Reassessing the Impact of Dynamic Slicing in SFL

First Author[1], **Second Author**[2], **Third Author**[3]
[1] First Affiliation
[2] Second Affiliation
[3] Third Affiliation
first@email.address, second@email.address, third@email.address

Abstract

[Sofia:[@todo**]]**

1 Introduction

Software debugging is difficult. The task of locating the faulty code (i.e., fault localization) is particularly challenging. For that reason, countless automated techniques have been proposed in the past to reduce the cost of fault localization.

Dynamic Slicing (DS) and Spectrum-Based Fault Localization (SFL) are two very popular techniques that use different principles to automate debugging. Dynamic Slicing Agrawal and Horgan looks for the statements in code that influence the evaluation of the fault-revealing assertion. Although the technique has been intensively investigated in research, few use cases gained notoriety—WhyLine [Ko and Myers, 2008] being an exception. SFL [Wong *et al.*, 2016a], in contrast, does not try to recover the influence region of faulty statements. It is a black-box statistical fault localization method that computes suspiciousness values associated with program entities (e.g., basic blocks). For that, it uses coverage information of passing and failing test cases. SFL produces a list of program entities ranked in decreasing order of suspiciousness. Similarly to DS, SFL received tremendous attention over the years, but its applicability to support debugging remains questionable [Ang *et al.*, 2017; Pearson *et al.*, 2017; Xie *et al.*, 2016]. Despite the skepticism of the research community, SFL has been shown useful in supporting downstream analyses, such as Automated Program Repair (APR) [Monperrus, ; Kim *et al.*, 2017], an increasingly popular technique that looks for fixes to buggy statements. Tools like JAFF [Arcuri, 2011], Prophet [Long and Rinard, 2016], SemFix [Nguyen *et al.*, 2013], and SPR [Long and Rinard, 2015] use SFL to guide the search for likely fixes.

DS and SFL are complementary. Intuitively, DS identifies irrelevant parts of the code whereas SFL ranks the relevant parts of the code. This paper reports the results of a comprehensive study to evaluate the impact of DS to improve SFL. The study involves 395 faults from six programs from the Defects4J (D4J) dataset [Just *et al.*, 2014], which is frequently used to evaluated fault localization research. The intuition for the DS+SFL combination is that several highly ranked statements, albeit covered by failing executions, may be unrelated to the fault. Prior work reported promising results in this combination [Wotawa, 2010; Alves *et al.*, 2011; Hofer and Wotawa, 2012; Lei *et al.*, 2012; Guo *et al.*,]. We found surprising that, despite these findings, reported nearly 5 years ago, no tool or client analyses use this combination today. Several reasons could justify that observation. One hypothesis is that results reported in prior work are over-optimistic. For example, most prior work evaluated improvements of SFL techniques using relative metrics, which are based on the position of the first faulty statements found in the ranking relative to the total number of ranked statements, which is often a large number. As such, it inflates actual improvements and deceives potential adopters of the technology. Ang et al. [Ang *et al.*, 2017] recently pointed to that fact and encouraged researchers to adopt more precise metrics, such as top-k [Wu *et al.*, 2014; Lucia *et al.*, 2014b; Wen *et al.*, 2016], which has been widely adopted to evaluate performance of information retrieval algorithms **[[**cite**]]**. This metric reports the percentage of faults captured by a technique when the rank is trimmed to the first k components.

The goal of this paper is to reassess whether the DS+SFL combination is worthwhile under the light of a more accurate metric and a larger dataset. The D4J dataset is about twice as large than the biggest test suite used in previous studies. It is worth noting that APR techniques should directly benefit from these results—either to improve their results or to be aware they should not invest on this integration. This study covers the following research questions:

[(un)soundness] *How often does DS miss faulty statements?*

[effectiveness] *How effective is the DS+SFL combination?*

The first question addresses an important problem associated with dynamic slicing techniques—that of missing faulty statements. Naturally, the consequence of that problem for debugging is that developers would never be able to locate faults. Different slicers can miss faulty statements for different reasons. In this paper, we used Critical Slicing [DeMillo *et al.*, 1996], which, in principle, could miss statements because of imprecise oracles that guide the slicing process. We evaluated how often that important problem happens in our experiments. The second research questions effectively evaluates the impact of the combination. To sum, our results show that **[[**write a paragraph about this...**]]**

The contributions of this work are as follows:

- A comprehensive study on the combination of SFL and DS for bug localization of *Java* faulty programs.
- The tools used to run the study. Benchmarks and experimental results are also publicly available.

2 Background

This section presents important background material for the rest of the paper.

