Program State Coverage: A Test Coverage Metric Based on Executed Program States

Khashayar Etemadi Someoliayi, Sajad Jalali, Mostafa Mahdieh, Seyed-Hassan Mirian-Hosseinabadi
Sharif University of Technology, Iran
{etemadi, sajalali, mahdieh}@ce.sharif.edu, hmirian@sharif.edu

***Abstract*—In software testing, different metrics are proposed to predict and compare test suites effectiveness. In this regard, Mutation Score (MS) is one of most accurate metrics. However, calculating MS needs executing test suites many times and it is not commonly used in industry. On the other hand, Line Coverage (LC) is a widely used metric which is calculated by executing test suites only once, although it is not as accurate as MS in terms of predicting and comparing test suites effectiveness. In this paper, we propose a novel test coverage metric, called Program State Coverage (PSC), which improves the accuracy of LC. PSC works almost the same as LC and it can be calculated by executing test suites only once. However, it further considers the number of distinct program states in which each line is executed. Our experiments on 120 test suites from four packages of Apache Commons Math and Apache Commons Lang show that, compared to LC, PSC is more strongly correlated with normalized MS. As a result, we conclude that PSC is a promising test coverage metric.**

***Index Terms*—Software Testing, Mutation Score, Line Coverage, Program State Coverage, Test Suite Effectiveness**

I. Introduction

Software developers create test suites to simulate the program states [1] in which the program might not work correctly. The created test suites are used to assess the program quality. However, every test suite simulates only a limited number of possible states of the program which means that there will still be the possibility of existing faults that the test suite cannot discover. Therefore, we need some metrics to evaluate the quality of test suites, measure their fault exposure capability, and compare them with each other.

There already exist a number of test coverage metrics; however, each of them has some deficiencies. For instance, using mutation score (MS) is considered as an appropriate way to evaluate the effectiveness of test suites in many studies [6], [9], [11]. Just et al. [11] showed that there is a "significant correlation between mutant detection and real fault detection". However, mutation analysis is rarely used in industrial projects because of its high calculation complexity. To calculate mutation score each test method should be executed on many different versions of the program (known as mutants). For example, on a subset of the JDK including around 7000 classes more than 200000 mutants are generated [13]. As a result, although MS is an accurate metric in terms of predicting and comparing test suites effectiveness, it could not be widely used.

On the other hand, Line Coverage[2] (LC) is a coverage metric that is used widely but it does not compare test suites effectiveness as accurately as MS does. LC is defined as the ratio of lines executed by the test suite to total executable lines in a program. In other words, LC assigns each line either 0 or 1 as the coverage point of the line and then considers the average coverage point of all executable lines as the total coverage. This coverage metric can be calculated by running the program only once; hence, it can be calculated faster than the MS and is used in many industrial projects [15]. However, LC has drawbacks in measuring test suite effectiveness. For instance, suppose an existing test suite which executes a line only in one state of the program. LC would consider that line as covered by the existing test suite. Nonetheless, it is possible that there is a non-detected fault in the executed line which affects the output only when the line is executed in a different state. In fact, in this situation, the existing test suite executes the line only in a specific state while an improved version of the test suite would execute the line in all possible distinct states. In contrast to an ideal metric, LC does not distinguish between the existing test suite and its mentioned improved version since it assumes that the line is completely covered by both of them. Thus, while LC can be calculated by running the test suite only once, it is not an ideal metric for comparing test suites effectiveness.

All these facts about MS, and LC show that it could be interesting to have a new test coverage metric with two important features. First, we should be able to calculate the value of the new metric by running the program only once. Second, the new metric should consider the distinct program states that the test suite executes. In this paper, we introduce the Program State Coverage (PSC) as a new coverage metric for evaluating test suites effectiveness. PSC performs much like the LC. The only difference between these two metrics is how they assign a coverage point to each line. PSC assigns each line a real number between 0 and 1 based on the number of distinct program states in which that line is executed (this point will be discussed more in Section II).

