Combining Dynamic Slicing and Spectrum-Based Fault Localization – A First Experimental Evaluation

Jonas Schleich
Graz University of Technology, Austria

Franz Wotawa[1,2]
Institute of Software Engineering and Artificial Intelligence, Graz University of Technology, Austria

Abstract

Identifying and localizing bugs in programs has always been considered a complex but essential topic. Whereas the former has led to substantial progress in areas like formal verification and testing with a high degree of automation, the latter has not been satisfactorily automated. Approaches like program slicing, model-based diagnosis, and, more recently, spectrum-based fault localization can be used to find possible causes of a misbehaving program automatically, but often come with high computational complexity or a larger list of diagnoses, which require additional manual effort. In this paper, we present the first experimental results of an approach that combines program slicing with spectrum-based fault localization aiming at improving the outcome of automated debugging methods. In contrast to previous work, where we illustrated potential improvements only by considering a particular use case, we present an evaluation based on 22 different example programs in this paper. The approach improves the wasted effort on average by around 5 to 15% on average.

2012 ACM Subject Classification Software and its engineering → Software testing and debugging; Computing methodologies → Causal reasoning and diagnostics

Keywords and phrases Software fault localization, program slicing, spectrum-based fault localization, automated debugging

Digital Object Identifier 10.4230/OASIcs.DX.2025.3

Supplementary Material *Software (Source Code)*: `https://github.com/SchleichJonas/JSR` archived at `swh:1:dir:062e5cd4325d9b47fafadb80f521129b36a5c8c6`

Funding *Franz Wotawa*: The work was supported by the Austrian Science Fund (FWF) Cluster of Excellence Bilateral AI under contract number 10.55776/COE12.

1 Introduction

Debugging comprises detecting, localization, and repairing faults in programs, which still is mainly carried out manually, causing a lot of effort. There are several approaches supporting the automation of fault detection in use, but almost none for fault localization and repair. However, automating debugging has been of interest for more than four decades, e.g., see Ehud Shapiro [21] or Mark Weiser [25, 26]. The latter investigated how programmers perform debugging utilizing program slices, which are calculated only considering the source code. Korel and Laski [16] introduced dynamic slicing, where program executions are used to eliminate part of the source code not involved in current executions. However, despite the interest in slicing from academia, its use in practice is limited, leading to other debugging approaches like spectrum-based fault localization or model-based debugging [6, 10, 31].

[1] Corresponding author
[2] Authors are listed in alphabetical order.

Regarding spectrum-based fault localization (SFL), Jones and Harrold [13] introduced the basic concepts in the corresponding tool Tarantula. In contrast to slicing, which utilizes program dependencies, SFL uses a probability-based approach considering different execution runs. In particular, the idea is to assign a suspicious value to each statement (or other part of a program), which can be motivated as follows. Any statement that is not executed in failing runs is very unlikely to be faulty. A statement that is only executed in failing runs is likely faulty. All other statements executed in passing or failing runs might be faulty. A suspicious value for each statement can be computed considering the execution of statements.

It is worth noting that SFL has gained much attention in the debugging community. Wong and colleagues [28] provided a survey showing that SFL holds the largest share of publications of about 30%, followed by slicing-based debugging approaches. However, there is only a little work combining slicing and SFL for improving debugging, i.e., Wen et al. [27], Hofer and Wotawa [11], Reis et al. [18], Soha [22], and more recently Wotawa [30]. In this paper, we mainly focus on the last cited publication and provide an initial experimental evaluation of the impact of the combined approach on the quality of the debugging results. In his original paper, Wotawa [30] only discussed the potential impact but did not provide any experimental evidence of the superiority of the combined slicing and SFL approach. It is further worth noting that combining approach is one possibility to bring different approaches together, which often leads to a better performance, e.g., see Mukhtar et al. [17]. The other way is to show that one approach can be subsumed by the other. For example, Wotawa [29] showed that the results of static slicing can be achieved using an abstract model of program statements considering only dependencies between variables and statements.

