

# The $\pi$ Measure \*

Borislav Nikolik

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

December 01, 2003

## Abstract

We propose a novel software measure, called the  $\pi$  measure, used for measuring test case independence. The  $\pi$  measure is interpreted as a degree of run-time control and data diversity at the code level, resulting from executing a program on a set of test cases. Unlike other well-known static complexity measures, the  $\pi$  measure is a dynamic measure, computed using only run-time information. The Diversity Analyzer computes the  $\pi$  measure for programs written in C, C++, C#, and VB in .NET.

Our experimental data shows a correlation between the  $\pi$  measure, test case independence, failure detection rates, and dynamic software complexity.

**Key Words:** Conditional diversity, data diversity, linear independence, orthogonality, diversity angle,  $\pi$  measure,  $\pi$ -strength hypothesis.

## 1 Introduction

Measuring software complexity has played an important part in the software engineering field. Software complexity measures are generally divided into static and dynamic types. The static measures are concerned with measuring program attributes at rest, such as program size and the complexity of its structure. The dynamic measures are a product of static complexity, computed by static analysis of the program, and an operational environment of the program[26]. Unlike these dynamic measures, the  $\pi$  measure does not involve static analysis of the program, rather, the  $\pi$  measure is based *only* on run-time code properties. According to the  $\pi$  metric, a software is more complex if it is executed in more diverse ways.

More complex code, based on a measure of diversity, contains higher degree of control and data "surprise" than less complex code. That is, code with low complexity tends to group same or similar executions, whereas highly complex code could change control flows and data flows in highly diversified ways. Test cases that cause same or similar control flows and data flows through the program tend to be dependent, since they all tend to be grouped around either correct

or faulty executions. We observe that good test case design and highly complex software intersect at test case independence. That is, test cases designed with higher degree of independence use the software in more complex ways, and, consequently, detect more faults.

Test case design is a fundamental activity involved in every type of testing. Without test case design there will be no test cases created and consequently no program would ever get tested. In general, test designers tend to operate under the independence of test cases principle, since they normally would stay away from creating obvious dependence, such as completely identical test cases, or test cases that are apparently different but are suspected to hit the same program fault[20]. Furthermore, stochastic independence of test cases is a basic assumption in software reliability engineering[24]. Software reliability models almost always assume that failures are independent by either assuming that failure times are independent of each other or by making the Poisson process assumption of independent increments[18]. In order for a statistical theory of correctness to work, the input samples have to be independent[5].

However, different test cases at the program interface level do *not* necessarily produce independence at the code level[8][5]. Apparently different test cases at the interface level, chosen at random, or chosen by following a specific design technique, could result in arbitrarily same or similar control flows and data flows at the code level[7][19]. We use the  $\pi$  measure as an indicator of independence of test cases. We argue that test cases with more independence exercise the program in more complex ways, and as such, expose more unrelated faults. In particular, more control and data variation in the code indicates more complex code, since such code carries more control and data surprise in execution. Code with no control and data variation always performs an identical computation, and as such is of trivial complexity regardless of other known measures, such as, for example, software science metrics[4], cyclomatic complexity[17], or a combination of the two[15]. On the other side of the spectrum, code with infinite control and data diversity always performs different computation, and as such is of highest complexity regardless of other known measures such as, for example, lines of code[25], relative program complexity[13] or bandwidth metric[21].

