitudes are dependent on the particular source code and the particular test cases. The same set of test cases could result in an arbitrarily different diversity values on different code. And conversely, the same program could yield different diversities on different test cases. The four test diversity measures are interpreted as follows. Conditional diversity is a measure of test-effort distribution and control variation, data diversity is a measure of data variation, standard deviation of diversity is a measure of control

²We use this simplified definition of state, which excludes the heap, for presentation purposes.

and data dispersion, and orthogonality is a measure of test case independence. Next we describe these four measures in more detail.

4.1 Conditional Diversity

Conditional diversity allows the tester to find out which portions of the code the testing is *concentrated* on. This is a measure of test-effort distribution that can be obtained per test case, or cumulatively per test suite. A balanced conditional diversity, where the frequencies of *true* and *false* branch evaluations are uniformly distributed, implies a test suite that is *not* concentrated on particular portions of the code. When a set of test cases results in maximally skewed conditional distribution, all the test cases cause branches in the code to either be taken or not taken. In general, deciding what the target conditional diversity should be is left to the tester, who bases the distribution-adequacy decision on factors such as code size, code criticality, code complexity, etc. For example, one might want to concentrate testing on a newly implemented, untested code with the premise that faults are more likely to lurk in that code. In that case the target conditional diversity should be skewed towards the new code, and possibly balanced in the new code, with the combined effect of exercising the new code more heavily and in a uniform fashion.

As this example hint, conditional diversity allows two degrees of freedom, one to target a particular section of code, and the other to target the diversity within the targeted section of code. Skewing and balancing is an iterative process of creating test cases, executing the program on the test cases, and measuring, along the lines of what is currently done in traditional coverage practice[13]. The iteration stops when either the target diversity is achieved, or other factors, such as schedule pressures, dictate termination of testing and product release.

4.2 Data Diversity

The effectiveness of testing depends heavily on the internal data variation given by test cases. Low data variation indicates a test suite where similar or same data flows through the program as it is being executed on the test cases. High data variation indicates that different data flows through the program. The converse, where different data flows through the program, does not necessarily mean different control is involved in testing. The data variation concept is called data diversity, and is tightly related to conditional diversity.

Let C_1 and C_2 be conditional diversity matrices of k conditional diversity vectors. The following condition establishes the relation between conditional diversity and data variation.

$$\vec{d}(C_1) \neq \vec{d}(C_2) \Rightarrow \exists_{t_i, t_j} (\vec{S}_{t_i}^P \neq \vec{S}_{t_j}^P), 1 \leq i \leq k.$$

Clearly if $\vec{d}(C_1) \neq \vec{d}(C_2)$ then there exist at least two corresponding vectors t_i, t_j in C_1 and C_2 such that $\vec{v}_i \neq \vec{v}_j$. If $\vec{v}_i \neq \vec{v}_j$ then there is at least one corresponding element c_p in \vec{v}_i and \vec{v}_j on which \vec{v}_i and \vec{v}_j differ. If the program state before each execution of e_p is identical for t_i and t_j then e_p will evaluate to the same value for both t_i and t_j . But this would give identical conditional diversity values c_p . This is

impossible since \vec{v}_i and \vec{v}_j differ on c_p . Therefore, t_i and t_j result in at *least* one different program state in $\vec{S}_{t_i}^P$ and $\vec{S}_{t_j}^P$.

The above condition has a practical implication in that branch-execution frequency counts can be used to reason about data variation. Intuitively, the above condition states that different conditional diversities guarantee different data flowing through the program on test cases t_i and t_j . That is, there is at least one conditional expression in the code for which t_i and t_j give different *true* and *false* frequency counts. When a test suite Θ_1 has a higher data diversity than a test suite Θ_2 , Θ_1 has at least as many different data states as Θ_2 has. For example, in Figure 3, test cases t_1, \dots, t_4 result in at least as many different states for e_1 as for e_2 . This is so because the number of different conditional diversities gives a lower bound on the number of different states.

Note that same conditional diversities does not imply same input or intermediate data. For example, in Figure 1 these two test cases ($\langle 1, 2, 3, 4 \rangle, 3$) and ($\langle 5, 6, 7, 8 \rangle, 7$) result in identical conditional diversities. Furthermore, in deterministic programs, same test cases should give the same conditional diversities. Therefore, same conditional diversities could be caused by both same or different data. These problems are a symptom of the fact that conditional diversity is a sufficient but not a necessary condition for data variation.

