Is Coverage a Good Measure of Testing Effectiveness?

An Assessment Using Branch Coverage and Random Testing

Yi Wei
Chair of Software Engineering
ETH Zurich
CH-8092 Zurich, Switzerland
yi.wei@inf.ethz.ch

Manuel Oriol
Dept. of Computer Science
University of York
YO10 5DD, United Kingdom
manuel@cs.york.ac.uk

Bertrand Meyer
Chair of Software Engineering
ETH Zurich
CH-8092 Zurich, Switzerland
bertrand.meyer@inf.ethz.ch

ABSTRACT

Most approaches to testing use branch coverage to decide on the quality of a given test suite. The intuition is that covering branches relates directly to uncovering faults. In this article we present an empirical study that applied random testing to 14 Eiffel classes for a total of 2520 hours and recorded the number of uncovered faults and the branch coverage over time. Our results show that: (1) in the tested classes, random testing reaches 93% branch coverage (2) it exercises almost the same set of branches every time, (3) it detects different faults from time to time, (4) during the first 10 minutes of testing while branch coverage increases rapidly, there is a strong correlation between branch coverage and the number of uncovered faults, (5) over 50% of the faults are detected at a time where branch coverage hardly changes and the correlation between branch coverage and the number of uncovered faults is weak.

These results provide evidence that branch coverage is not a good stopping criterion for random testing. They also show that branch coverage is not a good indicator for the effectiveness of a test suite.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and debugging, Test coverage of code, Testing tools

General Terms

Measurement, Experimentation

Keywords

random testing, branch coverage, experimental evaluation

1. INTRODUCTION

Various studies [16, 28, 23, 17, 4] show that random testing is an effective way of detecting faults. Random testing is also attractive because it is easy to implement and widely applicable. For example, when insufficient information is available to perform systematic testing, random testing is more practical than any alternative [15]. A question often asked about random testing is the branch coverage it achieves. The assumption is: the higher the branch coverage achieved by a test suite, the higher the number of faults it uncovers.

Branch coverage of a test suite is the percentage of branches of the program that the test suite exercises. As advocated by Myers [22], it is a weaker indicator of the quality of a test suite than other coverage criteria such as predicate coverage or path coverage. Although weak, branch coverage is widely used because of its ease of implementation and its low overhead on the execution of the program [29] under test.

This article presents an extensive study of the branch coverage that random testing achieves over time and its correlation with the number of faults uncovered. Despite the popularity of both random testing and branch coverage, there is little data available on the topic. We tested 14 Eiffel classes using our random testing tool for 2520 hours. It tested each class in 30 sessions with each session 6 hour long. The testing sessions are fully automated and consists of a single run of AutoTest [4, 5, 20], a random testing tool for Eiffel. For each test run, we recorded the exercised branches and detected faults over time. The main results of the study are as follows:

- Random testing reaches 93% of the branch coverage on average.
- Different test runs with different seeds for the pseudo random number generator of the same class exercise almost the same branches, but detect different faults.
- At the beginning of the testing session, both branch coverage and faults dramatically increase and they are strongly correlated.
- 90% of all the exercised branches are exercised in the first 10 minutes. After 10 minutes, the branch coverage level increases slowly. After 30 minutes, branch coverage further increases by only 4%.
- Over 50% of faults are detected after 30 minutes while the branch coverage level hardly increases after this time.
- There is a weak correlation between number of faults found and the coverage over the 2520 hours of testing.

The main implication of these results is that branch coverage is an inadequate stopping criteria for random testing. As AutoTest conveniently builds test suites randomly as it tests the code, the branch coverage achieved at any point in time corresponds to the branch coverage of the test suite built since the beginning of the testing session. Because there is a strong correlation between faults uncovered and branch coverage when the coverage increases, higher branch coverage implies uncovering more faults. Because with very little if any added coverage, 50% of the faults can be further discovered, and given that the correlation between the number of faults uncovered and the branch coverage is weak, this means that branch coverage by itself is not a good indicator of the quality of a test suite in general.

A package is available online[1] containing the source code of the AutoTest tool and instructions to reproduce the experiment.

Section 2 describes the design of our experiment. Section 3 presents our results. We discuss the results in Section 4 and the threats to validity in Section 5. We present related work in Section 6 and conclude in Section 7.

2. EXPERIMENT DESIGN

The experiment on which we base our results consists of running automated random testing sessions of Eiffel classes. We first describe contract-based unit testing for O–O program, then introduce AutoTest, and eventually present the classes under test, testing time and computing infrastructure.

2.1 Contract-Based Unit Testing for O–O Programs

In O–O programs, a unit test can be assimilated to a method call on an instance using previously created instances as arguments. Test engineers write unit tests and check that the result of calls are equal to pre-calculated values. In a Hoare-triple style this means that a unit test can be modelled as (v, o, o_1,... are variables, $init_o$,$init_{o_1}$... expressions that return instances, m the method called, and v_0 a value):

$$\{\}o := init_o; o_1 := init_{o_1}; ...; v := o.m(o_1, ..., o_n)\{v = v_0\}$$

In a contract-enabled environment, methods are equipped with contracts from the start:

$$\{Pre\}o.m(o_1, ..., o_n)\{Post\}$$

Unit tests can rely on contracts to check the validity of the call. It then consists only in writing the code to initialize instances that would satisfy the precondition of the method:

$$\{\}o := init_o; o_1 := init_{o_1}; ...\{Pre\}$$

In this article we use contract-based automated random testing. In such an approach the testing infrastructure automatically takes care of this last part. In practice, it generates the sequence of instructions at random and proceeds with the call.

When making a call, if the generated instances do not check the precondition of the method, the result of the call is ignored. After the precondition is checked, any contract violation or any exception triggered in the actual call then corresponds to a failure in the program.

As the random testing tool is not able to avoid executing similar test cases, it might uncover the same failure multiple times. Thus, it maps failures to faults by defining a fault as a triple:

$$< m, line\ number\ of\ the\ error, type\ of\ exception >$$

2.2 The AutoTest Tool

This section presents a general view of how AutoTest works. Note that more detailed explanations on AutoTest are available in previous publications [4].

[1] `http://se.inf.ethz.ch/people/wei/download/branch_coverage.zip`

AutoTest is a tool implementing a random testing strategy for Eiffel integrated in EiffelStudio 6.3 [3]. Given a set of classes and a time frame, AutoTest tries to test all their public methods in the time frame.

