Evaluation of Fault Localization Techniques using Pex© Parameterized Unit Tests

Luke Sandberg

University of Illinois

lsandbe2@illinois.edu Chris Line

University of Illinois

line1@illinois.edu

Mike Mercado

University of Illinois

mmerca8@illinois.eduRyan Gross

University of Illinois

gross14@illinois.edu

**ABSTRACT**

A number of studies have explored different test generation methods as well as several fault localization techniques, but combining both techniques for automated testing is an idea that has not been extensively explored. By combining Pex with three different fault localization techniques, we hope to demonstrate that it is possible to both detect and localize software faults with minimal user intervention.

**Categories and Subject Descriptors**

D.2.5 [**Software Engineering**]: Testing and Debugging – Testing tools*,* Symbolic execution,Debugging aids*.*

**General Terms**

Measurement, Reliability, Experimentation, Languages.

**Keywords**

Keywords are your own designated keywords.

# INTRODUCTION

A number of previous studies have demonstrated the feasibility and usefulness of automated testing techniques in software development. These methods seek to reduce the manual effort required by the testing process and increase the efficiency and, in some cases, reliability of software testing through the use of new tools or algorithms. Some of the techniques explored in these previous studies include methods for test generation (such as Pex [[1](#4cfdbea71d04b)] or Randoop [[2](#4cfdc085826c5)]) as well as those for fault localization (such as Tarantula or value replacement). Many of these techniques have been proven to be useful and have been applied to real software projects.

While fault localization and test generation have both been explored in depth, we have encountered only one previous study [[3](#4cfd96ebbf857)] that attempted to link the processes for a complete testing tool that can both reveal and localize faults without any need for user intervention. This study specifically targeted PHP applications, so many of its innovations were specific to programs that might generate error-causing output in the form of malformed HTML. In order to further explore this area as it might pertain to .NET development, we have attempted to combine the test-generation capabilities of Microsoft Pex with three popular fault-localization techniques, in order to determine the usefulness of Pex-generated tests for localizing software defects.

# Background

## Pex

Pex is an automated test generation tool developed for .NET. Currently available as a Visual Studio Power Tool, Pex is an example of a tool that utilizes “concolic” execution, meaning it combines aspects of concrete and symbolic execution techniques. Rather than being purely random when generating tests, Pex aims to maximize code coverage. Pex executes a program under test symbolically until it encounters a place where the execution path might branch, such as a conditional check. Pex then identifies different concrete values for its symbolically referenced variables -- often targeting edge cases, but also using a constraint solver to identify as many relevant values as possible -- that will pass or fail the different branch criteria. These values are then used to generate specific tests for the program that will, as a result, cover different execution paths in the code.

## Automated Fault Localization Techniques

The field of automated fault localization attempts to solve the following problem: Given a failing program, which portion of the program is responsible for the failure? There have been several attempts to solve this problem in the past decade, with varying degrees of success. The techniques can be divided into two categories: static techniques that utilize information from prior runs of the program and dynamic techniques that manipulate the state of a running program.

### Static Techniques

This section describes the set of fault localization techniques that rely only on the execution traces of failed vs. passing test cases, which we will call static techniques. These techniques are characterized by their low complexity, and therefore they are easier to set up and faster to produce their results than dynamic techniques.

The initial work in the field of static techniques was done by Agrawal and colleagues [[4](#4cfd96ebbfd71)] who specified a technique called program dicing. They first captured the program slices of each passing and failing test case, then used the set differences of the program statements in the slice of a passing test and the failing tests to determine which statements may be faulty.

Their work was expanded upon by Jones et al. in [[5](#4cfd96ebbfd0c)], where a statistical approach, called Tarantula, was proposed. Tarantula works by running the entire test suite, then determining how many passing test cases and failing test cases were executed on each particular program statement. This combination of coverage data and test result is called the diagnosis matrix. Next, each statement is assigned a probability of containing the fault according to the following formulas:

The intuition behind these formulas is that the faulty statement(s) will be more likely to be executed in the failing tests than in the passing tests, and that the suspiciousness is likely to be more accurate as more passing or failing tests are executed on a given statement.

