IEEE International Conference on Software Testing Verification and Validation Workshops

An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage

Nan Li, Upsorn Praphamontripong and Jeff Offutt
Software Engineering
George Mason University
Fairfax, VA 22030, USA
{nli1,uprapham,offutt}@gmu.edu

Abstract

With recent increased expectations for quality, and the growth of agile processes and test driven development, developers are expected to do more and more effective unit testing. Yet, our knowledge of when to use the various unit level test criteria is incomplete. The paper presents results from a comparison of four unit level software testing criteria. Mutation testing, prime path coverage, edge-pair coverage, and all-uses testing were compared on two bases: the number of seeded faults found and the number of tests needed to satisfy the criteria. The comparison used a collection of Java classes taken from various sources and hand-seeded faults. Tests were designed and generated mostly by hand with help from tools that compute test requirements and muJava. The findings are that mutation tests detected more faults and the other three criteria were very similar. The paper also presents a secondary measure, a cost benefit ratio, computed as the number of tests needed to detect each fault. Surprisingly, mutation required the fewest number of tests. The paper also discusses some specific faults that were **not** *found and presents analysis for why not.*

1 Introduction

Unit testing is increasingly becoming important for many software projects. The overall quality of the software has a stronger impact on the economic success of software systems, and therefore the companies that sell software. The increasing use of agile processes has an especially strong impact on testing. Test-driven development turns tests into major drivers for the software requirements. Programmers are expected to do more, and better, unit testing. Although programmers can study many different test criteria and adopt lots of tools, the research community has not developed enough knowledge about which criteria should be used and when.

This paper presents experimental data to try to help compare cost benefit tradeoffs among four test criteria. Formal test criteria are used to choose specific test values to test with. Informally, a test criterion is a goal or stopping rule for testing. Test criteria make it more likely that testers will find faults in the program and provide greater assurance that the software is of high quality and reliability.

Test requirements are specific elements of software artifacts that must be satisfied or covered [1]. An example test requirement for *statement coverage* is "Execute statement 7." A *test criterion* is a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests. That is, the criterion describes the test requirements for the artifact in a complete and unambiguous manner. Although test criteria can be based on lots of software artifacts, including formal specifications, requirements, and design notations, most unit testing is based on the program source.

Many test criteria are based on graphs, including three of the four criteria compared in this paper. Numerous graph-based criteria have been proposed, most notably on control flow analysis [13, 14] and data flow analysis [11, 12, 16, 21, 25]. Researchers have published theoretical studies and empirical comparisons among graph-based test criteria [6, 8, 9, 15, 25, 26, 27].

This research compares four test criteria; edge-pair, prime path, all-uses data flow and mutation. These criteria are defined formally in Section 3. The criteria are compared on the number of tests that were needed to satisfy the criteria and their ability of those tests to find faults in a collection of Java classes taken from free educational sources. Thus, this study is purely focused on unit testing. Faults were seeded into the classes by hand and tests were designed and generated mostly by hand with help from tools that compute test requirements and muJava [17, 18]. The findings in this pa-

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.30

per provide evidence that can help practical testers choose which test criteria to use and when.

The paper is organized as follows. Section 2 summarizes previous work on comparing test criteria and Section 3 describes the test criteria. Section 4 presents the goals and design of the experiment and how the experiment was carried out. Section 5 shows the results and Section 6 analyzes and discusses them. Section 7 provides conclusions and ideas for future work.

2 Previous Unit Level Criteria Comparisons

Our ability to compare test criteria theoretically is limited. Mathur and Wong [20] proved that mutation and the all-uses data flow criterion are theoretically "incomparable," that is, one does not subsume the other. Ammann and Offutt [1] showed that the prime path criterion subsumes the edge-pair criterion, but prime path is incomparable with all-uses and mutation. Thus, comparisons focus on empirically comparing unit level test criteria.

Two relationships are commonly used to compare test criteria. Weyuker, Weiss, and Hamlet [27] defined a relationship called ProbBetter: A test criterion C_1 is ProbBetter than C_2 for a program P if a randomly selected test set T that satisfies C_1 is more "likely" to detect a failure than a randomly selected test set that satisfies C_2. Mathur and Wong [19] suggested a different relationship called ProbSubsumes: A test criterion C_1 ProbSubsumes C_2 for a program P if a test set T that is adequate with respect to C_1 is "likely" to be adequate with respect to C_2. If C_1 ProbSubsumes C_2, C_1 is said to be more "difficult" to satisfy than C_2. ProbBetter is based on fault detection and ProbSubsumes is based on "cross scoring."

