Diversity Analysis *

Borislav Nikolik

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

September 20, 2004

Abstract

This paper describes a novel software-testing tool used for measuring control and data diversity. The Diversity Analyzer computes three original measures of control and data variation in testing: conditional diversity, data diversity, and standard deviation of diversity. Conditional diversity is a measure of control variation, data diversity is a measure of internal data variation, and standard deviation is a measure of control and data dispersion resulting from executing a program on a set of test cases. The Diversity Analyzer can currently measure test diversity for software written in C, C++, C#, and Visual Basic in the Windows and .NET environment.

Key Words: *conditional diversity, data diversity, standard deviation, diversity matrix, balancing, skewing.*

1 Introduction

An important aspect of software testing is determining the quality of a test suite. Many test-adequacy criteria have been proposed in order to evaluate the quality of a test suite. The existing test-adequacy criteria are based on control flow analysis[10], on data flow analysis[19][6], program mutation[15], and some combination of these. Control flow and data flow criteria are aimed at detecting faults of incorrect control and data flow. Mutation criteria are based on testing for a specific set of faults, such as for example operator reference faults[11]. The adequacy criteria are used to guide the test generation process and to determine when to stop testing and release the software[2]. The relative fault-detection abilities of many test-adequacy criteria have been compared[8][3][4]. Many commercial and research test tools have been built based on adequacy criteria, such as, for example, statement-coverage analyzers which give a percentage of executed program statements.

Criterion-based testing neither makes a distinction between the potential infinity of test suites that satisfy the given criteria nor it makes a distinction between test suites that do *not* satisfy the given criteria. Typically, some of the large number of suits that satisfy a criterion detect faults while other suites that satisfy the criterion do not detect faults[7] [3][20]. Conversely, some of the large number of suits that do not satisfy a criterion detect defects while other suites that do not satisfy the criterion do not detect defects[21]. It is our intent in this paper to improve on the binary nature of the test criterion. Diversity Analysis transcends test criteria by evaluating test suites regardless of the adequacy criterion used to generate it.

Black-box test criteria are based on exercising product functionality at the interface level. In reliability-based testing, test suites consist of randomly chosen test cases which are used to determine the probability of a failure of the program under test[14][13][18]. The quality of the test suite in reliability-based testing is a function of the number of randomly selected test cases. More test cases selected and executed gives a higher confidence bound that the program under test is of higher reliability[17]. A fundamental assumption of reliability-based testing is independence of test cases, which has to hold in order for the reliability predictions to be accurate.

Regardless of the test method used, test cases that yield similar or same control flows and data flows through the program exercise the program in a restricted, dependent, and concentrated manner. On the other hand, test cases with different control flows and data flows exercise the program in fundamentally different and independent ways. This observation is the basis for the notions of conditional diversity, data diversity, and standard deviation of conditional diversity.

Conditional Diversity is a measure that gives the test-effort distribution and the control variety resulting from executing a program on a set of test cases. 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 *true* and *false* branches in the code. Conditional diversity could be used in determining if adjustment of the distribution is necessary.

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 the test cases[16]. Higher data diversity indicates the test cases result in program executions with highly different internal data, whereas a lower data diversity indicates the code is covered with the same or similar data over and over again. In the limit, when data diversity is zero, all of the test cases result in the same data flow, ef-

*©2004 Vidak Quality. All rights reserved. Presented at PSQT/PSTT 2004 and 2005

fectively representing the same test case. We use conditional diversity to measure data diversity.

Test cases could result in control and data highly concentrated in specific regions of the control space and data space. In the worst case, all of the test cases are concentrated on the same control and data point, effectively representing the same test cases. *Standard Deviation* of the conditional diversities is used as a measure of control and data dispersion in the control and data space of the program under test. High standard deviation indicates the test cases exercise the program with dispersed control and data.

The Diversity Analyzer is a tool used for measuring test diversity; it calculates conditional diversity, data diversity, and standard deviation of conditional diversity. The tool first instruments the source code with branch-distribution collecting functions, which essentially count the number of times branches are taken and not taken. The instrumented program is then built and, typically, executed multiple times on a set of test cases. As the instrumented program executes, the branch-distribution collecting functions generate a test distribution. Next, this test distribution is used to calculate conditional diversity, data diversity, and standard deviation.

The Diversity Analyzer offers support for testing on multiple machines by providing merging capability for diversity metrics and test distributions. The instrumentation flexibility allows instrumenting whole projects, parts of projects, single source files, parts of source files, multiple projects, and files written in different programming languages. The tool also allows fine granularity of analysis, where diversity metrics could be computed per individual tests, or anywhere in the middle of a test in cases where the debugger interrupts the execution of the program. That is, diversity metrics could be computed even as the instrumented program under test is still running. The Diversity Analyzer can currently analyze C,C++,C# and Visual Basic source code in the Windows and .NET environment. The tool gives a summary of diversity metrics per source files, as well as diversity hot-spots view at the source-code level.