In this paper, the complexity of a program depends on

---

\*To be presented at IEEE METRICS 2004.

```

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

the actual execution of the program, that is, complexity is defined in terms of the actual diversity a set of test cases carries for a particular program. For instance, a program with 10 lines of code is of higher complexity if the test set exercises it in independent, highly diverse ways, than a program with 1000 lines of code whose test set contains only one identical test case repeated many times. This observation, called the  $\pi$ -strength hypothesis, is supported in this paper by failure data with respect to the two well-known static complexity measures: lines of code and number of branches.

We present an industrial-strength tool, called Diversity Analyzer, that computes diversity of test suites for code written in C, C++, C#, and Visual Basic for Microsoft Windows and .NET. The tool uses these diversity values to compute the  $\pi$  measure. Furthermore, the Diversity Analyzer computes suggested frequencies of branch executions that maximize the  $\pi$  measure. We have used the Diversity Analyzer to conduct an experimental evaluation of the  $\pi$  measure.

## 2 Background and Example

To demonstrate the central notions of this paper, we use the instrumented BubbleSort[2] code fragment in Figure 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 a *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  $c(\text{true}, i)$  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.

Consider the four test cases  $t_1, \dots, t_4$  given in Figure 3. Suppose the code of Figure 1 was executed on test cases  $t_1, \dots, t_4$ . The *true* execution counts T and *false* execution counts F for  $e_1$ ,  $e_2$  and  $e_3$  and test cases  $t_1, \dots, t_4$  are given in Figure 3. Conditional diversity for a conditional expression, such as  $e_1$  or  $e_2$  from Figure 1, is calculated as a *distance* between the actual distribution for that expression from the uniform dis-

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

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

Figure 2: Conditional Diversity

	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
$t_4$	$\langle 4, 3, 2, 1 \rangle$	4	1	6	4	6	0

Figure 3: Execution Frequency Counts

tribution for the same expression, as shown in Figure 2<sup>1</sup>. 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.

Data diversity for a conditional expression  $e_p$  is a percentage of distinct conditional diversities given by  $t_1, \dots, t_n$  for  $e_p$ .

The Diversity Analyzer uses the frequency counts produced by the instrumented version of the code fragment to calculate the conditional diversities for  $e_1$ ,  $e_2$  and  $e_3$  for  $t_1, \dots, t_4$ , as shown in Figure 4. Note that the columns under  $t_1, \dots, t_4$  in Figure 4 form *conditional diversity vectors*. These vectors form a *conditional diversity matrix* C, a matrix whose columns are conditional diversity vectors corresponding to executions of the test cases  $t_1, \dots, t_4$ . The code fragment of Figure 1 and the test cases of Figure 3 give the results in Figure 5.

The test case independence metric defined in this paper is a trigonometric measure involving angles between diversity vectors. The cosine of the angle between any two vectors  $a$  and  $b$  is given in [3] as:

$$\cos \theta = \frac{a^T b}{\|a\| \times \|b\|}.$$

The numerator is obtained by transposing  $\vec{a}$  and multiplying it with  $\vec{b}$ . The denominator is the product of the lengths of  $\vec{a}$  and  $\vec{b}$ . A length of a vector  $x$  is given by

$$\|x\|^2 = x_1^2 + x_2^2 + \dots + x_n^2 = x^T x.$$

We briefly recall the cosine values of some interesting angles. The cosine of 0 and  $2\pi$  radians is 1. The cosine of  $\pi/2$  and  $3\pi/2$  is 0. The  $\pi/2$  and  $3\pi/2$  angles are special cases in vector algebra since vectors under these angles are orthogonal and linearly independent. The cosine of  $\pi$  is  $-1$ .

<sup>1</sup>For presentation purposes, we prefer the given formula to the simpler versions of the formula.

	$t_1$	$t_2$	$t_3$	$t_4$
$e_1$	.3	0	0	.6
$e_2$	.4	.5	.5	.2
$e_3$	-.6	-1	-1	1

Figure 4: Conditional Diversity

$$\vec{c}_1 = \begin{bmatrix} .3 \\ .4 \\ -.6 \end{bmatrix} \quad \vec{c}_2 = \begin{bmatrix} 0 \\ .5 \\ -1 \end{bmatrix} \quad \vec{c}_3 = \begin{bmatrix} 0 \\ .5 \\ -1 \end{bmatrix} \quad \vec{c}_4 = \begin{bmatrix} .6 \\ .2 \\ 1 \end{bmatrix}$$

$$C = \begin{bmatrix} .3 & 0 & 0 & .6 \\ .4 & .5 & .5 & .2 \\ -.6 & -1 & -1 & 1 \end{bmatrix}$$

Figure 5: Diversity Matrix and Vectors