These relationships were used to compare mutation and all-uses in several papers. Mathur and Wong [19] used the ProbSubsumes relationship by manually generating test data to satisfy both criteria and comparing the scores. They used 4 programs and 30 sets of test cases per program and detected equivalent mutants and infeasible subpaths by hand. They found that mutation-adequate test sets were closer to being data flow-adequate than data flow-adequate test sets were to being mutation-adequate. Frankl, Weiss and Hu [10] compared mutation and all-uses using the ProbBetter relationship and found that mutation was more effective for five of their nine subjects, all-uses was more effective for two, and there was no difference for the other two. Offutt, Pan, Tewary and Zhang [23] compared mutation and all-uses with both ProbBetter and ProbSubsumes. They found that mutation-adequate test sets were closer to satisfying the all-uses criterion and detected more faults.

This paper uses the ProbBetter relationship and adds two criteria that have not been compared with mutation and all-uses: edge-pair and prime path coverage.

3 Test Criteria Used

Directed graphs form the foundation for many test criteria. The following definitions are taken from Ammann and Offutt [1]. A graph G is often drawn as a collection of circles connected by arrows, and formally is: a set N of *nodes*, where $N \neq \emptyset$; a set N_0 of *initial nodes*, where $N_0 \subseteq N$ and $N_0 \neq \emptyset$; a set N_f of *final nodes*, where $N_f \subseteq N$ and $N_f \neq \emptyset$; and a set E of *edges*, where E is a subset of $N \times N$. For graphs, test criteria define the set of test requirements in terms of properties of test paths in a graph G. Test requirements are *satisfied* by *visiting* specific nodes or edges or by *touring* specific paths or subpaths.

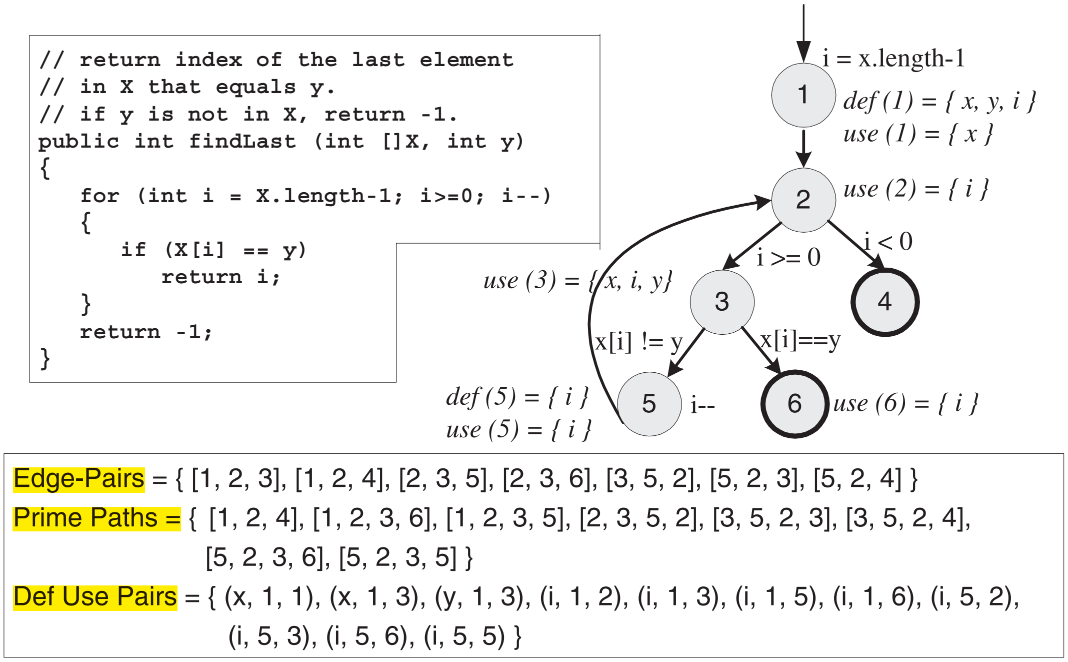
For convenience of expression, paths and subpaths that must be traversed (that is, that appear in test requirements) are separated from paths that test cases execute. A *test path* represents the execution of a test case on a graph. Test paths must start at an initial node and end at a final node. A test path p *tours* a subpath q if q is a subpath of p.

This paper evaluates two structural based test criteria. The **edge-pair (EP)** test criterion was originally defined for finite state machines by Pimont and Rault [24] and has also been called *two-trip* [4] and *transition-pair* [22]. In *edge-pair coverage*, tests must tour each reachable subpath of length less than or equal to 2 in G. The qualification "less than or equal to 2" is included specifically to ensure that edge-pair coverage subsumes edge coverage and node coverage in graphs that have only one edge or only one node.