4.3 Test Dispersion

The usual statistical interpretation of standard deviation is a measure of dispersion of a set of points. Test Diversity uses standard deviation of conditional diversity as a measure of dispersion of control. Let $\vec{st}(C)$ denote the standard deviation vector obtained from C . Let \vec{v}' denote the arithmetic mean of the elements of \vec{v} . If $\vec{st}(C_i)' > \vec{st}(C_j)'$ then the vectors of C_i are more dispersed than the vectors of C_j . For example, in Figure 3, $\vec{st}(C + v_2)' < \vec{st}(C + v_5)'$, where $v_5 = [1 \ -1]$, and the $+$ operation means add a vector as an additional column.

When the conditional diversity vectors are dispersed they reach more distant regions of the control space of the program. When the conditional diversity vectors are concentrated in a particular region other regions of the control space of the program go unexercised. Under a fault-dispersion assumption, the worse case occurs when all the conditional diversity vectors are identical, and as such, are concentrated in a restricted region in the control space. Such test cases, when apparently different at the user interface of the program, could mislead the tester into thinking that the program control is covered in a dispersed manner.

Given a set of more dispersed faults in the control space of the program, a more dispersed test suite has a higher probability of detecting all the defects in the code than a concentrated test suite. Let C_1 and C_2 be two conditional matrices containing the same number of rows and columns, that is, they represent same number of test cases. For example, assuming that faults are uniformly dispersed in the control space, if $\vec{st}(C_1)' > \vec{st}(C_2)'$ then the test cases of C_1 have a higher chance of detecting all the faults that the test cases of C_2 . If this is not the case, the faults have to be concentrated in some region where the test cases of C_2 are concentrated. This cannot be the case because, by assumption, faults are not concentrated in any particular region, but dispersed throughout

the control space of the program. This simple-minded argument hints at the fact that test diversity does not necessarily need to rely on the uniform fault-dispersion assumption. That is, if a tester makes an assumption about faults being concentrated in some region of the control space, \vec{st} could be used to measure the degree of concentration of the test in that area, with the mean value falling in the concentrated region. In that case, of course, a lower value for \vec{st} would indicate a better test suite. Test Diversity does allow flexibility in accomodating fault-dispersion assumptions, however, if the underlying fault dispersion/concentration assumption is wrong, the \vec{st} measure would be misleading.

4.4 Control Dependence and Patterns

Conditional diversity and data diversity is aimed at the problem of measuring control and data variation. Standard deviation of diversity is concerned with the problem of increasing the probability of fault detection under fault-dispersion assumptions. However, test suites may need to have other characteristics that are not related to measuring control variation, data variation, and measuring the likelihood of fault detection. When evaluating test suites it might be important to know how and if the test cases of the suite are related. For example, exploratory testing always seeks to execute the program on unrelated test cases in order to find new faults. Debugging might involve executing related test cases in order to gain insight into a particular fault. There could be many ways in which the control and data resulting from different test cases may be related. Test Diversity might be the tool for analyzing dependencies in control and data. For instance, Test Diversity could be used to measure the linear independence of program control in testing. Another possible application of Test Diversity is in analyzing control patterns. These two applications are discussed in more detail.

Let $\vec{v}_1, \dots, \vec{v}_k$ be conditional diversity vectors obtained by executing test cases t_1, \dots, t_k on some program. Let Σ^k be a k -dimensional vector space. If $\vec{v}_1, \dots, \vec{v}_k$ are orthogonal then they are also linearly independent in Σ^k . A well-known result from linear algebra states that a set of orthogonal vectors is linearly independent. The vectors $\vec{v}_1, \dots, \vec{v}_4$ of Figure 5 are not orthogonal, and they do not span Σ^4 . That is, every other vector in Σ^k cannot be expressed as a linear combination of $\vec{v}_1, \dots, \vec{v}_4$. To check any set of vectors $\vec{v}_1, \dots, \vec{v}_n$ for linear independence, form the matrix C whose n columns are the given vectors. If the system $Cp = 0$ has a solution $p \neq 0$ then the vectors are linearly dependent. Iterative methods for solving $Ax = b$, such as Gaussian elimination with worst case complexity of n^3 , are available.

