Is Branch Coverage a Good Measure of Testing Effectiveness?

Yi Wei[1], Bertrand Meyer[1] and Manuel Oriol[2]

[1] Chair of Software Engineering, ETH Zurich, Switzerland,
{yi.wei, bertrand.meyer}@inf.ethz.ch
[2] Department of Computer Science, University of York, United Kindom,
manuel@cs.york.ac.uk

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. The empirical study reported here 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. For the tested classes, (1) random testing reaches 93% branch coverage (2) it exercises almost the same set of branches every time, (3) it detects different faults from execution to execution, (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.

Keywords: random testing, branch coverage, experimental evaluation

1 Introduction

Various studies[11, 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 [10]. Many practitioners think that, to evaluate the effectiveness of a strategy, branch coverage –the percentage of branches of the program that the test suite exercises – is the criterion of choice. It is a weaker indicator of test suite quality than other coverage criteria such as predicate coverage or path coverage [15]. Branch coverage is widely used because of its ease of implementation and its low overhead on the execution of the program [18] under test. As an example the European Cooperation for Space Standardization (ECSS) gives 100% branch coverage as one of the measures to assure the quality of a critical software [6].[3]

[3] Section 6.2.3.2

Many practitioners and researchers dismiss random testing because it only achieves low branch coverage. We used AutoTest [4], an automatic, random-based testing tool for Eiffel, to gain insights on three questions: (1) the actual branch coverage achieved by testing Eiffel classes with AutoTest, (2) whether the achieved branch coverage correlates with the number of bugs found in the code, (3) whether branch coverage is a good stopping criterion for random testing. 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 fully automated random testing tool AutoTest for 2520 hours. AutoTest tested each class in 30 runs with each run 6 hour long. For each run, we recorded the exercised branches and faults detected over time. The main results are:

- Random testing reaches 93% 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, branch coverage and faults both increase dramatically and 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.
- There is a weak correlation between number of faults found and coverage over the 2520 hours of testing.

The main implication of these results is that branch coverage is an inadequate stopping criterion 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 coverage increases, higher branch coverage implies uncovering more faults. However, half of the faults can be further discovered with hardly any increase in coverage. This confirms that branch coverage by itself is not in general a good indicator of the quality of a test suite.

A package is available online[4] 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 in running automated random testing sessions of Eiffel classes. We first describe contract-based unit testing for object-oriented (O–O) programs, then introduce AutoTest, and present the classes under test, the testing time and the computing infrastructure.

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

2.1 Contract-Based Unit Testing for O–O Programs

In O–O programs, a unit test can be assimilated to a routine (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 routine 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, routines 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 of writing the code to initialize instances that would satisfy the precondition of the routine:

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

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 satisfy the precondition of the routine, the result of the call is ignored and not counted as a test. 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 unique triple:

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

When tests are executed, branch coverage is calculated in a straightforward manner as:

$$Branch\ Coverage = \frac{Number\ of\ exercised\ branches}{Number\ of\ branches}$$

2.2 The AutoTest Tool

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

AutoTest is a tool implementing a random testing strategy for Eiffel and is integrated to EiffelStudio 6.3 [2]. Given a set of classes and a time frame, AutoTest tries to test all their public routines in the time frame.

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

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

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

For each 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 conforming value from the object pool. This allows AutoTest to use old objects that may have had many routines called on them, resulting in states that would otherwise be unreachable.

Invoke routine under test. Eventually, the routine 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 [1] version 5.6. EiffelBase is production code that provides basic data structures and IO functionalities. It is used in almost every Eiffel program. The quality of its contracts should therefore be better than average Eiffel libraries. This is an important point because we assume the contracts to be 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.

Table 1. Metrics for tested classes

Class	LOC	Routines	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%**

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 used in the experiment is modified to include instrumentation for monitoring the branch coverage. AutoTest was the only CPU intensive program running during testing.

3 Results

This section presents results that answer five 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 unlikely that testing sessions achieve total coverage, or even just constant results over all tested classes. As an example, it might be extremely difficult to satisfy a complex precondition guarding a routine with such a random approach. Another example is that the visibility of a routine might not let AutoTest test it freely.

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 0.93, 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 similar: 93% of all exercised branches are exercised in the first 10 minutes, 96% in 30 minutes, and 97% in the first hour. Section 4 contains an analysis of branches not exercised.