Test Diversification is a process of generating diverse test suites. Diverse test cases are characterized with high data diversities and high standard deviations of conditional diversities. Test balancing and test skewing are the two underlying principles of test diversification. Balancing generates test cases that, as a whole, push the test distribution towards a balance state in which the *true* and *false* branch-execution counts are identical. Skewing generates test cases that, as a whole, push the test distribution towards its extreme points. The combined effect of balancing and skewing underlines every diverse test suite.

2 Diversity Fundamentals

To demonstrate the central notions of this paper we use an instrumented version of BubbleSort, shown here in Figure 1. The un-instrumented version of this code fragment appears frequently in textbooks on algorithms [1]. The code fragment consists of three conditional expressions, one controlling the *while* loop (denoted by e_1), one controlling the *for* loop (denoted by e_2), and one controlling the *if* statement (denoted by e_3). The Diversity Analyzer parses the code fragment, locates the three conditional expressions, and places the c function around e_1, e_2 and e_3 to count the number of times e_1, e_2, and e_3 evaluate to *true* and *false*, as shown in Figure 1. The second parameter of c denotes the sequential position of e_1, e_2 and e_3 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.

```
  bool didSwitch=true;
1 while(c(didSwitch,0)){     //e1
2  n=n-1;
3  didSwitch=false;
4  for(int j=0;c(j<n,1);j++) //e2
5   if(c(L[j]>L[j+1],2)){    //e3
6     Interchange(&L[j],&L[j+1]);
7     didSwitch=true;
    }
  }
```

Figure 1: Instrumented Bubble Sort

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

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

Figure 2: Conditional Diversity

A test-effort distribution consists of the *true* and *false* frequency counts. Suppose the code of Figure 1 was executed on test cases $t_1, ..., t_3$, given in Figure 3. The *true* execution counts T and the *false* execution counts F for e_1, e_2 and e_3 and test cases $t_1, ..., t_3$ are given in Figure 3. As these frequency counts indicate, most of the testing effort goes into executing e_2 in the *true* direction, and e_3 in the *false* direction.

Conditional diversity for a conditional expression is calculated as a *distance* between the actual distribution for that expression from the uniform distribution for that expression, as shown in Figure 2. A is the average of *true* hits and *false* hits for an expression, and D is the conditional diversity for the expression, a value in the $[-1, 1]$ range. The value of the conditional diversity is negative 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. We assign a specific value outside the $[-1, 1]$ range to denote an unexercised expression. The Diversity Analyzer uses the frequency counts produced by the instrumented version of the code fragment to calculate the conditional diversities for $e_1, ..., e_3$ and $t_1, ..., t_3$ utilizing the formulas of Figure 2.

Data diversity for a conditional expression e_p is a percentage of distinct conditional diversities given by a set of test

	L	e_1T	e_1F	e_2T	e_2F	e_3T	e_3F
t_1	$\langle 1,2,4,3 \rangle$	2	1	5	2	1	4
t_2	$\langle 1,2,3,4 \rangle$	1	1	3	1	0	3
t_3	$\langle 5,6,7,8 \rangle$	1	1	3	1	0	3

Figure 3: Execution Frequency Counts

	t_1	t_2	t_3	dd	std
e_1	.3	0	0	.3	.27
e_2	.4	.5	.5	.3	.05
e_3	$-.6$	-1	-1	.3	.32

Figure 4: Diversity Statistics

cases for e_p. If two test cases t_i and t_j cause different conditional diversities for e_p, than *different* data flows through e_p on t_i and t_j. In effect, the test-effort distribution is used as an indicator of data variation among test cases.

Standard deviation, when applied to conditional diversity, is a measure of dispersion of conditional diversities around the conditional diversity mean. Higher standard deviation means more coverage of the program's control space is achieved. Given a set of 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.

The conditional diversities, data diversities, and standard deviations for the code fragment of Figure 1, and test cases $t_1, ..., t_3$ are given in Figure 4. Note that the columns under $t_1, ..., t_3$ in Figure 4 form *conditional diversity vectors.* These vectors form a *conditional diversity matrix* C, a matrix whose columns are the conditional diversity vectors corresponding to executions of the test cases $t_1, ..., t_3$. Each row of the diversity matrix is used to compute a *standard deviation vector* $\vec{st}$ using the formula commonly found in textbooks on statistics:

$$\sigma = E[(X - \mu)^2],$$

where X holds the values of the rows of C, and $\mu = E(X)$ is the arithmetic mean of the values of X. The *data diversity vector* $\vec{d}$ is a vector of values in the range $[0, 1]$, where each value in the vector corresponds to the percentage of *distinct* values in the rows of C. The code fragment of Figure 1 and the test cases of Figure 3 give the results in Figure 5.

