

# Test Suite Oscillations \*

Borislav Nikolik

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

October 20, 2004

## Abstract

This paper proposes a set of new software test-diversity measures based on control oscillations of test suites. Oscillation Diversity uses conversion, inversion, and phase transformation to vary test suite amplitudes, frequencies, and phases. Resistance and inductance are defined as measures of diversification difficulty. The experimental results show correlation between some Oscillation Diversity measures and fault-detection effectiveness.

**Key Words:** testing strategies, test coverage, testing tools, converter, amplitude, terz, inverter, oscillation.

## 1 Introduction

An important focus of software reliability has been the problem of maximizing the probability of detecting software faults. As a consequence of the fact that in general the distribution of faults in the software is not known prior to testing, software reliability has relied on various diversification approaches in order to increase the chances of fault detection. For instance, design diversity is based on executing diverse implementations of the same program specifications, with the goal of fault detection and recovery[5]. Data diversity is based on executing the same code with diverse inputs[4][1][17]. Note that the code coverage criteria might also be viewed as a form of diversification of program control and data. In particular, data flow and control flow coverage are based on covering diverse data flow and control flow relations in the code, such as def-use or c-use paths[11]. Mutation testing also includes an element of diversification, since it introduces variations in program operators and operands[7]. Random testing[10] is a form of diversification since varied random input samples are selected according to an operational profile. In fact, every test method in existence can be viewed as one form of another of diversification. The problem, however, addressed in this paper is how to measure diversity in a general way, and adjust it based on the obtained measures.

This paper proposes a novel test diversification approach called Oscillation Diversity. This approach involves measur-

ing the degree to which test cases are diversified with respect to the program's control and data. This approach could be used to validate a particular fault-distribution assumption. For instance, under a fault-dispersion assumption, which is the one used in the experiments of section 4, test cases that cause more control and data variation are better, since they increase the chance of detecting dispersed faults. Oscillation Diversity could be used in conjunction with the existing testing methods - most notably the code coverage criteria - in order to increase their effectiveness. For example, the control dispersion of code coverage methods could be measured, and then the dispersion could be adjusted relative to a particular fault-distribution assumption.

Unlike existing testing approaches, Oscillation Diversity is based on introducing variations in the direction and magnitude of the control flow of the program. The differences between the *true* and *false* execution counts of the conditional expressions in the code give control amplitudes. Amplitude diversification could be accomplished by a converter, which varies the magnitudes of the execution counts between test cases in a test suite. A measure of resistance is used as an indicator of the program run time necessary to diversify amplitudes. Standard deviation of amplitudes, resulting from executing multiple test cases, is used as a measure of control diversity.

Test suite frequencies are calculated based on the number of changes of the amplitude direction, where a true-dominated conditional-expression count indicates a positive direction, and a false-dominated count indicates a negative direction. Frequency diversification could be accomplished by an inverter, which varies amplitude direction changes between test suites. Test suite frequencies are used as indicators of internal data combination, a mix of data that causes conditional expressions to evaluate to *true*, and data that causes conditional expressions to evaluate to *false*. Standard deviation of frequencies is used as a measure of control diversity between test suites. Inductance is a measure of opposition to changes in control flow, and it indicates the difficulty of frequency diversification. Test suite phases, which are related to test suite frequencies, represent shifts in the test periods. A phase transformer is used to diversify test phases by selecting different starting points of the test.

Oscillation Diversity has the desirable property of diver-

---

\*Address correspondence to boris@vidakquality.com

sifying the internal program data as a byproduct of control diversification. Internal data diversification is important since the existing testing approaches - in particular code coverage - suffer from the problem of potentially repeatedly covering the program with the same or similar internal data. Diverse data at the program interface level, such as random input data as used in random testing[10], does not guarantee data diversity at the code level. Oscillation Diversity could be used as an indicator of data variation at the code level. The conditions for data diversity presented in this paper are sufficient conditions for data variation.

The experimental results, performed with an industrial strength tool called Diversity analyzer, show that higher amplitude variations and higher test frequencies result in higher fault detection rates.