2.1 Critical Slicing

Program slicing is a program understanding method to identify the relevant parts of the program with respect to given points of interest. This paper focuses on dynamic slicing for its application in automated software debugging [Binkley *et al.*, 2014] where failing tests exist. We focused on Critical Slicing [DeMillo *et al.*, 1996] for its simplicity/generality. Critical Slicing prescribes a black-box language-semantics-agnostic recipe to computing executable slices. Critical Slicing minimizes the program by deleting statements such that the sliced program preserves critical observations (e.g., assertion violations). The simplification mechanism used in Critical Slicing is rather simple, consisting of mutating the program and observing the output.

We used the Mozilla Lithium tool [1] as the implementation for Critical Slicing. Lithium is conceptually similar to the DD-min Delta Debugging algorithm [Zeller and Hildebrandt, 2002], but operates on code instead of test inputs. It takes as input a file and produces as output a simplified version of that file that satisfies a user-defined oracle. For dynamic slicing, the oracle is defined such that the test produces the same failure manifestation. The Lithium minimization process starts by determining the initial size of chunks—in number of lines—to delete from the input file. For that, it chooses the highest power of two number smaller than the file size. For example, if the file has 1,000 lines, Lithium sets the initial chunk size to 512 lines. Then, the tool starts a local search looking for chunks to remove from the file that satisfy the oracle. When no more chunks of that given size can be removed, Lithium divides the chunk size by two and repeats the search. This iterative process continues until no more line can be removed. If n is the size of the input file and m is the size of the 1-minimal file found by Lithium, then Lithium usually performs $O(m \cdot \lg(n))$ iterations. Proofs can be found elsewhere[2].

2.2 Spectrum-based Fault Localization

Spectrum-based fault localization (SFL) is a statistical fault localization technique that takes as input a test suite including at least one failing test and reports on output a ranked list of components likely to be in fault [Wong *et al.*, 2016b; Jones and Harrold, 2005; Lucia *et al.*, 2014a; Abreu *et al.*, 2009]. The following are given in SFL: a finite set $\mathcal{C} = \{c_1, c_2, ..., c_M\}$ of M system *components*[3]; a finite set $\mathcal{T} = \{t_1, t_2, ..., t_N\}$ of N system transactions, which correspond to records of a system execution, such as test cases; the error vector $e = \{e_1, e_2, ..., e_N\}$, where $e_i = 1$ if transaction t_i has failed and $e_i = 0$ otherwise; and an $N \times M$ coverage matrix $\mathcal{A}$, where $\mathcal{A}_{ij}$ denotes the coverage of component c_j in transaction t_i. The pair $(\mathcal{A}, e)$ is commonly referred to as spectrum [Harrold *et al.*, 2000]. Figure 1 shows an example spectrum.

$\mathcal{T}$	c_1	c_2	$\cdots$	c_M	e
t_1	$\mathcal{A}_{11}$	$\mathcal{A}_{12}$	$\cdots$	$\mathcal{A}_{1M}$	e_1
t_2	$\mathcal{A}_{21}$	$\mathcal{A}_{22}$	$\cdots$	$\mathcal{A}_{2M}$	e_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
t_N	$\mathcal{A}_{N1}$	$\mathcal{A}_{N2}$	$\cdots$	$\mathcal{A}_{NM}$	e_N

Figure 1: An example spectrum.

Several types of spectra exist. The most commonly used is called hit-spectrum, where the coverage matrix is encoded in terms of binary *hit* (1) and *not hit* (0) flags, i.e., $\mathcal{A}_{ij} = 1$ if t_i covers c_j and $\mathcal{A}_{ij} = 0$ otherwise. The pair $(\mathcal{A}, e)$ serves as input to the fault localization technique. With this input, the next step in this coverage-based technique consists of determining what columns of the matrix A resemble the error vector e the most. For that, an intermediate component frequency aggregator $n_{pq}(j)$ is computed $n_{pq}(j) = |\{i \mid \mathcal{A}_{ij} = p \wedge e_i = q\}|$. $n_{pq}(j)$ denotes the number of runs in which the component j has been active during execution ($p = 1$) or not ($p = 0$), and in which the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved ($p = 1$) in failing executions ($q = 1$), whereas $n_{10}(j)$ counts the number of times component j has been involved in passing executions. We then calculate similarity to the error vector by means of applying *fault predictors* to each component to produce a score quantifying how likely it is to be faulty. Components are then ranked according to such likelihood scores and reported to the user. Ochiai is one of those fault predictors that has shown to perform well [Pearson *et al.*, 2017; Wong *et al.*, 2016a]. The Ochiai formula [Rui Abreu and van Gemund, 2006] is given by

$$n_{11}(j)/(\sqrt{n_{11}(j) + n_{01}(j)} + \sqrt{n_{11}(j) + n_{10}(j)})$$

3 Critical Slicing + SFL

The combinination of critical slicing and SFL consists of the following steps: (1) Compute the execution spectra as discussed in Section 2.2, (2) Compute slices associated with failing tests, (3) Adjust the spectra computed in Step 1 with results obtained in Step 2, and (4) Compute suspiciousness scores. Figure 2 illustrates the update of the spectra and suspiciousness rankings as result of slicing the program against the failing test t_2. The faulty statement in this example is the component c_2. The slice obtained from t_2 is $\{c_2, c_5\}$. Intuitively, slicing enables identification of cells in the matrix (/spectra) whose values can be set to zero.