To generate test cases for methods in specified classes, AutoTest repeatedly performs the following three steps:

Select method: AutoTest maintains the number of times that each method has been tested, then it randomly selects one of the least tested methods as the next method under test, thus trying to test methods in a fair way.

Prepare objects: To prepare objects needed for calling the selected method, AutoTest distinguishes two cases: basic types and reference types.

For a basic type such as INTEGER, DOUBLE and BOOLEAN, AutoTest maintains a predefined value set. For example, for INTEGER, the predefined value set is $0, +/-1, +/-2, +/-10, +/-100, maximum$ and $minimum\ integers$. It then chooses at random either to pick a predefined value or to generate it at random.

AutoTest also maintains an object pool with instances created for all types. When selecting a value of a reference type, it either tries to create a new instance of a conforming type by calling a constructor at random or it retrieves a confroming value from the object pool. This allows AutoTest to use old objects that may have had many methods called on them, resulting in states that would otherwise be unreachable.

Invoke method under test: Eventually, the method under test is called with the selected target object and arguments. The result of the execution, possible exceptions and its branch coverage information is recorded for later use.

2.3 Experiment Setup

Class selection. We chose the classes under test from the library EiffelBase [2] version 5.6. EiffelBase is production code that provides basic data structures and IO functionalities. It is used in almost every Eiffel program, so the quality of its contracts should be better than average Eiffel libraries. This is an important point, because as described in section 2.1, we assume the present contracts are correct. In order to increase the representativeness of the test subjects, we tried to pick classes with various code structure and intended semantics. Table 1 shows the main metrics for the chosen classes. Note that the branches shown in Table 1 is the number of testable branches, obtained by subtracting dead branches from the total number of branches in the corresponding class.

Test runs. We tested each class in 30 runs with different seeds with each run 6 hour long. This supposedly made the test runs long enough so that branch coverage level reaches a plateau. But we found out that even after 16 hours, random testing is still capable of exercising some new branches with a very low probability. We chose 6 hour runs because the branch coverage level already increases very slowly after that, and because 6 hours corresponds to an overnight testing session.

Computing infrastructure. We conducted the experiment on 9 PCs with Pentium 4 at 3.2GHz, 1GB of RAM, running Linux Red Hat Enterprise 4. The version of AutoTest in EiffelStudio 6.3 used in the experiment is modified to include instrumentation for branch coverage monitoring. AutoTest was the only CPU intensive program running during testing.

3. RESULTS

Table 1: Metrics for tested classes

Class	LOC	Methods	Contract assertions	Faults	Branches	Branch Coverage
ACTIVE_LIST	2433	157	261	16	222	92%
ARRAY	1263	92	131	23	118	98%
ARRAYED_LIST	2251	148	255	22	219	94%
ARRAYED_SET	2603	161	297	20	189	96%
ARRAYED_STACK	2362	152	264	10	113	96%
BINARY_SEARCH_TREE	2019	137	143	42	296	83%
BINARY_SEARCH_TREE_SET	1367	89	119	10	123	92%
BINARY_TREE	1546	114	127	47	240	85%
FIXED_LIST	1924	133	204	23	146	90%
HASH_TABLE	1824	137	177	22	177	95%
HEAP_PRIORITY_QUEUE	1536	103	146	10	133	96%
LINKED_CIRCULAR	1928	136	184	37	190	92%
LINKED_LIST	1953	115	180	12	238	92%
PART_SORTED_TWO_WAY_LIST	2293	129	205	34	248	94%
Average	**1950**	**129**	**192**	**23**	**189**	**93%**
Total	**27302**	**1803**	**2693**	**328**	**2652**	**93%**

This section presents results that answer the five following main questions:

1. Is the level of the branch coverage achieved by random testing predictable?
2. Is the branch coverage exercised by random testing similar from one test run to another?
3. Is the number of faults discovered by random testing predictable?
4. Are the faults uncovered by different test runs similar?
5. Is there a correlation between the level of coverage and the number of faults uncovered?

3.1 Predictability of coverage level

Because AutoTest might not be able to test all branches of a class due to its random nature, it is very unlikely that testing sessions achieve total coverage, let alone constant results over all tested classes. As an example, it might be extremely difficult to satisfy a complex precondition guarding a method with such a random approach. Another example is that the visibility of a routine might not let AutoTest test it freely. The branch coverage level itself is calculated in a straightforward manner as:

$$\frac{Number\ of\ exercised\ branches}{Number\ of\ branches\ in\ that\ class}$$

This intuition is confirmed by the results presented in Figure 1 which shows the median of the branch coverage level for each class over time. The branch coverage level ranges from 0 to 1. As a first result, we can see that the branch coverage of some classes reaches a plateau at less than 0.85 while most of them have a plateau at or above 0.9. The thick curve in Figure 1 is the median of medians of the branch coverage level of all the classes. Over all 14 classes, the branch coverage level achieved after 6 hours of testing ranges from 0.82 to 0.98. On average, the branch coverage level is 0.92, with a standard deviation of 0.04, corresponding to 4.67% of the median.

While the maximum coverage is variable from one class to another, the actual evolution of branch coverage compared to the maximum coverage achieved through random testing is very similar: 93% of all exercised branches are exercised in the first 10 minutes, 96% are exercised in 30 minutes, and 97% are exercised in the first hour. Section 4 contains an analysis of branches not exercised.

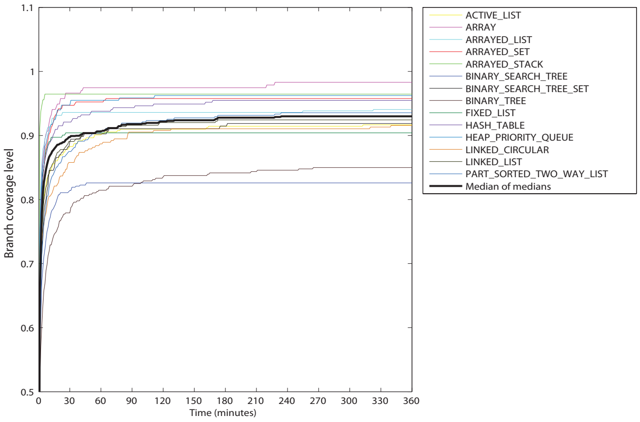

Figure 1: Medians of the branch coverage level for each class over time and their median

In short, the branch coverage level achieved by random testing depends on the structure of the class under test and it increases very fast in the first 10 minutes of testing and then very slowly afterwards.