## 2 Background and Example

We use the C++ string-matching function of Figure 1 to exemplify the oscillation notions presented in this paper. This function implements a straightforward string matching of a pattern P against a text string T[3]. The function returns the index of T where a copy of P begins, if a match for P is found. The index will be the length of T plus one if no match for P is found. Starting at the beginning of P and T, the *match* function compares characters, one after the other, until either P is exhausted or a mismatch is found. When a mismatch is found, the function slides P one more place forward over T and starts again, comparing the first character of P with the next character in T. Note that *i* is not needed since *i* can be expressed as *j-k+1*. We denote the string-matching function of Figure 1 by M.

The control flow in programs is determined by conditional statements, which in turn contain conditional expressions. A *conditional expression* is an expression that evaluates to either *true* or *false*. The number of conditional expression in a program is denoted by  $\beta$ . For example, M consists of three conditional expressions,  $\beta = 3$ , one controlling the *while* loop (denoted by  $e_1$ ), one controlling the *if* statement at line 4 (denoted by  $e_2$ ), and one controlling the *if* statement at line 8 (denoted by  $e_3$ ).

Suppose M was executed on four test cases  $t_1, \dots, t_4$ , which are given in the second column of Figure 2<sup>1</sup>. The *true* and *false* execution counts for  $e_1, e_2$  and  $e_3$ , and the ordered test suite  $\Theta = \{t_1, \dots, t_4\}$  are also given in Figure 2. These execution counts are used to measure the amplitudes and frequency of  $\Theta$ . For example, the amplitude of  $t_1$ , whose units are denoted by D, with respect to  $e_1$  is  $e_1^T - e_1^F = +5D$ , and the amplitude of  $t_1$  with respect to  $e_3$  is  $e_3^T - e_3^F = -1D$ . The frequency of  $\Theta$  with respect to  $e_1$  is zero cycles(0Tz) since the sign of the amplitude does not change. The frequency of  $\Theta$  with respect to  $e_3$  is 1Tz since the sign of the amplitude changes once per  $\Theta$ . The test case  $t_1$  results in direct control flow for  $e_1$ , and  $t_1$  results in alternate control flow for  $e_3$ .  $\Theta$  does not cause any variation in amplitude for  $e_1$  since  $e_1^T = 6D$  and  $e_1^F = 1D$  for each test case in  $\Theta$ . When considered in isolation, the two subsets of  $\Theta$ :  $\{t_1, t_2\}$  and  $\{t_3, t_4\}$ , do not cause any variation in amplitudes for neither

```

int match (char *T, char *P){
1  int i=0,j=0,k=0;
2  int plen=P.length(),tlen=T.length();
3  while (j < tlen && k < plen)    //e1
4    if (T[j] == P[k])            //e2
5      j++, k++;
6  else
7    ++i, j=i, k=0;
8  if (k >= plen)                  //e3
9    return i;
10 else
11 return j;

```

Figure 1: String Matching

	T,P	$e_1^T$	$e_1^F$	$e_2^T$	$e_2^F$	$e_3^T$	$e_3^F$
$t_1$	abcdef,xyz	6	1	0	6	0	1
$t_2$	ghijkl,pqr	6	1	0	6	0	1
$t_3$	abcdef,def	6	1	3	3	1	0
$t_4$	ghijkl,jkl	6	1	3	3	1	0
$t_5$	abcdef,abc	3	1	3	0	1	0
$t_6$	abcdef,bcz	8	1	2	6	0	1
$t_7$	$\epsilon$ ,abcdef	0	1	0	0	0	1

Figure 2: Execution Frequency Counts

of  $e_1, \dots, e_3$ . Rearranging the execution of the test cases of  $\Theta$  to yield  $\Theta' = \{t_2, t_3, t_4, t_1\}$  results in frequency of 3Tz for  $e_3$ . Note that  $\Theta'$  is  $\Theta$  with a phase shift of one, that is,  $\Theta'$  starts with  $t_2$  and ends with  $t_1$  as opposed to starting with  $t_1$  and ending with  $t_4$ . Figure 3 shows the amplitudes for  $e_1$  and  $\Theta'' = \{t_1, \dots, t_7\}$ , with a maximum amplitude of 7D, and overall frequency of 1Tz.