Notice that reversing the sign of  $\vec{b}$  reverses the sign of  $\cos \theta$  and changes the angle by  $\pi$  radians. The approximate angles among the vectors of Figure 5 are summarized in Figure 6. The biggest angle is between vectors  $\vec{c}_2$  and  $\vec{c}_4$ , and vectors  $\vec{c}_3$  and  $\vec{c}_4$ . The smallest angle is between vectors  $\vec{c}_2$  and  $\vec{c}_3$ . Changing the sign of  $\vec{c}_3$  makes  $\vec{c}_2$  and  $\vec{c}_3$  exactly under  $\pi$  radians.

Test cases whose diversity vectors are under 0 or  $2\pi$  radians are effectively the same test case since they result in similar control flow through the program under test. Test cases under  $\pi$  radians represent opposite test cases since the sign of the corresponding diversity values in the diversity vectors are opposite of each other. Such test cases result in opposite frequency counts for each conditional expression in the code. That is, if one test case results in a *true* dominated count, the other test results in a *false* dominated count.

The  $\pi$  measure is a central notion in this paper, which is primarily motivated by our intuition that "opposite" control flows are less likely to hit the same faults in the program. That is, sharper diversity angles mean more test case dependence, and such test cases exercise the code in less complex ways. Next, we give a precise definition of the  $\pi$  measure.

Let  $v$  be a set of all pairs of distinct vectors in the vector set  $m$ . Let  $v$  have cardinality of  $n$ . The  $\pi$  measure is computed

$\angle$	$\cos \theta$	$\theta^\circ$
$c_1 c_2$	0.93	24.6
$c_1 c_3$	0.93	24.6
$c_1 c_4$	-0.36	111
$c_2 c_3$	1	0
$c_2 c_4$	-0.69	113
$c_3 c_4$	-0.69	113

Figure 6: Diversity Angles

as a ratio in the interval  $[0, 1]$  as

$$S_v^m = \frac{\pi - \frac{\sum_{i=1}^n (\pi - \angle(v_i))}{n}}{\pi - \frac{\pi(n-kq)}{\bar{m}}},$$

where  $k$  is the floor of the number of vectors in  $m$  divided by 2. The variable  $q$  is the number of vectors in  $m$  minus  $k$ , and  $\bar{m}$  is the cardinality of the set  $m$ .  $\angle(v_i)$  denotes the angle in radians between the vectors in the pair  $v_i$ . The numerator of  $S_v^m$  is the actual measure of angle sharpness among vectors in  $m$ . In particular, it is the average of the distances between a  $180^\circ$  angle and the actual angles. The denominator is the best case of angle flatness among vectors in  $m$ . For example, if  $m'$  is the set of vectors

$$m' = \vec{c}_1, \vec{c}_2, \vec{c}_3,$$

then the set of all distinct pairs is

$$v' = \langle c_1 c_2 \rangle, \langle c_1 c_3 \rangle, \langle c_2 c_3 \rangle,$$

and the  $\pi$  measure is  $S_{v'}^{m'} \approx 16.4^\circ / 120^\circ \approx 0.136$ . If  $m''$  is the set of vectors

$$m'' = \vec{c}_1, \vec{c}_2, \vec{c}_4,$$

then the set of all distinct pairs is

$$v'' = \langle c_1 c_2 \rangle, \langle c_1 c_4 \rangle, \langle c_2 c_4 \rangle,$$

and the  $\pi$  measure is  $S_{v''}^{m''} \approx 82^\circ / 120^\circ \approx 0.7$ . In this example, the vectors of  $m'$  are under sharper angles than the vectors in  $m''$  since  $m''$  has a higher value for the  $\pi$  measure than  $m'$ . The  $\pi$  measure reflects the fact that  $m''$  is a much superior test (w.r.t diversity) than  $m'$ . While  $m'$  uses two thirds of the test cases to test already sorted lists in ascending order,  $m''$  replaces one third of that with sorting a list which is already sorted in *descending* order.

Using the formula for computing the cosine of the angle between two vectors, the  $\pi$  formula becomes

$$S_v^m = \frac{\pi - \frac{\sum_{i=1}^n (\pi - \text{acos}(\frac{a_i^T b_i}{\|a_i\| \|b_i\|}))}{n}}{\pi - \frac{\pi(n-kq)}{\bar{m}}},$$

where the diversity vectors  $a_i$  and  $b_i$  are a pair in the set  $v$ , and  $\text{acos}(x)$  is the arcus cosinus of  $x$ . This formula lends itself to a straightforward conversion to an algorithm for computing the  $\pi$  measure.

### 3 Complexity

The angles between conditional diversity vectors could be used as means of measuring test case independence, and measuring dynamic code complexity. The  $\pi$  measure captures the notion of diversity angles relative to a test set. That is, a software is as complex as test cases show it to be. As a result, the same code can have multiple complexities, depending on its actual execution. It is the angles between the particular test cases that determine the complexity of the program.

For example, test cases  $t_2$  and  $t_3$  of Figure 3 show a lower complexity than test cases  $t_1$  and  $t_4$  for the code of Figure

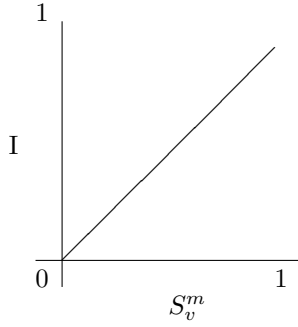


Figure 7: Independence and Diversity

1. Test cases  $t_1$  and  $t_4$  show diversity of  $111^\circ$ . Test cases  $t_2$  and  $t_3$  are under  $0^\circ$  and are effectively identical with respect to control flow. Note that if test case  $t_3$  is rotated by  $\pi$  radians,  $t_2$  and  $t_3$  would show higher complexity than  $t_1$  and  $t_4$  because they would represent opposite test cases, positioned exactly under  $180^\circ$ , and as such represent opposite control flow. Vector rotation is a frequent operation in linear algebra. For example, the matrix  $Q$  rotates every vector of two elements through the angle  $\theta$ .