Ammann and Offutt defined the **prime path (PP)** test criterion to ensure strong coverage of loops without requiring an infinite number of paths [1]. A path from node n_i to n_j is *simple* if no node appears more than once in the path, with the exception that the first and last nodes may be identical. That is, simple paths have no internal loops, although the entire path itself may be a loop. The *roundtrip* path defined by Chow [5] is a special case of a simple path–with nonzero length that starts and ends at the same node. Prime paths are *maximal length* simple paths: A simple path from node n_i to n_j is a *prime path* if it is simple and does not appear as a proper subpath of any other simple path. Prime paths do not have any internal loops, although the entire path may be a loop.

In prime path coverage, tests must tour each prime path in the graph G. Prime path coverage requires touring all subpaths of length 0 (all nodes), of length 1 (all edges), length 2, 3, etc. Thus it subsumes node coverage, edge coverage and edge-pair coverage.

The edge-pair and prime path criteria are defined on unannotated graphs; in unit level testing, the graphs are usually control flow graphs of the methods. **Data flow** criteria require tests that tour subpaths from specific definitions of



```
// return index of the last element
// in X that equals y.
// if y is not in X, return -1.
public int findLast (int []X, int y)
{
   for (int i = X.length-1; i>=0; i--)
   {
      if (X[i] == y)
         return i;
   }
   return -1;
}
```

Edge-Pairs = { [1, 2, 3], [1, 2, 4], [2, 3, 5], [2, 3, 6], [3, 5, 2], [5, 2, 3], [5, 2, 4] }

Prime Paths = { [1, 2, 4], [1, 2, 3, 6], [1, 2, 3, 5], [2, 3, 5, 2], [3, 5, 2, 3], [3, 5, 2, 4], [5, 2, 3, 6], [5, 2, 3, 5] }

Def Use Pairs = { (x, 1, 1), (x, 1, 3), (y, 1, 3), (i, 1, 2), (i, 1, 3), (i, 1, 5), (i, 1, 6), (i, 5, 2), (i, 5, 3), (i, 5, 6), (i, 5, 5) }

Figure 1. findLast(), its control flow graph, and test requirements

variables to specific uses. Nodes where a variable is assigned a value are called *definitions* (or *defs*), and nodes where the value of a variable is accessed are called *uses*. A definition d for a variable x *reaches* a use u if there is a path from d to u that has no other definitions of x (*def-clear*). The *all-uses (AU)* criterion requires tests to tour at least one subpath from each definition to each reachable use.

Figure 1 shows an example Java method, its annotated control flow graph, and the edge-pairs and prime paths from the graph. Nodes 4 and 6 are final nodes, corresponding to the `return` statements. Node 2 is introduced to capture the `for` loop; it has no executable statements. The edge-pairs, prime paths, and DU pairs for `findLast` are listed in the figure. Edge-pairs and prime paths are given as sequences of nodes. DU pairs are shown as a variable name followed by the def node, then the use node. These test requirements are fairly similar for this simple program.

The fourth criterion, **mutation coverage**, is based on syntactic faults. For code-based unit level testing, small changes are introduced into the program (*mutants*) and tests are required to cause each mutant to result in incorrect output (*killing* the mutant).

3.1 Satisfying Test Criteria

We generated test values by hand with the support of several tools. MuJava [17, 18] generates mutants, runs tester supplied tests against the mutants, and computes the mutation score. MuJava uses mutant schemata generation (MSG) and bytecode translation and is run through a graphical user interface. We generated test values by hand and used muJava to evaluate the tests.

Tests for edge-pair, prime path, and all-uses coverage were designed with the help of the graph coverage web applications provided online as a supplement to Ammann and Offutt's textbook [3]. These tools provide a web-based interface that accepts definitions of graphs and defs and uses of variables. They compute test requirements for structural criteria including prime paths and DU paths, as well as test paths to satisfy the criteria. The graphs were generated by hand, then submitted to the tools to create the test requirements. We then generated test values by hand. Section 4.3 gives more details on how values were created.

4 Experimental Design and Conduct

The goal of this experiment was to compare mutation, prime path, edge-pair, and all-uses coverage on the basis of effectiveness and efficiency. Effectiveness is approximated by the number of faults detected (the PROBBETTER relationship). Efficiency is approximated by the number of tests needed to satisfy the criteria. We would also like to consider the effort associated with applying each criterion. However, this could vary significantly by the automated support available. The tools used in this study are fairly primitive (intended for research and educational use, rather than commercial use), and did not include automatic test data generation, thus any measure based on our effort would be unrealistic.