3.2 Similarity of coverage

Another important question is whether different test runs for the same class exercise different branches. Since we are more interested in branches which are difficult to exercise, we raised the question: Do different test runs for the same class leave the same set of branches unexercised? To answer the latter question, we need to measure the difference between the sets of unexercised branches in two test runs for the same class. We use an array per testing session, containing a flag for each branch, indicating whether it was visited or not.

To measure the difference of two sets of unexercised branches, we use the Hamming distance [18]. For two strings of equal length, the Hamming distance between them is the number of positions at which the corresponding symbols are different. For example, the Hamming distance between string 1011101 and 1001001 is 2 because the number of positions with different values is 2 (the third and the fifth position).

Because we only focused on the branches difficult to exercise, we defined the notion of difficult branches as: A branch in a class is difficult to exercise if and only if it has not been exercised at least once through the 30 runs for that class.

The *difficult branch coverage vector* of a test run for a class with n difficult branches is a vector of n elements, where the i-th element is a flag for the i-th difficult branch in that class, with one of the following value: 0, indicating that the corresponding branch has not been exercised in that test run, or 1, indicating that the corresponding branch has been exercised in that test run.

The *branch coverage distance* D_{BC} between two vectors u and v of the a class with N_b difficult branches is the Hamming distance between them:

$$D_{BC} = \sum_{i=1}^{N} u_i \oplus v_i$$

where u_i and v_i is the value at the i-th position of u and v respectively, and $\oplus$ means xor operation. D_{BC} is in the range between 0 and N_b. The larger the distance, the more different branches are covered by these two runs.

The *branch coverage similarity* is defined as:

$$\frac{N_b - D_{BC}}{N_b}$$

The intention of the similarity is that the smaller the branch coverage distance, the higher the similarity and the similarity should range between 0 and 1. The similarity among $k > 2$ vectors is calculated as the median of the similarity values between each two vectors: there are $\frac{k(k-1)}{2}$ pairs of k vectors, for each pair, a similarity value is calculated, and the overall similarity is the median of those $\frac{k(k-1)}{2}$ values.

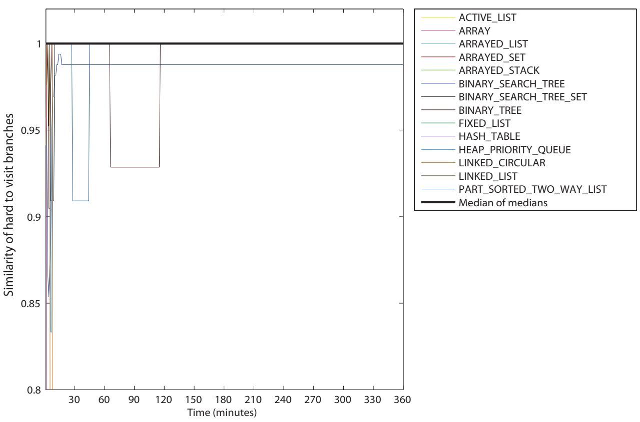

Figure 2: The branch coverage similarity for each class over time; their median

The difficult branch coverage similarity for each class over time is plotted in Figure 2. The thick curve in Figure 2 is the median of the branch coverage similarity over all classes. Figure 3 shows the standard deviation of the branch coverage similarity for each class. Figure 2 shows that the similarity of difficult branch coverage is already 1 only after a few minutes of testing, and Figure 3 shows that the standard deviation of difficult branch coverage similarity is almost 0.

The high median of similarity means that in general, the set of branches from a class that are difficult to exercise are very similar from test run to test run (for the same class), the small standard

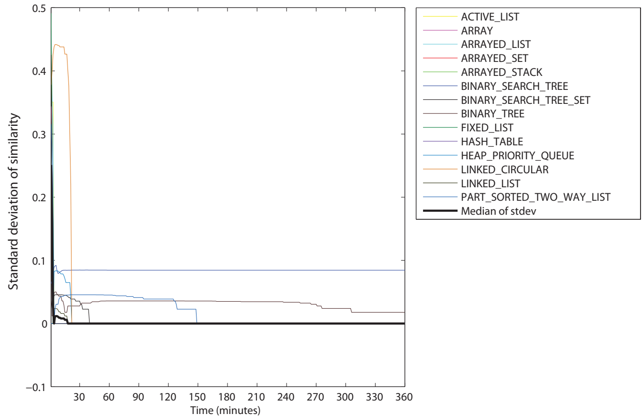

Figure 3: Standard deviation of the branch coverage similarity for each class over time; their median

deviation means that this phenomenon was constantly observed through all the runs.

The consequence drawn from Figure 2 and Figure 3 is that if a branch is not exercised by a test run, it is unlikely that it will be exercised by other runs for the same class. In other words, by applying random testing with different seeds to the same class does not help to improve branch coverage of that class. Unexercised branches will stay unexercised.

3.3 Predictability of number of faults

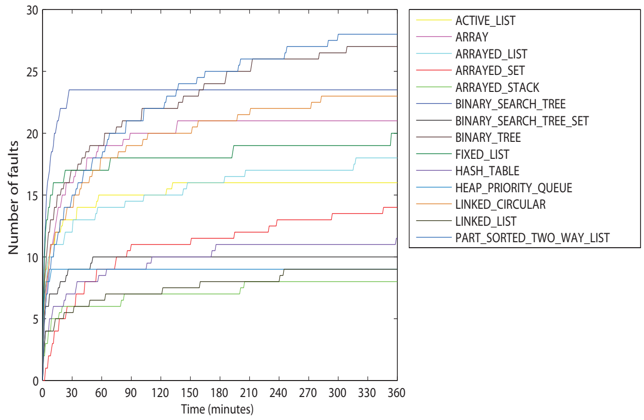

Figure 4: Medians of the number of faults detected in each class over time

The question of predictability of number of faults was already addressed in a previous study [6]. Our results confirm that study and extend it to much longer testing sessions (6-hour sessions rather than 90-minute ones), they are also using the most recent version of AutoTest that underwent significant performance improvements. The median of the number of faults detected for each class over time is plotted in Figure 4. Note that all the faults found are real faults in a widely used Eiffel library. This also shows that our testing tool is effective in finding faults. Figure 4 shows that 54% of the faults are detected in the first 10 minutes, 70% are detected in 30 minutes, and 78% are detected in 1 hour. About 22% of the faults are detected after 1 hour. This means after 30 minutes of testing, 70% of the faults are detected although only 4% additional branches are exercised.