$$Q = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix}.$$

Rotated diversity vectors could be used by a tester as a *guide* for constructing vectors that maximize the  $\pi$  measure.

Test cases that form flat angles under  $180^\circ$  represent "opposite" test cases, and as such are highly independent. On the other hand, test cases that form angles of  $0^\circ$  and  $360^\circ$  are essentially control equivalent and as such are highly dependent. In between these extremes falls a special class of test cases that form angles under  $90^\circ$ . These are orthogonal test cases resulting in linear independence of program control. Linearly independent diversity vectors are *not* a linear combination of each other.

### 3.1 Test Case Independence

Test case independence is an important assumption of software reliability engineering, and it plays an important role in test case design. Therefore, it is important to be able to measure it and improve it.

Statistical independence is defined in terms of marginal probability density functions. In particular, let two random variables  $X_1$  and  $X_2$ , representing failure of a program on test cases, have the joint probability density function  $f(x_1, x_2)$ .  $X_1$  and  $X_2$  are stochastically independent if

$$f(x_1, x_2) = f_1(x_1)f_2(x_2),$$

where  $f_1(x_1)$  and  $f_2(x_2)$  are marginal probability density functions[23]. Test case independence, denoted by  $I$  in this paper, is the probability of *unrelated* failures occurring when a program is executed on a set of test cases. That is,  $I$  is higher when chances of a set of test cases exposing unrelated failures is higher. Unrelated failures occur when the test cases hit unrelated faults in the code. Note that test cases could result in the same branch coverage and yet hit unrelated faults in the code, since they could execute the exact same code in different ways. The  $\pi$  measure is a suitable indicator of test

case independence, since it captures the intuition that test cases with opposite control flow are less likely to hit the same faults in the program. We hypothesize that test case independence,  $I$ , is proportional to the  $\pi$  measure, as given in the following formula, and represented graphically in Figure 7.

$$I = S_v^m,$$

where  $S_v^m$  is the  $\pi$  measure which is defined in Section 2. The linear relation between  $I$  and  $S_v^m$  is supported by experimental data in Section 5.

The reasoning behind choosing test cases with high  $I$  values is that such test cases result in control flow that is further apart. For example, if the diversity values of a test case  $t_j$  are the values of  $t_i$  negated, the diversity vectors for the two test cases  $t_i$  and  $t_j$  are under  $\pi$  radians. That means that for every frequency count dominated by the *true* count in  $t_i$ , the same frequency count is dominated by the *false* count in  $t_j$ . An example of such diversity values are 1 and  $-1$  of  $\vec{c}_3$  and  $\vec{c}_4$  from Figure 5.

Linear independence is a special case of the  $\pi$  measure where neither test case results in a conditional diversity vector that is a linear combination of the other conditional diversity vectors. Linearly dependent test cases result in linearly dependent frequency counts. For example, two linearly dependent test cases might differ in that one test case causes every *false* branch to be taken one more time than the other test case, given the *true* counts are identical for both test cases. Such execution patterns are clearly undesirable. Next we recall the linear independence and orthogonality definitions.

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. 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 linear independence[3].

Two vectors  $\vec{x}$  and  $\vec{y}$  are *orthogonal* if

$$x_1y_1 + \dots + x_ny_n = 0.$$

The vectors  $\vec{c}_1, \dots, \vec{c}_4$  from Figure 5 are not linearly independent and orthogonal since  $\vec{c}_2 = \vec{c}_3$ . However,  $\vec{c}_1, \vec{c}_2, \vec{c}_4$  are linearly independent and orthogonal. To check any set of vectors  $\vec{v}_1, \dots, \vec{v}_n$  for linear independence, form a diversity 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.

It is important to note that control and data are tightly related with respect to diversity in the following way. 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  $t_j$ . Obviously, 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  $t_j$ . This would give identical conditional diversity values  $e_p$ . But this is impossible since the conditional diversities differ for  $e_p$ . Therefore,  $t_i$  and  $t_j$  result in different program states. Effectively, the  $\pi$  measure is indicative of data diversity as well. That is, higher test case independence involves higher degree of data variation among the test cases.