A *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  $\vec{s}$ . For example, in Figure 1, the program state before the first execution of line 3 on test case  $t_1$  is

$$\langle [i \ j \ k \ tlen \ plen \ T \ P][0 \ 0 \ 0 \ 6 \ 3 \ (abcdef) \ (xyz)] \rangle.$$

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  $P$ .

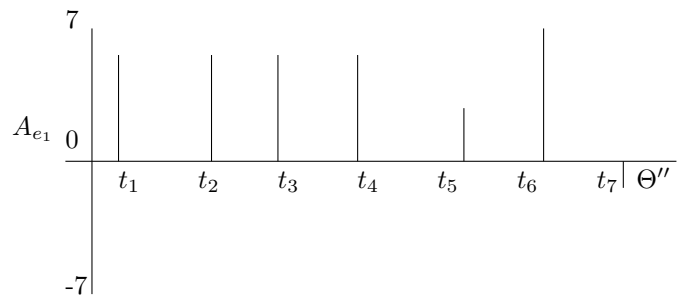


Figure 3: Amplitude and Frequency for  $e_1$  and  $\Theta''$

<sup>1</sup> $\epsilon$  represents an empty string.

A conditional expression  $e_p$  partitions the state space into the set of all states that cause  $e_p$  to evaluate to *true*, and into the set of all states that cause  $e_p$  to evaluate to *false*. For example, the *true* partition of  $e_2$  contains all the states involving  $i, j, k, \text{tlen}, \text{plen}, T$  and  $P$  for which the following holds:

$$j < \text{tlen} \wedge k < \text{plen} \wedge T[j] = T[k].$$

The state partitions defined by conditional expressions are called *conditional expression partitions*.

The difference between  $\vec{S}_{t_i}^P$  and  $\vec{S}_{t_j}^P$ , denoted by  $\vec{S}_{t_i}^P \Delta \vec{S}_{t_j}^P$ , is the number of different program states between  $\vec{S}_{t_i}^P$  and  $\vec{S}_{t_j}^P$ .

The *control space* is a set of all feasible complete paths. The *data space* is the set of all program states.

### 3 Test Oscillation Fundamentals

Oscillation Diversity involves two related processes: measuring amplitude and frequency diversity, and diversifying amplitudes, frequencies and phases of test suites. First, measuring the diversity of amplitudes and frequencies is discussed. The discussion of diversifying amplitudes, frequencies, and phases follows.

#### 3.1 Measuring Diversity

A few formal definitions are needed in order to discuss measuring oscillation diversity. The *count difference* of a test case  $t_i$  for an expression  $e_p$  is the difference of the total execution counts, given by

$$A_{e_p}^{t_i} = e_p^T - e_p^F.$$

The values of  $A_{e_1}^{t_1}, \dots, A_{e_m}^{t_m}$  form an *amplitude vector* of size  $m$ . Figure 4 shows the amplitude vectors for  $M$  and  $\Theta''$ . The difference in amplitudes for two vectors  $a_i$  and  $a_j$ , denoted by  $a_i \Delta a_j$ , is the number of different corresponding elements in  $a_i$  and  $a_j$ . For example,  $a_1 \Delta a_3 = 2$  and  $a_1 \Delta a_1 = 0$ .

The average count difference of a test suite  $\Theta$  of  $n$  test cases  $t_1, \dots, t_n$  for an expression  $e_p$  is given by

$$A_{e_p}^\Theta = \frac{\sum_{i=1}^n A_{e_p}^{t_i}}{n}.$$

The average count difference of a test suite  $\Theta$  for a program  $P$  with  $m$  conditional expressions  $e_1, \dots, e_m$  is given by

$$A_P^\Theta = \frac{\sum_{i=1}^m A_{e_i}^\Theta}{m}.$$

The symbol for count difference is  $\delta$  and its unit is abbreviated by D. Next, we turn our attention to test suite frequencies.