Hence, in this paper, we contribute to automated software debugging as follows. We present a first experimental evaluation that considers a combined debugging approach utilizing SFL and dynamic slicing. The experimental evaluation was carried out considering available tools for Java programs. The research objective of the study was to clarify whether the combined debugging approach improves debugging or not. The question is important because adding slicing to debugging comes with a substantial computational overhead compared to SFL. Hence, without substantial improvements, a real benefit of the combined method may not be arguable. In addition to the study, we also discuss challenges and issues we experienced when carrying out the experimental evaluation. In particular, there is a huge influence of tools on the outcome, which cannot be neglected and where only partial mitigation is possible. Note that the provided experimental evaluation cannot be considered an exhaustive one incorporating a vast number of faulty versions of different programs. It is the first study to clarify whether conducting additional experimental evaluation is worthwhile.

We organize the paper as follows: We first introduce the foundations behind SFL, dynamic slicing, and the combined approach. Afterward, we describe the experimental setup, followed by a detailed presentation of the results. We discuss and summarize the obtained results and finally conclude this paper.

2 Foundations

In this section, we discuss the foundations behind SFL and its variant that utilizes dynamic slicing. To illustrate definitions and underlying ideas, we make use of an illustrative example program. In Figure 1, we have a simple Java program `Car.warn` that computes the braking distance using the current velocity, the friction coefficient `mu`, and the physical equation given in Line 12. This braking distance is used together with a given distance to a vehicle in front of a car to raise a warning in the case where we cannot stop within 80% of the provided

```
1. public class Car {
2.     public static double mu = 1.10758097;
3.     public static double constG = 9.80665;
4.     public static void warn(
5.           double velocity, double distance, boolean raining) {
6.       double braking_distance;
7.       double current_mu = mu;
8.       boolean warning;
9.       if (raining) {
10.          current_mu = 0.8*current_mu;
11.      }
12.      braking_distance = (velocity*velocity)/(2*current_mu*constG);
13.      if (braking_distance > 0.8*distance) {
14.          warning = true;
15.      } else {
16.          warning = false;
17.      }
18.      store(warning,braking_distance);
19.    }
20. }
```

Figure 1 An illustrative example program computing an expected braking distance and comparing it with a distance to a vehicle in front for raising alarm messages.

Table 1 A test suite for program `Car.warn`.

test case	velocity	distance	raining	warning	braking_distance
T_1	15	10	false	true	10.357533856871605
T_2	30	64	false	false	41.430135427486420
T_3	30	64	true	true	51.787669284358020
T_4	15	64	true	false	12.946917321089504

distance. The 80% is a safety margin. In addition, a situation where we have rain influencing the friction and, therefore, the braking distance is considered. In the case of rain, the friction coefficient is set to 80% of its original value.

To test `Car.warn`, we need test cases. A test case comprises given input values and the resulting expected output values. In our case, we consider the warning and the calculated braking distance as output. Table 1 comprises 4 test cases for `Car.warn`. Via executing `Car.warn`, we are able to confirm the correct behavior of the program. Let us now assume that we have a bug in the program. Instead of Line 10, we have `current_mu = 0.85*current_mu`. We refer to this faulty variant as `Car.warn'`. Executing `Car.warn'` on the same test suite, however, leads to a different outcome, which we depict in Table 2. Obviously, the values of the braking distance are wrong whenever it is raining, but only once the warning is false instead of true.

After identifying deviations from expectations, we are interested in identifying the root cause behind them. This is similar to ordinary model-based diagnosis [19, 7] but considers a program instead of a system comprising interacting (hardware) components.

Table 2 Running the test suite from Table 1 on program `Car.warn'`. Values in **bold** face indicate differences to expectations.

test case	velocity	distance	raining	warning	braking_distance
T_1	15	10	false	true	10.357533856871605
T_2	30	64	false	false	41.430135427486420
T_3	30	64	true	**false**	**48.74133579704284**
T_4	15	64	true	false	**12.18533394926071**

Table 3 The program spectrum of `Car.warn'` considering the 4 test cases.