Measuring conditional diversity allows the tester to find out which portions of the code the testing is *concentrated* on. Test cases $t_1, ..., t_3$ are concentrated on the *true* branch for e_2 and on the *false* branch for e_3. The test cases are balanced for e_1, but they are especially skewed for e_3. 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. 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[12]. The iteration stops when either the target diversity is achieved, or other factors, such as schedule pressures, dictate termination of testing and product release.

The effectiveness of testing depends on the internal data

$$\vec{st} = \begin{bmatrix} .27 \\ .05 \\ .32 \end{bmatrix} \vec{d} = \begin{bmatrix} .3 \\ .3 \\ .3 \end{bmatrix}$$

$$C = \begin{bmatrix} .3 & 0 & 0 \\ .4 & .5 & .5 \\ -.6 & -1 & -1 \end{bmatrix}$$

Figure 5: Diversity Vectors and Matrix

variation given by test cases. Data variation is tightly related to conditional diversity. Different conditional diversities guarantee different data flowing through the program on test cases. The data diversity for $t_1, ..., t_3$ is low since the conditional diversities for t_2 and t_3 are identical for e_1, e_2, and e_3; that is, 2/3 of the test suite $\{t_1, ..., t_3\}$ is identical with respect to program control. The actual data that flows through the code on t_2 and t_3 is different; however, the data variation resulting from executing test cases t_2 and t_3 is trivial with respect to control variation.

Low standard deviation of conditional diversities means the differences between control executions in the code are low. For example, low standard deviation might mean that there is only one loop iteration difference in the *true* direction between the test cases. Figure 5 indicates that the standard deviation is especially low for e_2. Therefore, on test cases $t_1, ..., t_3$ the control space of Bubblesort is covered in a concentrated rather than the desirable dispersed fashion. Such concentrated test cases, when apparently different at the user interface of the program, could mislead the tester into thinking that the program internals are covered thoroughly. For example, test cases t_2 and t_3 are different at the interface level of Bubblesort, but they result in identical test distribution since both test cases represent four-element lists that are already sorted.

Test cases that result in high data diversity and high standard deviations give a dispersed control and data coverage, exercise the program with different and dispersed data, and are *less* control and data dependent. In general, distinct test cases at the interface level do *not* guarantee high data diversity and high standard deviation, since these metrics are properties not only of test cases but of code as well. 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. Therefore, the test cases have to be analyzed in the context of the source code to determine diversity. The Diversity Analyzer is used for this purpose.

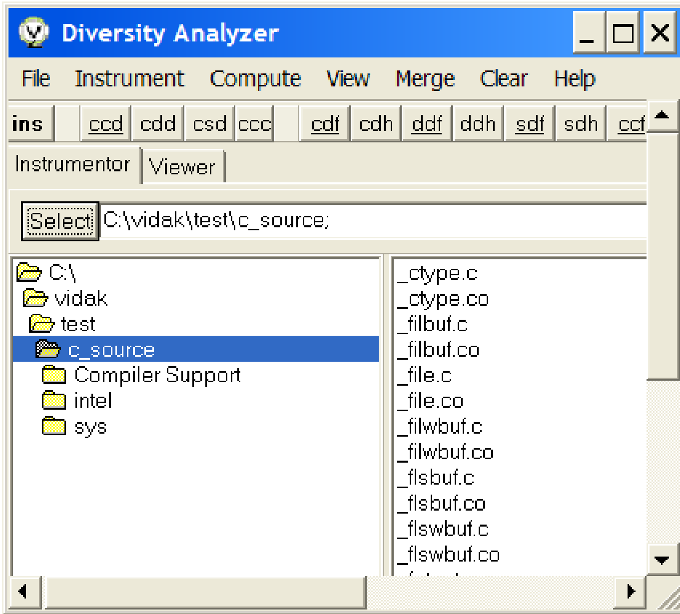

Figure 6: Instrumentor

3 Diversity Analyzer

The Diversity Analyzer is a tool that measures test diversity for projects written in C, C++, Visual Basic, and C# in the Windows and .NET environment. The tool can be extended to Unix/Linux environments, and other languages, such as Java. The tool can handle mixed-language projects, multiple projects simultaneously, project components such as dlls, and multi-tasking code. Next we describe the tool in more detail.

3.1 Measurement Steps

The Diversity Analyzer measures control and data diversity resulting from testing. The following simple steps need to be performed to obtain conditional diversity, data diversity, and standard deviation of diversity: instrument, build, run, compute, and view. In this section, we refer to three internal files generated by the Diversity Analyzer: master.dfr contains the *true* and *false* count for the conditional expressions, master.cdm contains the conditional diversity matrix, and master.ccm contains the branch-coverage matrix.