We performed an experiment on 120 test suites (including 1040 test cases in total) from Apache Commons Math [2] and Apache Commons Lang [1] to evaluate PSC. In the

[1]The program state of a running program is determined by the values of the PC, i.e., program counter, and all the program variables [4].

[2]Researchers usually use the notion of Statement Coverage instead of Line Coverage. Despite some minor differences, these two metrics work almost the same. In this paper, we use the notion of Line Coverage because it helps us to better explain our analysis about our proposed metric.

experiment, we used Java Debug Interface (JDI) [3] to find the distinct program states in which each line is executed. Our results show that the Kendall τ correlation between PSC and normalized MS is more than the Kendall τ correlation between LC and normalized MS (see Section III). Moreover, PSC can be calculated by running the test suite only once. Therefore, PSC seems a promising test coverage metric that can be used to compare test suites effectiveness. Thus, this paper makes the following contributions:

- It proposes PSC as a new test coverage metric which can be calculated by executing the test suite only once while it compares test suites effectiveness better than LC does.
- It reports an evaluation comparing the correlation between MS and PSC with the correlation between MS and LC and shows PSC is a promising test coverage metric.

The rest of this paper is organized as follows: Section II explains how PSC is calculated and compares it with other coverage metrics, Section III discusses the experiment and evaluation, Section IV reviews the related works, and Section V concludes the paper with future work.

II. Program State Coverage as a New Test Coverage Metric

In this section, we define program state coverage and describe how it is calculated. We also show how PSC is different from LC, and MS by reviewing an example.

```
1  public class App {
2    public static int countNonMultipleOfThree(int[] arr){
3      int counter = 0;
4      for(int i = 0; i < arr.length; i++){
5        int n = arr[i];
6        int minPositiveRemainder = 1;
7        if(n % 3 >= minPositiveRemainder)
8          counter++;
9      }
10     return counter;
11   }
12 }
```

Fig 1. App is a correct Java program used for counting the numbers in an array that are not multiple of three.

```
1 public class TestSuite1 {
2   @Test
3   public void test(){
4     int cnt = App.countNonMultipleOfThree(new int[]{1});
5     assertEquals(cnt, 1);
6   }
7 }
```

Fig 2. TestSuite1 tests App.countNonMultipleOfThree() by passing an array that does not contain any member which is multiple of three.

```
1 public class TestSuite2 {
2   @Test
3   public void test(){
4     int cnt = App.countNonMultipleOfThree(new int[]{1, 2, 3});
5     assertEquals(cnt, 2);
6   }
7 }
```

Fig 3. TestSuite2 tests App.countNonMultipleOfThree() by passing an array that contains a member which is multiple of three.

```
1  public class AppF{
2    public static int countNonMultipleOfThree(int[] arr){
3      int counter = 0;
4      for(int i = 0; i < arr.length; i++){
5        int n = arr[i];
6        int minPositiveRemainder = 0;
7        if(n % 3 >= minPositiveRemainder)
8          counter++;
9      }
10     return counter;
11   }
12 }
```

Fig 4. AppF is a faulty version of App.

A. Definition and Calculation of Program State Coverage

In order to calculate the PSC of a test suite, at first, we run all the test cases. Let $PS(l)$ be the set of all distinct program states in which line l is executed by the test suite. For each line l_i, we find the content of $PS(l_i)$. Afterward, using an increasing function, which we call the PAF (i.e., Point Assigner Function), a point between 0 and 1 is assigned to l_i due to the size of $PS(l_i)$. The input of PAF is the size of $PS(l_i)$ which can be any natural number and its output is the point assigned to l_i which is a real number between 0 and 1. Finally, the average point for all the executable program lines is reported as the total coverage.