The independent variable in this experiment is the test criterion used. The experiment used four criteria: mutation, edge-pair, prime path and all-uses. The two dependent variables were the number of tests and the number of faults found.

4.1 Experimental Subjects

Twenty-nine Java classes were used. Since this study is strictly about unit testing, we did not seek large packages, but typical (mostly small) classes. They were taken from various sources, including open source software websites, the accompanying CD to the Java textbook by Deitel and Deitel [7], and the accompanying website to the testing textbook by Ammann and Offutt [2].

Space precludes giving data on individual classes, so Table 1 gives totals, and the minimum and maximum for the number of executable lines of code, the number of classes, and the number of methods.

Table 1. Summary of experimental subjects

	Min	Max	Total
Lines	26	618	2909
Classes	1	11	51
Methods	1	50	174
Faults	1	19	88

4.2 Seeded Faults

Faults were seeded into the Java classes by the second author. To avoid possible interactions among the faults, each fault was seeded into a separate copy of the class, except for a few cases when it was impossible for the faults to interact. To avoid bias, she did not participate in the test generation process. A slight side-effect of this independence was that some faults are similar to mutants and other faults are exact mutants created by muJava. This effect is explored in Section 5. The total number of faults is in Table 1.

4.3 Test Set Values

One source of small differences when comparing test criteria is the effect of having different values satisfy the same test requirements. Most test requirements can be satisfied by many input values, and sometimes different test sets that satisfy the same criterion will find different faults. Thus, small differences in the values in different test sets can produce accidental differences in which faults are found, a potential problem with internal validity.

We controlled for this "test value noise" by drawing all test sets from the same collection of values. Our intent was to make sure the four sets of tests for each program overlapped as much as possible. The process was as follows. First, we generated tests to satisfy the edge-pair criterion (T_{ep}) (344 tests total, as shown in Table 2). Next we added 92 tests to satisfy prime paths (T_{pp}). We then used the T_{pp} tests to satisfy all-uses (T_{au}), but only needed a total of 362 tests. Next, we ran T_{pp} on the mutants, and eliminated tests that did not contribute to mutation coverage. This left 208 tests, but some mutants were still alive. Finally, we added an additional 61 tests to kill the remaining mutants (T_{mut}). After this process, $T_{pp} \supset T_{ep}$, $T_{pp} \supset T_{au}$, $| T_{au} \cap T_{ep} | = 297$, and $| T_{pp} \cap T_{mut} | = 208$. All test values were generated by hand.

Table 2. Number of test requirements and tests

	Requirements	Infeasible	Tests
Edge-Pair	684	7	344
All-Uses	453	62	362
Prime Paths	849	175	436
Mutation	2919	562	269

4.4 Measuring Effectiveness & Efficiency

Each fault was seeded into a separate copy of the original Java class, thus making it obvious during execution which fault was detected. Efficiency was measured by taking the ratio of the number of tests over the number of faults found. The ratio estimates the number of tests required to find a fault for each test criterion; test criteria with lower ratios can be considered to be more efficient.

5 Results

Table 3 shows the data from the experiment, broken out by each class studied. The first two columns give program index numbers and the number of seeded faults. The next four columns show the number of faults found by each criterion for each program and the next four the number of tests for each criterion. The bottom row has the totals.

Figure 2 shows the total number of tests needed for each criterion across all subjects in bar chart form. The prime path criterion needed the most tests and mutation the least. Figure 3 shows the total number of test requirements for each criterion. The feasible numbers of test requirements

Table 3. Experimental results

		Number of faults found				Number of test cases			
	Faults	Edge-pair	All-uses	Prime path	Mutation	Edge-pair	All-uses	Prime path	Mutation
J1	1	0	0	0	1	2	5	7	4
J2	1	1	1	1	1	3	4	5	5
J3	1	1	1	1	1	2	3	7	4
J4	1	0	0	0	0	3	4	5	4
J5	1	1	1	1	1	2	5	7	4
J6	1	0	0	0	0	2	5	7	3
J7	2	1	1	1	2	3	3	4	5
J8	1	0	0	0	1	2	2	2	5
J9	1	1	1	1	1	2	2	3	3
J10	2	1	1	2	2	5	11	12	8
J11	2	1	1	1	2	3	3	3	5
J12	6	3	3	3	3	25	25	25	9
J13	2	2	2	2	2	6	4	6	3
J14	2	2	2	2	2	9	6	9	4
J15	1	1	1	1	1	2	6	10	3
J16	19	12	12	12	16	108	106	121	42
J17	4	0	0	0	3	8	5	8	3
J18	6	2	2	2	4	18	18	18	10
J19	1	1	1	1	1	12	11	12	10
J20	1	1	1	1	1	12	11	12	10
J21	1	0	0	0	1	3	3	3	12
J22	4	1	0	1	4	11	8	11	9
J23	6	4	4	4	4	27	26	27	25
J24	4	3	3	3	4	16	15	20	21
J25	3	3	3	3	3	6	6	6	9
J26	2	2	2	2	2	10	10	14	5
J27	2	1	1	1	2	14	12	16	21
J28	4	4	4	4	4	16	21	28	11
J29	6	6	6	6	6	12	22	28	12
Totals	**88**	**55**	**54**	**56**	**75**	**344**	**362**	**436**	**269**