An empirical study of Tarantula by these authors [[6](#4cfd96ebbfcaf)] introduced the standard accuracy measurement that is used for fault localization: the number of statements that must be examined by the programmer before the fault is found, if the programmer examines the statements in the order returned by the algorithm.

The Tarantula method has become the basis for most of the work in the field of static techniques, and has been expanded upon several times. In [[7](#4cfd96ebbf9ba)] Abreu, Zoeteweij, and van Gemund evaluate several formulas for calculating the suspiciousness of a program statement, and determine that the Ochiai formula:

produces the most accurate results.

In [[8](#4cfd96ebbfde5)], Masri et al. investigate branch coverage and definition-use pair coverage as alternatives to statement coverage, concluding that both offer better accuracy. Santelices et al. expand upon this work in [[9](#4cfd96ebbf94a)] by examining combinations of the three coverage types, concluding that an average of the statement, branch, and du-pair values for each statement performs better than branch or du-pair coverage alone.

Baudry et al. [[10](#4cfd96ebbfc47)] examine the characteristics of a test suite that best enable fault localization, which leads them to the concept of the dynamic basic block. They define a DBB as a set of statements that, given a test suite, are executed by the same subset of the test suite. With this definition, they show that the statements within each dynamic basic block are indistinguishable to any static fault localization algorithm, thereby establishing the theoretical bound on the effectiveness of a static technique to be the size of the DBB containing the fault. This leads to the conclusion that the test suite used for fault localization should minimize the size of the DBBs in the program.

### Dynamic Techniques

Dynamic fault localization techniques are varied, but generally use a custom runtime environment to manipulate or observe the dynamic program state. Dynamic techniques have the benefit of vastly more information than most statistical techniques; however, they generally suffer greatly in terms of run time and implementation complexity. For example, the *Cause Transitions* technique introduced by Cleve and Zeller [[11](#4cfd9767cc6b5)] describes a dynamic technique in which they dynamically compare the runtime state of a failing run with a similar passing run. They then use *Delta Debugging* [[12](#4cfd974f71773)] to rule out extraneous differences between the two runs leaving a set of program locations where the state diverged between the two runs. Other dynamic techniques include *Dynamic Program Slicing* [[4](#4cfd96ebbfd71)] in which the target program is monitored in order to determine all program locations that contribu­­­­­te to the final faulty result.

In this paper we will focus on the dynamic technique *Value Replacement* introduced by Jeffrey, Gupta and Gupta [[13](#4cfd96ebbfb79)]. The *Value Replacement (VR)* technique attempts to find statements that can be shown to affect the final outcome of the run. This is done by inspecting the faulty test cases and then for each statement of the faulty run, substituting different variable values and then re-running the test in order to try to change the final outcome. If they can find a different set of values for a statement such that a failing run becomes a passing run then that statement is likely to either be faulty itself or closely related to a faulty statement.

if (info >= 0.1) //changed 0.0 to 0.1

**Example fault from a SIR** [[14](#SIR)]**project**

The above line of code represents a typical fault we should try to find using value replacement. If info has a value of *0.05* then it will fail the test when we know it should pass (if the condition were correct). If there were a revealing test case for this error then, using value replacement, we would substitute new values for info at this line. If we were to substitute *0.2* for *info* then the condition would pass and the test case would pass. The *Value Replacement* technique searches for these alternate value mappings and then proposes them as potential faults.

## Process Overview

Imagine that you are tasked with refactoring a legacy module to make it reusable for a new module. The module is old, complex, poorly documented, and essential to the function of the other modules that currently use it . In this case, it would be good to have a way to know which of your changes may have broken some legacy behavior of the system. If there were a way to generate a suite of tests that quickly captured the state of the current program to tell you that you have broken some of the legacy behavior this would be useful. However, the process of finding which change caused the failure could be time consuming. Thus, if these generated tests were accurate enough to use one of the aforementioned fault localization techniques, the efficiency of making these changes could be greatly improved.