Statement	T_1	T_2	T_3	T_4	a_{00}	a_{01}	a_{10}	a_{11}	c_O
4. public static void warn(									
5. double velocity, double distance, boolean raining) {									
6. double braking_distance;	1	1	1	1	0	0	2	2	0.707
7. double current_mu = mu;	1	1	1	1	0	0	2	2	0.707
8. boolean warning;	1	1	1	1	0	0	2	2	0.707
9. if (raining) {	1	1	1	1	0	0	2	2	0.707
10. **current_mu = 0.85*current_mu;**	0	0	1	1	2	0	0	2	1.000
11. }									
12. braking_distance = (velocity*velocity)/...;	1	1	1	1	0	0	2	2	0.707
13. if (braking_distance > 0.8*distance) {	1	1	1	1	0	0	2	2	0.707
14. warning = true;	1	0	0	0	1	2	1	0	0.000
15. } else {									
16. warning = false;	1	0	1	1	1	0	1	2	0.816
17. }									
18. store(warning,braking_distance);	1	1	1	1	0	0	2	2	0.707
19. }									
Error vector	0	0	1	1					

2.1 Spectrum-Based Fault Localization

We first explain the ideas behind SFL [13, 2, 3, 1]. SFL utilizes the program spectrum for debugging. A program spectrum is a matrix considering the program statements on one axis and the test cases on the other. A cell of the matrix has a value of 1 if the statement is executed in its corresponding test case and 0, otherwise. To access an element of the program spectrum in row i and column j, we write x_{ij}. In addition, we have an error vector e_j indicating whether a test case is passing or failing for each column j. A passing test case is a test case where the program delivers the expected outcome. Otherwise, a test case is said to be a failing one. Table 3 shows the program spectrum and the error vector for our illustrative example program `Car.warn'`.

To compute a suspicious value for each statement, we need information on whether a statement is executed in a passing or failing run. SFL introduces for this purpose 4 metrics values a_{ij} for each statement, i.e., row i, which are defined as follows: $a_{nm}(i) = |\{j | x_{ij} = n \wedge e_j = m\}|$. Hence, each a_{nm} indicates whether a statement is executed or not in a passing or failing test case. We find the metrics information for `Car.warn'` in Table 3. What is missing is the computation of a suspicious value. In SFL, the four metrics values are used. There are a lot of papers introducing the computation of different suspicious values, which are also called SFL coefficients, that lead to a ranking of statements. The one with the highest value is the most suspicious, followed by the next value, etc.

In our experiments, we used three different SFL coefficients, i.e., Tarantula [13] c_T (Equation 1), Ochiai [2] c_O (Equation 2), and Sarhan-Beszedes [20] c_S (Equation 3):

$$c_T = \frac{\frac{a_{11}}{a_{11}+a_{01}}}{\frac{a_{10}}{a_{10}+a_{01}} + \frac{a_{11}}{a_{11}+a_{01}}} \tag{1}$$

$$c_O = \frac{a_{11}}{\sqrt{(a_{11}+a_{01}) \cdot (a_{11}+a_{10})}} \tag{2}$$

$$c_S = a_{11} + \left(\frac{a_{11} - a_{01}}{a_{11} + a_{01} + a_{10}} \right) \tag{3}$$

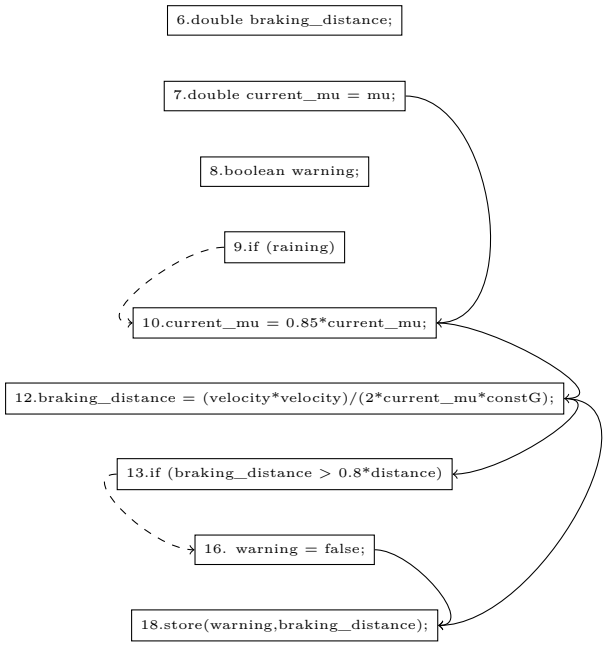
For our running example, Table 3 shows the results when applying the Ochiai coefficient. We see that the statement comprising the fault, i.e., Line 10, is the most suspicious with a c_O value of 1.0, followed by Line 16. Hence, in this case, SFL would enable a programmer to localize the fault in one step, only having a look at the statement in Line 10.