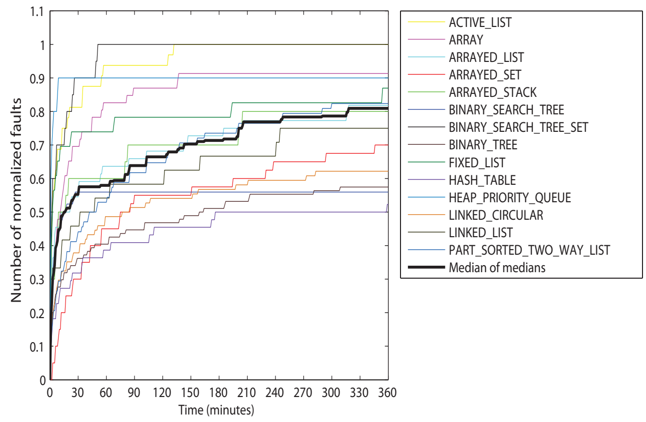

Figure 5: Medians of the normalized number of faults detected for each class over time; their median

Different classes contain different number of faults. In order to compare fault detection across different classes, we use the normalized number of faults, obtained by dividing the number of faults detected by each test run by the total number of faults found in all test runs for that particular class. The number of normalized faults for a particular test run represents the percentage of faults found in that test run against all faults that we know in the class. The medians of the number of the normalized faults detected over time for each class are shown in Figure 5. The thick curve in Figure 5 is the median of the medians of the number of normalized faults detected over time for all classes.

For most of the classes, the median of the normalized number of faults does not reach 1 at the end of testing, indicating that different runs detect different faults. Because if every test run for a class found the same number of faults, the number of normalized faults for those runs should be 1.

3.4 Similarity of faults

As in the case of the branch coverage level, we are interested in the similarity of detected faults for the same class among test runs. The detected faults are similar when different test runs find the same faults. Similarly to section 3.2 we introduce the fault detection vector, distances and similarity.

The *Fault detection vector* of a class in a particular test run is a vector of n elements, with n being the total number of faults detected for that class over all runs. Because we do not know the actual number of faults in a class, we can only use the total number of faults that are found by AutoTest for that class as an estimation. Each element in the vector has one of the following values: 0, indicating that the corresponding fault is not detected in this particular run, or 1, indicating that the corresponding fault is detected.

Given two fault detection vectors r and s for the same class, in which the total number of found faults is N_f, the *fault detection distance* D_f between r and s is defined as

$$D_f = \sum_{i=1}^{N} r_i \oplus s_i$$

where r_i and s_i is the value at the i-th position of r and s respectively, and $\oplus$ means xor operation. D_f is in the range between 0 and N_f.

The *fault detection similarity* between them is then defined as:

$$\frac{N_f - D_f}{N_f}$$

The fault detection similarity ranges from 0 to 1. The larger the similarity, the more faults are detected in both test runs or in neither. Fault detection similarity among more than two vectors is calculated similarly to branch coverage similarity.

Figure 6 shows the similarity of detected faults in different test runs for each class. The median of the fault detection similarity for all classes (the thick curve in Figure 6) ranges from 0.84 to 0.90. As can be seen in Figure 6, most of the faults can be detected in every test run, but (because the median does not reach 1.0) in order to get as many faults as possible, multiple test runs for that class are necessary. Figure 7 shows the standard deviation of the fault detection similarity for each class. The median of the standard deviation of the fault detection similarity (the thick curve in Figure 7) ranges from 0.07 to 0.05, corresponding to 8% to 5% of the median of fault detection similarity for all classes.

This implies that most faults are discovered by most testing runs, but several runs produce better results. Seeds have a stronger impact on fault detection than on branch coverage.

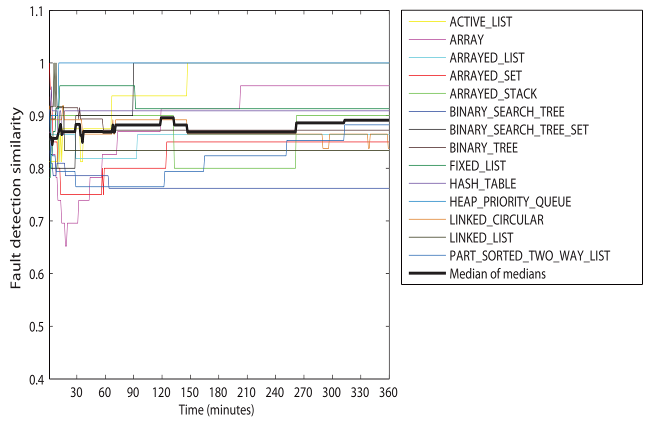

Figure 6: Fault detection similarity for each class over time; their median

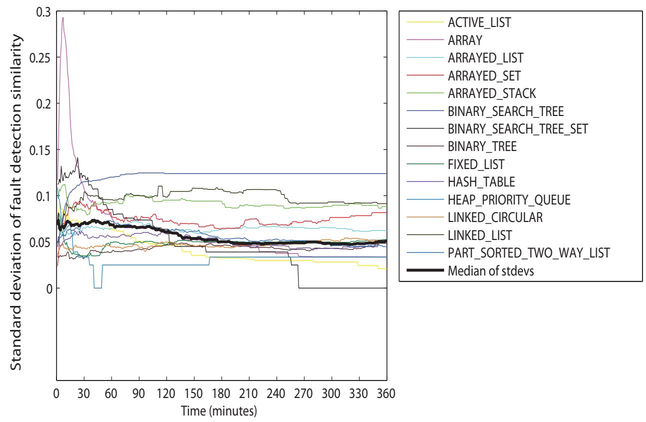

Figure 7: Standard deviation of the fault detection similarity for each class over time; their median