Because of the nature of bounded exhaustive test generation tools such as Pex, the test suites generated by these tools should be able to produce a test suite that is well suited to distinguishing between blocks of code. Therefore, we have developed a process to capture the state of an existing program in a generated test suite, and then utilize that test suite to localize faults caused by changes to that program.

The process is straightforward. First, the test generation tool is applied to generate a suite of test inputs, capturing the current state of the program. Next, as changes are made to the program, the suite of generated tests is run periodically. When a fault is detected by the tests, then one or more fault localization algorithm is applied in order to determine the suspiciousness of the lines in the program, which is then presented back to the programmer.

## Implementation Overview

To implement this process for .Net, we utilize Pex as our Test Generator, run the tests using the MSTest testing framework that is built into Visual Studio, and have written custom tools to utilize the output of the test runs to utilize the fault localization algorithms.

The tool we have developed for static localization simply parses the code coverage files generated by the MSTest runtime to generate the diagnosis matrix and calculate the statistical probabilities for each program statement. Due to a limitation in the Microsoft coverage data, we were only able to obtain line number information, so we cannot evaluate branch coverage or du-pair coverage in this paper.

The tool we have developed for dynamic localization utilizes the Mono.cecil [[15](#MonoCecil)] project to dynamically instrument the CIL code that all .NET languages compile to. The instrumentation replaces all variable accesses with a call into a custom method that allows the code to inject a new value for each variable any time it is accessed. The code for these tools is available on GitHub [[16](#GitHubAccount)].

# Evaluation

## Experimental Setup

### Siemens Suite

In order to properly evaluate the effect of using Pex-generated test inputs has on the selected fault localization techniques, it is necessary to work with a standardized set of test data also used by other fault localization experiments. Other research papers, including those on Tarantula and Ochiai, use a set of programs known as the Siemens suite [[14](#SIR)] to assess the ability for a particular fault localization technique to locate seeded faults. The Siemens suite consists of 7 individual programs, with a total of 132 faulty versions of the programs. Every program has several faulty versions containing exactly one seeded fault, and it also comes with a set of test inputs and test cases that achieve full coverage of the program.

Research on Tarantula and Ochiai use the provided test inputs and test cases to generate the suspiciousness rankings used in locating a particular fault. However, in this experiment we replace the given test inputs and test cases with those generated by Pex, and compare the results of the Tarantula and Ochiai research.

Since Pex is specifically used for C# programs, all the Siemens suite program must be ported from C to C# in order to have Pex generate test inputs for them.

### Data Acquisition

Once the original version and all the faulty versions of each Siemens suite program were ported to C#, the goal is to create a test suite based on the original fault-free version of each program. To do this, we create parameterized unit tests for the program and then use Pex to generate interesting test inputs to these PUTs that achieve maximum code coverage.

After there is a test suite that passes all tests for the original version of the program, each faulty version is then run against that test suite and eventually different tests will fail due to the seeded fault. We only used the faulty versions that actually caused some tests in the test suite to fail. This test result data is passed to the fault localization technique logic to evaluate the suspiciousness rankings of each line based on the passed and failed tests that executed it. To automate this process, we developed a test runner that runs each faulty version against the test suite and relays the results to the fault localization logic.

### Evaluation Metric

The fault localization logic outputs several metrics for each suspected line of code. Both Ochiai and Tarantula compute a suspiciousness value, which essentially measures the likelihood that a particular line of code contains a fault. Once these suspiciousness values are calculated suspiciousness rankings are assigned to each line of code based on its suspiciousness value. The Score metric is a value between 0 and 1, and uses the suspicious ranking to determine the percentage of suspected lines that do not need to be examined to find the fault. The closer this value is to 1, the more effective the fault localization technique performed. These common metrics are used in other research papers and can be directly compared with the results of this experiment.