Another way that test cases could be related is through control and data patterns. Consider for example the following control pattern, denoted by $t_i \otimes t_j$, and defined by the following condition

$$(T_i = T_j = 0 \vee F_i = F_j = 0) \vee (T_i = F_i \wedge T_j = F_j),$$

where T_i and F_i refer to the number of times an expression evaluates to *true* and *false* respectively, when executed on a test case t_i . For example, in Figure 3, *no* such control pattern exists among the test cases t_1, \dots, t_4 . The notion of

conditional diversity could be used to detect control patterns. In particular,

$$\vec{v}_i = \vec{v}_j \Rightarrow t_i \otimes t_k.$$

Obviously, if $\vec{v}_i = \vec{v}_j$ then \vec{v}_i and \vec{v}_j are identical in all corresponding elements. Applying the definition of conditional diversity for each element in \vec{v}_i and \vec{v}_j yields the following

$$\frac{|A_i - T_i| + |A_i - F_i|}{T_i + F_i} = \frac{|A_j - T_j| + |A_j - F_j|}{T_j + F_j},$$

which holds when

$$T_i = T_j = 0 \vee F_i = F_j = 0 \vee T_i = F_i \wedge T_j = F_j \vee T_i = T_j \wedge F_i = F_j.$$

For example, in Figure 3 no vectors are equal, and therefore, as a whole these vectors do not imply control patterns. However, \vec{v}_1 and \vec{v}_2 imply a control sub-pattern for e_2 , where the conditional distribution is maximally skewed in the *false* direction, that is, $e_2 T = 0$ for both t_1 and t_2 . As the example hints, a sub-pattern for t_i and t_j is defined as a control pattern for sets of corresponding elements in v_i and v_j .

Note that, since control and data are tightly related with respect to diversity, control patterns may imply data patterns. Data patterns are defined as the set of all the program states that cause the same control patterns. For example, in Figure 3 $t_1 \otimes t_2$ for e_2 happened to be characterized by all the test cases for which the search does *not* find the element *item* in the list *list*. It is *not* our objective to give such precise characterizations, which, in general, might be possible to obtain using diversity.

5 Diversity Analyzer

The Diversity Analyzer is an industrial-strength tool that can analyze projects written in C, C++, Visual Basic, and C# in the Windows and .NET environment. The tool is relatively easily extendable to Unix/Linux environments, and other languages, such as Java. The tool can handle mixed-language projects, multiple projects simultaneously, dlls, ocxs, and multi-tasking code. The Diversity Analyzer has been used to efficiently analyze industrial C++, C# and VB code of more than 500K lines of source code. The tool efficiently analyzes conditional diversity, data diversity, and standard deviation for conditional matrices of $100,000 \times 500$ and above. All of the diversity algorithms are of linear complexity, except data diversity, which is of $k^2 \times n$ worst-case complexity, where k is the number of test cases, and n is the constant size of the conditional diversity vectors, that is, the number of conditional expressions in the code. Diversity metrics are obtained with the Diversity Analyzer in five simple steps: instrument source code, build (compile and link) instrumented code, run tests, compute diversity metrics, and view diversity metrics. Next we describe these steps in more detail.

One or more computer software source files, which are part of one or many projects, one or many executables, and/or one or many libraries, written in potentially different programming languages are selected for diversity analysis. The Diversity Analyzer user interface for instrumenting source files is given in Figure 6. If testing of parts of code is desired, zooming-in is performed for obtaining fine diversity, where the