Theorem 1 *The faulty statement may not be included in the critical slice of failing test cases.*

[1] Lithium is available at `https://github.com/MozillaSecurity/lithium` (accessed November 2018).

[2] Algorithm complexity available at `https://github.com/MozillaSecurity/lithium/blob/master/src/lithium/docs/algorithm.md` (accessed November 2018).

[3] A component can be any source code artifact of arbitrary granularity such as a class, a method, a statement, or a branch [Harrold *et al.*, 2000].

$\mathcal{T}$	c_1	c_2	c_3	c_4	c_5	e		$\mathcal{T}$	c_1	c_2	c_3	c_4	c_5	e
t_1	1	0	1	1	0	✓		t_1	1	0	1	1	0	✓
t_2	0	1	1	1	1	✗		t_2	0	1	0	0	1	✗
t_3	1	0	1	0	0	✗	⇒	t_3	1	0	1	0	0	✗
t_4	0	1	0	0	1	✓		t_4	0	1	0	0	1	✓
t_5	1	0	0	1	1	✓		t_5	1	0	0	1	1	✓

(a) Spectra update.

1	c_4 (0.59)		1	c_2 (0.35)
2	c_3 (0.55)		2.5	c_1 (0.32)
3	c_2 (0.35)	⇒	2.5	c_5 (0.32)
4.5	c_1 (0.32)		4	c_3 (0.29)
4.5	c_5 (0.32)		5	c_3 (0.24)

(b) Ranking update.

Figure 2: Modifications on spectra and ranking as result of slicing code against test t_2.

It is possible that a slice that discards the faulty statement still produces the expected message checked by the oracle. This can happen, for example, if the oracle is too general.

[Sofia:[@todo: Add a short 3-4 lines at most example based on one of our subjects**]]** Section 3.1 explains the heuristics we used to infer oracles for critical slicing that mitigates this issue and Section 4 shows how often critical slicing misses faulty statements.

Theorem 2 *The rank of faulty statements cannot decrease if the slice includes the faulty statements.*

Considering the Ochiai formula, for those statements j which are not part of the slice of a failing test, the combination reduces the value of $n_{11}(j)$ and increases the value of $n_{01}(j)$. Therefore, suspiciosness of those statements decrease. Similar argument applies to other fault predictors.

In the example from Figure 2, components c_3 and c_4, which are not in the slice of t_2, have their suspiciousness reduced, enabling the faulty component c_2 to rise from the third to the first position in the ranking.

3.1 Implementation

[Marcelo:[Sofia, I think this was way too low-level for intro and decided to move here. please check how to fit.**]]** **[[**A couple of different tools were designed to perform this empirical study: *morpho* and *lithium-slicer*. *morpho* retrieves as an output the input for *lithium-slicer*. *lithium-slicer* uses the input to reduce the search domain for SFL and outputs the statements that resulted from the minimization. *morpho* uses this to update the spectrum matrix, performs the before and after SFL evaluation using the corresponding matrix, and outputs the metrics that report the SFL performance for both cases — before and after using DS. **]]**

Two different tools were developed to support this research: *morpho* and *lithium-slicer*. *morpho* was designed to calculate the suspiciousness of all statements of a project before, and after the top-k minimization performed by *lithium-slicer* whereas *lithium-slicer* is responsible for reducing the search domain of each `Java` class of the project for the after-SFL analysis. *morpho* uses the spectrum matrix (Figure 1), and the pair *name#location* for each statement to calculate the respective ranking. All rankings are ordered from highest to lowest ranked. This information is retrieved along with each test case stack trace in a `.json` file which serves as an output to *lithium-slicer*. *lithium-slicer* starts by generating the inputs (Algorithm 1, line 2) for the top-k classes of each failing test based, mainly, on the output of *morpho* — ranked list of statements and the test cases stacktraces. Then, the tool iterates each class from the top-k classes (c) of each test t. In each iteration, classes are refined using an external java program (Algorithm 1, line 6) that substitutes all the line comments ($\backslash\backslash$), block comments ($\backslash*$ to $*\backslash$) and javadoc comments ($/**$ to $*/$) using the `JavaParser`[4] library. This step was added to *lithium-slicer* because it turns `MozillaLithium` faster since the empty lines are ignored. Then, `MozillaLithium` performs the class minimization using a function of interest (Algorithm 2) which compares the output of the test with the expected one which is given as an input. Finally, the location of all relevant statements (Algorithm 1, line 9) is saved in a `.json` which is used for the before-SFL analysis. *morpho* uses the output from *lithium-slicer* to create a copy of the older spectrum matrix and updates it according to the explanation provided on Figure 2a where the statements that are not in the slice of the test suffer a suspiciouness reduction. In the end, *morpho* performs the SFL analysis for both matrixes and calculates the probability of the first line being faulty, the probability of the last line being faulty, and the mean and median of the position of the faulty line in the ranking. These are the metrics used to evaluate how considerable is the improvement obtained when combining DS with SFL.