## Process

[Luke]

## Results

### Siemens Suite

In many cases, the code under test that we ported from the Siemens suite contained faults that were not revealed by Pex-generated tests. In the case of one program, PrintTokens, all of the faulty versions were able to pass 100% of the Pex-generated tests. We suspect that this is caused in many cases by the fact that the seeded faults did not cause total program failures, but perhaps manifested instead as output issues, which *Pex* cannot detect. For the remaining programs in the Siemens suite, however, we were able to find some versions that contained faults that were revealed by Pex. The results shown in this section cover only those versions for which Pex-generated tests revealed the seeded faults.

For the remaining Siemens test programs that we analyzed, the combination of Pex test generation with some kind of fault localization technique was able to find the targeted fault with a high degree of accuracy. For Tarantula and Ochiai, the rankings and associated scores were similar for each tested program; we attribute the high number of identical scores to the small number of tests generally associated with a given error, resulting in a very high suspiciousness rating for a faulty line regardless of the technique used. This same shortage of tests resulted in a failure of value replacement in some cases (specifically the Siemens program tcas, for which Pex only generated 10 tests), and in these cases the value replacement defaults to the Tarantula score.

Tarantula and Ochiai were both able to achieve a rating of greater than 90% in 16/22 versions tested, and greater than 80% in 21/22 versions tested, meaning that a developer would need to examine at most 20% of the code under test in order to identify the faulty statement. For the remaining version, the faulty line was not covered by the generated tests (either due to the alteration of a conditional statement, or due to the fact that the fault itself was an omission of code). While this fault still manifested itself as a failing test, the fault localization techniques were unable to localize the faulty line of code. We considered these lines to have no rank and a score of 0%.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Score | | |
| Program | Version | Tarantula | Ochiai | VR |
| Tcas | 6 | 0.952381 | 0.952381 | 0.952381 |
|  | 9 | 0.873016 | 0.873016 | 0.873016 |
|  | 10 | 0.952381 | 0.952381 | 0.952381 |
|  | 11 | 0.952381 | 0.952381 | 0.952381 |
|  | 21 | 0.873016 | 0.873016 | 0.873016 |
|  | 28 | 0.936508 | 0.936508 | 0.936508 |
|  | 30 | 0.952381 | 0.952381 | 0.952381 |
|  | 33 | 0.904762 | 0.904762 | 0.904762 |
|  | 35 | 0.936507 | 0. 936507 | 0.936508 |
|  | 38 | 0.825397 | 0.825397 | 0.825397 |
| printtokens2 | 5 | 0.994652 | 0.994652 | 0.994652 |
|  | 6 | 0.983957 | 0.983957 | 0.994652 |
|  | 7 | 0.983957 | 0.983957 | 0.973262 |
|  | 10 | 0.967914 | 0.967914 | 0.994652 |
| Schedule | 1 | 0.972222 | 0.965278 | 0.986111 |
|  | 5 | \* | \* | 0.888888 |
|  | 6 | 0.972222 | 0.965278 | 0.972222 |
| schedule2 | 1 | 0.861446 | 0.849398 | 0.861445 |
|  | 3 | 0.945783 | 0.969879 | 0.981927 |
|  | 4 | 0.963855 | 0.951807 | 0.927710 |
|  | 5 | 0.993975 | 0.843373 | 0.951807 |
|  | 7 | 0.891566 | 0.963855 | 0.993975 |

Fault Localization Scores for the Siemens Suite

\*indicates the faulty line was not covered by the Pex-generated tests

The table above also shows results for fault localization using the value replacement technique. In the cases when value replacement failed altogether, the score used was the same as that of Tarantula to reflect the technique used by the original authors of the value replacement study. Thus in many cases value replacement performed no worse than Tarantula, but as one can see in the results, there were a number of situations where it proved to be significantly more effective than either Tarantula or Ochiai. The *Value Replacement* technique generally performed better when there were a large number of tests from which it could gather additional values to substitute. The *tcas* program was problematic for the VR algorithm because *Pex* generated very few tests and thus there were not enough successfully runs from which we could gather alternate values to attempt.