### 3.2 Maximizing the $\pi$ Measure

Since higher values for the  $\pi$  measure indicate higher test case independence, and exposes higher code complexity, we are interested in maximizing the  $\pi$  measure. The  $\pi$  measure is at its maximum when  $S_v^m = 1$ . Using the definition of  $S_v^m$  from Section 2, and simplifying, yields

$$\frac{\sum_{i=1}^n (\pi - \arccos(\frac{a_i^T b_i}{\|a_i\| \|b_i\|}))}{\frac{n}{\pi(n-kq)}} = 1.$$

Further simplification gives the average of diversity angles as

$$\overline{\angle(v_i)} = \frac{\overline{m} + kq - n}{\overline{m}} \pi.$$

For example, if there are three diversity vectors, then  $\overline{m} = 3$ ,  $n = 3$ ,  $k = 1$ ,  $q = 2$ , and  $\angle(v_i) = 2/3\pi = 120^\circ$  maximizes  $S_v^m$ . To maximize the  $\pi$  measure, the average of the angles between diversity vectors has to be as high as possible. Here we are particularly interested in the case where some arbitrary test suit already exists, and we need to generate an additional test case such that its diversity vector maximizes the  $\pi$  value.

Given a set of an arbitrarily chosen diversity vectors  $m$ , the additional vector  $c_k$  that maximizes  $S_v^p$ , where  $p$  is  $m$  with the additional vector  $c_k$ , is computed as follows. For each vector  $c_i$  in  $p$ ,  $c_k$  is formed by negating  $c_i$ . If  $c_k$  is unique,  $S_v^p$  is computed. The such formed vector  $c_k$  that produces the highest value for  $S_v^p$  maximizes  $S_v^p$ . Intuitively, the maximization is obtained by forming a new vector under  $180^\circ$  degrees for each existing vector in the set, and selecting the new vector that maximizes the  $\pi$  measure.

However, the  $c_k$  vector may *not* be unique. More importantly, it is uncomputable to determine if there are test cases that form a particular angle. Therefore, the computed vector is only a suggested vector, intended to be used as a guide to what needs to be constructed. Additionally, maximizing the  $\pi$  measure may mean interfering with the operational profile of the program, since only particular operational profiles may allow maximizing the  $\pi$  measure. An *operational profile* is a function that assigns to each program input a probability that in normal usage the input will be selected for execution[5].

In practice, maximizing the  $\pi$  measure is an iterative process of creating test cases that are believed to maximize the  $\pi$  measure, executing the program on the test cases, and measuring, along the lines of what is currently done in traditional coverage practice[14]. The iteration stops when either the target value of the  $\pi$  measure is achieved, or other factors, such as schedule pressures, dictate termination of testing and product release. In order to maximize the  $\pi$  measure, the tester could use the suggested vector that maximizes the  $\pi$  measure as a general guide, in conjunction with other relevant information, such as perhaps knowledge of the code base, and

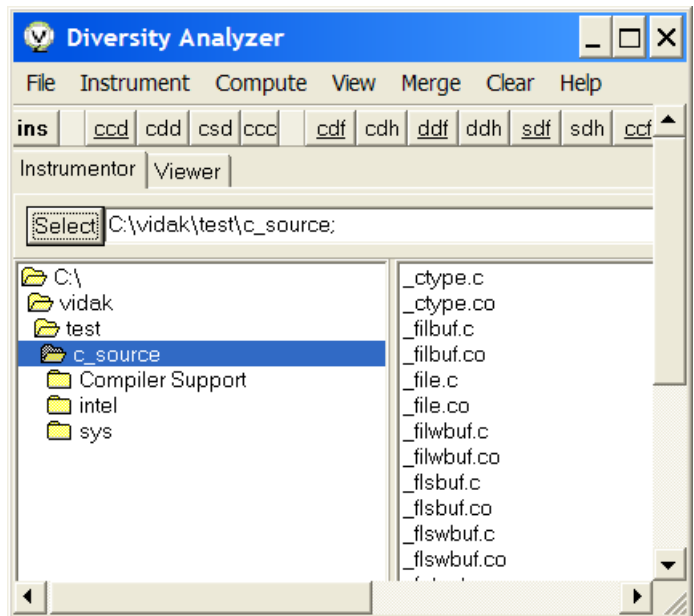


Figure 8: Instrumentor

maximization insights from similar versions of the code. The Diversity Analyzer computes suggested diversity vectors that would maximize the  $\pi$  measure if such vectors are feasible.

## 4 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<sup>2</sup>. 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 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 8. If testing of parts of code is desired, zooming-in is performed for obtaining fine diversity, where the parser only locates the conditional statements delimited by the zoom keywords: ZOOM\_BEGIN and ZOOM\_END<sup>3</sup>.

<sup>2</sup>The tool is relatively easily extendable to Unix/Linux environments, and other languages, such as Java.

<sup>3</sup>The tester may place these keywords in the code around sections of code of particular interest.

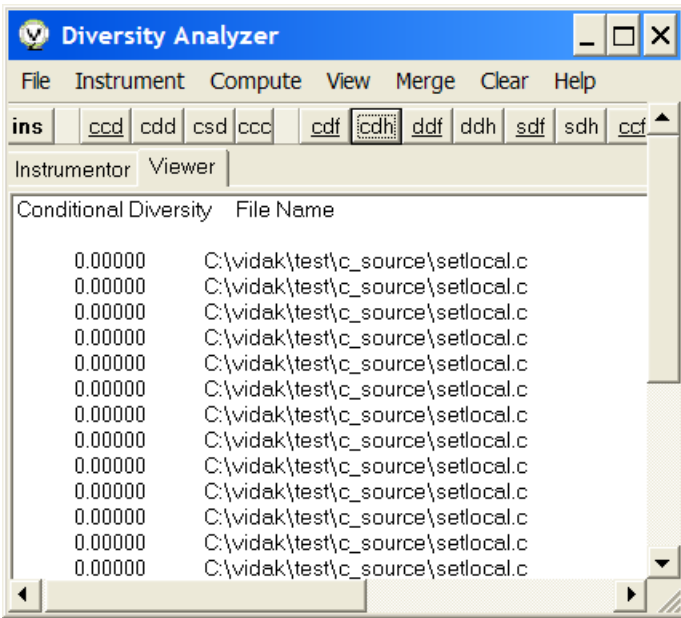


Figure 9: Viewer

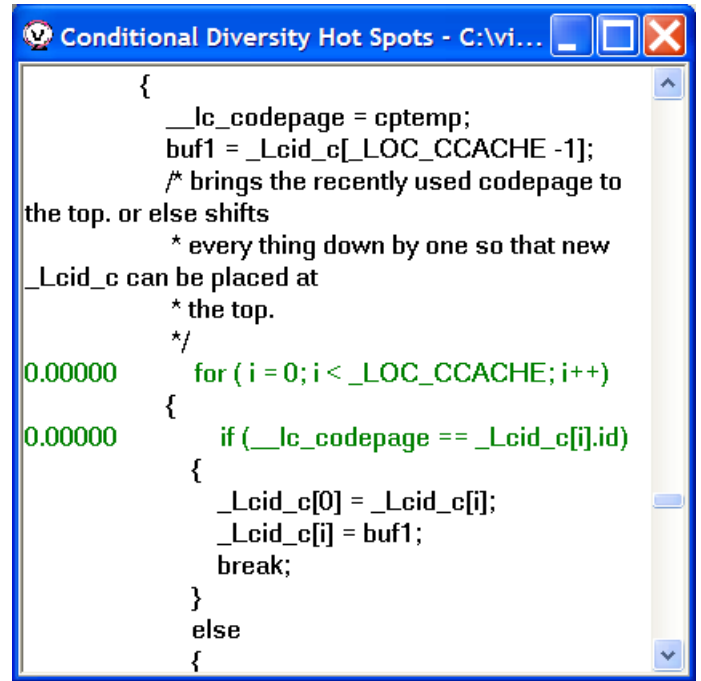


Figure 10: Source Code Viewer

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 Analyzer uses the diversity matrix to compute the  $\pi$  measure for the matrix. It also computes suggested diversity vectors that maximize the  $\pi$  measure.

The diversity metrics could be viewed in a summary fashion given by source files, as shown in Figure 9, 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 10.

## 5 Experimental Results

We have used the Diversity Analyzer 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.