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

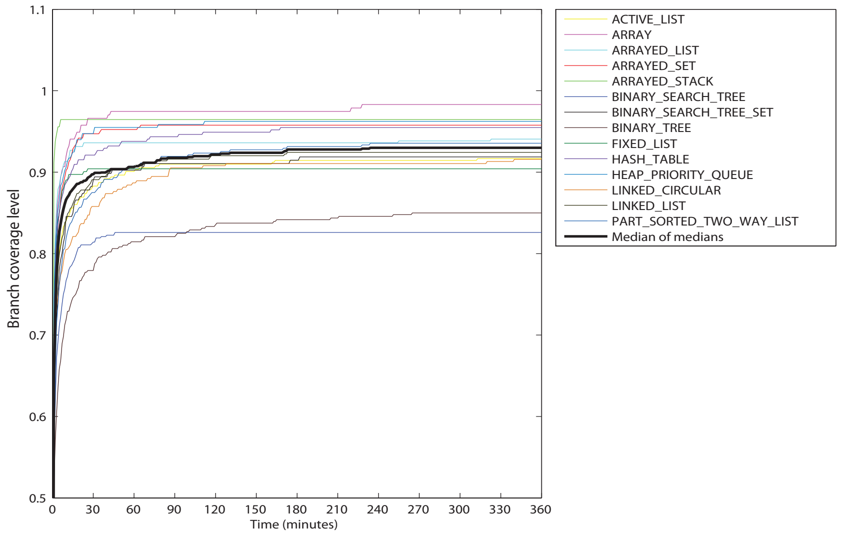

Fig. 1. Medians of the branch coverage level for each class over time and their median

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 difficult to exercise, the more specific question is: Do different test runs for the same class leave the same set of branches not visited? To answer this question, we need to measure the difference between the sets of branches not visited in two test runs for the same class. We use an array per testing run, containing a flag for each branch indicating whether it was visited.

To measure the difference between two sets of non-visited branches, it is appropriate to use the Hamming distance [12]: the number of positions, in two strings of equal lengths, where symbols differ. For example, the Hamming distance between 1011101 and 1001001 is 2 since the values differ at two positions, 3 and 5.

For the purposes of this study, a branch is said to be "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 *difficult 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:

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

where u_i and v_i are the values at the i-th position of u and v respectively, and $\oplus$ is exclusive or. 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 *difficult 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.

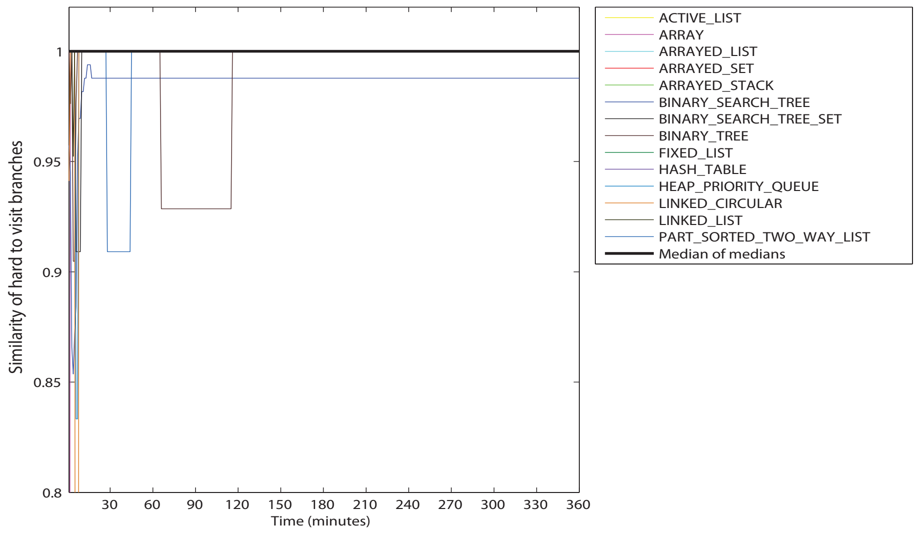

Fig. 2. The branch coverage similarity for each class over time; their median

Figure 2 shows the difficult branch coverage similarity for each class over time. The thick curve is the median of the difficult branch coverage similarity over all classes. Figure 2 reveals that the similarity of difficult branch coverage is already 1 only after a few minutes of testing, Figure 3 shows the standard deviation of the branch coverage similarity for each class. It reveals 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 deviation means that this phenomenon was constantly observed through all the runs.