<describe additional VR result statistics>

If we consider the fact that these results omit all versions of the Siemens suite containing faults that are not exposed by Pex, then the actual performance of our solution is not nearly as good as the scores might imply. However, based on this data and the relative success that was attained for a subset of the Siemens suite, we can conclude that the combination of Pex with some fault localization technique is feasible, though some additional work would be required beyond basic test generation in order to produce enough tests to reliably uncover a more useful selection of faults.

### FunctionalDotNet

Pex offers the ability to add assumptions and high-level assertions to its parameterized unit tests, presenting the user with the ability to test for certain conditions that might not generate a runtime error but are nonetheless indicative of a problem with the code (such as output issues). While this goes beyond the basic test generation approach we used for analyzing the Siemens suite, the resulting unit tests are nonetheless generated by Pex and provide a degree of automation that allow for more efficient testing of an application. As a result, there are real-world applications and libraries that make use of Pex PUTs that are significantly more sophisticated than just what Pex generates by default.

We were interested in testing some of these applications’ Pex test suites in order to see if our results improved. One application we tested was FunctionalDotNet, a library for functional data structures that uses a test suite based on Pex. As this library was already passing all of its Pex-generated test cases, we seeded a fault and then ran the resulting test suite through our tool.

<FP results go here>

# Related Work

[Mike]

# Threats to Validity

*Porting Issues*. One of the major undertakings of our project was to port the Siemens Test Suite from SIR to *C#*. In porting the projects, we faced many issues trying to retain the logic and structure of the original C*.* The C code used many techniques that made it difficult to port to run using our tools, such as explicit calls to *exit()* which would bring down our test runner. In working around these difficulties it is likely that we introduced some inconsistencies into the performance of the ported versions. Finally working with these programs without always having a full understanding of their intended use made it difficult to maintain accuracy.

*Test Writing.* However, the main difficulty we faced beyond the time intensive work of porting was in writing appropriate *Pex* PUTs for the programs under test. The SIR unit test suite is incredibly extensive, with many programs having hundreds and even thousands of test cases that can be used to fully exercise the program. Even in the more complex programs such as *Replace*, where *Pex* generated a large number of test inputs, the generated tests were not nearly as extensive as the provided suite. Additionally, in writing the tests we had to come up with an appropriate amount of assumptions and assertions in order to guide the *Pex* explorations. This was particularly difficult as the programs often had complex input requirements and difficult to validate output. Someone with more expertise with the programs would probably be able to generate a more valuable test suite. Future work would include exploring projects actively writing their own *Pex* unit tests, in order to draw on that expertise. Initial work has been done in this direction with our preliminary analysis of the *FunctionalDotNet* project.

*Coverage Tools.* We created a large number of automated tools in order to aid in the generation and gathering of the data. We used the tools available with *Visual Studio* to automate the *mstest* test runner and the automatic building, running and parsing of the coverage data. This enabled us to quickly get data, but from a performance perspective was extremely limiting. The coverage data produced by the test tools didn’t distinguish which lines were run by which tests. So we had to individually run each test to properly gather the data; this lead to an extremely long data gathering process. Future work would include writing a custom coverage tool, which would greatly increase performance.

*Value Replacement Implementation.* Our VR implementation was very different from the implementation described in [[13](#4cfd96ebbfb79)]. Since we would instrument the assembly and then run all the value replacement runs from within the same process, this made our technique unable to handle faults that would immediately halt execution or faults that would corrupt memory. Additionally, if the program relies on static state and doesn’t properly clear it after each test then it could unpredictably influence later test runs. And finally, despite every effort to accurately create a *Value Replacement* tool equivalent to the one described in the original paper, due to the vastly different implementation techniques, we expect that there is some difference in performance. Future work would include: enabling value replacement to run in a sandboxed environment (to detect hard crashes and easily restore static state), expanding support of generic typed values (to enable handling modern .NET programs) and incorporating some of the technique enhancements introduced by the authors in later papers [[17](#4cfdc2a1d6d2e)].