2.2 Dynamic Slicing

Dynamic slicing [16] is different compared to SFL as it utilizes the data and control dependencies of a program considering statements that are executed for a particular test case. Dynamic slicing only uses one test case at a time for extracting statements that cause a value of a given variable at a certain position of the execution. A particular execution can be seen as a trace, i.e., the execution trace. Together with all dependencies, we obtain a directed graph, i.e., the execution trace graph. Let us illustrate this using our `Car.warn'` method and test case T_3. When calling `warn(30,26,true)` the following statements are executed ignoring syntax details:

```
6.      double braking_distance;
7.      double current_mu = mu;
8.      boolean warning;
9.      if (raining) {
10.         current_mu = 0.85*current_mu;
12.     braking_distance = (velocity*velocity)/(2*current_mu*constG);
13.     if (braking_distance > 0.8*distance) {
16.         warning = false;
18.     store(warning,braking_distance);
```

We now add information regarding data and control dependencies and obtain the following graph:



If we now want to compute a dynamic slice for a variable, e.g., `braking_distance`, we only need to mark the node from the end of the graph where the variable is defined for the last time. For `braking_distance` this is Line 12. Afterward, we traverse the graph backward and mark all nodes we can reach. For `braking_distance` we obtain lines 10, 9, and 7. The final dynamic slice comprises all marked nodes, which are 7, 9, 10, and 12 for `braking_distance`. For `warning`, the dynamic slice is: 7, 9, 10, 12, 13, 16. Considering a single fault assumption, we may only want to focus on the intersection of both slices, which finally is the slice for `braking_distance`. Note that in dynamic slicing, we cannot rank statements in the slice. We further do not incorporate knowledge about failing or passing runs. Dynamic slicing is mainly used for failing test cases to identify root causes for misbehavior. However, due to using dependency information, dynamic slicing allows distinguishing statements in a block that would not be possible when only considering SFL. Therefore, a combination of the approaches seems to be a good idea to compensate for the weaknesses of the respective other approach.

2.3 Dynamic slicing enhanced spectrum-based fault localization

To be self-contained, we briefly explain the idea and algorithm of the combined approach. For more details, we refer to the original publication of Wotawa [30]. The idea behind combining slicing and SFL is simple. Instead of considering statement executions and solely the outcome of a test, we distinguish all output variables and consider the dynamic slices for them. When distinguishing the outputs of a test, we can also add the information of whether an output is correct or incorrect directly into SFL. Moreover, statements that are not in any dynamic slice can be ignored. The computation of the coefficients is then working as usual. Hence, it is only the way a test is separated into parts that distinguish the hybrid method from ordinary SFL.

In Figure 4, we depict the outcome of the hybrid method for our running example program `Car.warn'`. We see that the buggy statement is still the highest-ranked one. It is worth noting that, for this example, the hybrid method does not provide an improvement. However, for other example programs, there are improvements and, therefore, we are interested in a more detailed evaluation considering different faults and programs.

Table 4 The program spectrum of `Car.warn'` considering the 4 test cases and the hybrid approach. Note that the columns named `B` and `W` are the ones for variables `braking_distance` and `warning` respectively. Further, note that values are only computed for statements appearing in slices.