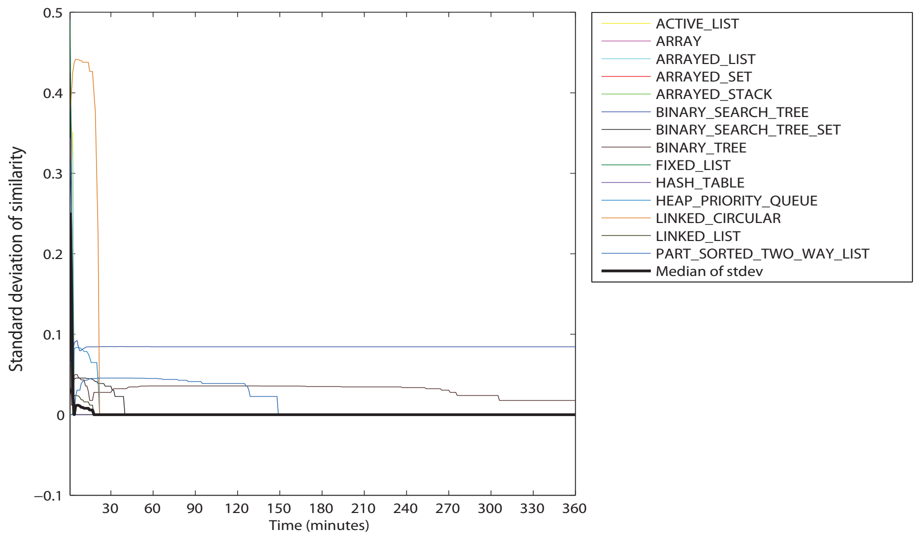

Fig. 3. Standard deviation of the branch coverage similarity for each class over time; their median

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, applying random testing with different seeds to the same class does not improve branch coverage for that class. Branches not exercised in one run are not visited in subsequent runs.

3.3 Predictability of number of faults

The question of predictability of the number of faults found by random testing was already addressed in a previous study [5]. The new results confirm that study and extend it to longer testing sessions (6-hour sessions rather than 90-minute ones), they are also using the most recent version of AutoTest which benefits from 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 at finding faults. Figure 4 shows that 54% of the faults are detected in the first 10 minutes, 70% in 30 minutes, and 78% in 1 hour. About 22% of the faults are detected after 1 hour. This means that after 30 minutes of testing, 70% of the faults have been detected even though only 4% additional branches have been exercised.

Different classes contain different numbers of faults. 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

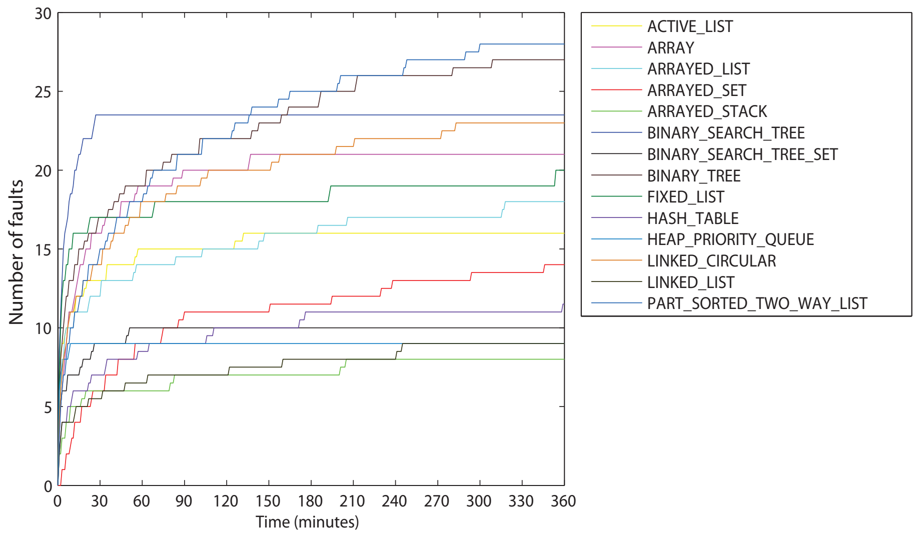

Fig. 4. Medians of the number of faults detected in each class over time

time for each class are shown in Figure 5. The thick curve 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 does not reach 1. This indicates different runs detect different faults (since median 1 would mean that every run finds the same faults).