Given a pair of ordered test cases  $\langle t_i, t_j \rangle$ , where  $t_i, t_j \in \Theta$ , a *cycle* for an expression  $e_p$ , denoted by  $C_{e_p}^{\langle t_i, t_j \rangle}$ , is given by

$$(A_{e_p}^{t_i} > 0 \wedge A_{e_p}^{t_j} < 0) \vee (A_{e_p}^{t_i} < 0 \wedge A_{e_p}^{t_j} > 0),$$

where the value of this logical expression is 1 if it evaluates to *true*, and its value is 0 if it evaluates to *false*. Given an ordered sequence of  $n$  test cases  $\langle t_1, \dots, t_n \rangle$ ,  $t_1, \dots, t_n \in \Theta$ , and a program  $P$  with  $m$  conditional expressions  $\langle e_1, \dots, e_m \rangle$ , the ordered set  $S^\Theta$ , of cardinality  $k$ , contains all the pairs  $\langle t_i, t_j \rangle$  of adjacent test cases in the sequence  $\langle t_1, \dots, t_n \rangle$ . Let  $S_i^\Theta$  denote

$$\vec{a}_1 = \begin{bmatrix} 5 \\ -6 \\ -1 \end{bmatrix} \quad \vec{a}_2 = \begin{bmatrix} 5 \\ -6 \\ -1 \end{bmatrix} \quad \vec{a}_3 = \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix} \quad \vec{a}_4 = \begin{bmatrix} 5 \\ 0 \\ 1 \end{bmatrix}$$

$$\vec{a}_5 = \begin{bmatrix} 2 \\ 3 \\ 1 \end{bmatrix} \quad \vec{a}_6 = \begin{bmatrix} 7 \\ -4 \\ -1 \end{bmatrix} \quad \vec{a}_7 = \begin{bmatrix} -1 \\ 0 \\ -1 \end{bmatrix}$$

Figure 4: Amplitude Vectors

the  $i$ -th pair  $p_i$  in the  $S^\Theta$  sequence  $\langle p_1, \dots, p_{i-1}, p_i, p_{i+1}, \dots, p_k \rangle$ . The *frequency* of  $\Theta$  for an expression  $e_p$  is given by

$$F_{e_p}^\Theta = \sum_{i=1}^k C_{e_p}^{S_i^\Theta}.$$

The frequency of  $\Theta$  for a program  $P$  of  $m$  conditional expressions is given by

$$F_P^\Theta = \frac{\sum_{i=1}^m F_{e_i}^\Theta}{m}.$$

The symbol for test frequency is  $\nu$  and its unit is Terz abbreviated by Tz.

$$1\text{Terz} = 1\text{Tz} = 1 \text{ oscillation per test suit} = 1^{-\theta}.$$

Related to the frequency is the period  $T$  of the motion, which is the number of test cases required to complete one oscillation, given by

$$T = \frac{1}{\nu}.$$

Next, measuring amplitude and frequency diversity is discussed. Standard deviation could be used as a measure of dispersion of amplitudes and frequencies, given by

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

where  $X$  is a random variable of the discrete type representing either  $\delta$  or  $\nu$ , and  $\mu = E(X)$  is the arithmetic mean of the values of  $X$ . In the amplitude case,

$$\mu = A_{e_p}^\Theta \quad X = \{A_{e_p}^{t_1}, \dots, A_{e_p}^{t_n}\},$$

and  $\sigma$  is denoted by  $\sigma_\delta^2$ . In the frequency case,

$$\mu = \frac{\sum_{i=1}^n F_P^{\Theta_i}}{n} \quad X = \{F_P^{\Theta_1}, \dots, F_P^{\Theta_n}\},$$

and  $\sigma$  is denoted by  $\sigma_\nu$ .  $\sigma_\delta$  and  $\sigma_\nu$  could be used to measure the effectiveness of test suites in fault detection. For instance, if faults are dispersed in the control space of the program, higher values of  $\sigma_\delta$  and  $\sigma_\nu$  indicate a better test suite because of course such test suite yields more dispersion in the control space. However, if the faults are concentrated in a particular control sub-space, that is, they are concentrated around particular amplitudes and frequencies, higher  $\sigma_\delta$  and  $\sigma_\nu$  would be misleading, since test cases concentrated

<sup>2</sup>If not specified otherwise,  $\sigma_\delta$  denotes the average of the individual standard deviations for each expression  $e_1, \dots, e_n$  in a program, i.e.,  $\sigma_\delta = \frac{\sum_{i=1}^n \sigma_\delta^i}{n}$ .