Algorithm 1 Class Minimization Algorithm

```
Input: $proj$ - project name
       $bug$ - bug number
       $k$ - number of top ranked classes
       $stk$ - expected stacktrace
Output: Top-k classes minimization for each test
 1: procedure LITHIUM-SLICER($proj$, $bug$, $k$, $stk$)
 2:     $testsInfo \leftarrow$ generateInputs($proj$, $bug$, $k$, $stk$)
 3:     for all $t \in testsInfo$ do
 4:         $classes \leftarrow$ getClasses($proj$, $bug$)
 5:         for all $c \in classes$ do
 6:             $unc \leftarrow$ removeComments($c$)
 7:             $min \leftarrow$ MozillaLithium($iFunc$, $unc$, $t$, $stk$)
 8:         end for
 9:         $slicer \leftarrow$ getLocation($c$, $t$, $min$)
10:     end for
11:     return $slicer$
12: end procedure
```

3.2 Function of Interest

The `MozillaLithium` tool takes as input a function of interest that can be designed according to the problem to be solved. The function of interest (Algorithm 2) determines if

[4] JavaParser is available at http://javaparser.org/ (accessed November 2018).

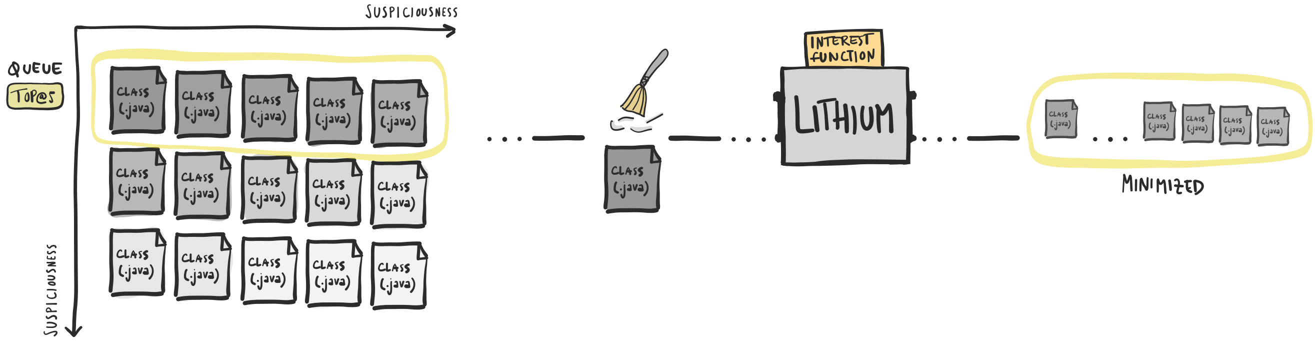

Figure 3: Simple illustration of the process to obtain the top-5 minimized classes from a project carrying a bug

Algorithm 2 Function of interest (iFunc)

Input: c - class to be minimized
t - testcase name
$expStk$ - expected stacktrace
Output: $interest$ - $True$ if it is interesting, $False$ if not

```
1: function IFUNC($c$, $t$, $expStk$)
2:     $testStk \leftarrow$ runTest($t$)
3:     $testOracle \leftarrow$ getOracleToCompare($testStk$)
4:     $expOracle \leftarrow$ getOracleToCompare($expStk$)
5:     return $testOracle == expOracle$
6: end function
```

the test case output is interesting or not. In this study, interesting means that the test execution path is equal to the expected one. This information can be deducted from `Java` stack traces which show the stack of functions called until the thrown of an exception. The authors assume that it is only necessary to compare both stack traces until the call where the test fails. For each chunk selected by `MozillaLithium`, the test is executed without considering the chunk under evaluation (Algorithm 2, line 2). Both test ($testStk$) and expected ($expStk$) stack traces are filtered according to the position of the test call fails. These operations are performed by line 3 and 4 from Algorithm 2. The result is an oracle such as the one provided in the box below:

```
java.lang.IndexOutOfBoundsException: Index: -1, Size: 1,
java.util.ArrayList.rangeCheckForAdd(ArrayList.java:643),
java.util.ArrayList.add(ArrayList.java:455),
org.jfree.data.xy.XYSeries.addOrUpdate(XYSeries.java:564),
org.jfree.data.xy.XYSeries.addOrUpdate(XYSeries.java:527),
org.jfree.data.xy.junit.XYSeriesTests.
testBug1955483(XYSeriesTests.java:479),
...
```