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. Definitions of distances, similarity and fault detection vector, similar to those of section 3.2, are appropriate.

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 found by AutoTest. Each vector element is 1 if the corresponding fault has been detected and 0 otherwise.

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

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

where r_i and s_i is the value at the i-th position of r and s respectively. D_f is in the range between 0.. N_f.

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

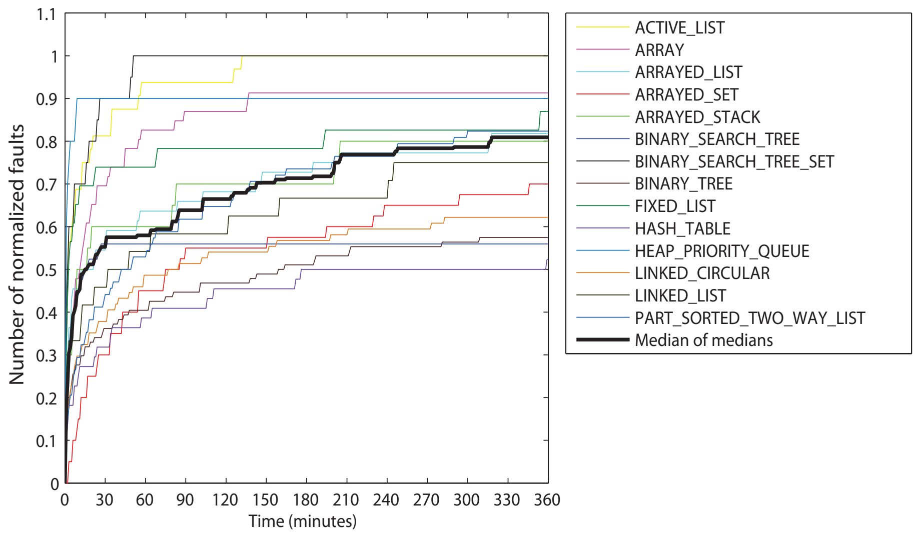

Fig. 5. Medians of the normalized number of faults detected for each class over time; their median

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

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) ranges from 0.84 to 0.90. The figure indicates that 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 (the thick curve) ranges from 0.07 to 0.05, corresponding to 8% to 5% of the median for all classes.

This implies that most faults are discovered by most testing runs, but several runs produce better results. The choice of seed has a stronger impact on fault detection than on branch coverage.

3.5 Correlation between branch coverage and number of faults

Here we take a closer look at the correlation between branch coverage and the number of detected faults. Although higher coverage does uncover more faults overall, it is clearly not sufficient an indicator.

To study the correlation between branch coverage level and fault detection ability, Figure 8 superimposes the median of the branch coverage level and the median of the normalized number of faults for the tested classes. In the first few minutes of testing,

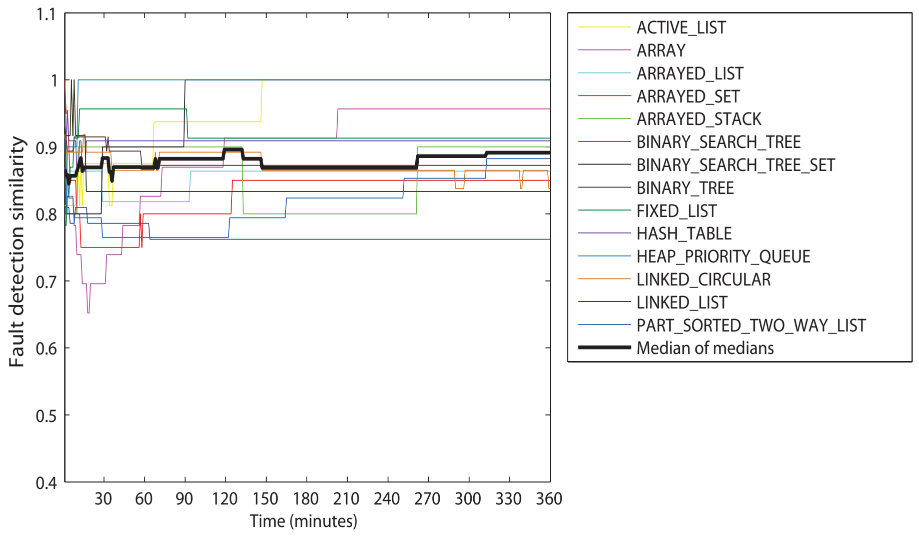

Fig. 6. Fault detection similarity for each class over time; their median

when the branch coverage level increases quickly, faults are also found quickly. After a while, the increase of branch coverage slows down. The speed of fault detection also decreases, although less dramatically. After 30 minutes, the branch coverage level only increases slightly, but many faults are detected in that period.