are shown in parentheses. Despite the fact that mutation needed the fewest tests, it had far more test requirements than the other criteria.

Figure 4 shows the total number of faults found. Mutation detected the most faults and the other three criteria are roughly the same. Prime paths required the most tests and mutation the least.

Table 4 summarizes these data by dividing the faults into those that are equivalent to mutants generated by muJava (muJava faults), faults that look like mutants but that are not generated by muJava (mutant-like faults), faults that do not look like mutants (non mutant-like faults). The first category results in a slight bias toward the mutation test sets, however, mutation only found four more of these faults than the all-uses and prime path tests, so the effect is negligible.

Table 5 gives an approximation of the cost-benefit ratio of the four criteria on the experimental subjects. It is important to note that this table only counts the number of tests, not the cost of creating those tests. The creation cost would vary dramatically by the amount of automation used, particularly if automatic test data generation was available. As is, these data indicate that mutation is the most efficient, and prime path coverage the least. Anecdotally, we found

Table 4. Different types of faults found

	Num	Num faults found			
Type of faults		EP	AU	PP	Mut
muJava faults	9	5	5	5	9
Mutant-like faults	19	13	13	14	17
Non mutant-like faults	60	37	36	37	49
Sum	88	55	54	56	75

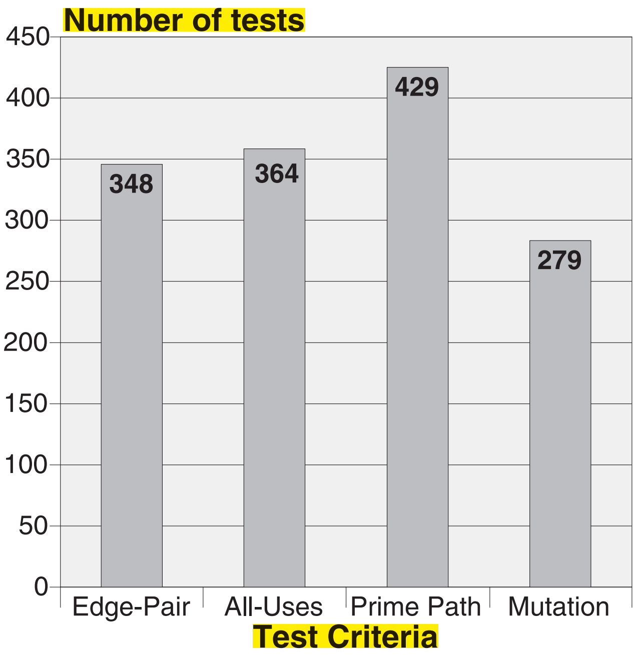


Figure 2. Number of Tests for Each Criterion

that generating and satisfying the test requirements for all-uses was more difficult than for edge-pair and prime paths. Finding values to kill the last few mutants was quite time consuming as well as intellectually challenging.

Table 5. Cost versus benefits ratio

	Tests	Faults	Cost / Benefit
Edge-pair	344	55	6.3
All-uses	362	54	6.7
Prime path	436	56	7.8
Mutation	269	75	3.6

6 Analysis and Discussion

All edge-pairs are simple paths, so edge-pair coverage is subsumed by prime path coverage [1]. Since every DU path (from a def to a use) is either a simple path or a prime path, and some prime paths are not DU paths, prime path coverage also formally subsumes all-uses [1]. Thus, our expectations were prime path coverage would found more faults than edge-pair and all-uses coverage. However, it is surprising that the tests from all three criteria found almost the same number of faults.

We were surprised that prime path coverage required more tests than mutation coverage. Mutation certainly has significantly more test requirements. However, most tests kill many mutants, whereas many prime paths require a unique test.