3.1.1 Instrument

Source-code instrumentation inserts executable statements in the original source code while not affecting the logic of the original source code[6]. The instrumented version is unavoidably more complex, and more memory and time consuming than the original version. To instrument the source code with test-distribution collecting functions, the following steps need to be performed: prepare the source code for instrumentation, choose target directories containing code to be instrumented, and redistribute instrumented source files to their corresponding projects.

In preparing for instrumentation, one needs to make sure the source files build in the original projects. If the source files compile and link in the original projects, the Diversity Analyzer will successfully parse and instrument these files as well. The tool has an Export/Import feature for saving and retrieving diversity results. The Export feature could be used before instrumenting to save previous diversity results, since instrumentation overwrites the internal files.

The instrumenter allows great flexibility in instrumentation. For instance, individual files may be chosen from different projects and placed in a single target directory. In addition, the target directories may contain source files of different programming languages. Furthermore, the source files could be marked with the keywords ZOOM_BGN and ZOOM_END to delimit the scope of analysis to particular sections of code. If zoom statement are placed in the code, only the code that appears between a ZOOM_BGN and ZOOM_END pair is instrumented; otherwise, the complete source file is instrumented.

Target directories are picked for instrumentation using the simple GUI, as shown in Figure 6, by double clicking on a directory in the directory list, and clicking on the Select button. Clicking on the **ins** button instruments all the source files in the selected directories. When the instrumentation completes, the instrumented files from the target directories are ready to be integrated into their original projects, and are ready to be built; if the source files were picked-and-chosen from various directories and projects, they need to be redistributed to the corresponding directories in the projects. The Clear-Zoom main-menu selection reverses the instrumentation by removing all the ZOOM_BGN and ZOOM_END statements from the source files. The Clear-Source Directories main-menu selection restores the source files in the target directories to the original, un-instrumented versions.

3.1.2 Build and Run

Before the instrumented code is built, the test-distribution collection libraries vidak.lib and vidak.dll need to be integrated into the instrumented projects. The library vidak.dll uses the COFF Microsoft format. If the build environment uses the Omf import library file format, such as Borland Builder, a conversion tool, such as Coff2Omf, needs to be used to convert vidak.dll to Omf format. After integrating the vidak.lib and the vidak.dll files, the projects should be build as they would be normally build.

The instrumented build could be run on a single machine or on multiple machines. If testing is done only on the machine where the build was made, no additional steps are necessary prior to running the test cases. However, if tests are to be run on multiple machines the following steps need to be performed: copies of master.dfr need to be placed on the target machines, and the file vidak.dll needs to be placed on the target machines as well. The instrumented code is then executed on a set of test cases. The test distribution is recorded in master.dfr.

The Clear-Distribution main menu selection clears the master.dfr frequency record. Running a new test after the Clear-Distribution selection has been made, results in obtaining frequency data only for the new test. If cumulative results are needed the Clear-Distribution selection should not be used between test runs.

3.1.3 Compute

The Compute feature uses the master.dfr file to compute diversity and coverage. If the tests were run on multiple machines, the master.dfr files need to be moved from the other machines to the machine where the Diversity Analyzer is installed and where diversity is to be computed. The main-menu command Merge-Distributions is used to obtain a combined, cumulative conditional frequency distribution. After the merge has completed, the master.dfr will contain the combined, cumulative results from all the .dfr files. The main menu command Merge-Diversity Matrices and/or Merge-Coverage Matrices is used to obtain consolidated master.cdm and master.ccm matrices. After the merge has completed, the master.cdm/master.ccm files will contain all the other .cdm/.ccm file appended to the masters.

The Compute main-menu selection is used to compute diversity and coverage. The Conditional Diversity selection computes the conditional diversity vector and appends this vector to the conditional diversity matrix master.cdm. This selection also computes the average of the conditional diversity vector.

The Data Diversity selection computes the data diversity vector based on the conditional diversity vectors in the conditional diversity matrix master.cdm. This selection also computes the average of the data diversity vector.

The Standard Deviation selection computes the standard deviation vector based on the conditional diversity vectors in master.cdm. This selection also computes the average of the standard deviation vector.

The Coverage Matrix compute-selection computes a coverage vector and appends the vector to master.ccm. This selection also computes the average of the coverage vector.

The Clear-Diversity Matrix main-menu selection clears the master.cdm file. All the conditional diversity vectors are erased from master.cdm. The Clear-Coverage Matrix main-menu selection clears the master.ccm file. All the coverage vectors are erased from master.ccm.

3.1.4 View

The View tab of the Diversity Analyzer UI shows the view panel where diversity and coverage results are displayed as shown in Figure 7. The View main menu selection allows choosing the type of results to view.