We also calculated the correlation between branch coverage and normalized number of faults. It varies much from class to class, 0.3 to 0.97 and there seems to be no common pattern among the tested classes as shown in Figure 9.

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 provide material for answering three questions:

- Is branch coverage a good stopping criterion for random testing?
- Is it a good measure of 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 not a feasible stopping criterion. In practice, the percentage

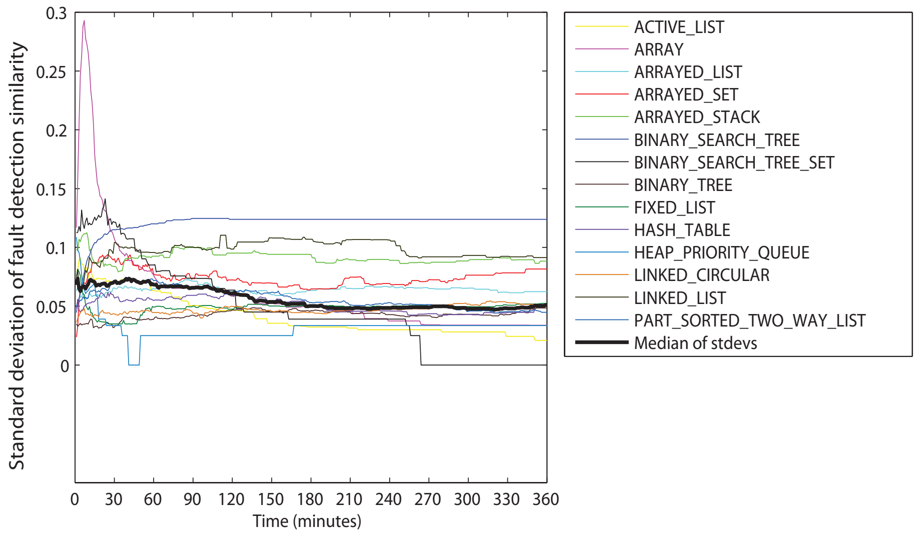

Fig. 7. Standard deviation of the fault detection similarity for each class over time; their median

of code coverage is often used as an adequacy criterion: the higher the percentage, the more adequate the testing [19]; and testing can be stopped if the generated test suite reached a 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 reaching full coverage. 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 evolves closely with the coverage only in the first few minutes of testing. On testing sessions longer than 10 minutes, the correlation degrades. In fact, about 50% of the faults are found in a period where the branch coverage level hardly increases any more. This means that branch coverage is not a good predictor for the number of faults remaining to be found.

The correlation greatly varies from class to class. For some classes such as BINARY_SEARCH_TREE, the correlation coefficient is 0.98 and the correlation is almost linear, but for others such as ARRAYED_STACK the correlation is weak (0.3), especially with 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 drastically improves the number of faults found [5]: to find as many faults as possible, a class should be random-tested multiple times with different seeds, even if the same branches are exercised every time.

Our conclusion is that branch coverage alone cannot be used as a stopping criterion for random testing.

4.2 Branch Coverage as Measure of Testing Effectiveness

To assess branch coverage as a measure of testing effectiveness, one must 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 routine calls are executed on the class under test. Each routine 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 [14] for a detailed explanation of test case construction and simplification). To push the analogy further, testing a class in different runs is the same as providing different test suites for that class.

Our experiments test production code in which the existing number of faults is unknown. They do not seed faults in the code but merely tested the discrepancy between the contracts and the code. As a result, it is not possible to use the ratio of detected faults against the total number of faults to measure the effectiveness of testing. Instead, we assess testing effectiveness through two parameters: the number of faults 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 test suites satisfying the same branch coverage criterion may detect different faults.

These two observations indicate that 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. Applying random testing to a class always yields the same level of branch coverage adequacy. Also, for all the tested classes, the branch coverage adequacy level stabilizes after some time (1 hour in our case).