The oracle includes an exception error message (*java.lang.IndexOutOfBoundsException: Index: -1, Size: 1*) and the program execution path until the class test where it fails (*XYSeriesTests.java:479*). The regular expression `Test(.*).java` retrieves the test class position in the stack trace. If the test class is not found, the entire stack trace is compared. There is, however, an exception. When the failure is a *StackOverflowError*, there is no test class mentioned in the stack trace because the exception is a runtime error. This exception is triggered when the JVM encounters a situation where there is no space for a new stack frame to be created which is, normally, the outcome of infinite recursion. Consequently, the stack trace contains many times the failed call. So, when the failure message contains a *StackOverflowError*, the oracle is only considered until the call is repeated for the fifth time.

```
java.lang.StackOverflowError,
sun.reflect.generics.reflectiveObjects.
TypeVariableImpl.hashCode(TypeVariableImpl.java:198),
java.util.HashMap.hash(HashMap.java:362),
java.util.HashMap.getEntry(HashMap.java:462),
java.util.HashMap.get(HashMap.java:417),
org.mockito.internal.util.reflection.GenericMetadataSupport.
getActualTypeArgumentFor(GenericMetadataSupport.java:182),
org.mockito.internal.util.reflection.GenericMetadataSupport.
getActualTypeArgumentFor(GenericMetadataSupport.java:185),
org.mockito.internal.util.reflection.GenericMetadataSupport.
getActualTypeArgumentFor(GenericMetadataSupport.java:185),
...
```

Some of the error messages can be different even if the test is executed with the same inputs and in the same environment. One example is the one provided in the box below, where the memory positions (*class@<memory-position>*) may differ from the ones in the expected message. In order to solve this, it was designed a mechanism to check this kind of exception patterns. However, the current version of the interesting function only considers this pattern. Other patterns will be added soon into the next version of *lithium-slicer*.

```
junit.framework.AssertionFailedError:
expected:<org.jfree.chart.util.ShapeList@d0e55451> but
was:<org.jfree.chart.util.ShapeList@d951b68f> ...
```

4 Evaluation

[Marcelo:[———————— estou aqui**]]**

This section reports results of our study to assess the impact of the DS+SFL combination.

4.1 Objects of Analysis

Following other recent studies, we used the Defects4J benchmark in our experiments [Just *et al.*, 2014]. The Defects4J benchmark includes six subjects and 395 faults. Apache commons-lang is a library that provides a set of helper utilities for the `java.lang` API. Apache commons-math is a lightweight library of self-contained mathematics and statistics components. The Closure compiler is a toolset for turning JavaScript files into smaller scripts for faster download and execution in the browser. JFreechart is a library with a full-featured charts user interface for Java. Joda-Time is a lightweight library that aims to replace the default Java `java.util.Date` classes providing simpler APIs. Mockito is a mocking testing framework.

Project	Size (kLOC)	# Tests	# Defects
Apache commons-lang	111751	6057	65
Apache commons-math	306276	26797	106
Closure compiler	149521	27930	133
JFreechart	230159	8458	26
Joda-Time	141610	3289	27
Mockito	22787	8835	38

Table 1: Characterization of Defects4J subjects.

4.2 Experiments

These experiments only consider 213 of 395 bugs of the Defects4J. Each bug has one or more failing tests associated. For each test, it is verified if all the faulty statements of the bug are in the slicer. #Ev is the total number of verifications performed by bug. For each bug (i) of the project, this experiment checks if all the faulty statements of the bug (FS_i) are in the slicer retrieved by each test (T_i). n is the total number of bugs considered for each project.

$$\#Ev = \sum_{i=1}^{n} T_i * FS_i$$

[Sofia:[It does not make much sense to report #Bugs here. If we end up not using the all dataset I think it would be better to report this in another section.**]]** To evaluate the performance of DS alone, the following metrics were taken into consideration: the total number of faulty statements ($\#FS$) in each project; the total number of tests ($\#T$) in each project; the number of times that each FS may be in the slicer of each T, $\#Ev$; the number of bugs considered per project to the experiments, $\#Bugs$; the percentage of FS that are not in the slicers of T, $\% \overline{in}$; the percentage of FS that are not found in the top-5 of the classes with the highest ranks of suspiciousness; and, the percentage of FS that are in the slicers of T. Regarding the SFL+DS combination, performance is evaluated based on four different metrics: the probability of the first line being the faulty one, $\%imp(f)$; the probability of the last line being the faulty one, $\%imp(l)$; and, finally the mean, $\%imp(\overline{m})$, and median $\%imp(median)$ of the position of the faulty line in the ranking. This combination is evaluated considering all the classes of the project (all) and only the classes of the project involved in the failing tests execution (loaded).