Statement	T_1		T_2		T_3		T_4		a_{00}	a_{01}	a_{10}	a_{11}	c_O
	W	B	W	B	W	B	W	B					
4. public static void warn(													
5. double velocity, ...) {													
6. double braking_distance;													
7. double current_mu = mu;	1	1	1	1	1	1	1	1	0	0	5	3	0.612
8. boolean warning;													
9. if (raining) {	1	1	1	1	1	1	1	1	0	0	5	3	0.612
10. **current_mu = 0.85*cur...**	0	0	0	0	1	1	1	1	4	0	1	3	0.866
11. }													
12. braking_distance = (velocity...;	1	1	1	1	1	1	1	1	0	0	5	3	0.612
13. if (braking_distance > 0.8*...) {	1	1	1	1	1	1	1	1	0	0	5	3	0.612
14. warning = true;	1	0	0	0	0	0	0	0	4	3	1	0	0.000
15. } else {													
16. warning = false;	0	0	1	0	1	0	1	0	3	2	2	1	0.333
17. }													
18. store(warning,braking_distance);													
19. }													
Error vector	0	0	0	0	1	1	0	1					

2.3.1 Implementation

We implemented the hybrid SFL approach utilizing the existing JSR framework that computes the checked coverage of Java programs [23, 15, 14]. Checked coverage is a coverage measure that considers whether statements influence the values of a given test property, which are used, for example, in JUnit as test oracles. The JSR framework utilizes the Java code coverage library JaCoCo [9] and the dynamic slicing tool Slicer4J [4] for computing checked coverage.

Using JSR, we obtain the spectrum for a program and a test suite as follows: For each test and output variable, we generate a JUnit test that comprises a test property for the variable. We run JSR on the test suite and compute the checked coverage for all test properties, which means to compute all statements that influence the variable value and are executed. This information is added to the program spectrum. For the error vector, we record whether the test property indicates a passing or a failing run and finally obtain all the needed information.

It is worth noting that we made some changes to JSR and also implemented the computation of the already introduced SFL coefficients, i.e., Ochiai, Tarantula, and Sarhan-Beszedes. The implementation is available on GitHub `https://github.com/SchleichJonas/JSR`. Besides the implementation, we put all programs for obtaining the experimental evaluation into the repository to ensure reproducibility.

3 Experimental evaluation

The objective of the experimental evaluation is to show the impact of adding dynamic slicing to spectrum-based fault localization. In particular, we are interested in answering the question of whether utilizing dynamic slicing improves the outcome of spectrum-based

Table 5 Programs used for the initial experiments.

Program	LoC	Test cases	Inputs	Outputs
BMI	24	6	2	1
Expint	88	5	2	1
Fisher	74	5	3	1
Gammq	91	5	2	1
Luhn	91	7	2	1
Middle	27	5	3	1
Tcas	152	8	12	1

fault localization when applied to ordinary programs. To answer this question, we first need to define what "improves" means in this context. From the literature, there have been several metrics specified for comparing different debugging methods, e.g., Hit Ratio@k and the wasted effort [12]. Whereas the former indicates how often a debugging approach ranks the correct diagnosis within the first n elements, the latter captures how many non-faulty statements are ranked before (or have to be inspected before) the buggy statement. The wasted effort is an absolute evaluation measure and not a percentage rank. Note that wasted effort is also sometimes referred to as EXAM score and is considered favorable to assess software debugging techniques [5]. In the context of this paper, we use the following definition of wasted effort (WE) where s_f is the suspicious score of the actual error:

$$WE = |\text{lines with score} > s_f| + 0.5 \cdot (|\text{lines with score} = s_f|) - 1)$$

Note that the hit ratio given in this paper is always the best, i.e., if there are multiple statements having the same SFL coefficient value, we assume that the statement comprising the fault is ranked the best.

For the experimental evaluation, we used different sets of programs. We introduced bugs and test suites that are able to capture the bug. For every program, we only introduced one fault manually. We ran the different debugging approaches and obtained the hit ratio and wasted effort for each program. We finally computed average values. In the following, we report the obtained results, considering each of the program sets separately. We start with the initial experiments.