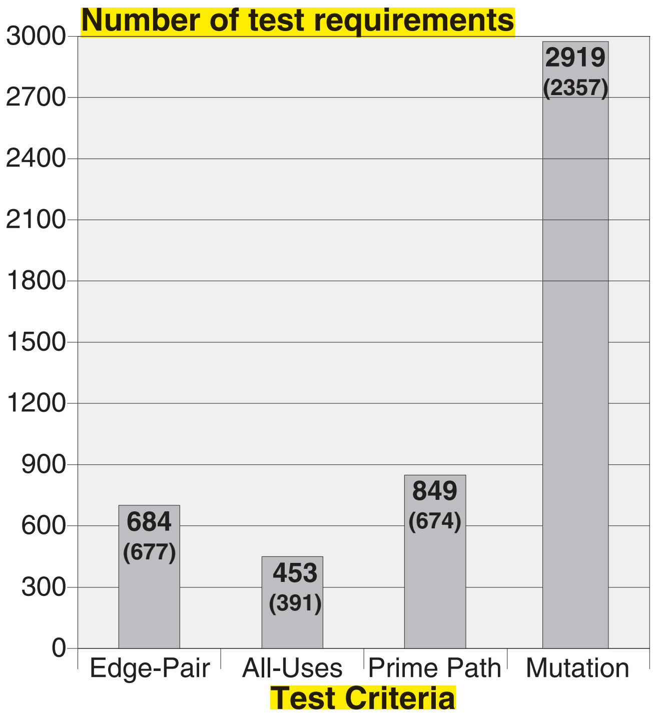


Figure 3. Number of Test Requirements by Each Criterion

These data shed important light on the question "which test criterion is best?", but do not provide a final answer. The answer depends on at least four issues: (1) how difficult it is to compute test requirements, (2) how difficult it is to generate tests, and (3) how well the tests reveal faults. The first two, of course, depend greatly on the level of automation used. In fact, a fourth issue may be how difficult it is to build automated tools. Computing all-use test requirements by hand is quite a bit more difficult than computing prime paths, which is in turn more difficult than computing edge-pairs. And it is obviously not practical to compute mutants by hand.

6.1 Analysis of Specific Faults

It is interesting to analyze some of the faults that were not found by some of the test sets. The mutation tests found all of the faults that the other three criteria found, but missed the following two faults.

The first fault is in method `lastZero()`, and is shown in Figure 5. The fault is that the for loop should search from the last element to the first, but searches from the first to the last. The correct statement should be:

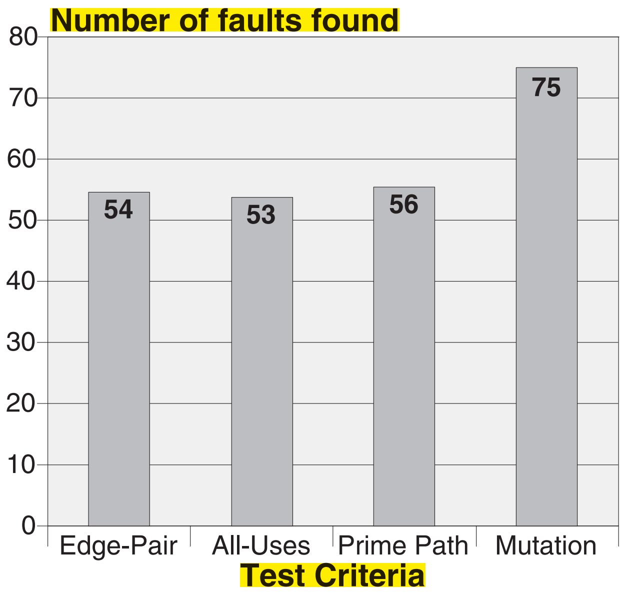


Figure 4. Number of Faults Found by Each Criterion

“`(for int i = x.length; x >= 0; i--)`”

muJava generated 22 mutants for `lastZero()`. The tests {0}, {2}, {1, 0}, and {-1} killed twenty and the rest are equivalent. The fault in Figure 5 could have been found by the test {0, 1, 0}. The expected output of this case is 2 and the actual output is 0 because the program returns the index of the array `x` once it identifies a `0`.