4.3 Results of Dynamic Slicing

We describe the results of dynamic slicing alone in the following sections.

How often DS misses faulty statements?

How effective is the SFL+DS combination for bug localization?

4.4 Threats to validity

5 Related Work

5.1 Program Slicing

Program Slicing is an old technique with several applications in PL and SE research [Weiser, 1981]. Slicing can be implemented in different ways. Dynamic Slicing has found its main application in fault localization [Agrawal and Horgan, 1990]—in this application context, failing tests exist. Research in this area has mainly focused on slicing code efficiently [Wang and Roychoudhury, 2008; Wang and Roychoudhury, 2004] and avoiding data and control omission errors [Zhang *et al.*, 2007; Lin *et al.*, 2018]. Omission errors correspond to the cases where tests fail because some part of the code was not executed when it should. Dynamic fault localization techniques cannot hope to soundly report bugs in those cases. The issue is that incorporating static information to capture those cases can result in unacceptably large slices, which defeats the purpose of reducing the fault search space. It is worth noting that, conceptually, the DS+SFL combination can be used with any form of dynamic slicing. We chose an implementation of Critical Slicing [DeMillo *et al.*, 1996] for its generality.

5.2 SFL

Statistics-based techniques (e.g., [Pearson *et al.*, 2017]) are popular automated fault localization techniques within the Software Engineering community. They correlate information about program fragments that have been exercised in multiple program execution traces (also called *program spectra*) with information about successful and failing executions. By doing that, statistics-based approaches yield a list of suspect program fragments sorted by their likelihood to be at fault. Since this technique is efficient in practice, it is attractive for large modern software systems [Zoeteweij *et al.*, 2007].

Spectrum-based fault localization (SFL) is amongst the most common statistical fault localization technique that takes as input a test suite including at least one failing test and reports on output a ranked list of components likely to be in fault [Wong *et al.*, 2016b; Jones and Harrold, 2005; Lucia *et al.*, 2014a; Abreu *et al.*, 2009].

5.3 DS+SFL

Prior work has investigated the combination of dynamic slicing and spectrum-based fault localization [Wotawa, 2010; Alves *et al.*, 2011; Hofer and Wotawa, 2012; Lei *et al.*, 2012; Guo *et al.*,]. Although the methodology used in these papers vary, the overall message is that the combination is valuable.

Project	% top-5	% $\overline{in}$	% in
JFreechart	0%	13.2%	86.8%
Closure compiler	3.7%	0%	96.3%
Apache commons-lang	0%	0%	100%
Apache commons-math	2.1%	8%	89.9%
Mockito	3.5%	3.3%	93.2%
Joda-Time	0%	1.2%	98.8%

Table 2: Report of the capability of DS on missing faulty statements

Project	all				loaded			
	% imp (f)	% imp (l)	% imp ($\overline{m}$)	% imp (median)	% imp (f)	% imp (l)	% imp ($\overline{m}$)	% imp (median)
JFreechart	20.98%	20.32%	20.05%	19.75%	20.98%	20.32%	20.05%	19.75%
Closure compiler	4.17%	0.00%	0.00%	3.68%	4.17%	0.00%	0.00%	3.68%
Apache commons-lang	7.91%	5.84%	6.47%	6.80%	7.91%	5.84%	6.47%	6.80%
Apache commons-math	8.58%	6.59%	7.65%	8.74%	8.58%	6.59%	7.65%	8.74%
Mockito	15.73%	17.37%	17.63%	15.74%	15.73%	17.37%	17.63%	15.74%
Joda-Time	8.83%	10.70%	11.25%	10.86%	8.83%	10.70%	11.25%	10.86%

Table 3: SFL+DS performance evaluation

Note that slicing alone does not provide the guarantee of improvement to SFL as statements discarded with slicing could have been ranked lower compared to faulty statements. This paper differs from prior work in that it uses a much larger dataset of programs and faults and a more rigorous experimental methodology.

Applications

Ishii and Kutsuna [Li *et al.*, 2013] proposed a technique involving dynamic slicing and SFL applied to MATLAB/Simulink models. The technique consists of iteratively generating tests that cause an error using satisfiability modulo theories (SMT) so that the slices generated are distinct from each other. For slicing, their technique uses a program dependence graph (PDG) that express both data and control dependencies between blocks, so that the slice is built only on blocks that affect a specific line in the model - *effective blocks*. Their goal with the slicing was (1) to keep suspiciousness of a fault maximum if a target model has only one fault, and (2), to reduce the number of fault candidates obtained by the slicer. For this, they utilize only failed test cases generated with SMT solvers [Peleska *et al.*, 2011] to calculate suspiciousness, as passed test cases might lower the suspiciousness of a fault. The calculated suspiciousness is the number of times that a block belongs to the *effective blocks* group by the number of total failed test cases. Although their experiments and results showed promise with small models, the accuracy of the technique decreases when applied to models with multiple faults instead of one, as does the efficiency when exposed to large models, number of generated tests, sizes of sliced models and so on.