3.5 Correlation between branch coverage and number of faults

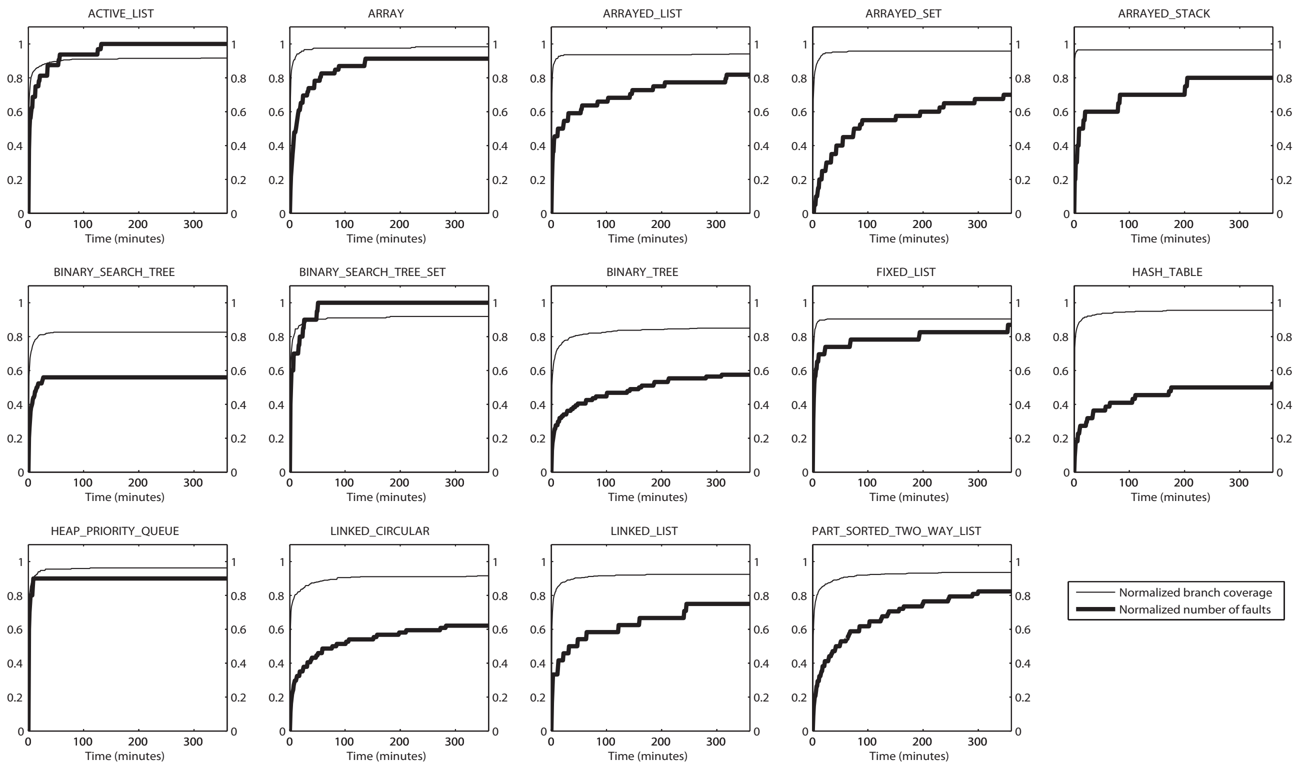

Figure 8: Median of the branch coverage level and median of the normalized number of faults for each class over time

As previously written, the correlation between branch coverage and number of faults is something that is usually taken for granted. Here we take a closer look at it, and it seems that it is not as clear as one might expect. While it is true that a higher coverage gives higher number of faults, it is clearly not sufficient as an indicator.

In order to study the correlation between branch coverage level and fault detection ability, the median of the branch coverage level and the median of the normalized number of faults for the tested classes are superimposed in Figure 8. Figure 8 shows that in the first few minutes of testing when the branch coverage level increases quickly, faults are also found quickly. After a while, the increase of the branch coverage slows down. The speed of fault detection also decreases, although less dramatically than the branch coverage level. After 30 minutes, the branch coverage level only increases slightly, but during that period, many faults are detected.

The correlation between the branch coverage level and the normalized number of faults and for each class across all test runs is shown in Figure 9. Each subgraph shows the value of the correlation coefficient r. The correlation between the branch coverage level and the normalized number of faults shows a positive correlation, but varies much from class to class, from 0.3 to 0.97 and there seems to be no common pattern among the tested classes.

The implications of these results are twofold: (1) when coverage increases, faults discovered increase as well, (2) when coverage stagnates, faults are still found. Thus increasing the branch coverage clearly increases the number of faults found. It is however clearly not sufficient to have a high value of the branch coverage to assess the quality of a testing session.

The next section further elaborates on these findings as well as their limitations.

4. DISCUSSION

The results of the previous section enable us to answer the three following main questions:

- Is branch coverage a good stopping criterion for random testing?
- Is branch coverage a good measure of the testing effectiveness?
- What are the unexercised branches?

4.1 Branch Coverage as Stopping Criterion for Random Testing

Since in general, random testing cannot achieve 100% branch coverage in finite time, total branch coverage is certainly not a feasible stopping criterion. In practice, the percentage of code coverage is often used as an adequacy criterion, the greater the percentage, the more adequate the testing [30], and testing can be stopped if the generated test suite reached certain level of adequacy. In our experiments, after 1 hour, the branch coverage level hardly increases, so it will be unpractical to extend the testing time until full coverage is reached. Instead, the only reasonable way to use branch coverage would be to evaluate the expectation of finding new faults. As shown in the previous section, the number of faults only evolves closely with the branch coverage level in the first few minutes of testing. On testing sessions longer than 10 minutes, the correlation between number of faults and branch coverage degrades. In fact, about 50% of the faults are found in the period where the branch coverage level hardly increases. This means that branch coverage is not a good predictor for the number of faults remaining to be found.