The View-Diversity-Conditional-File selection shows the average conditional diversities by source file, computed from the most recent conditional diversity vector. This selection gives a summary view of the files and their average conditional diversities. The View-Diversity-Conditional-Hot-Spots selection shows the conditional diversities, computed from the most recently computed conditional diversity vector. They are displayed based on conditional expressions in the code, and sorted in ascending order based on conditional diversity. The tool also gives by-file or by-hot-spots view of data diversity, standard deviation, and branch coverage.

Clicking on a particular row in the View panel opens a new window with the source code annotated with diversity/coverage data, as shown in Figure 8. The Diversity button at the bottom of the View panel opens the diversity matrix in Excel, as shown in Figure 9. The last column gives the

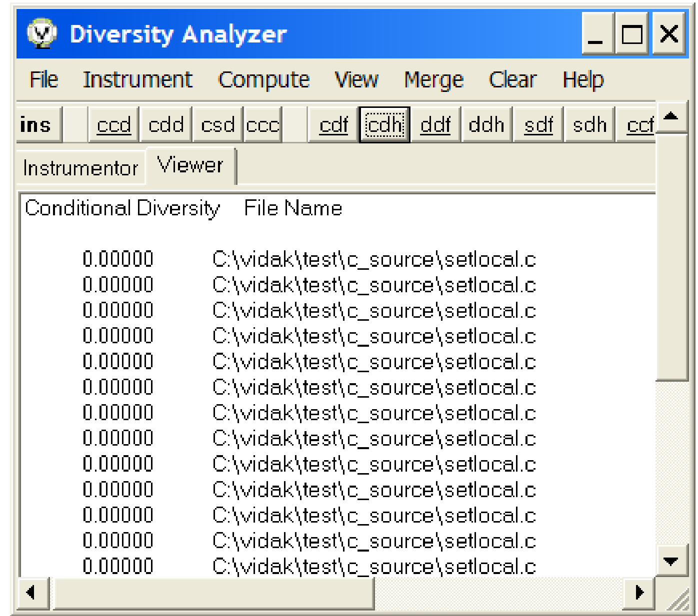

Figure 7: Summary Viewer

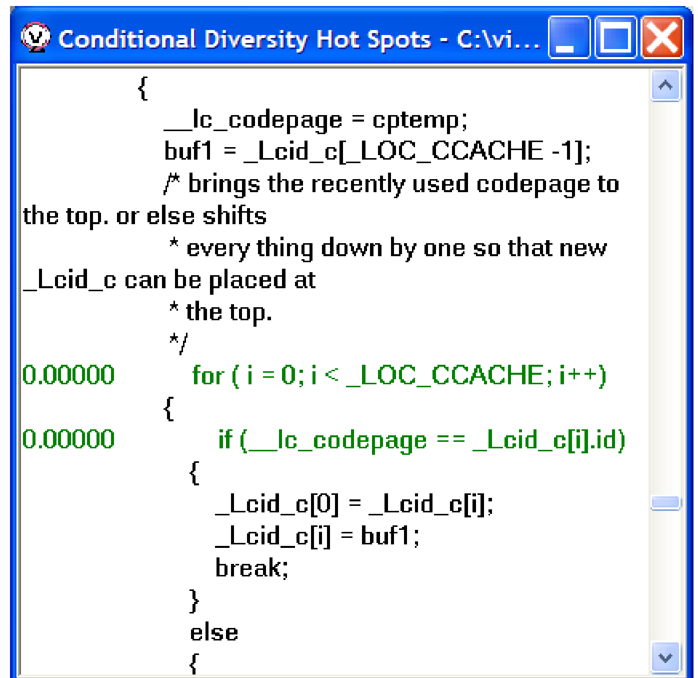

Figure 8: Source Code Viewer

	A	B	C	D	E	F
1	####					
2	File	Line	Condition	CD	CD	CD
3				-0.1667	0.2667	0.1873
4	C:\Do	13	didSwitch	0	0.6	0.3333
5	C:\Do	16	j<n	0.5	0.2	0.4286
6	C:\Do	17	L[j]>L[j+1]	-1	0	-0.2
7						

Figure 9: Diversity Matrix in Excel

most recently computed vector: conditional, data or standard deviation. The Coverage button at the bottom of the View panel opens the coverage matrix in Excel. The latest column gives the most recently computed coverage vector. These Excel results could be used to perform operations such as sorting and graphing for presentation purposes. The cdf and cdh buttons are used to view conditional diversity by file and by hot spots. The ddf and ddh buttons show data diversity by file and hot spots in Excel. The sdf and sdh buttons open the standard deviation results by file and hot spots. The ccf and cch buttons show conditional coverage by file and hot spots respectively. The frequency distribution button shows the actual *true/false* counts for each conditional expressions.

3.2 Performance

The performance of the Diversity Analyzer is evaluated based on time and memory it takes to: instrument source code, build instrumented code, run tests, and compute diversity/coverage. The performance evaluations are given relative to the performance of the un-instrumented, original source code. The example execution times in this section are based on a reference machine: Intel Pentium(R) Mobile 1400Mhz, 384MB RAM. The execution time and memory needed for running the Diversity Analyzer depends on few factors: number of conditional expressions in the code, number of times instrumented conditional expressions are executed, and sizes of conditional diversity matrix and coverage matrix.