3.1 Initial experiments

We carried out the initial experiments considering simple programs such as a BMI calculator, the Luhn algorithm that is used for credit card number verification, TCAS, and others, for which we summarize some metrics information in Table 5. These programs have been used in the context of software testing and also debugging research. The calculations vary in complexity from elementary calculations to more complex ones. In each class, multiple intentional errors were added manually, triggering SFL and dynamic slicing. The introduced errors differ, aiming at achieving as much variety as possible. Every calculation has multiple test cases, including passing and failing test cases. Table 6 depicts the applied changes for obtaining the programs used for the initial experiments.

For each program, we manually developed 5–8 test cases, ensuring that some are passing and some are failing. For the initial evaluation, we only considered one variable as the output variable. Hence, we did not expect any negative impact of the hybrid method on the SFL result. There might be a positive impact on the results because the hybrid SFL method utilizes data dependencies, which might allow the elimination of statements that do not contribute to the final outcome and, therefore, improve the ranking.

Table 6 Alteration of the source code used in the initial experiments.

Program	LineNr.	Original code	Altered code
BMI	17	else if (bmi < 30)	else if (bmi > 30)
BMI2	7	bmi_score = weight / (height * height);	bmi_score = weight / (height);
BMI3	12	calculateBMI(height, weight);	calculateBMI(weight, height);
Expint	43	h *= del;	h = del;
Expint2	47	return h*Math.exp(-x);	return h*Math.exp(x);
Expint3	38	a = -i*(nm1+i);	a = -i*nm1+i;
Fisher4	10	a = 2*(m/2)-m+2;	a = 2*(m/2)-n+2;
Fisher2	27	d = 0.5*p*z/w;	d = 0.4*p*z/w;
Fisher3	54	p = p*zk+w*z*(zk-1.0)/(z-1.0);	p = p*zk+w*(zk-1.0)/(z-1.0);
Gammq	37	an = -i*(i-a);	an = i*(i-a);
Gammq2	72	gamser=sum*Math.exp(-x+a*Math.log(x)-gln);	gamser=sum*Math.exp(-x+a*Math.log(x));
Gammq3	84	return 1-gamser;	return gamser-1;
Luhn	14	if (number.length() != 16 ||	if (number.length() != 15 ||
Luhn2	76	for (int i = number.length-2; i > -1; i-=2) {	for (int i = number.length-2; i > 0; i-=2) {
Middle	6	if((a<b && b<c)|| (c<b && b<a)){	if((a<b && b<c)|| (c<b && b>a)){
Middle2	15	return a;	return b;
Tcas	54	return ((Climb_Inhibit!=0) ? Up_Separation + NOZCROSS : Up_Separation);	return ((Climb_Inhibit!=0) ? Up_Separation - NOZCROSS : |Up_Separation);
Tcas2	124	alt_sep = UPWARD_RA;	alt_sep = UNRESOLVED;

In tables 7 and 8, we depict the obtained results of the initial experiments considering SFL alone and the hybrid approach, respectively. We see that there are differences both in the rank and the wasted effort (WE). In some cases, WE improves when using the hybrid approach, and in three cases, the WE becomes worse. This holds especially for the Expint and the Tcas programs, where we obtained severe declines in performance, which is also visible in the following table, which captures the Hit ratio and the average values of the WE results for the three SFL coefficients:

	Ochiai			Tarantula			Sarhan-Beszedes		
	Hit@1	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**
SFL	0.8333	1.0000	4.8056	0.7222	1.0000	5.8611	0.8333	1.0000	4.8056
Hybrid	0.7778	0.9444	6.4722	0.7222	0.9444	7.6389	0.7778	0.9444	6.4722

For all three SFL coefficients, we see a decline of the WE when using the hybrid approach, which is unexpected. Therefore, we further investigated the underlying reasons.