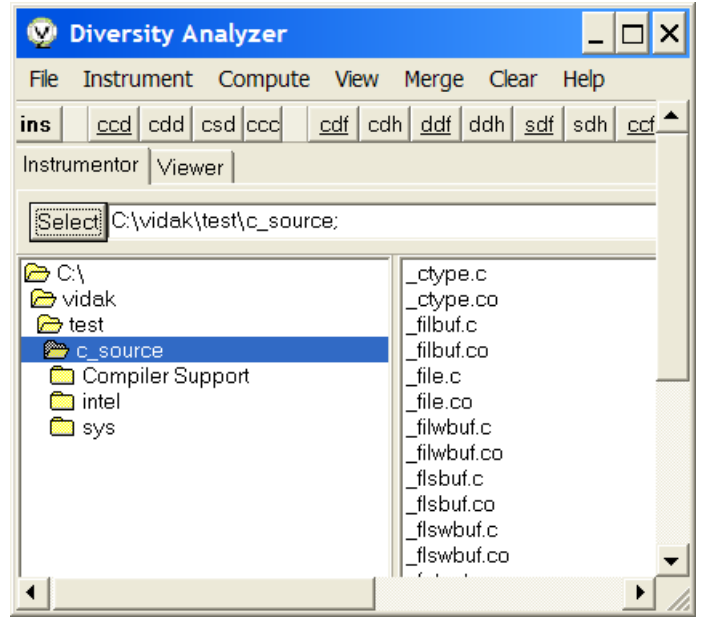


Figure 6: Instrumentor

parser only locates the conditional statements delimited by the zoom keywords: ZOOM.BEGIN and ZOOM.END. The tester may place these keywords in the code around sections of code of particular interest. After the parser locates a conditional statement, the instrumenter inserts a compact conditional distribution function call at that location in the code. The basic information about the conditional statement, such as the file name where it appears, the line number where it begins, and place holders for the number of times the conditional expression in that statement evaluates to *true* and *false*, are kept in a data structure which, at the end of the parse process, is permanently recorded. The instrumenter also places glue code in the source files to link the implementation of the conditional distribution function placed at the conditional statements. The instrumented source files are then compiled in their corresponding projects, and typically, executed many times with different test cases. The conditional distribution functions that were placed in the conditional statements keep track of the number of times conditional expressions evaluate to *true* and *false* by updating the permanent record of conditional statements, which were produced and initialized by the parse process. The permanent record for conditional statements that contains the *true* and *false* evaluation frequencies of the conditional expressions is used to calculate the conditional diversity vector at any point in the program execution. After the conditional vector is computed, it is placed as a column in the conditional diversity matrix. The user can then clear the permanent record, execute the next test case, and compute the next conditional diversity vector.

The diversity metrics could be viewed in a summary fashion given by source files, as shown in Figure 7, or in a per conditional expression fashion, sorted so that the tester has a quick view of the diversity hot spots. The Diversity Analyzer allows view of the source code, annotated with diversity metrics, as shown in Figure 8.

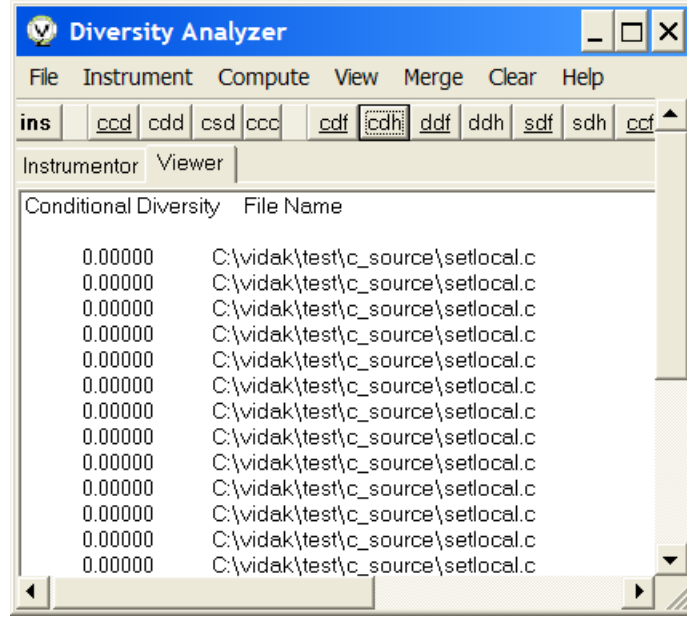
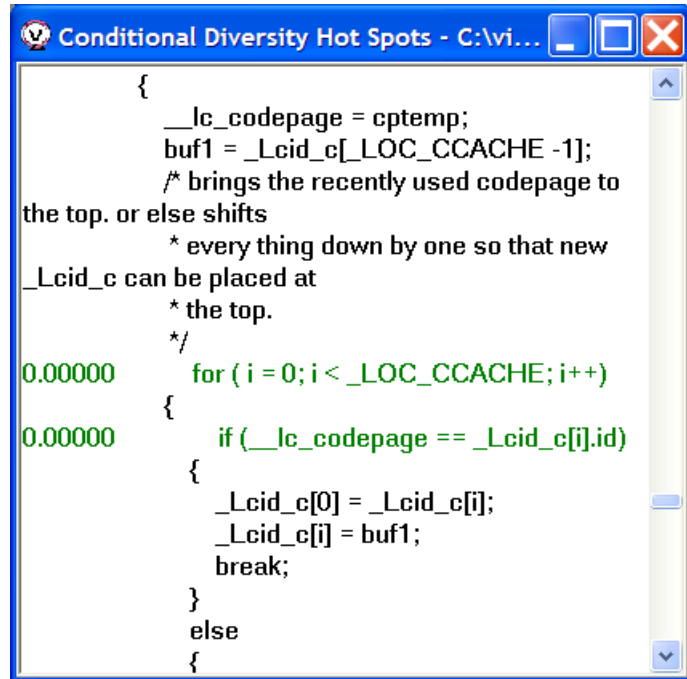