	$\sigma_\delta$	$\nu$	$L$	$R$	$T$	$\delta$
$\Theta$	1.3	0.6Tz	0.1	0.2	1.6	0.66D
$\Theta''$	2.2	0.8Tz	0.13	0.2	1.25	0.66D

Figure 5: Diversification Statistics

in sub-spaces of high fault density have higher fault detection effectiveness than dispersed test cases.

It is of interest to know the difficulty involved in amplitude and frequency diversification. *Resistance*, denoted by  $R$ , is related to the amount of run time it takes to diversify amplitudes, and it is given by the following necessary condition

$$R = \frac{\delta}{\beta}.$$

In order to achieve higher  $\delta$  the difference between the *true* and *false* execution counts has to be greater, which is obtained by executing the conditional expressions more times. And conversely, executing conditional expressions less times achieves lower  $\delta$ . The resistance is reverse proportional to the number of conditional expressions in the code, that is, lower  $\beta$  means less conditional expressions there are in the code, and as a result, less execution time is needed to diversify amplitudes. *Inductance*, denoted by  $L$ , opposes changes in control flow, and it is given by

$$L = \frac{\nu}{|\Theta| - 1},$$

where  $|\Theta|$  is the cardinality of  $\Theta$ .  $L$  is the ratio of the actual number of control flow changes per  $\Theta$  and the maximum number of Tz for a test suite of cardinality  $\Theta$ .

Figure 5 shows the particular values for the above definitions for two test suites,  $\Theta = \{t_1, \dots, t_4\}$  and  $\Theta'' = \{t_1, \dots, t_7\}$ . Note that  $\Theta''$  has higher  $\sigma_\delta$  than  $\Theta$  which reflects the fact that  $\Theta''$  diversifies amplitudes better.  $\Theta''$  has 0.2Tz more than  $\Theta$ , which gives a shorter period  $T$ , and a higher induction for  $\Theta''$ . The resistance is the same for  $\Theta''$  and  $\Theta$  since both have the same  $\delta = .66D$  and same  $\beta = 3$ .

### 3.2 Diversifying Test Suites

This section discusses procedures for diversifying amplitudes, frequencies and phases. Given that, in general, the problem of generating test cases that result in particular amplitudes and frequencies is undecidable, heuristics are suggested for diversifying amplitudes and frequencies. A *Converter*, denoted by  $C$ ,  $C : \delta \rightarrow \delta$ , diversifies amplitudes by utilizing a heuristics where by to increase  $\delta$ ,  $C$  generates test cases that run the code for a longer time, and in order to decrease  $\delta$ ,  $C$  generates test cases that run the code for a shorter time. This could usually be done by running the program with higher or lower volumes of inputs. For example,  $T = P = \epsilon$  is a low volume input for  $M$ , which decreases  $\delta$  relative to each  $t_1, \dots, t_6$ . A high-volume input for  $M$ , where  $T = P$  consisting of  $10^3$  characters, would significantly increase  $\delta$  relative to each  $t_1, \dots, t_7$ .

An *Inverter*, denoted by  $I$ ,  $I : \nu \rightarrow \nu$ , increases the Tz value of a test suite by changing the test case execution sequence

in cases where  $Tz > 0$ . For example, to increase  $\nu$  for  $M$ ,  $\Theta''$  could be rearranged to give  $\Theta' = \{t_1, t_5, t_2, t_3, t_6, t_4, t_7\}$ .  $\nu$  could be maximized by forming sets of permutations of  $\Theta$ , computing  $\nu$  for each set, and selecting the set which has the highest  $\nu$  value. In practice, diversifying amplitudes and frequencies would be done along the lines of what is currently done in test coverage practice, that is, repeated test creation, execution, and measurement until desired diversity is achieved, or until the product has to be released due to market pressures. Given a fixed number of test cases to be generated, this trial and error diversification procedure might in general not be more expensive than code coverage, and in many cases it might be cheaper than coverage when the above-mentioned heuristics are used.