The number of conditional expressions in the source code determines the time and memory it takes to instrument the source code. The time to instrument the source code is a small fraction of the time it takes to make a complete build of the original, un-instrumented code. It takes less than 5% of the original-code build time to perform the instrumentation. For example, it takes 2.5 minutes on the reference machine to instrument 3084 source files containing a total of 20078 conditional expressions, approximately 100K lines of source code. The increase in size of the instrumented source code and its executable relative to the original code and its executable is negligible. The instrument time and memory requirements are linear to the source-code size, that is, linear to the number of conditional expressions in the source code.

The time to build the instrumented code is almost identical to the time it takes to build the original, un-instrumented code. No significant increase in memory use is needed to build the instrumented code.

The increase in run time of the instrumented code is linear to the number of times conditional expressions in the source code get evaluated. To run the instrumented version it can take $1.4 - 1.6$ of the time it takes to run the original code. We believe this increase in run-time is acceptable, especially since usually testing is distributed among multiple machines which could carry test execution in parallel. The Diversity Analyzer has convenient features to combine distribution frequencies and conditional diversity matrices. 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. The perceived execution-time difference of the instrumented version and the original version depends on the ratio of branch statements to the non-branch statement in the code. Higher this ratio is, the perceived difference will be greater. According to published data [9], conditional branch frequency is $11\% - 17\%$ of the total instruction in the code. No significant increase in memory use is needed to run tests on the instrumented version.

The time to compute diversity and coverage depends on the conditional vector size (the number of conditional expressions in the source code), and on the conditional diversity matrix size (number of test cases analyzed, i.e., number of conditional vectors in the diversity matrix. The time it takes to compute a conditional diversity vector is linear in the number of conditional expressions in the code. For example, it takes about 15 seconds to compute conditional diversity on the reference machine for 20078 conditional expressions. This includes the time it takes to write the results to the corresponding .xls files. The time it takes to compute a standard deviation vector is linear in the conditional vector size, and linear in the number of conditional diversity vectors in the conditional diversity matrix. For example, it takes about 10 seconds on the reference machine to compute a standard deviation vector of a diversity matrix of 11 vectors, each of size 20078. This includes the time it takes to write the results to the corresponding xls files. Memory is needed to load the diversity matrix: size of float times the number of matrix entries.

The time it takes to compute a data diversity vector is in the order of $n \times logn$ times the size of a conditional diversity vector, where n is the number of vectors in the diversity matrix. For example, it takes about 14 seconds to compute a data diversity vector on the reference machine of a diversity matrix of 11 vectors, each of size 20078. This includes the time it takes to write the results to the corresponding .xls files.

The time it takes to compute a conditional coverage vector is liner in the conditional vector size, and linear in the number of conditional diversity vectors in the conditional diversity matrix. For example, it takes about 9 seconds on the

	L	e_1T	e_1F	e_2T	e_2F	e_3T	e_3F
t_5	$\langle 1,2,4,3\rangle$	2	1	5	2	1	4
t_6	$\langle 3,4,5,6\rangle$	1	1	3	1	0	3
t_7	$\langle 6,7,8,9\rangle$	1	1	3	1	0	3
t_8	$\langle 2,3,4,5\rangle$	1	1	3	1	0	3

$$C_{t_5-t_8} = \begin{bmatrix} .33 & 0 & 0 & 0 \\ .4 & .5 & .5 & .5 \\ -.6 & -1 & -1 & -1 \end{bmatrix}$$

$$\vec{st}_{t_5-t_8} = \begin{bmatrix} .28 \\ .06 \\ .34 \end{bmatrix} \vec{d}_{t_5-t_8} = \begin{bmatrix} .25 \\ .25 \\ .25 \end{bmatrix}$$

Figure 10: Non-Diversified Test Cases

reference machine to compute a conditional coverage vector of a diversity matrix of 11 vectors, each of size 20078. This includes the time it takes to write the results to the corresponding xls files. Memory is needed to load the coverage matrix: size of integer times the number of matrix entries.

The Diversity Analyzer is efficient in the instrumentation phase. The run-time penalty of the instrumented code is acceptable during testing, given that the intended use of the Diversity Analyzer is not to instrument release code. The time and memory requirements of computing conditional diversity and standard deviation is linear in the number of conditional expressions in the code.

3.3 Test Diversification

Test Diversification is the process of generating test suites that result in diverse control and data. Diversification could also be applied to already existing test suites, which are concentrated in specific regions of the control and data space. Test suites of poor diversity skew the conditional diversity, result in low data diversity, and low standard deviation of conditional diversities.