For example, consider how PSC is calculated when the test suites in Fig. 2 and Fig. 3 are executed on the program in Fig. 1. App.countNonMultipleOfThree() method in Fig. 1 counts the number of members of arr that are not multiple of three. TestSuite1 and TestSuite2 in Fig. 2 and Fig. 3 are created to test App.countNonMultipleOfThree() method. At first both test suites are executed on App. The size of $PS(l_i)$ for each executable line is shown in Table I. Next, we need an increasing function (PAF) to assign a point to each line when each test suite is executed.

For instance, *f* defined in (1) can be a PAF. This function assigns 0.5 to the lines that are executed only in one program state and assigns 1 to the lines that are executed in two or more distinct program states. The output of *f* would be zero for the lines that are not executed at all. Table I shows the point assigned by *f* to each executable line of App when each of TestSuite1 and TestSuite2 is executed.

$$f\colon\mathbb{N} \to [0,1]\,,\ \ f(x) = \begin{cases} 0 & x = 0 \\ 0.5 & x = 1 \\ 1 & x \geq 2 \end{cases} \qquad (1)$$

In the final step of PSC calculation, the average point assigned to the executable lines is calculated as the total PSC. Therefore, due to the information in Table I, the PSC of TestSuite1 and TestSuite2 would be 0.5 and 0.87, respectively.

B. Comparing Program State Coverage with Line Coverage and Mutation Score

In order to show the difference between PSC and LC, we create a faulty version of App by changing ‘1’ to ‘0’ in line 6 of App. The result would be AppF in Fig. 4. The

TABLE I
SIZE OF $\mathbf{PS}\left(\mathbf{l_i}\right)$ FOR EXECUTABLE LINES OF App AND THE POINT ASSIGNED BY f TO EACH LINE.

Line#	**Size of $PS\left(l_i\right)$**		**Point (Assigned by g)**	
	TestSuite1	TestSuite2	TestSuite1	TestSuite2
3	1	1	0.5	0.5
4	1	3	0.5	1
5	1	3	0.5	1
6	1	3	0.5	1
7	1	3	0.5	1
8	1	2	0.5	1
9	1	3	0.5	1
10	1	1	0.5	0.5
AVG	1	2.37	**0.5**	**0.87**

point is that this fault can be detected only if the input of AppF.countNonMultipleOfThree() contains a member which is a multiple of three. This is the reason why TestSuite1.test() passes on AppF while TestSuite2.test() does not. In other words, the fault in AppF can be detected only if line 7 is executed when n is a multiple of 3. Therefore, there might be some test suites, such as TestSuite1, that execute this line of code in a program state that does not lead to detecting the fault while some other test suites, such as TestSuite2, detect the fault.

Since both TestSuite1 and TestSuite2 execute all the lines of App.countNonMultipleOfThree(), they have the same LC while, as it was described in previous section, TestSuite2 has a higher PSC. LC metric does not distinguish between TestSuite1 and TestSuite2 because it does not consider different program states in which each line of the program is executed by the test suite. On the other hand, PSC distinguishes between these test suites because it assigns a higher point to a line when the line is executed in more number of distinct program states. This example shows that at least in some cases PSC is a better metric than LC for comparing test suites effectiveness.

We also note that the mutation scores of TestSuite1 and TestSuite2 are different because some of the generated mutants for App.countNonMultipleOfThree() are killed by TestSuite2 but not by TestSuite1. For example, one of the mutation operators for Java programs which is implemented in PIT [7] (INLINE_CONSTS), would replace '1' in line 6 of App with '0'[3]. The result of applying this mutation operator on the mentioned line would be AppF in Fig. 4 which can be killed by TestSuite2 but not by TestSuite1. Therefore, MS distinguishes between TestSuite1 and TestSuite2.

III. EVALUATION

In order to evaluate PSC, we conducted an experiment on four packages of Apache Commons Math and Apache Commons Lang. For each of these packages, we calculated LC, PSC, and MS for 30 randomly chosen test suites and compared LC and PSC due to their Kendall τ[4] correlation with MS.