### 5.1 Performance

The instrumentation of these projects took less than 5% of the time it took to build 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[9]. 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. We believe this increase in run-time is acceptable, especially in the cases where testing is distributed among multiple machines, which could carry test execution

	$S_v^m$	total	unrltd	# tst	cov	NB	LOC
$p_1$	0.54	38	27	850	63%	30K	300K
$p_2$	0.43	27	14	710	72%	6K	25K
$p_3$	0.35	21	11	690	68%	5K	30K

Figure 11: Overall Project Statistics

$S_v^m$	[0, 0.2]	[0.2, 0.4]	[0.4, 0.6]	[0.6, 0.8]
$p_1$	9%	21%	29%	41%
$p_2$	5%	11%	32%	52%
$p_3$	12%	20%	31%	37%

Figure 12: Unrelated Failure Distribution

in parallel. The Diversity Analyzer has convenient features to combine distribution frequencies and conditional diversity matrices. However, 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.

## 5.2 $\pi$ Measure Evaluation

The goal of the experimental evaluation was to explore the correlation among the  $\pi$  measure, test cases independence, and failures. As a starting point to our 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. Figure 11 shows the overall statistics for the three projects. This figure shows, going from left to right, the  $\pi$  measure obtained by the regression tests, the total number of naturally-occurring faults detected with the regression tests, the total number of unrelated faults, the total number of test cases executed, the branch coverage obtained by the tests, the total number of branches in the code, and the number of lines of code.

The evaluation of the correlation between the  $\pi$  measure and fault-detection capabilities is performed by identifying subsets of the regression test suite and measuring, for each subset, its  $\pi$  measure 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%. These subsets were identified in such a way that the  $\pi$  measure of 25 subsets fell in the range [0, .2], the  $\pi$  measure of 25 of the subsets fell in the range [.2, .4], the  $\pi$  measure of 25 of the subsets fell in the range [.4, .6], and the  $\pi$  measure of 25 subsets fell in the range [.6, .8]. The number of faults detected by each subset was noted for each project. Figure 12 shows the distribution of unrelated failures per range of the  $\pi$  measure for each project.

The findings of the experimental evaluation are summarized as follows. Test cases with flatter angles tend to detect more unrelated failures. Test cases which are at sharper angles tend to detect more related failures. That is, test cases

that had higher values for the  $\pi$  measure were more independent than the test cases that had lower  $\pi$  measure, as measured by the number of unrelated failures. On average, the data shows a linear relation between test case independence, I, and the  $\pi$  measure.

Note that  $p_1$  is more complex than  $p_2$  and  $p_3$  with respect to NB and LOC, but depending on the  $\pi$  measure, the less complex  $p_2$  and  $p_3$  could show more failures than  $p_1$ . Similarly,  $p_3$  is more complex than  $p_2$  with respect to LOC, but the  $\pi$  measure controls which project shows more failures. This data suggests that the LOC and NB complexity metrics are dependent on the  $\pi$  measure. We call this observation the  $\pi$ -strength hypothesis, and we explain it as follows. Since the existing complexity metrics, such as LOC and NB, are based on static program analysis, they give only a *potential* for the particular complexity they indicate. However, the actual complexity is determined at execution time. In particular, for the potentially more complex code, based on static analysis, to actually be more complex in execution, it has to be executed in at least as diverse ways as the less complex code. That is, higher value of the  $\pi$  measure gives stronger correlation between existing complexity measures and software faults. For example,  $p_1$  is only potentially more complex than  $p_2$ . It is the  $\pi$  measure, as the data of figure 12 suggests, that controls the actual relative complexities of  $p_1$  and  $p_2$ , as measured by the number of faults detected.

Traditional coverage analysis treats test cases that do not increase code coverage as undesirable. Our experimental data shows that there is 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 to be expected, since branch coverage is only a special case of conditional diversity, where the true/false distribution frequency counts are binary values.