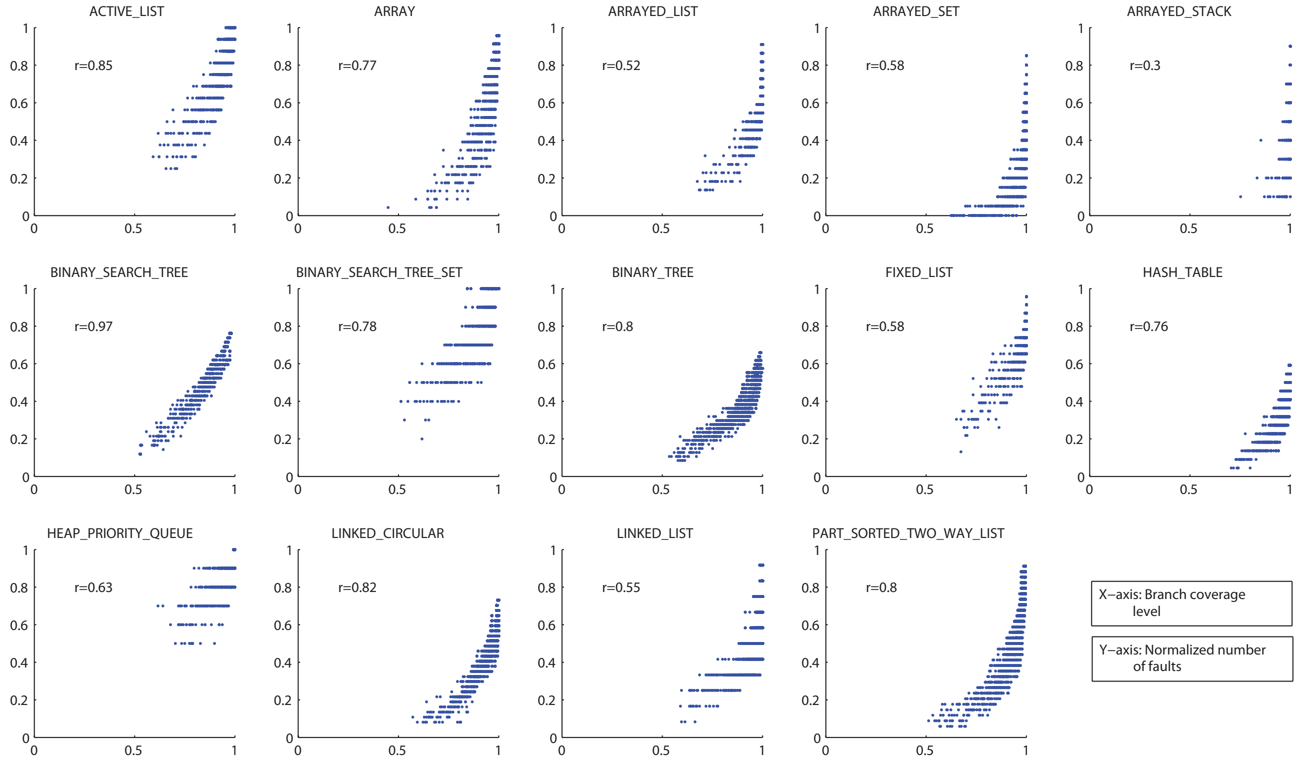

Figure 9: Correlation between the branch coverage level and the normalized number of faults for each class over 360 minutes

The correlation between the branch coverage level and the number of detected faults greatly varies from class to class. For some classes such as BINARY_SEARCH_TREE, the correlation coefficient is 0.98 and the scatter graph shows the correlation is almost linear, but for other classes such as ARRAYED_STACK, the correlation is weak (0.3), especially for longer testing sessions. This variation on the class under test reduces the precision if branch coverage is used as a stopping criterion.

Random testing also detects different faults in different test runs while it exercises almost the same branches. This confirms that multiple restarts improve drastically the number of faults found [6]: in order to find as many faults as possible, a class should be random tested multiple times with different seeds, even though the same branches are exercised every time.

Thus branch coverage alone cannot be used as a stopping criterion for random testing.

4.2 Branch Coverage as Measure of Testing Effectiveness

As a preamble, it is important to understand that running random testing longer is the same as adding new test cases into a test suite. The reason is that testing for a longer time means that more method calls are executed on the class under test. Each method call is actually the last line of a test case that contains all previous calls participating to the state of data used in the call (see [20] for a detailed explanation of test case construction and simplification). To push further the analogy, testing a class in different runs is the same as providing different test suites for that class.

Our experiments tested production code in which the existing number of faults is unknown. As a result, we cannot use the ratio of detected faults against the total number of faults to measure the effectiveness of testing. Instead, we measure testing effectiveness in two ways: (the number of) faults that are detected and the speed at which those faults are detected.

Two results show that different faults can be detected at the same level of branch coverage: (1) in a test run, new faults were detected in a period where branch coverage hardly changes; (2) in different test runs for the same class, different faults were detected while almost the same branches were exercised. In other words, different tests suites satisfying the same branch coverage criterion may detects different faults.

These two observations indicate that the degree of test adequacy in terms of branch coverage level is highly predictable, not only in how many branches are covered, but also in what the covered branches are. Put another way, when applying random testing to a class, the same level of branch coverage adequacy is always achieved. Also, for all the tested classes, the branch coverage adequacy level stabilized after some time (1 hour in our case), which means even continue the testing for a much longer time, the branch coverage level would not going to be increased.

But when looked at the testing effectiveness in terms of number of found faults, random testing can continue find new faults when the branch coverage level is not increasing. Although we do not know how many faults are to be found in those tested classes, the fact that over 50% of new faults were found in the period when the branch coverage level stagnates is not neglectable.

These results provided evidence of the lack of reliability [12] of branch coverage criterion achieved by random testing. Reliability requires that a test criterion always produce consistent results. In the context of our experiments, reliability requires that two test runs achieving the same branch coverage of a class should deliver similar number of faults. But our results show that the number of faults from different test runs will differ from each other by at least 50%.

In terms of speed of fault detection, we can only consider ran-

Table 2: Unexercised branches

Reason	% of branches
Branch condition not satisfied	45.6%
Linear constraint not satisfied	12.9%
Call site not exercised	13.7%
Unsatisfiable branches	13.7%
Crash before branch	8.6%
Implementation limitation	2.5%
Concurrent context needed	1.7%

dom testing. In the first few minutes of random testing, the branch coverage level increases quickly, and the number of faults increases accordingly. There is a strong correlation between the number of faults found and the coverage during that period. This means that branch coverage is good in measuring testing effectiveness in the first few minutes. But after a while, the branch coverage level hardly increases, the fault detection speed also slows down but less dramatically than the branch coverage level. In fact, many faults are detected in the period where the branch coverage hardly changes. This means in the later period, branch coverage is not a good measure for testing effectiveness.

In general, to detect as many faults as possible, branch coverage is necessary but not sufficient.