In the rest of this section, at first, we describe how we implemented PSC calculation. Then we review the experiment that we conducted to evaluate PSC. Finally, some threats to the validity of our arguments are discussed.

A. Implementation of Program State Coverage

In order to calculate PSC of a given test suite, for each line, we have to find all the program states in which that line is executed. For instance, when TestSuite1 is executed, line 8 of App is executed only in one program state and this program state is determined by the values of counter, i, n, arr, and the program counter. We used JDI to find all the executed program states. JDI enables us to get values of all variables while the program is running. In order to avoid excessive time consumption, for the variables that refer to objects, we find the content of the objects up to 3 levels in depth.

In the calculation process, at first, all the test cases are executed one by one and during the execution of each program line, we save the current program state as a hash code. Although we can save the whole content of the program state, we save it as a hash code in order to reduce the size of the saved objects. When all the program states executed by each test case are found, we combine the saved data for all the test cases and compute $PS\left(l_i\right)$ for all the program lines.

Afterward, we calculate the PSC of the test suite using k in (2) as the PAF.

$$k:\mathbb{N} \rightarrow [0,1]\,,\;\; k\left(x\right) = \min(1, \frac{x}{10}) \tag{2}$$

Using k as the PAF means that if a line is executed in 10 or more distinct program states, that line will get the maximum point (namely, 1). Besides, if a line is executed in x distinct program states and x is less than 10, k assigns $\frac{x}{10}$ to that line. Therefore, when we use k as the PAF, we do not have to know the total number of possible distinct program states in which a line can be executed.

B. Correlation between Program State Coverage and Mutation Score

As it was mentioned in Section I, in some cases, two test suites have different PSCs while their LCs are the same. Actually, LC can be different only when a different number of lines are covered while PSC can be different even when the number of covered lines is the same but different number of program states are executed. Therefore, it seems that PSC is more likely to be different for any given pair of test suites. However, we have to ensure that the difference that PSC makes between test suites can be used effectively to compare test suites effectiveness. In order to evaluate whether a higher PSC for a test suite means a higher effectiveness, we calculated the Kendall τ correlation between PSC and MS for 120 randomly chosen test suites and compared it with the correlation between LC and MS in those test suites. A higher correlation between a metric and MS means that the metric is more accurate in evaluating test suites effectiveness. This method is widely used for evaluating different test coverage metrics [8], [9]. The idea

[3]Inline Constant Mutator (http://pitest.org/quickstart/mutators/)

[4]Kendall τ is a measure of rank correlation [12]. A higher Kendall τ correlation between a coverage metric and MS means that we can better predict the rank order of MS values of different test suites given the rank order of their coverage values [9].

TABLE II
LC, PSC, NORMALIZED MS, AND NON-NORMALIZED MS FOR SOME OF THE TEST SUITES.

Test Suite ID	**Tested Package**	**Package LOC**	**LC**	**PSC**	**Normalized MS**	**Non-normalized MS**
S1	org.apache.commons.math4.fraction	431	0.26	0.03	0.34	0.07
S2	org.apache.commons.math4.analysis.polynomials	374	0.30	0.20	0.86	0.21
S3	org.apache.commons.lang3.math	715	0.07	0.02	0.85	0.03

TABLE III
THE KENDALL τ CORRELATION BETWEEN NORMALIZED AND NON-NORMALIZED MS AND EACH OF LC AND PSC.

Tested Package	**LC**		**PSC**	
	Normalized	Non-normalized	Normalized	Non-Normalized
org.apache.commons.math4.fraction	0.19	0.66	0.23	0.5
org.apache.commons.math4.analysis.polynomials	0.26	0.52	0.46	0.65
org.apache.commons.lang3.math	0.01	0.89	0.14	0.87
org.apache.commons.lang3.text.translate	0.45	0.65	0.54	0.79

behind this method is that a higher MS of a test suite means the test suite is more effective in detecting possible program faults [11].