The second fault is in method `oddOrPos()`, and is shown in Figure 6. The fault is that the `if` statement only counts odd positive values, and misses odd negative values. The correct statement should be “`if (x[i]\%2 == -1 || x[i] >0)`,” which counts odd negative values.

```
public static int lastZero (int[] x)
{ // if x==null throw NullPointerException
  // else return index of the LAST 0 in x
  // Return -1 if 0 is not in x
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
```

Figure 5. First fault missed by mutation tests

```
public static int oddOrPos (int[] x)
{ // if x==null throw NullPointerException
  // else return number of elements in x
  // that are odd, positive or both
  int count = 0;
  for (int i = 0; i < x.length; i++)
  {
    if (x[i]%2 == 1 || x[i] > 0)
    {
      count++;
    }
  }
  return count;
}
```

Figure 6. Second fault missed by mutation tests

```
1. First fault missed by non-mutation tests
public static int countPositive (int[] x)
{  // if x==null throw NullPointerException
   // else return the number of
   // positive elements in x.
   int count = 0;
   for (int i = 0; i < x.length; i++)
   {
      if (x[i] >= 0)
      {
         count++;
      }
   }
   return count;
}
```

Figure 7. First fault missed by mutation tests

muJava generated 23 mutants for `oddOrPos()`. The test cases {2}, {1, 0}, and {2, 2, 2} killed twenty-one and the rest are equivalent. The fault in Figure 6 could have been found by the test case {-1}. The expected output of this case is 1 but the actual output is 0.

The edge-pair, prime path and all-uses coverage tests missed the same faults that the mutation tests missed, plus other faults that look like mutants. We illustrate two of these faults here. In the class `countPositive()`, shown in Figure 7, instead of counting positive values, the test in the `if` statement also counts values of 0. The correct statement should be “`if (x[i] > 0)`,” which counts odd negative values.

The tests {1}, {1, -1}, {1, 1}, {-1}, {}, and {-1, -2} were created to satisfy prime path coverage. The fault was not found because no 0s were included in the tests. The test case {0} could find the fault, and would have satisfied the same test requirements that test case {1} satisfied. In this case, the fault can be found. The other fault that prime path coverage missed is similar.

The second fault that the non-mutation tests missed was in the main class of project `ATMCaseStudy`, shown in Figure 8. The fault in the program is that "`pin != 0`" is added to the `if` condition in method `authenticateUser()`. The prime path coverage, edge-pair coverage, and all-uses tests missed this fault because the fault creates an additional possible test path: one test value (12345, 54321) goes through one test path if a user is authenticated and another test value (1235, 54321) goes through the other test path if not. In the `bankDatabase` object, the username and pin for three users were stored for testing: (12345, 54321), (98765, 56789), (1, 0). The added condition "`pin != 0`" is missed. In mutation testing, five test cases are needed to kill all 118 mutants, one of which, (1, 0), detects the fault.

6.2 Threats to Validity

As with all empirical studies involving software artifacts, **external validity** is questionable because we cannot be sure the subjects were representative. The subjects were from open source collections and textbooks, rather than part of industrial software products.

One threat to **internal validity** is because faults were seeded by hand. To minimize the effect of already knowing mutation, the second author tried to generate faults that were more complicated than altering a single change. We eliminated most faults that were obviously similar to mutants. Although we considered asking somebody who did **not** know mutation to seed the faults, we did not because experience with previous studies have convinced us that most faults people would naturally seed into programs would be mutants.

Another potential threat to internal validity is the specific tools used. We cannot be sure they were correct, but any effects would probably be similar across all techniques.

A potential threat to **construct validity** is that the faults were generated by only one person. Further studies could be carried out to see if using different people would result in different results. Another potential construct validity threat is that the test cases were generated manually. If we had access to better automatic test data generation tools, we could eliminate this type of threat.

7 Conclusions and Future Work

As expected, these results indicate that mutation testing will find more faults than other criteria. Perhaps a little surprising is that, despite its widespread reputation as being the "most expensive test criterion," mutation did not require significantly more tests. In fact, based on the efficiency ratio in Table 5, mutation is the **most efficient** criterion. This leads to an argument that the expense of mutation is worthwhile because it will help the tester find more faults.

The faults that mutation did **not** catch can be instructive. It may be possible to design additional mutation operators that can detect these faults.

Of course this study has several limitations. As in all studies that use software as subjects, external validity is limited by the number of subjects and the fact that we have no way of knowing whether they are representative of the general population. Most of the classes were reasonably simple, and we must leave it to a future replicated study to see if the results would be similar for larger and more complicated classes.

References

[1] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, Cambridge, UK, 2008. ISBN 0-52188-038-1.

[2] Paul Ammann and Jeff Offutt. Introduction to software testing website. Online, 2008. http://www.cs.gmu.edu/∼offutt/softwaretest/, last access January 2009.