Figure 7: Viewer



| | # tests | coverage | # faults | cd | dd | std |
|-------|---------|----------|----------|-----|-----|------|
| p_1 | 850 | 63% | 38 | 0.5 | 0.2 | 0.1 |
| p_2 | 710 | 72% | 27 | 0.4 | 0.1 | 0.15 |
| p_3 | 690 | 68% | 21 | 0.6 | 0.3 | 0.09 |

Figure 9: Project Statistics

6 Experimental Results

The goal of the experimental evaluation was twofold: to see how the Diversity Analyzer would perform in an industrial environment with respect to speed and memory requirements, and to evaluate the fault-detecting effectiveness of test diversity. The Diversity Analyzer was used to analyze three commercial project written in C++, running under Windows: an accounting software, a printer driver, and a part-placement scheduling engine. The accounting software, denoted by p_1 , had about 300K lines of source code, with total of 30K conditional expressions. The printer driver software, denoted by p_2 had about 25K lines of source code, with a total of 6K conditional expressions. The scheduling engine software, denoted by p_3 , had about 30K lines of source code, with a total of 5K conditional expressions. Each of these projects are distributed in 50 – 100 source files containing between 200 and 500 classes. These experimental subjects cannot be biased in any way since they were developed by different organizations.

6.1 Performance

The instrumentation of these projects took less than 5% of the time it took to build (compile and link) the original, un-instrumented projects. The time it took to build the instrumented projects was almost identical to the time it took to build the un-instrumented projects, which points to the efficiency of the instrumentor. However, the execution of the instrumented version was slower than the execution of the un-instrumented version, because of the counting nature of the distribution-recording functions placed in the instrumented code. The conditional branch frequency for the projects was in the 11% – 17% of total instructions, which is the actual published range for conditional branch frequency [11]. Given this range, even if the instrumented conditional branches are slower 5 times than the un-instrumented branches, the overall time increase for the instrumented version is 1.4 – 1.6 times in the best and worst case respectively. The Diversity Analyzer has convenient features to combine distribution frequencies and conditional diversity matrices for testing executed in parallel on multiple machines. The Diversity Analyzer gives an option of collecting data only at application exit, which effectively means the end of a test cases is defined by application exit. In this case, the time increase over the original code is on average less than 1.2.

6.2 Effectiveness Evaluation

Another goal of the experimental evaluation was to explore the relationship between conditional diversity, data diversity,

standard deviation of conditional diversity and failures. In particular, the hypothesis being tested is that faults are dispersed in the control and data space of programs rather than concentrated. Data that shows an increase in faults as *std* and *dd* grow and *cd* gets closer to zero validates the hypothesis. Data that shows the opposite invalidates the hypothesis. Of course, one could always write a program that invalidates the hypothesis.

As a starting to the experimental evaluation we took already existing regression tests, ones that were run after every minor software release of p_1, \dots, p_3 . These tests are functional black-box tests mostly automated at the GUI level of the programs. The total number of test cases executed, the number of naturally-occurring faults detected with the regression tests, and the branch coverage obtained by the tests are given in Figure 9. The conditional diversity(*cd*), the data diversity(*dd*), and the standard deviation of conditional diversity(*std*) obtained by the regression tests are also given in Figure 9. The single values for *cd*, *dd* and *std* were obtained as averages of the conditional diversity matrix, the data diversity vectors, and the standard deviation vector respectively. For example, in Figure 5 $cd = (.3 + .6 + .5 + 0 - 1 - 1 - .3 + 0)/8 = -.9/8 = -.11$, $dd = (1 + .5)/2 = .75$, and $std = (.23 + .19)/2 = .21$. For all three projects, the data diversity and the standard deviations are very low. And yet the branch coverage obtained is around 70%, which is relatively high. This points to the fact that high coverage does not necessarily imply high diversity.