A *Phase Transformer*, denoted by  $\Phi$ ,  $\Phi : \Theta \rightarrow \Theta'$ , selects a starting point  $t_i$  in  $\Theta$  and places in  $\Theta'$   $t_i$  and all of the test cases following  $t_i$  in the sequence  $\Theta$ , and then placing in  $\Theta'$  all the test cases preceding  $t_i$  in  $\Theta$ . For example, a phase transformation of two of  $\Theta''$ , gives  $\{t_3, t_4, t_5, t_6, t_7, t_1, t_2\}$ .

Program control is related to internal program states. In particular, we observe that higher amplitude differences result in higher data diversity, that is, that differences in total program state between the test cases are higher. The following lower-bound condition captures this observation. Let  $a_1$  and  $a_2$  be amplitude vectors of  $k$  count differences.

$$(\vec{a}_i \Delta \vec{a}_j) = k \Rightarrow (\vec{S}_{t_i}^P \Delta \vec{S}_{t_j}^P) \geq k.$$

If  $(\vec{a}_i \Delta \vec{a}_j) = k$  then there are  $k$  corresponding elements in  $a_i$  and  $a_j$  that differ, that is, there are  $k$  conditional expressions,  $e_1, \dots, e_k$ , for which  $a_i$  and  $a_j$  differ. If the program state before each execution of  $e_{p, 1 \leq p \leq k}$  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 corresponding values for  $a_i$  and  $a_j$ . This is impossible since  $\vec{a}_i$  and  $\vec{a}_j$  differ on  $e_p$ . Therefore,  $t_i$  and  $t_j$  result in at least one different program state in  $\vec{S}_{t_i}^P$  and  $\vec{S}_{t_j}^P$  for each  $e_p$ . Since there are  $k$  such conditional expressions,  $(\vec{S}_{t_i}^P \Delta \vec{S}_{t_j}^P) \geq k$ .

The practical implication of this argument is in the fact that conditional-expression execution counts can be used to reason about internal data variation. A corollary of the condition is that the different states between  $t_i$  and  $t_j$  are distributed across  $k$  different expressions<sup>3</sup>. Note that state differences at a conditional expression  $e_m$  do not guarantee difference on other expressions that follow  $e_m$  in the execution chain, since the difference could, but do not necessarily have to, be annulled by assignment statement following  $e_m$ [16]. Note that same conditional-expression counts do not imply same input or intermediate data. For example, in Figure 2 test cases  $t_1$  and  $t_2$  result in identical counts. Furthermore, in deterministic programs, same test cases should give the same counts. Therefore, same counts could be caused by both same or different data. These problems are a symptom of the fact that different conditional-expression counts is a sufficient but not a necessary condition for data variation.

Test suite frequencies are also related to data diversity. That is, if  $F_P^{\Theta'} > F_P^\Theta$  then  $\Theta'$  results in a more combination of data across conditional-expression partition boundaries. For

<sup>3</sup>This is not necessarily a uniform distribution since at least one of the  $k+$  different states is attributed to each  $e_p$ .

	$\sigma_\delta$	$\nu$	faults	tests	coverage
$p_1$	0.54	35Tz	38	850	63%
$p_2$	0.43	20Tz	27	710	72%
$p_3$	0.35	9Tz	21	690	68%

Figure 6: Overall Project Statistics

$\sigma_\delta$	[0,.2]	[.2,.4]	[.4,.6]	[.6,.8]	baseline
$p_1$	5%	7%	30%	58%	10%
$p_2$	2%	15%	17%	66%	5%
$p_3$	10%	11%	15%	64%	7%

$\nu$	[0,10]	[10,20]	[20,30]	[30,40]	baseline
$p_1$	20%	22%	25%	33%	15%
$p_2$	15%	20%	20%	45%	11%
$p_3$	20%	25%	25%	30%	12%

Figure 7: Failure Distribution over  $\sigma_\delta$  and  $\nu$

example, the test cases of  $\Theta''$  result in more data combination than  $\Theta$  across expression partitions for all  $e_1, \dots, e_3$  in  $M$ .