Although we do not know how many faults remain in tested classes, it was astonishing to discover that over 50% of found faults only appear in the period when branch coverage stagnates.

These results provide evidence of the lack of reliability [8] of branch coverage criterion achieved by random testing. Reliability requires that a test criterion always produce consistent results. In the experiments reported here, this goal requires that two test runs achieving the same branch coverage of a class should deliver similar numbers of faults. But the results show that the number of faults found in different test runs will differ from each other by at least 50%.

What about the speed of fault detection? In the first few minutes of random testing, branch coverage increases quickly, and the number of faults increases accordingly, with a strong correlation. 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 Branches not exercised

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 routines. Table 2 shows the reasons why certain branches were not exercised and the percentage of branches not exercised that fall into that each reason. In Table 2 the categories are as follows:

Table 2. Branches not exercised

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%

Branch condition not satisfied: branch not exercised because its branch condition is not met. This is the most common case.

Linear constraint not satisfied: in the branch condition there is a linear constraint, and they were not satisfied by the random strategy. This is actually a special case of branch condition, but important on its own because a random strategy usually has great difficulty satisfying these constraints.

Call site not exercised: no calls of a routine containing the branch were executed.

Unsatisfiable branch: the branch depends on conditions that can never be satisfied.

Fault before branch: there was always a fault found before exercised.

Implementation limitation: branch not exercised because of a limitation of AutoTest.

Concurrent context needed: the branch is 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 branches not exercised fall into the first two reasons (*Branch condition not satisfied, linear constraint not satisfied*).

A follow-up 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 and propagate them up to the routine entry. Symbolic executors however induce a large overhead in the general case.

We analyzed branches falling into the first two categories to see how often a symbolic executor would help: in 32.3% of cases, we need a symbolic executor to propagate path conditions, for the remaining 67.7%, it is only needed to concatenate all dominating path conditions and select inputs at the routine entry – a linear constraint solver is needed when there is linear constraint in the concatenated path condition. Even if in some cases it is not possible to solve the constraints, it seems useful to investigate further this lead.

For *Faults before branch*, the faults should either be fixed first or avoided while testing. For the *Implementation limitation* and *Concurrent context needed* categories, we need to further improve AutoTest.

5 Threats to Validity

Four observations may raise questions about the result.

Representativeness of chosen classes. Despite being chosen from the widely used Eiffel library EiffelBase and varying in terms of various code metrics and intended semantics, the chosen classes may not be fully representative of general O–O programs.

Representativeness of AutoTest's variant of random testing. We tried to keep the algorithm of AutoTest as general as possible, but other implementations of random testing may produce different results.

Branch coverage below 100%**.** We do not know whether the correlation between branch coverage and number of faults still holds when all branches are exercised. We consider this very likely, since if we considered the application trimmed of all the branches that were not visited, we would then achieve 100% branch coverage in most cases.

Size of test suite. A recent formal analysis [3] of random testing showed that the number of tests made has a great influence on the results found with random testing. It might be possible that while our study relies on many more tests than previous ones, we did not execute enough tests. We consider this unlikely because of the high similarity of the faults found in the present experiments.

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. Some studies [17, 16] 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.

Ciupa et al. [5] investigated the predictability of random testing and showed that in terms of the number of faults detected over time, random testing is predictable. Figure 5 and Figure 6 confirm those results.

Many studies compare branch coverage for assessing the effectiveness of test strategies. With other criteria in. Frankl et al. [7] compared the branch coverage criterion with the all-uses criterion and concluded that for their programs, 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. The present study focuses more on 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. [13] 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: 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. [9] 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.

7 Conclusions and Future Work

This article has shown that the branch coverage level achieved by random testing varies depending on the structure of the program under test but was very high on the classes we tested (93% on average). Most of the branches exercised by random testing 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. Different test runs on the same class detect roughly 10% different faults.

Our results also confirm that branch coverage in general is not a good indicator of the quality of a test suite. In the experiments, more than 50% of the faults are uncovered while coverage is at a plateau. Although many studies showed the weakness of branch coverage, there is little evidence showing that random testing finds new faults while the branch coverage stagnates.

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, even if the branch coverage level stops increasing or only increases very slowly.

For the continuation of this work, we are investigating how to reach even higher branch coverage (100% or very close), and how to devise a good stopping criterion for random testing.