The evaluation of the correlation between diversity and fault-detection capabilities is performed by identifying subsets of the regression test suite and measuring, for each subset, its diversity and its fault-detection capability. For each project, we have randomly identified a hundred subsets, all of the same cardinality of thirty test cases, and similar branch coverage of about 10%. There were no significant differences in the number of times conditions were executed for these subsets. This is important in order to eliminate the possibility that there is a strong correlation between frequency of execution and fault detection independently of diversity. The subsets were identified in such a way that the *cd* of 25 subsets fell in the range $[0, .2]$, the *cd* of 25 of the subsets fell in the range $[.2, .4]$, the *cd* of 25 subsets fell in the range $[.4, .6]$, and the *cd* of 25 subsets fell in the range $[.6, .8]$. The number of faults detected by each subset was noted for each project. The experiment was repeated for *dd* and *std*. As a baseline 25 subsets were chosen in a random way and the percentage of faults detected was noted. The comparison of fault-detecting effectiveness of diversity is summarized in Figure 10. Figure 10 shows the percentage of faults detected in each *cd*, *dd*, and *std* range. It also shows the percentage of faults detected by the baseline test. The data of Figure 10 shows that subsets with higher diversity values for *dd* and *std* are more effective in detecting faults for projects p_1, \dots, p_3 and their corresponding regression tests. Note that by definition closer *cd* is to zero more conditional diversity there is. The exponential rate of growth of the fault percentages across diversity ranges for *std* is different for each project; however, the steepest growth occurs when the *std* values are in the $[.6, .8]$ range. This observation gives a motivation for obtaining high levels of diversification in order to get maximum fault-detection rates. The initial experiments are somewhat promising but more evaluation is

| cd | [0,.2] | [.2,.4] | [.4,.6] | [.6,.8] | baseline |
|-------|--------|---------|---------|---------|----------|
| p_1 | 55% | 30% | 8% | 7% | 15% |
| p_2 | 66% | 17% | 15% | 2% | 20% |
| p_3 | 64% | 15% | 11% | 10% | 17% |

| dd | [0,.2] | [.2,.4] | [.4,.6] | [.6,.8] | baseline |
|-------|--------|---------|---------|---------|----------|
| p_1 | 20% | 25% | 25% | 30% | 25% |
| p_2 | 15% | 25% | 30% | 30% | 5% |
| p_3 | 11% | 12% | 15% | 64% | 15% |

| std | [0,.2] | [.2,.4] | [.4,.6] | [.6,.8] | baseline |
|-------|--------|---------|---------|---------|----------|
| p_1 | 5% | 7% | 30% | 58% | 20% |
| p_2 | 2% | 15% | 17% | 66% | 12% |
| p_3 | 10% | 11% | 15% | 64% | 13% |

Figure 10: Diversity Effectiveness

needed in order to validate or invalidate the fault-dispersion hypothesis. The additional experiments are best performed by independent researchers in the context of substantial industrial samples possibly with domains completely different than the ones of p_1, \dots, p_3 .

Traditional coverage analysis treats test cases that do not increase code coverage as undesirable. The experimental data shows that there may be a value in such undesirable test cases, since there was an upward fault-detection trend in the presence of constant branch coverage for subsets across different diversity ranges. This is probably to be expected, since branch coverage is only a special case of conditional diversity, where the true/false distribution frequency counts are binary values.

7 Related Work

The general idea of test diversification is related to many existing testing methods. For example, adaptive random testing is a black-box approach that attempts to ensure an even spread of test cases[24]. However, note that in general there is no relation between an even spread of test cases and conditional diversity, data diversity, and standard deviation of diversity. Unlike adaptive random testing, Test Diversity measures diversity at the code level. Automating statistical testing involves a heuristic that is aimed at uniform generation of execution paths[6]. Our measures are not based on heuristics, they can be effectively obtained by the Diversity Analyzer, and the notions of test dispersion, data diversity, and test orthogonality are novel.