When faults are dispersed in the data space of the program, amplitude  $\Delta$  could be used as a measure of the effectiveness of test suites in fault detection. However, if the faults are concentrated in particular internal data regions, that is, they are concentrated around particular program states, amplitude  $\Delta$  would be misleading.

## 4 Experimental Evaluation

The experimentation uses an industrial-strength tool, called Diversity Analyzer, that can obtain execution counts and calculate various diversity metrics for source code written in C, C++, C#, Java and VB in .NET and Windows environments[18]. The goal of the experimentation was to explore the relationship between  $\sigma_\delta$ ,  $\nu$  and failures. In particular, the hypothesis being tested is that the number of faults detected grows as  $\sigma_\delta$  and  $\nu$  grow, which in turn suggests a fault-dispersion assumption, since these two measures are indicators of control dispersion. Of course, one could always write a program that either validates or invalidates the hypothesis.

The Diversity Analyzer has been 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. The cost of instrumenting the source code, collecting frequency data, and calculating diversity does not add any significant cost as compared to traditional code coverage used in commercial tools[18].

As a starting point to the experimentation evaluation already existing regression tests were taken, 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 overall  $\sigma_\delta$  and  $\nu$  for each project, the total number of naturally-occurring faults detected, the number of total test cases executed, and the branch coverage obtained by the tests are given in Figure 6.

The evaluation of the correlation between  $\sigma_\delta$ ,  $\nu$  and the fault-detection capabilities is performed by identifying subsets of the regression test suite and measuring, for each subset, its  $\sigma_\delta$ ,  $\nu$  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. These subsets were identified in such a way that the  $\sigma_\delta$  and  $\nu$  of 25 subsets fell in the range [0, .2] and [0, 10] respectively, the  $\sigma_\delta$  and  $\nu$  of 25 of the subsets fell in the range [.2, .4] and [10, 20], the  $\sigma_\delta$  and  $\nu$  of 25 subsets fell in the range [.4, .6] and [20, 30], and the  $\sigma_\delta$  and  $\nu$  of 25 subsets fell in the range [.6, .8] and [30, 40]. The number of faults detected by each subset was noted for each project. 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  $\sigma_\delta$  and  $\nu$  is summarized in Figure 7. It also shows the percentage of faults detected by the baseline test. Figure 7 gives the percentage of total faults detected in each  $\sigma_\delta$  and  $\nu$  range. The data of Figure 7 shows that subsets with higher  $\sigma_\delta$  and  $\nu$  values are more effective in detecting faults for projects  $p_1, \dots, p_3$ . The steepest fault growth occurs in the [.6, .8] and [30, 40] range, which gives a motivation for obtaining high levels of  $\sigma_\delta$  and  $\nu$  in order to get maximum fault detection rates. The data shows that the relation between  $\sigma_\delta$  and faults is much stronger than the relation between  $\nu$  and faults.

More evaluation is needed to either validate or invalidate the fault-dispersion hypothesis. Of particular interest would be results from other researchers with different types of industrial experimental subjects, and possibly different types of experiments.

## 5 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[23]. However, note that in general there is no relation between an even spread of test cases and diversity at the code level. Unlike adaptive random testing, Oscillation Diversity measures diversity at the code level. Automating statistical testing involves a heuristic that is aimed at uniform generation of execution paths[8].  $\sigma_\delta$  and  $\sigma_\nu$  are not based on heuristics, they can be effectively obtained by

the Diversity Analyzer, and the notions of amplitude, oscillation, conversion, inversion, phase transformation, resistance, and inductance 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. Oscillation 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%. Oscillation 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.