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

References

1. EiffelBase. Eiffel Software. `http://www.eiffel.com/libraries/base.html`.
2. EiffelStudio. Eiffel Software. `http://www.eiffel.com/`.
3. A. Arcuri, M. Iqbal, and L. Briand. Formal analysis of the effectiveness and predictability of random testing. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 219–230. ACM, 2010.
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, 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.
6. European Cooperation for Space Coordination. *Space product assurance - Software product assurance, ECSS-Q-ST-80C*. ESA Requirements and Standards Division, 2009.
7. 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.
8. J. B. Goodenough and S. L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.
9. 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.
10. 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.
11. R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
12. R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.
13. 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.
14. 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, 2007.
15. G. J. Myers. *The Art of Software Testing, 2nd edition*. John Wiley and Sons, 2004.
16. 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.
17. E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.
18. 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.
19. 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.

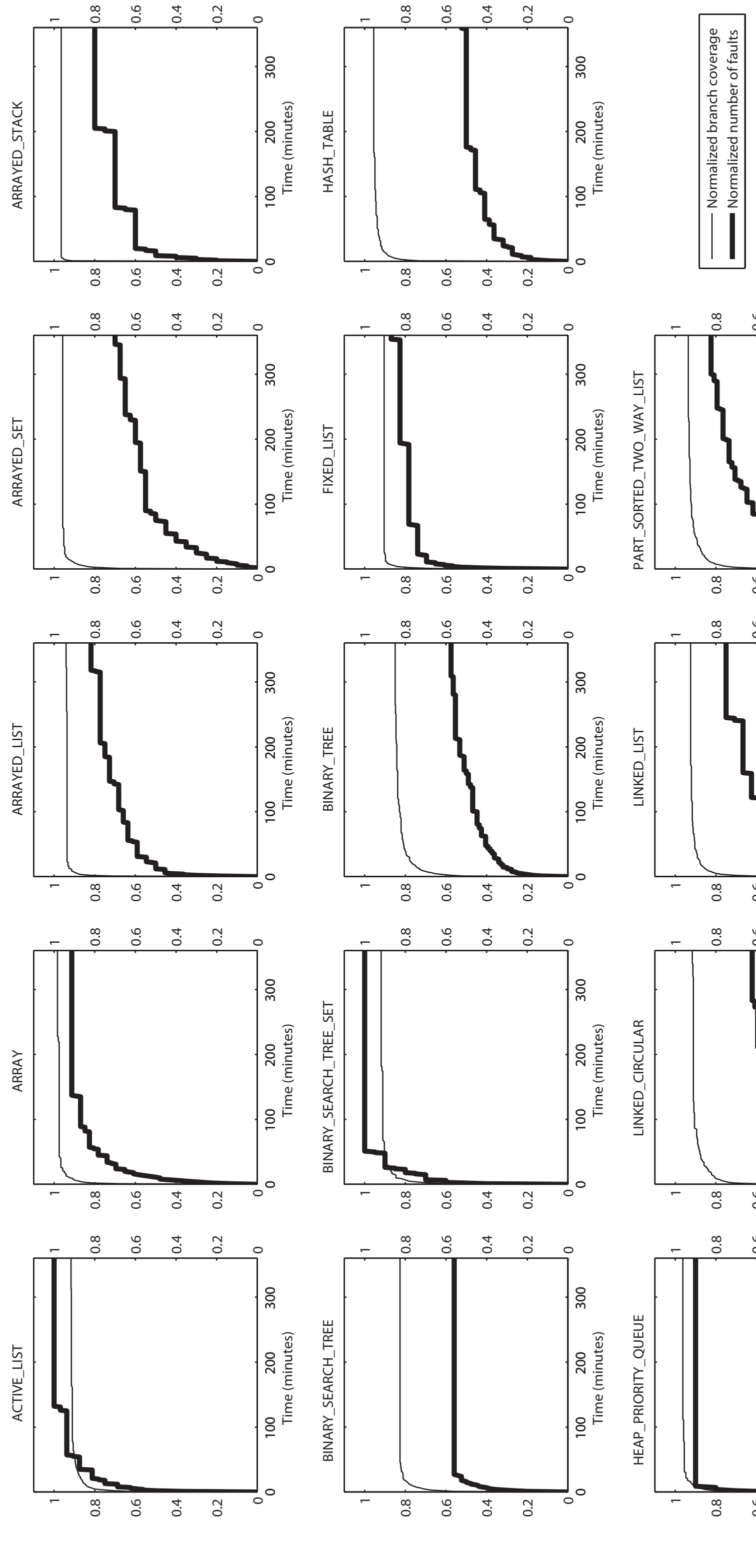

Fig. 8. Median of the branch coverage level and median of the normalized number of faults for each class over time