Consider the four test cases in Figure 10, perhaps generated by branch coverage, and executed against the code of Figure 1 to give 100% branch coverage. These test cases give highly skewed conditional diversity for e_2 and e_3; e_2 is skewed in the *true* direction, and e_3 is maximally skewed in the *false* direction. Only the expression e_1 is perfectly balanced for $t_6, ..., t_8$.

Different data does flow through the program on these test cases; however, that data is trivially different for $t_6, ..., t_8$ since it results in control-equivalent executions caused by already sorted four-element lists. Such trivial data variation does not improve data diversity.

Test cases $t_5, ..., t_8$ result in low standard deviation of conditional diversities, especially for e_2. This is due to the fact that e_2 is skewed in the *true* direction on all test cases. For the other two expressions, t_5 results in some variety, which is reflected in a higher standard deviation for e_1 and e_3 than for e_2.

Test cases $t_5, ..., t_8$ are of low quality since they result in skewed conditional diversity, low data diversity, and low standard deviation of conditional diversities for the code of Figure 1.

	L	e_1T	e_1F	e_2T	e_2F	e_3T	e_3F
t_9	$\langle 2,3,4,1\rangle$	4	1	6	4	3	3
t_{10}	$\langle 1,4,2,3\rangle$	2	1	5	2	2	3
t_{11}	$\langle 4,3,1,2\rangle$	3	1	6	3	5	1

$$C_{t_9-t_{11}} = \begin{bmatrix} .6 & .33 & .5 \\ .2 & .42 & .33 \\ 0 & -.2 & .66 \end{bmatrix}$$

$$\vec{st}_{t_9-t_{11}} = \begin{bmatrix} .19 \\ .16 \\ .64 \end{bmatrix} \vec{d}_{t_9-t_{11}} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Figure 11: Diversification Test Cases

$$C_{t_5-t_{11}} = \begin{bmatrix} .33 & 0 & 0 & 0 & .6 & .33 & .5 \\ .4 & .5 & .5 & .5 & .2 & .4 & .33 \\ -.6 & -1 & -1 & -1 & 0 & -.2 & .66 \end{bmatrix}$$

$$\vec{st}_{t_5-t_{11}} = \begin{bmatrix} .62 \\ .27 \\ 1.5 \end{bmatrix} \vec{d}_{t_5-t_{11}} = \begin{bmatrix} .28 \\ .28 \\ .57 \end{bmatrix}$$

Figure 12: Diversification Results

Improving the quality of a test suite takes the form of test distribution balancing and skewing. The goal of balancing and skewing is to increase the values of the diversity vector and the standard deviation vector. Balancing generates test cases that push the conditional diversity towards zero. Skewing generates test cases that push the test distribution towards its extreme points of -1 and 1.

The current conditional diversity of a test suite is used as a starting point for balancing. For code segments with positive conditional diversities, test cases are needed that give negative conditional diversities, and vice versa, for code segments with negative conditional diversities, test cases are needed that give positive conditional diversities. For example, to balance the negative diversities for e_3 produced by test cases $t_5, ..., t_8$, the test cases $t_9, ..., t_{11}$ from Figure 11 could be used. For example, test case t_{11} results in a conditional diversity of .66 for e_3, working to balance the negative conditional diversities for e_3 on test cases $t_5 - t_8$.

The challenge of balancing is to balance the un-balanced conditional diversities without un-balancing the balanced diversities. For example, t_{11} works to balance e_3 but it contributes to further skew e_1 in the *true* direction. In some cases it may not be possible to get significant negative or positive diversities to counter-weight the given diversities. In such cases balancing is obtained by choosing test cases that push the diversity closer to zero without necessarily producing in diversity of the opposite sign.

The current conditional diversity of a test suite is also used as a starting point for skewing. For code segments with positive diversities, test cases are needed that give higher positive

conditional diversities, and vice versa, for code segments with negative conditional diversities, test cases are needed that give higher negative conditional diversities. For example, t_9 skews e_1 in the *true* direction.

In order to balance particular sections of code, other sections of code might need to be skewed, and vice versa, to skew particular sections of code, other sections of code might need to be balanced. For example, to either skew or balance e_3, e_2 needs to be skewed in the positive direction. In general, test cases produce a combined effect of balancing and skewing. For example, test case t_9 skews e_1 but it balances e_2 and e_3.

The cumulative diversity results for test cases $t_5 - t_{11}$ are given in Figure 12. The combined effects of balancing and skewing of the initial distribution given by $t_5 - t_8$ are evident in the improvements in data diversity and standard deviation of diversity. There is a significant improvement in $\vec{st}_{t_5-t_{11}}$ over $\vec{st}_{t_5-t_8}$. The improvement is more modest for $\vec{d}_{t_5-t_{11}}$, with the exception for e_3 where the improvement in $\vec{d}_{t_5-t_{11}}$ over $\vec{d}_{t_5-t_8}$ is circa 100%.