In our evaluation, we computed the correlation between normalized and non-normalized MS and each of PSC and LC. We use the notion of normalized MS as it is defined in [9]. The normalized MS is defined as the number of mutants killed by a test suite divided by the number of non-equivalent mutants that the test suite covers. A test suite covers a mutant if it executes the line of the original program that is altered in the mutant. Non-normalized MS is the number of killed mutants divided by the number of all non-equivalent mutants.

We created the test suites by choosing a number of test cases from the test cases existing in the online repositories of the projects. In order to do this, for each of the four tested package, we made 10 suites of each of the following sizes: 3 methods, 8 methods, 15 methods (120 test suites in total). All the test methods in the generated suites were randomly chosen. Afterward, we calculated LC, PSC, and normalized and non-normalized MS for each of these test suites. The results of this calculation for three different test suites are shown in Table II. Finally, we computed Kendall τ correlation between normalized and non-normalized MS and each of PSC and LC. The results of this experiment are shown in Table III.

As it can be seen in Table III, compared to LC, PSC is more strongly correlated with normalized MS in all the packages. This fact shows that PSC is more accurate than LC for determining which test suite better reveals the possible faults in the covered lines. Moreover, since LC and PSC are both strongly correlated with non-normalized MS, we can also conclude that when more lines or program states are covered, more possible faults can be found. PSC dominates LC regarding its correlation with normalized MS in all the four tested packages. Nevertheless, neither LC nor PSC dominates the other one regarding its correlation with non-normalized MS in all the tested packages. Considering these facts, it seems that PSC could compare test suites effectiveness better than LC does.

Furthermore, another benefit of using PSC for evaluating the effectiveness of a test suite is that the PSC could be calculated by executing the test suite only once. Contrarily, in order to calculate the MS, we have to execute the test suite on all of the generated mutants one by one.

Since PSC can compare test suites effectiveness better than LC does and it can be calculated by executing test suites only once while calculating MS needs executing test suites on all the mutants, PSC seems to be a promising test coverage metric.

C. *Threats to validity*

There are four main threats to the validity of this study. First, it may be argued that the two projects that we used for the evaluation are both written in Java; however, the results might be different for other languages. We believe that even though there is much difference between Java and other programming languages, the assumptions that we made in our work are common among most programming languages. Namely, the assumptions that 1) "mutation testing is widely believed to be a computationally expensive testing technique" [10], 2) it is possible that two test suites have the same LC while they are different regarding their effectiveness, and 3) a test suite that executes a line in different program states is more likely to find possible faults in that line, compared to a test suite which executes the same line only in one program state. Nevertheless, it makes sense to evaluate PSC on projects with different programming languages.

Second, the limited number of the programs that we used in our experiment is another threat to the validity of our analysis. We used only 120 test suites in the experiment. However, the fact that the correlation between PSC and normalized MS is more than the correlation between LC and normalized MS in all the tested packages without any exceptions shows that the results for other datasets are likely to be the same as current results. Nonetheless, repeating the experiment with different subject programs could make us more confident in the results.

Third, we have compared PSC only with LC and MS while there are some other metrics that PSC can be compared with. However, Inozemtseva et al. [9] showed that normalized MS has approximately the same Kendall τ correlation with each of statement coverage (which works almost the same as LC), decision coverage, and modified condition coverage.

Therefore, since PSC works better than LC, it will probably work better than decision coverage and modified condition coverage, too. Nevertheless, PSC should be compared with other test coverage metrics in the future.

Finally, as it was explained in Section III-A, we use some heuristics to make PSC calculation faster. For instance, in order to find the current program state, we consider the content of the objects only up to 3 levels in depth which seems an ad hoc heuristic. However, our experiment shows that our proposed method works well with this heuristic although it might work even better with different heuristics.