## 6 Related Work

The  $\pi$  measure is related to dynamic complexity measures. In particular, it is related to operational complexity. Operational complexity is based on the complexity of the code that is to be executed, that is, the complexity of a program will vary based on the particular subgraph selected by a particular operational profile[15]. Unlike operational complexity, the  $\pi$  measure is computed by executing the program. As such, it is fundamentally different than the existing static and dynamic complexity measures. The limitation of static and dynamic complexity measures is that they measure the program at rest. There could be great discrepancies between the complexity of a program at rest and of a program in execution.

In the evolution model, as the system progresses through a series of builds, system complexity will tend to rise[12]. However, the  $\pi$  measure rises even when the *same* build is exercised in more diverse ways. There seems to be no apparent relation between the  $\pi$  measure and the existing measures, such as lines of code, software science and cyclomatic complexity. That is, the  $\pi$  measure does not tend to rise as the other measures rise. In particular, more lines of code, higher program length and volume, higher cyclomatic complexity, and higher bandwidth metric[21] do not necessarily increase

the  $\pi$  measure.

There is a direct relationship and high correlation between the lines of code metric and software faults[25]. Correlation between cyclomatic complexity and software faults has been established. Correlation between software science and software faults has been established as well. The strength of these correlations seems to be related to the  $\pi$  measure, at least in the context of the LOC and NB metrics. That is, higher  $\pi$  measure tends to detect more failures regardless of the particular values of LOC and NB. In particular, when the  $\pi$  measure is low, the correlation between LOC/NB metrics and faults is low. When the  $\pi$  measure is high, the correlation between LOC/NB metrics and faults is high. In effect, the other measures only indicate potential for complexity, the  $\pi$  measure shows the degree to which this potential has been realized.

Our work is related to the operational abstraction approach[16], to cluster analysis[27], to the software tomography approach[11], and to the bug isolation approach[1]. 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 uses multivariate analysis for forming clusters of related program executions. However, test case independence is higher when clustering is lower. The  $\pi$  measure could, therefore, be seen as an indicator of clustering. In particular, low values of the  $\pi$  measure mean high degree of test clustering/concentrating, and high values of the  $\pi$  measure mean low degree of test clustering. 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 test cases independence, variation of control and data as described in this paper, and are not aimed at measuring software complexity.

Diversity analysis is related to code coverage[10][6]. In particular, diversity analysis is related to conditional, or decision coverage [28], 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 same decision-coverage criterion could result in very different  $\pi$  measures. Decision coverage is *not* aimed at control and data variation, as defined in this work.

The data-flow methods are related to data diversity in that data-flow testing criteria require certain dataflow properties to be 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 [22]. Data flow testing does not require any variation in the internal data of the program, as defined in this work. That is, data flow methods do not distinguish the potentially infinity of paths that satisfy a particular data flow property.

Diversity analysis is also related to performance optimization. Source code optimization and profiling give execution-time distributions at the source-code level, and at the object-code level[29]. 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. In addition, profiles are based on statistical sampling in order to reduce the amount of performance data gathered. Test diversity is based on the exact run time information.

## 7 Summary

Independence of test cases is a fundamental assumption of reliability engineering. Independence of test cases also tends to be the universal law of test case design, evident outside of the reliability engineering arena. The  $\pi$  measure is a novel, dynamic software measure for test cases independence, based on the degree of program control and data variation, relative to a test suite. Higher value of  $\pi$  indicates more control and data variation in the code, more complex code, more independent test cases, and more faults detected. Test cases designed with higher degree of independence use the software in more complex ways and consequently detect more faults.

We observe that the existing complexity measures give *potential* complexity. The  $\pi$  measure indicates the degree to which this potential has been realized in execution. The  $\pi$ -strength hypothesis states that the correlation between existing complexity metrics and faults might be related to the  $\pi$  measure.