# Conclusion

Our results have shown that the use of Pex for test generation can be paired with common fault localization techniques in order to detect and localize software faults in a highly automated fashion. When faults are detected using Pex-generated tests, Tarantula, Ochiai and value replacement all proved to be effective in localizing the causing fault when provided with the resulting test coverage data. However, the number of faults that went undetected indicates that this technique cannot be successful without some degree of user intervention. Specifically, it seems as though

<FP analysis? Promising or not?>

# ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

# REFERENCES

x

|  |  |
| --- | --- |
| 1 | Tillmann, N and De Halleux, J. Pex: white box test generation for. NET. ( 2008), 134--153. |
| 2 | Pacheco, C, Lahiri, S K, Ernst, M D, and Ball, T. Feedback-directed random test generation. ( 2007), 75--84. |
| 3 | Artzi, S, Dolby, J, Tip, F, and Pistoia, M. Practical fault localization for dynamic web applications. ( 2010), 265--274. |
| 4 | Agrawal, H, Horgan, J R, London, S, and Wong, W E. Fault localization using execution slices and dataflow tests. ( 2002), 143--151. |
| 5 | Jones, J A, Harrold, M J, and Stasko, J. Visualization of test information to assist fault localization. ( 2002), 477. |
| 6 | Jones, J A and Harrold, M J. Empirical evaluation of the tarantula automatic fault-localization technique. ( 2005), 273--282. |
| 7 | Abreu, R, Zoeteweij, P, and van Gemund, A J. An evaluation of similarity coefficients for software fault localization. ( 2006), 39--46. |
| 8 | Masri, W. Fault localization based on information flow coverage. *Software Testing, Verification and Reliability*, 20 (2010), 121--147. |
| 9 | Santelices, R, Jones, J A, Yu, Y, and Harrold, M J. Lightweight fault-localization using multiple coverage types. ( 2009), 56--66. |
| 10 | Baudry, B, Fleurey, F, and Le Traon, Y. Improving test suites for efficient fault localization. ( 2006), 82--91. |
| 11 | Cleve, H and Zeller, A. Locating causes of program failures. ( 2005), 342--351. |
| 12 | Zeller, A and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* (2002), 183--200. |
| 13 | Jeffrey, D, Gupta, N, and Gupta, R. Fault localization using value replacement. ( 2008), 167--178. |
| 14 | *http://sir.unl.edu*. |
| 15 | *http://www.mono-project.com/Cecil*. |
| 16 | *https://github.com/lukesandberg/PexFaultLocalization*. |
| 17 | Jeffrey, D, Gupta, N, and Gupta, R. Effective and efficient localization of multiple faults using value replacement (2009). |
| 18 | Schulz, M H and Auth, E. ESSENTIAL: An efficient self-learning test pattern generation algorithm for sequential circuits. ( 2002), 28--37. |
| 19 | Wong, W E and Debroy, V. Software Fault Localization. |
| 20 | Debroy, V, Wong, W E, Xu, X, and Choi, B. A Grouping-Based Strategy to Improve the Effectiveness of Fault Localization Techniques. ( 2010), 13--22. |
| 21 | Eric Wong, W, Debroy, V, and Choi, B. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83 (2010), 188--208. |
| 22 | Jones, J A, Bowring, J F, and Harrold, M J. Debugging in parallel. ( 2007), 16--26. |
| 23 | Agrawal, H and Horgan, J R. Dynamic program slicing. *ACM SIGPLAN Notices*, 25 (1990), 246--256. |

x