The fact that different test criteria target different kinds of faults, and hence should be used in combination, is well known and has been empirically observed in several experiments. However, since there is a plethora of test criteria (in theory there is an infinity of them) the tester is left with the problem of figuring out which methods to combine. Test Diversity attacks the problem of diversity at its root which is control and data. In fact every existing individual test

criteria could be seen as a form of test diversification. For example, branch coverage attempts to diversify execution of program branches, where for instance 70% branch coverage means more branch-execution diversity than 20%. Test Diversity goes beyond ordinary test criteria, which is not to say that it cannot be used in conjunction with the existing criteria. One could potentially satisfy any given criteria with an arbitrarily concentrated test distribution. Test suites that satisfy a given criteria could then be evaluated by the measures defined in this work.

Test Diversity is closely related to code coverage. In particular, diversity analysis is related to conditional, or decision coverage [21], which is a special case of conditional diversity in which the frequency counters are Boolean values indicating if the branches are covered or not covered. However, test suites that satisfy the decision-coverage criterion could result in very different conditional diversity, data diversity and standard deviation. Decision coverage is *not* aimed at control and data variation, as defined in this work. Since there could be many sets of test cases that satisfy decision coverage, diversity analysis could be performed to determine the quality of decision coverage. This does *not* evaluate the quality of the criterion itself as is done in [4], but rather it evaluates the particular test suite.

The data-flow methods are somewhat related to data diversity in that data-flow testing criteria require certain dataflow properties covered [13]. For instance, the definition-use dataflow criterion requires covering of at least one path that connects a statement where a variable is assigned a value and a statement where the variable is used [17]. However, paths that cover definition-use pairs could all be the same, and/or the data on the paths could be the same as well. The data-flow testing criteria are *not* aimed at control and data variation, as defined in this work. Since there could be many sets of test cases that satisfy data-flow coverage, diversity analysis could be performed to determine the quality of data-flow coverage by measuring data diversity for the particular test suite. In fact, diversity analysis could be used to determine the quality of any test suite, regardless of the method used to generate the test suite. Some adequacy criteria might be predisposed to produce test cases that give higher diversities.

Test Diversity is also related to performance optimization. Branch prediction makes static and dynamic guesses on the branch that would be executed next[8]. The static analysis is done at compile time where it is based on general knowledge of loop executions³, while the dynamic analysis is done at the microprocessor pipeline level, where predictions are based on, for example, one-bit or two-bit prediction schemes[11]. Source code optimization and profiling give execution-time distributions at the source-code level, and at the object-code level[23]. In many cases it may be trivial to add counting, and some profilers allow counting of the number of times certain execution paths are covered. Profilers are commonly found as parts of compilers and widely available. However, in general, there is *no* relation between the time spent in a section of code and the number of times branches that lead up to that point are executed. Furthermore, profilers do not perform data diversity, and they do not compute standard deviation

³Guesses based on findings such as loop branches are taken with 90% probability[11].

of conditional diversities. They do not check for linear independence of control. In addition, profiles are mainly based on statistical sampling in order to reduce the amount of performance data gathered. Test Diversity is based on exact run time information.

Test Diversity is related to the operational abstraction approach[15], to cluster analysis[19], to the software tomography approach[12], and to the bug isolation approach[2]. Operational Abstraction is a *specification*-based approach where executable statements, capturing semantic requirements, are placed in the code to evaluate test cases at execution time. Cluster Analysis is a form of observation testing, based on multivariate analysis for forming groups of clusters of related program executions. The software tomography and the bug isolation approaches are aimed at light-weight instrumentation for the purpose of collecting run-time data for various purposes, such as to aid in the analysis of problems found in the field after product release. All of these approaches are not aimed at measuring variation of control and data as described in this paper.

8 Conclusion

Test Diversity is a novel software test method which can be used to measure the test effort distribution and test effectiveness at the source code level. Test Diversity consists of four new measures of internal control and data variation: conditional diversity, data diversity, standard deviation of diversity, and test orthogonality. These four measures could be used to assess the quality of any test suite regardless of the method used to generate the test suite.