4.3 Unexercised branches

We analyzed the 179 branches in all 14 classes that were not exercised in our experiments. Among these branches, there are 116 distinct branches, and 63 duplicated branches because they appear in inherited methods. Table 2 shows the reasons why certain branches were not exercised and the percentage of unexercised branches that fall into that each reason. In Table 2 the categories are as follows:

Branch condition not satisfied means that those branches were not exercised because their branch condition is not met. This is the most common case.

Linear constraint not satisfied means that in the branch condition, there is a linear constraint, and they were not satisfied by the random strategy. Actually, linear constraint is a special case of branch condition, but we think it is an important category because a random strategy usually has great difficulty satisfying these constraints.

Call site not exercised means that the call site of a method containing the unexercised branches were not executed.

Unsatisfiable branches means that the branch checks on conditions that will never be satisfied because the condition can never be true.

Fault before branch means that there was always a fault found before those unexercised branches, interrupting the execution.

Implementation limitation means that because of the limitation of the AutoTest tool, those branches were not exercised.

Concurrent context needed means that those branches are only exercisable when tested in a concurrent context. But our experiments were conducted in a sequential setting.

Table 2 shows that 58.5% of the unexercised branches fall into the first two reasons (*Branch condition not satisfied, linear constraint not satisfied*). A following question would be how to satisfy these branch conditions. A common solution to satisfy branch conditions is to use symbolic execution to collect path conditions under which certain statement can be executed and propagate the path conditions up to the method entry so particular inputs can be generated. However, symbolic executors often entails a great complexity and they usually come with a large overhead. We analyzed those unexercised branches falling into the first two reasons to see how often a symbolic executor is needed: In 32.3% of cases, a symbolic executor to propagate path conditions is needed, for the rest 67.7%, simply concatenate all dominating path conditions and select inputs at the method entry satisfying the concatenated path condition would suffice to exercise those branches (a linear constraint solver is needed when there is linear constraint in the concatenated path condition). Of course, there is no guarantee that certain branches will be exercised by doing this (for example, it is possible that there is no object satisfying the concatenated path condition for the testing strategy to choose from), but it may direct the testing strategy more effectively in exercising more branches. Also, by doing this, it is possible to reduce the number of unexercised in the *Call site not exercised* category because if the branch containing the call sit of a method is exercised, the branches in side the called method may be exercised also. 67.7% is high enough not to be ignored, so we think this method is worthy trying.

For those unexercised branches in the *Fault before branch* category, the faults must be fixed first and then retest the class again. For branches in the *Implementation limitation* and *Concurrent context needed* categories, we can enhance the AutoTest tool to support the creation of agents and to support testing in a concurrent environment.

5. THREATS TO VALIDITY

We detail mainly five threats to validity for our results.

First, although the classes under test in our experiment are from the widely used Eiffel library EiffelBase and they vary in terms of various code metrics and intended semantics, we make no claim regarding their *representativeness* of O–O programs in general.

Second, AutoTest is one implementation of a random testing strategy. It uses a pseudo-random number generator, and chooses some interesting values for primitive types such as integers, reals and characters with some probability. We tried to keep the algorithm of AutoTest as general as possible, but *other implementations of random testing* may produce different results.

Third, the heuristics used in mapping from failures to faults mostly assume the correctness of contracts: a precondition violation on method entry identifies the caller of that method as faulty, and a postcondition violation on method exit identifies the method itself as faulty. This may cause AutoTest to miss some faults because if the contracts of a method are not correct callers might contain faults that are unnoticed. Unfortunately, in case of a contract violation, deciding whether it is because of a wrong contract of a wrong implementation cannot be done automatically. To limit this risk, we chose classes from the EiffelBase library, whose contracts are used and reviewed by many programmers during a long period of time. After testing, we manually inspected the faults that are suggested by the heuristic, and in all the cases, the faulty method is correctly identified. The fact that AutoTest relies on contracts as test oracle does not limit its applicability to languages without contracts because when AutoTest runs on Eiffel classes with runtime contract monitoring turned off (essentially equals to test classes without contracts), it can catch faults due to exceptions other than contract violation, for example, null pointer dereferencing, division by zero, system level error. However, without contracts specifying what a class is supposed to do, an automatic testing tool only can found low level faults. According to our experience, 60% of the faults found by AutoTest are contract related.

Fourth, in the experiment, classes are tested in isolation in each test run. This means that in each test run, only methods from that particular class under test are tested directly (methods they transitively call will be tested indirectly). The result may be different if the library is tested as a whole, meaning that all classes from the library are tested together.

Last, for all classes, the *branch coverage level is below* 100%, due to the limitations of the random testing strategy and the way branch coverage is calculated. We do not know if the correlation between the branch coverage level and the number of faults still holds when all branches are exercised. Also, we only have data on faults detected by random testing, we don't know how many faults are still undetected in the chosen classes even if the number of faults found by random testing is much higher in general than manual testing [5].

6. RELATED WORK

Intuitively, random testing cannot compete in terms of effectiveness with systematic testing because it is less likely that randomly selected inputs will be interesting enough to reveal faults in the program under test. However, some studies [16, 28, 23] have shown that random testing is as effective as some systematic methods such as partition testing. Our results also showed that random testing is effective: in the experiment, random testing detected 328 faults in 14 classes in EiffelBase library while in the past 3 years, only 28 faults were reported by users.

The advantage of cheap implementation and easy applicability makes random testing attractive. It has been used in testing operating systems [21, 9], as well as O–O programs [7, 8, 25, 24, 26].

Ciupa et al. [6] investigated the predictability of random testing and showed that in terms of the number of faults detected over time, random testing is predictable, meaning that different runs for the same class will detects roughly the same number of faults, while in terms of the kind of faults, random testing is unpredictable, meaning that different runs for the same class detects different kinds of faults. Figure 5 and Figure 6 confirm their results.

The branch coverage criterion for measuring testing strategy effectiveness is compared with other criteria in many studies. Frankl et al. [10] compared the branch coverage criterion with the all-uses criterion and concluded that for their programs under test, all-uses adequate test sets performs better than branch adequate test sets, and branch adequate test sets do not perform significantly better than null-adequate test sets, which are test sets containing randomly selected test cases without any adequacy requirement. They defined the effectiveness of an adequacy criterion to be the probability that a test set selected randomly according to that adequacy criterion will expose an error. Based on this definition, they evaluated branch coverage effectiveness in test sets as small as possible. In our study, we are more interested the branch coverage level achieved by random testing in a certain amount of time and the number of faults found in that period.