Diversification is an iterative process of measuring diversity, and of diversifying control and data. This process is similar to traditional coverage analysis of iterative measuring and creating test cases to increase coverage. The problem of determining if a particular test distribution can be obtained is, in general, undecidable. Therefore, the process of balancing and skewing consists of generating test cases, based on code insights and perhaps on knowledge of similar code, executing the test cases, and measuring until satisfactory diversity results are obtained.

4 Related Work

Diversity analysis is related to branch coverage[21], data-flow coverage[12], and code optimization[22]. Branch coverage or decision coverage is a special case of conditional diversity in which the frequency counters are Boolean values indicating if the branches are covered or not covered. Test suites that satisfy the decision-coverage criterion could result in very different conditional diversity, and low data diversity and standard deviation. 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.

Data-flow testing criteria require certain dataflow properties covered. 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 [19]. Paths that cover definition-use pairs could all be the same, and/or the data on the paths could be the same as well. 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.

Branch prediction makes static and dynamic guesses on the branch that would be executed next[5]. The static analysis is done at compile time where it is based on general knowledge of loop executions, 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[9]. Neither of the branch prediction schemes count the number of times branches get executed.

Source code optimization and profiling give execution-time distributions at the source-code level and at the object-code level. 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. Therefore, diversity analysis cannot be effectively replaced by an already available profiler. Furthermore, profilers do not perform data diversity, and they do not compute standard deviation of conditional diversities. They do not check for linear independence of control.

5 Conclusion

Test Diversity is a novel software-test measurement method used to measure the variety of control and data in testing. Conditional diversity gives the test-effort distribution, and control and data variation. This measure could be used to determine if balancing and/or skewing of conditional diversity is needed. When conditional diversity is at its extreme points, the tests hit *only* the *true* or *false* branches. As the conditional diversity approaches zero, the tests hit the *true* and *false* branches more uniformly.

Conditional diversity is used to calculate data diversity, which is a measure of data variation in testing. 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 dataflow is less likely to find faults. Conditional diversity, which is a measure simple to obtain, is used to compute data diversity. Standard deviation of the conditional diversity matrix is used to determine the degree of dispersion of control and data resulting from testing. Higher standard deviation means more dispersion of control and data. Closer to zero the standard deviation is, more concentrated the control and data are.

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.

Test Diversification is the process of diversifying control and data. Balancing diversifies the test by minimizing the difference between the frequencies of taken and not taken branches in the code. Skewing diversifies the test by pushing the difference between the frequencies of taken and not taken branches towards its extreme points. Balancing and skewing are complimentary diversifying techniques.

References

[1] Sara Baase. Computer algorithms. *Addison-Wesley*, Second Edition:16, 1988.

[2] M. D. Davis and E. J. Weyuker. A formal notion of program-based test data adequacy. *Information and Control*, 56:52–71, 1983.

[3] 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.

[4] P.G. Frankl and E.J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *IEEE Trans. on Soft. Eng.*, 19:202–213, 1993.

[5] 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.

[6] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. *Proceedings of 15th International Conference on Software Engineering*, pages 118–129, 1993.

[7] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. on Soft. Eng.*, 16:1402–1411, 1990.

[8] R. Hamlet. Theoretical comparison of testing methods. *In Proc. 3rd Symp. Testing, Analysis, and Verification*, pages 28–37, 1989.

[9] J.L. Hennesey and D. A. Patterson. Computer architecture a quantitative approach. *Morgan Kaufmann Publishers, INC.*, pages 251–343, 1990.

[10] J. Huang. An approach to program testing. *ACM Computing Surveys*, 7:113–128, 1975.

[11] K.A.Foster. Sensitive test data for logic expressions. *ACM SIGSOFT Software Eng. Notes*, 9:120–126, 1984.

[12] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Soft. Eng.*, SE-9:347–354, 1983.

[13] N. G. Leveson. Safeware system safety and computers. In *Addison-Wesley*, 1995.

[14] M. R. Lyu. Handbook of software reliability engineering. *IEEE Computer Society Press*, 1995.

[15] R.A. De Millo, R.J. Lipton, and F.G.Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 1978.

[16] Borislav Nikolik. Constraint preservation through loops. *Information Processing Letters*, 55:143–148, 1995.

[17] Borislav Nikolik. Ultrareliability of programs specified with equational specifications. *Ph.D. Dissertation*, University of Oregon, 1998.

[18] D. L. Parnas, A. van Schouwen, and S. Kwan. Evaluation of safety-critical software. *Comm. of the ACM*, 33:638–648, 1990.

[19] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, SE-11:367–375, 1985.

[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] Michael Wolfe. Optimizing supercompilers for supercomputers. *MIT Press, Cambridge, Mass.*, 1989.