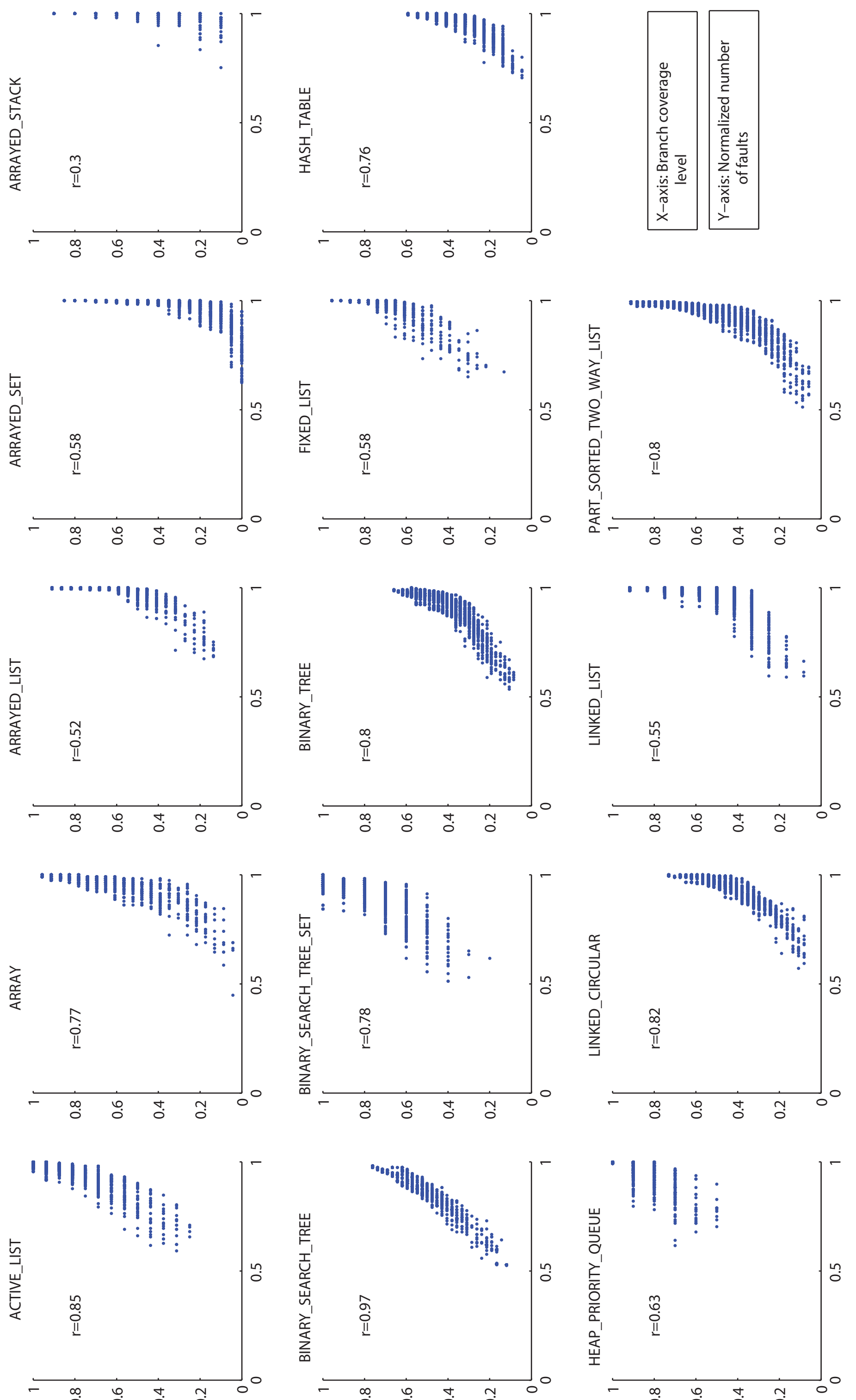

Fig. 9. Correlation between the branch coverage level and the normalized number of faults for each class over 360 minutes