Hutchins et al. [19] also compared the effectiveness of the branch coverage criterion and the all-uses criterion. They found that for both criteria, test sets achieving coverage levels over 90% showed significantly better fault detection than randomly selected test sets of the same size. This means that a lot of faults could be detected when the coverage level approaches 100%. They also concluded that in terms of effectiveness, there is no winner between branch coverage and all-uses criterion. Our results on the correlation between the branch coverage level and the number of detected faults also shows a similar pattern that many faults are detected at higher coverage levels, in our experiment, however, the branch coverage level did not reach 100%, while in their study, manually written test sets guaranteed total branch coverage. Also, in their study, programs under test were seeded with faults, while in our experiment, programs were tested as they are.

Gupta et al. [13] compared the effectiveness (the ability to detect faults) and efficiency (the average cost for detecting a fault) of three code coverage criteria: predicate coverage, branch coverage and block coverage. They found that predicate coverage is the most effective but the least efficient, block coverage is the least effective but most efficient, while branch coverage is between predicate coverage and block coverage in terms of both effectiveness and efficiency. Their results suggest that branch coverage is the best among those three criteria for getting better results with moderate testing efforts.

Many methods have been proposed to maximize branch coverage, many of which are based on random testing or use random testing in an initial phase. Gupta et al. [14] presented a method to dynamically switch to a path that offers relatively less resistance to generation of an input to force execution to reach an unexercised branch. DART [11] combined random testing and symbolic execution to achieve path coverage. Pex [27] also uses symbolic execution to achieve high branch coverage. Our experiment provided results showing how random testing performs in terms of branch coverage, it can be used as a benchmark in evaluating the enhancement of those branch coverage maximizing methods.

7. CONCLUSIONS AND FUTURE WORK

We have assessed how random testing performs in terms of branch coverage. Our results show that the branch coverage level achieved by random testing varies depending on the structure of the program under test, but on average, is very high (93%). Within the branches that are exercised by random testing, most of them are exercised very quickly (in the first 10 minutes of testing) regardless of the class under test. For the same class, branches exercised in different test runs are almost the same. In terms of fault detection, different test runs for the same class will detect roughly 10% different faults. Over 50% of the faults are detected in the period when branch coverage hardly changes.

Our results indicate that branch coverage is not a good stopping criterion for random testing. One should test a program in multiple test runs to find as many faults as possible even though by doing so the branch coverage level will not be increased in general. Also, one should not stop random testing when the branch coverage level stops increasing or only increases very slowly.

Another deduction from our results is that branch coverage in general is not a good indicator of the quality of a test suite. In our experiments, more than 50% of the faults are uncovered while coverage is at a plateau. Although many studies showed that branch coverage is weak, we found little evidence showing a random testing strategy continues finding faults when the branch coverage stagnates.

Future work includes investigating how to reach even higher branch coverage (100%) as well as to analyze the reasons for not reaching total coverage.

Acknowledgement We thank Ilinca Ciupa, Andreas Leitner, Simon Poulding, and Stephan van Staden for their insightful comments.

8. REFERENCES

[1] ECMA-367 Eiffel: Analysis, Design and Programming Language, 2nd Edition.

`http://www.ecma-international.org/publications/standards/Ecma-367.htm`.

[2] EiffelBase. Eiffel Software. `http://www.eiffel.com/libraries/base.html`.

[3] EiffelStudio. Eiffel Software. `http://www.eiffel.com/`.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis 2007 (ISSTA'07)*, pages 84–94, 2007.

[5] I. Ciupa, B. Meyer, M. Oriol, and A. Pretschner. Finding faults: Manual testing vs. random+ testing vs. user reports. In *19th International Symposium on Software Reliability Engineering (ISSRE'08)*, pages 157–166.

[6] I. Ciupa, A. Pretschner, A. Leitner, M. Oriol, and B. Meyer. On the predictability of random tests for object-oriented software. In *First International Conference on Software Testing, Verification, and Validation 2008 (ICST'08)*, pages 72–81.

[7] C. Csallner and Y. Smaragdakis. Jcrasher: an automatic robustness tester for java. *Softw. Pract. Exper.*, 34(11):1025–1050, 2004.

[8] C. Csallner and Y. Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 245–254. ACM, 2006.

[9] J. E. Forrester and B. P. Miller. An empirical study of the robustness of windows nt applications using random testing. In *WSS'00: Proceedings of the 4th conference on USENIX Windows Systems Symposium*, pages 6–6, Berkeley, CA, USA, 2000. USENIX Association.

[10] P. Frankl and S. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Software Engineering, IEEE Transactions on*, 19(8):774–787, Aug 1993.

[11] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[12] J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[13] A. Gupta and P. Jalote. An approach for experimentally evaluating effectiveness and efficiency of coverage criteria for software testing. *Int. J. Softw. Tools Technol. Transf.*, 10(2):145–160, 2008.

[14] N. Gupta, A. Mathur, and M. Soffa. Generating test data for branch coverage. *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*, pages 219–228, 2000.

[15] D. Hamlet. When only random testing will do. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 1–9, New York, NY, USA, 2006. ACM.

[16] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Trans. Softw. Eng.*, 16(12):1402–1411, 1990.

[17] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[18] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[19] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 191–200, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[20] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*, pages 417–420, November 2007.

[21] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[22] G. J. Myers. *The Art of Software Testing, 2nd edition*. John Wiley and Sons, 2004.

[23] S. Ntafos. On random and partition testing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48, New York, NY, USA, 1998. ACM.

[24] C. Oriat. Jartege: A tool for random generation of unit tests for java classes. In *Quality of Software Architectures and Software Quality, First International Conference on the Quality of Software Architectures, QoSA 2005 and Second International Workshop on Software Quality, SOQUA 2005*, pages 242–256, 2005.

[25] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

[26] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Washington, DC, USA, 2007. IEEE Computer Society.

[27] N. Tillmann and J. de Halleux. Pex-white box test generation for .net. In *Tests and Proofs, Second International Conference(TAP'08)*, pages 134–153, 2008.

[28] E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[29] Q. Yang, J. J. Li, and D. Weiss. A survey of coverage based testing tools. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 99–103, New York, NY, USA, 2006. ACM.

[30] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.