[3] Paul Ammann, Jeff Offutt, and Wuzhi Xu. Coverage computation web applications. Online, 2008. http://cs.gmu.edu:8080/offutt/coverage/, last access January 2009.

[4] Special Interest Group in Software Testing British Computer Society. *Standard for Software Component Testing, Working Draft 3.3*. British Computer Society, 1997. http://www.rmcs.cranfield.ac.uk/∼cised/sreid/BCS_SIG/.

[5] T. Chow. Testing software designs modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[6] Lori A. Clarke, Andy Podgurski, Debra J. Richardson, and Steven J. Zeil. A formal evaluation of data flow path selection criteria. *IEEE Transactions on Software Engineering*, 15:1318–1332, November 1989.

[7] Harvey Deitel and Paul Deitel. *Java: How to program.* Pearson Education, Inc., sixth edition, 2005.

[8] Phyllis G. Frankl and O. Iakounenko. Further studies of test effectiveness. In *Sixth Symposium on Foundations of Software Engineering*, pages 153–162, Orlando, FL, November 1998. ACM SIGSOFT.

```
public class ATM
{
   ...
   // attempts to authenticate user against database
   public void authenticateUser()
   {
      screen.displayMessage ("\nPlease enter your account number: ");
      screen.displayMessage ("\nEnter your PIN: "); // prompt for PIN
      // set userAuthenticated to boolean value returned by database
      userAuthenticated = bankDatabase.authenticateUser (accountNumber, pin);
      // check whether authentication succeeded
      if (userAuthenticated && pin != 0)
      {
         currentAccountNumber = accountNumber; // save user's account #
      } // end if
      else
         screen.displayMessageLine ("Invalid account number or PIN. Please try again.");
   } // end method authenticateUser
   ...
   public int setPin (int pin1)
   {
      pin = pin1;
      return pin;
   }
   public int setAccountNumber (int accountNumber1)
   {
      accountNumber = accountNumber1;
      return accountNumber;
   }
} // end class ATM
```

Figure 8. Second fault missed by non-mutation tests

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

[10] Phyllis G. Frankl, Steven N. Weiss, and Cang Hu. All-uses versus mutation testing: An experimental comparison of effectiveness. *The Journal of Systems and Software*, 38(3):235–253, 1997.

[11] Phyllis G. Frankl and Elaine J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[12] P. Herman. A data flow analysis approach to program testing. *Australian Computer Journal*, 8(3):92–96, November 1976.

[13] W. E. Howden. Reliability of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, 2(3):208–215, September 1976.

[14] J. C. Huang. An approach to program testing. *ACM Computing Surveys*, 7(3):113–128, September 1975.

[15] Marlie Hutchins, H. Foster, Thomas Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, Sorrento, Italy, May 1994. IEEE Computer Society Press.

[16] Janusz Laski and Bogdan Korel. A data flow oriented program testing strategy. *IEEE Transactions on Software Engineering*, SE-9(3):347–354, May 1983.

[17] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. Mujava : An automated class mutation system. *Software Testing, Verification, and Reliability*, 15(2):97–133, June 2005.

[18] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. muJava home page. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/, http://salmosa.kaist.ac.kr/LAB/MuJava/, last access December 2008.

[19] Aditya P. Mathur and W. Eric Wong. An empirical comparison of data flow and mutation-based test ade-

quacy criteria. *Software Testing, Verification, and Reliability*, 4(1):9–31, March 1994.

[20] Aditya P. Mathur and W. Eric Wong. A theoretical comparison between mutation and data flow based test adequacy criteria. In *Proceedings of the 22nd annual ACM Conference on Computer Science*, pages 38–45, Phoenix, Arizona, 1994.

[21] S. C. Ntafos. On required element testing. *IEEE Transactions on Software Engineering*, 10(6):795–803, Nov 1984.

[22] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.

[23] Jeff Offutt, Jie Pan, Kanupriya Tewary, and Tong Zhang. An experimental evaluation of data flow and mutation testing. *Software–Practice and Experience*, 26(2):165–176, February 1996.

[24] S. Pimont and J. C. Rault. A software reliability assessment based on a structural behavioral analysis of programs. In *Proceedings of the Second International Conference on Software Engineering*, pages 486–491, San Francisco, CA, October 1976.

[25] Sandra Rapps and Elaine J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, April 1985.

[26] Elaine J. Weyuker. The cost of data flow testing: An empirical study. *IEEE Transactions on Software Engineering*, 16(2):121–128, February 1990.

[27] Elaine J. Weyuker, Stewart N. Weiss, and Richard G. Hamlet. Comparison of program testing strategies. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 1–10, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.