The Diversity Analyzer is an industrial-strength tool that computes diversity of test suites for code written in C, C++, C#, and Visual Basic for Microsoft Windows and .NET. The tool uses these diversity values to compute the  $\pi$  measure. Furthermore, the Diversity Analyzer computes suggested frequencies of branch executions that maximize the  $\pi$  measure.

We have used the Diversity Analyzer to conduct an independence experiment on four industrial projects. The experimental results show that test cases with higher  $\pi$ -measure values are more independent, show more complex software, and detect more unrelated faults. The results also show evidence for the  $\pi$ -strength hypothesis in the context of LOC and NB metrics.

More extensive empirical evaluation of the  $\pi$  measure is needed. A comparative evaluation of the  $\pi$  measure and other possible control and data diversity methods is needed as well. Furthermore, more empirical data is needed to validate the  $\pi$ -strength hypothesis in the context of more existing complexity metrics. It might be possible to extrapolate the  $\pi$ -strength observation to other complexity metrics.

## References

- [1] Liblit B, Aiken A, Zheng A, and Jordan M. Bug isolation via remote program slicing. *PLDI 2003*, pages 141–154, 2003.
- [2] Sara Baase. Computer algorithms. *Addison-Wesley*, Second Edition:16, 1988.
- [3] Strang G. Linear algebra and its applications. *Harcourt Brace Jovanovich, INC*, December, 1988.

- [4] M.H Halstead. Elements of software science. *Elsevier North-Holland, New York*, 1977.
- [5] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.
- [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 J. Voas. Faults on its sleeve: amplifying software reliability. In *International Symposium on Software Testing and Analysis*, pages 89–98, Boston, MA, 1993.
- [8] R. G. Hamlet. Probable correctness theory. *Info. Proc. Letters*, 25:17–25, 1987.
- [9] J.L. Hennesey and D. A. Patterson. Computer architecture a quantitative approach. *Morgan Kaufmann Publishers, INC.*, pages 251–343, 1990.
- [10] W.E. Howden. Reliability of the path analysis testing strategy. *IEEE Trans. on Soft. Eng.*, SE-2,3:208–216, 1976.
- [11] Bowring J, Orso A, and Harrold M.J. Monitoring deployed software using software tomography. *ACM SIGPLAN-SIGSOFT PASTE 2002*, pages 2–9, November, 2002.
- [12] Munson J.C and Khoshgoftaar. Applications of a relative complexity metric for software project management. *Journal of system and software*, 12:283–291, 1990.
- [13] Munson J.C and Khoshgoftaar T.M. Regression modeling of software quality: empirical investigation. *Journal of information and software technology*, 32:105–114, March 1990.
- [14] J. Laski and B. Korel. A data flow oriented program testing strategy. *IEEE Trans. on Soft. Eng.*, SE-9:347–354, 1983.
- [15] M. R. Lyu. Handbook of software reliability engineering. *IEEE Computer Society Press*, 1995.
- [16] Harder M, Mellen J, and Ernst M. Improving test suites via operational abstraction. *ICSE 2003*, pages 60–71, May, 2003.
- [17] T.J. McCabe. A complexity metric. *IEEE Transactions on Software Engineering*, 2:308–320, December, 1976.
- [18] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill, New York, NY, 1987.
- [19] Borislav Nikolik. Constraint preservation through loops. *Information Processing Letters*, 55:143–148, 1995.
- [20] Borislav Nikolik. Ultrareliability of programs specified with equational specifications. *Ph.D. Dissertation*, University of Oregon, 1998.
- [21] Lind R and Vairavan K. An experimental investigation of software metrics and their relationship to software development effort. *IEEE Transactions on Software Engineering*, 15:649–653, May, 1989.
- [22] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Soft. Eng.*, SE-11:367–375, 1985.
- [23] Hogg R.V and Craig A.T. Introduction to mathematical statistics. *Macmillan Publishing Co., INC*, 1978.
- [24] M. L. Shooman. *Software Engineering Design, Reliability, and Management*. McGraw-Hill, New York, NY, 1983.
- [25] Khoshgoftaar T.M and Munson J.C. A measure of software system complexity and its relationship to faults. *Proceedings of the 1992 international simulation technology conference, Huston, Texas*, pages 267–272, 1992.
- [26] Khoshgoftaar T.M, Munson J.J, and Lanning D.L. Dynamic system complexity. *Proceedings of IEEE-CS international software metrics symposium, Baltimore, MD*, pages 129–140, May 1993.
- [27] Dickinson W, Leon D, and Podgurski A. Finding failures by cluster analysis of execution profiles. *ICSE 2001*, pages 339–348, 2001.
- [28] 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.
- [29] Michael Wolfe. Optimizing supercompilers for supercomputers. *MIT Press, Cambridge, Mass.*, 1989.