Conditional diversity is a measure of test distribution, pointing to portions of source code of high and low test concentration. This measure could be used to evaluate the test effort distribution of a test suite. Data diversity is a measure of internal data variation given by a set of test cases. When data diversity is at its maximum, *all* of the test cases result in a different data flow through the program under test. Different data flow is highly desirable since executing the program with test cases that result in the same internal data is less likely to find faults. Standard deviation of conditional diversity is a measure of control and data dispersion. Highly concentrated control and data are less likely to detect defects than dispersed control and data, given the defects are dispersed in the control and data space of the program. Orthogonality is a measure of test case independence. In particular, orthogonal test cases do not have control flows that can be expressed as a linear combination of each other. Independent test cases are an indication of unrelated executions of the program under test.

The Diversity Analyzer is a new, industrial-strength tool used for analyzing C, C++, C# and Visual Basic programs in Windows and .NET environment. The Diversity Analyzer instruments source code, computes conditional diversity, data diversity, and standard deviation. It allows viewing of the diversity metrics in a summary fashion, and viewing of the source coded annotated with diversity metrics.

Our experimental results, using the Diversity Analyzer, show that the Diversity Analyzer can efficiently be used in

various industrial environments. Our experimental data suggests that more diverse test suites may be more effective in detecting faults.

References

- [1] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Trans. on Soft. Eng.*, 11:1491–1501, 1985.
- [2] Liblit B, Aiken A, Zheng A, and Jordan M. Bug isolation via remote program slicing. *PLDI 2003*, pages 141–154, 2003.
- [3] Sara Baase. Computer algorithms. *Addison-Wesley*, Second Edition:16, 1988.
- [4] M. D. Davis and E. J. Weyuker. A formal notion of program-based test data adequacy. *Information and Control*, 56:52–71, 1983.
- [5] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–43, 1978.
- [6] Thevenod Fosse and Waeselynck Helene. Software statistical testing based on structural and functional coverage. *11th International Quality Week*, May San Francisco, CA:4–12, 2003.
- [7] P. Frankl and E. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. on Soft. Eng.*, 19:202–213, 1993.
- [8] A.L. Goel and K. Okumoto. Time dependent error detection rate model for software and other performance measures. *IEEE Transactions on Reliability*, R-28:206–211, 1979.
- [9] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. *Proceedings of 15th International Conference on Software Engineering*, pages 118–129, 1993.
- [10] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.
- [11] J.L. Hennesey and D. A. Patterson. Computer architecture a quantitative approach. *Morgan Kaufmann Publishers, INC.*, pages 251–343, 1990.
- [12] Bowring J, Orso A, and Harrold M.J. Monitoring deployed software using software tomography. *ACM SIGPLAN-SIGSOFT PASTE 2002*, pages 2–9, November, 2002.
- [13] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Soft. Eng.*, SE-9:347–354, 1983.
- [14] N. G. Leveson. Safeware system safety and computers. In *Addison-Wesley*, 1995.
- [15] Harder M, Mellen J, and Ernst M. Improving test suites via operational abstraction. *ICSE 2003*, pages 60–71, May, 2003.
- [16] Borislav Nikolik. Constraint preservation through loops. *Information Processing Letters*, 55:143–148, 1995.
- [17] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, SE-11:367–375, 1985.
- [18] M. Z. Tsoukalas, J. W. Duran, and S. C. Ntafos. On some reliability estimation problems in random and partition testing. *IEEE Trans. on Soft. Eng.*, 19:687–697, July 1993.
- [19] Dickinson W, Leon D, and Podgurski A. Finding failures by cluster analysis of execution profiles. *ICSE 2001*, pages 339–348, 2001.
- [20] S.N. Weiss. What to compare when comparing test data adequacy criteria. *Software Engineering Notes*, 14:42–49, 1989.
- [21] E. J. Weyuker, S. N. Weiss, and R. G. Hamlet. Comparison of program testing strategies. In *Symposium on Testing, Analysis, and Verification (TAV4)*, pages 1–10, Victoria, BC, 1991.
- [22] E.J. Weyuker. Can we measure software testing effectiveness. *Proc. IEEE-CS Int. Software Metrics Symposium*, pages 100–107, 1993.
- [23] Michael Wolfe. Optimizing supercompilers for supercomputers. *MIT Press, Cambridge, Mass.*, 1989.
- [24] Chen T. Y., Kuo F. C., Merkel R. G., and Ng S. P. Mirror adaptive random testing. *Third International Conference on Quality Software*, November Dallas TX, 1998.