IV. Related Work

There are a lot of studies in the literature investigating different coverage metrics and their uses. Some studies try to show how different coverage metrics perform in evaluating and comparing test suites effectiveness [8], [9]. There also exist a number of studies proposing new test coverage metrics for evaluating test suite effectiveness. For example, Vanoverberghe et al. [17] propose Object Sensitive State Coverage as the ratio of the number of state updates read by assertions to the total number of state updates. This metric is different from PSC because PSC considers the content of all the objects as the current program state while, in the mentioned study, a state update is a pair of the reference of an updated object and the code location of the update.

Since MS seems to be an accurate metric to measure the test suite effectiveness, some researchers have tried to reduce MS calculation time in order to make it more useful in real-world projects. Some studies propose and evaluate different methods to select only some of the mutation operators and generate mutants based on them [5], [16], [18]. Lima et al. [14] show that choosing 20% of the generated mutants by random is one of the best conventional methods for reducing MS calculation time.

Our study can be categorized as a study which proposes a new coverage metric for evaluating test suite effectiveness. This new coverage metric can also be used for test generation, test selection, and many other test related activities.

V. Conclusions and Future Work

To sum up, in this paper, we proposed Program State Coverage as a novel test coverage metric which considers the number of distinct program states in which each line is executed. Our experiment on 120 test suites on four Java program packages shows that the correlation between PSC and normalized MS is higher than the correlation between LC and normalized MS in all the four tested packages. Besides, PSC can be calculated by executing the test suite only once while MS calculation needs executing the test suite on all the generated mutants. As a result, we conclude that PSC is a promising test coverage metric.

In the future, we should evaluate our proposed metric on larger projects with various programming languages. The ad hoc heuristics used in this paper could also be further studied and replaced by better heuristics. Moreover, we use a simple increasing function (k in (2)) to assign a point to each line due to the set of the program states in which that line is executed. Nonetheless, it makes sense to study the effect of using other methods to assign points to the lines. Finally, since test coverage metrics have different uses, we can also investigate the effects of using PSC for test generation, test selection, test minimization, and other activities that could be based on different test coverage metrics.

References

[1] Apache commons lang. https://github.com/apache/commons-lang. Accessed: 2018-11-20.

[2] Apache commons math. https://github.com/apache/commons-math. Accessed: 2018-11-20.

[3] Java debug interface. https://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi. Accessed: 2018-11-20.

[4] P. Ammann and J. Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[5] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for c. *Software Testing, Verification and Reliability*, 11(2):113–136, 2001.

[6] X. Cai and M. R. Lyu. The effect of code coverage on fault detection under different testing profiles. *ACM SIGSOFT Software Engineering Notes*, 30(4):1–7, 2005.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. Pit: a practical mutation testing tool for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 449–452. ACM, 2016.

[8] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[9] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, pages 435–445. ACM, 2014.

[10] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2011.

[11] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[12] M. G. Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[13] P. Krishnan, J. Loh, R. O'Donoghue, and L. Meinicke. Evaluating quality of security testing of the jdk. In *Proceedings of the 8th ACM SIGSOFT International Workshop on Automated Software Testing*, pages 19–20. ACM, 2017.

[14] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. Silva, H. V. Ehrenfried, et al. Evaluating different strategies for reduction of mutation testing costs. In *Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing*, page 4. ACM, 2016.

[15] S. Park, B. Hossain, I. Hussain, C. Csallner, M. Grechanik, K. Taneja, C. Fu, and Q. Xie. Carfast: Achieving higher statement coverage faster. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 35. ACM, 2012.

[16] A. Siami Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th international conference on Software engineering*, pages 351–360. ACM, 2008.

[17] D. Vanoverberghe, J. de Halleux, N. Tillmann, and F. Piessens. State coverage: Software validation metrics beyond code coverage. In *International Conference on Current Trends in Theory and Practice of Computer Science*, pages 542–553. Springer, 2012.

[18] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 435–444. ACM, 2010.