Oscillation Diversity is closely related to code coverage. In particular, it 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 values for the measures defined in this paper. 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, Oscillation Diversity could be used to determine the quality of decision coverage. This does *not* evaluate the quality of the criterion itself as is done in [6], 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 [14]. 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]. 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, Oscillation Diversity could be used to determine the quality of data-flow coverage by measuring data diversity for the particular test suite. In fact, Oscillation Diversity 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.

Oscillation Diversity is also related to performance optimization. Branch prediction makes static and dynamic guesses on the branch that would be executed next[9]. The static analysis is done at compile time where it is based on general knowledge of loop executions<sup>4</sup>, while the dynamic analysis is done at the microprocessor pipeline level, where predictions are based on, for example, one-bit or two-bit pre-

<sup>4</sup>Guesses based on findings such as loop branches are taken with 90% probability[12].

diction schemes[12]. Source code optimization and profiling give execution-time distributions at the source-code level, and at the object-code level[22]. 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 compute the measures defined in this paper. In addition, profiles are mainly based on statistical sampling in order to reduce the amount of performance data gathered. Oscillation Diversity is based on exact run time information.

Oscillation Diversity is related to the operational abstraction approach[15], to cluster analysis[20], to the software tomography approach[13], 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.

Oscillation Diversity is related to the  $\pi$  measure, which is a complexity metric aimed at measuring statistical independence of test cases[18]. The  $\pi$  metric is based on measuring the angles between control diversity vectors in order to assess test case independence.

## 6 Summary

Oscillation Diversity involves measuring amplitudes and frequencies of test suites in order to qualify the control and data diversity of a test suite. Higher amplitude variations among test cases result in higher control diversity, and in higher internal data diversity among test cases. Resistance is a measure related to the amount of run time it takes to diversify amplitudes. Inductance is a measure of opposition to changes in control flow. Test suites with higher inductance values result in lower Terz values. Standard deviation of amplitudes and frequencies is used as a measure of control and data diversity. Diversifying program control and data could be done by utilizing converters, inverters, and phase transformers by varying test suite amplitudes, frequencies and phases.

The experimentation done in this paper shows that higher standard deviations of amplitudes and higher test suite frequencies results in higher fault detection rates.

## References

- [1] P. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37:418–425, 1988.

- [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. Blum and S. Kannan. Designing programs that check their work. *JACM*, 42:269–291, 1995.
- [5] Susan Brilliant, J. C. Knight, and N.G. Leveson. Analysis of faults in an n-version software experiment. *IEEE Trans. on Soft. Eng.*, 16, 1990.
- [6] M. D. Davis and E. J. Weyuker. A formal notion of program-based test data adequacy. *Information and Control*, 56:52–71, 1983.
- [7] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: help for the practicing programmer. *Computer*, 11:34–43, 1978.
- [8] 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.
- [9] 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.
- [10] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.
- [11] D. Hamlet, B. Gifford, and B. Nikolik. Exploring dataflow testing of arrays. *Proceedings of 15th International Conference on Software Engineering*, pages 118–129, 1993.
- [12] J.L. Hennesey and D. A. Patterson. Computer architecture a quantitative approach. *Morgan Kaufmann Publishers, INC.*, pages 251–343, 1990.
- [13] Bowring J, Orso A, and Harrold M.J. Monitoring deployed software using software tomography. *ACM SIGPLAN-SIGSOFT PASTE 2002*, pages 2–9, November, 2002.
- [14] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Soft. Eng.*, SE-9:347–354, 1983.
- [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] Borislav Nikolik. Ultrareliability of programs specified with equational specifications. *Ph.D. Dissertation*, University of Oregon, 1998.
- [18] Borislav Nikolik. The  $\pi$  measure. *IEEE METRICS 2004* [www.vidakquality.com/meas.pdf](http://www.vidakquality.com/meas.pdf), September, 2004.
- [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] Dickinson W, Leon D, and Podgurski A. Finding failures by cluster analysis of execution profiles. *ICSE 2001*, pages 339–348, 2001.
- [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.
- [23] 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.