Automatic Program Repair

Automated repair techiniques have received considerable attention on the last decade and consists of three basic steps: fault localization, patch generation, and patch validation. Regarding the fault localization step, Automated Program Repair (APR) tools can be guided by statistical fault localization techiniques to modify code which most likely contains the fault. On this section, we will briefly describe some of these tools. **[Davino:**[Not necessarily on the paper and not necessarily on a related work section. To be revised/moved/sliced later. **]]**

GenProg[[cite]] is a state of the art tool based on genetic programming to repair defects in off-the-shelf C programs. GenProg's strategy to fault localization consists of instrumenting the program so that all lines visited when executing failing test cases are recorded and used on the genetic search algorithm. The algorithm involves serching for other parts of the program, considering that a program that contains an error in one area likely implements the correct behaviour elsewhere. The tool proved to be effective in reparing 16 programs involving 1.25 millions LOC, and efficient as the algorithm search space takes advantage of test case coverage information, reusing existing program statements.

Debroy and Wong[[cite]] presented a techinique for automatically fixing faults in a program combining both mutation and statistical fault localization. Using the Tarantula fault localizer[[cite]], the techinique focus the top ranked statements in terms of suspiciousness for the mutation operations. The percentage of statements examined in search for the fault is based on the total number of statements of the program, with the 10 percent being a reasonable number on the subject programs studied.

JAFF[[cite]] is a Java automatic repair tool based on evolutionary principles that uses fault localization for ranking the statements in the code in terms of suspiciousness level. Before searching for the fix, the Java program is translated into a syntax tree composed of nodes related to the top ranked statements being used on the search process. The tool uses a specific formula to get the number of nodes that will be used on the search based on the number of statements with the same rank of suspiciousness, with the range between 1 to 20 nodes being used on the cases studied.

SemFix is based on symbolic execution, constraint solving and program synthesis applied to software automatic repair. The tool uses statistical fault localization (SFL) with Tarantula to obtain the rank of suspiciousness and iteratively takes the most suspicious statement that has not been tried out to be used on the search for a fix. Compared to GenProg, the presented tool took consideredably less time to find a fix for programs utilizing the same test suites. However, the authors mention that the choice of the SFL techinique could potentially affect the effectiviness of the tool, particularly with large programs.

SPR[[cite]] **[Davino:**[Fan Long]] is based on staged program repair with condition synthesis. SPR uses fault localization to identify target statements that will be transformed in search for a fix on the program. The search space is built with a list of statements ranked by suspiciousness level, where only the first 200 ranked statements are considered. The authors observed that increasing the search space to the top 2000 statements would bring additional repairs, but with a obvious cost of searching a large space. **[Davino:**[Another tool presented by the same authors is Prophet, which uses the same search space, top 200.]]

MintHint [[cite]] uses statistical analysis with SFL to rank statements by their likelihood of being faulty. Given the ranked list, the tool analysis the top *k* statements, where *k* is a threshold set by the user, and proceeds with the repair algorithm assuming the fault can be repaired by changing a single statement.

[Davino:[Another set of automatic program repair tools that doesn't use fault localization includes: CodePhage, ClearView, PAR, RSRepair, Kali, and AutoFix, among others.]]

6 Conclusions and Future Work

Acknowledgments

References

[Abreu *et al.*, 2009] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[Agrawal and Horgan, 1990] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 246–256, New York, NY, USA, 1990. ACM.

[Alves *et al.*, 2011] Elton Alves, Milos Gligoric, Vilas Jagannath, and Marcelo d'Amorim. Fault-localization using dynamic slicing and change impact analysis. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 520–523, Washington, DC, USA, 2011. IEEE Computer Society.

[Ang *et al.*, 2017] Aaron Ang, Alexandre Perez, Arie van Deursen, and Rui Abreu. Revisiting the practical use of automated software fault localization techniques. In *2017 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, volume 00, pages 175–182, Oct. 2017.

[Arcuri, 2011] Andrea Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494 – 3514, 2011.

[Binkley *et al.*, 2014] David Binkley, Nicolas Gold, Mark Harman, Syed Islam, Jens Krinke, and Shin Yoo. Orbs: Language-independent program slicing. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 109–120, New York, NY, USA, 2014. ACM.

[DeMillo *et al.*, 1996] Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 121–134, New York, NY, USA, 1996. ACM.

[Guo *et al.*,] Anbang Guo, Xiaoguang Mao, Deheng Yang, and Shangwen Wang. An empirical study on the effect of dynamic slicing on automated program repair efficiency. To Appear in ICSME 2018 (short paper).

[Harrold *et al.*, 2000] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[Hofer and Wotawa, 2012] Birgit Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *ECAI 2012 - 20th European Conference on Artificial Intelligence. Including Prestigious Applications of Artificial Intelligence (PAIS-2012) System Demonstrations Track, Montpellier, France, August 27-31 , 2012*, pages 420–425, 2012.

[Jones and Harrold, 2005] James A. Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 273–282, 2005.

[Just *et al.*, 2014] René Just, Darioush Jalali, and Michael D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 437–440, New York, NY, USA, 2014. ACM.

[Kim *et al.*, 2017] Sunghun Kim, Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair (dagstuhl seminar 17022). *Dagstuhl Reports*, 7(1):19–31, 2017.

[Ko and Myers, 2008] Andrew J. Ko and Brad A. Myers. Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 301–310, New York, NY, USA, 2008. ACM.

[Lei *et al.*, 2012] Y. Lei, X. Mao, Z. Dai, and C. Wang. Effective statistical fault localization using program slices. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 1–10, July 2012.

[Li *et al.*, 2013] F. Li, W. Huo, C. Chen, L. Zhong, X. Feng, and Z. Li. Effective fault localization based on minimum debugging frontier set. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 1–10, Feb 2013.

[Lin *et al.*, 2018] Yun Lin, Jun Sun, Lyly Tran, Guangdong Bai, Haijun Wang, and Jinsong Dong. Break the dead end of dynamic slicing: Localizing data and control omission bug. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 509–519, New York, NY, USA, 2018. ACM.

[Long and Rinard, 2015] Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015. ACM.

[Long and Rinard, 2016] Fan Long and Martin Rinard. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 298–312, New York, NY, USA, 2016. ACM.

[Lucia *et al.*, 2014a] Lucia, David Lo, Lingxiao Jiang, Ferdian Thung, and Aditya Budi. Extended comprehensive study of association measures for fault localization. *Journal of Software: Evolution and Process*, 26(2):172–219, 2014.

[Lucia *et al.*, 2014b] Lucia, David Lo, and Xin Xia. Fusion fault localizers. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 127–138, New York, NY, USA, 2014. ACM.

[Monperrus,] Martin Monperrus. Automatic software repair: a bibliography. Accepted for publications in ACM Computing Survey. `http://www.monperrus.net/martin/survey-automatic-repair.pdf`.

[Nguyen *et al.*, 2013] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 772–781, May 2013.

[Pearson *et al.*, 2017] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE'17, pages 609–620, Piscataway, NJ, USA, 2017. IEEE Press.

[Peleska *et al.*, 2011] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 298–312, Berlin, Heidelberg, 2011. Springer-Verlag.

[Rui Abreu and van Gemund, 2006] Peter Zoeteweij Rui Abreu and Arjan J. C. van Gemund. An evaluation of similarity coefficients for software fault localization. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, pages 39–46, 2006.

[Wang and Roychoudhury, 2004] Tao Wang and Abhik Roychoudhury. Using compressed bytecode traces for slicing java programs. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 512–521, Washington, DC, USA, 2004. IEEE Computer Society.

[Wang and Roychoudhury, 2008] Tao Wang and Abhik Roychoudhury. Dynamic slicing on java bytecode traces. *ACM Trans. Program. Lang. Syst.*, 30(2):10:1–10:49, March 2008.

[Weiser, 1981] Mark Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[Wen *et al.*, 2016] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 262–273, New York, NY, USA, 2016. ACM.

[Wong *et al.*, 2016a] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[Wong *et al.*, 2016b] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey of software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[Wotawa, 2010] Franz Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *Proceedings of the 2010 10th International Conference on Quality Software*, QSIC '10, pages 161–170, Washington, DC, USA, 2010. IEEE Computer Society.

[Wu *et al.*, 2014] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. Crashlocator: Locating crashing faults based on crash stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 204–214, New York, NY, USA, 2014. ACM.

[Xie *et al.*, 2016] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 808–819, New York, NY, USA, 2016. ACM.

[Zeller and Hildebrandt, 2002] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002.

[Zhang *et al.*, 2007] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta. Towards locating execution omission errors. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 415–424, New York, NY, USA, 2007. ACM.

[Zoeteweij *et al.*, 2007] Peter Zoeteweij, Rui Abreu, Rob Golsteijn, and Arjan J. C. van Gemund. Diagnosis of embedded software using program spectra. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS '07, pages 213–220, Washington, DC, USA, 2007. IEEE Computer Society.