After having a look at the usual suspects for causes of the unexpected results, like the implementation or the experimental setup, which seemed to be correct, we further investigated the dynamic slicer used in the JSR framework. A detailed analysis of the resulting traces and slices showed that sometimes the slicer does not give back statements comprising faults. For both programs Expint and Tcas we obtained a similar outcome. The slicer uses Jimple, which is a 3-address intermediate representation, to simplify analysis and transformation of Java bytecode [24] as the internal representation of the program. Unfortunately, this transformation removes important parts of the program, not allowing the slicer to return a correct output in every case. It is worth noting that dynamic slicers themselves do not always deliver back a correct slice, which leads to the development of critical slicing [8] and other variants.

Table 7 Results of the initial experiments only considering SFL without slicing. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
BMI	1	0.0	1	1.5	1	0.0
BMI2	1	0.5	3	5.0	1	0.5
BMI3	1	1.0	1	1.0	1	1.0
Expint	1	9.0	1	9.0	1	9.0
Expint2	1	9.0	1	9.0	1	9.0
Expint3	1	9.0	1	9.0	1	9.0
Fisher	1	8.0	1	19.0	1	8.0
Fisher2	1	0.5	1	0.5	1	0.5
Fisher3	2	4.5	2	4.5	2	4.5
Gammq	1	10.0	1	10.0	1	10.0
Gammq2	1	6.0	1	6.0	1	6.0
Gammq3	1	6.0	1	6.0	1	6.0
Luhn	1	0.0	2	2.0	1	0.0
Luhn2	1	6.5	1	6.5	1	6.5
Middle	2	1.5	2	1.5	2	1.5
Middle2	1	0.0	1	0.0	1	0.0
Tcas	4	15.0	4	15.0	4	15.0
Tcas2	1	0.0	1	0.0	1	0.0

Table 8 Results of the initial experiments considering the hybrid approach with dynamic slicing. An arrow up indicates an improvement of the result compared to the one given in Table 7, and an arrow down the opposite. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
BMI	1	0.0	1	1.5	1	0.0
BMI2	1	0.0↑	3	4.5↑	1	0.0↑
BMI3	1	1.0	1	1.0	1	1.0
Expint	2↓	26.0↓	2↓	26.0↓	2↓	26.0↓
Expint2	1	7.5↑	1	7.5↑	1	7.5↑
Expint3	1	7.5↑	1	7.5↑	1	7.5↑
Fisher	1	4.5↑	1	19.5↓	1	4.5↑
Fisher2	1	0.5	1	0.5	1	0.5
Fisher3	2	4.5	2	4.5	2	4.5
Gammq	1	8.0↑	1	8.0↑	1	8.0↑
Gammq2	1	4.5↑	1	4.5↑	1	4.5↑
Gammq3	1	5.0↑	1	5.0↑	1	5.0↑
Luhn	1	0.0	1↑	0.0↑	1	0.0
Luhn2	1	6.5	1	6.5	1	6.5
Middle	2	5.5↓	2	5.5↓	2	5.5↓
Middle2	1	0.0	1	0.0	1	0.0
Tcas	6↓	35.5↓	6↓	35.5↓	6↓	35.5↓
Tcas2	1	0.0	1	0.0	1	0.0

Table 9 Programs used for second experiments.

Program	LoC	Test cases	Inputs	Outputs
Armstrong	49	7	1	1
Bubblesort	38	7	1	1
ChineseRemainder	47	7	2	1
Factorial	18	7	1	1
GCD	27	7	2	1
InverseCounter	26	7	1	1
IsPrime	20	7	1	1
LCM	24	7	2	1
LogExp	23	7	2	1
Minimax	50	7	1	1
ModInverse	32	7	2	1
Mult	30	7	2	1
RSA	58	7	3	1
RussianPeasant	34	7	2	1
Sqrt	23	7	1	1

Hence, we can summarize the findings obtained from the initial experiment as follows:

Finding 1: There is a huge impact of the underlying slicer on the obtained results of the hybrid method in some cases leading to worst results when using the hybrid SFL approach.

More experiments are required to further investigate the effect of the impact of the underlying slicer and on other influencing factors like the way an error was introduced in a program. Therefore, we carried out a second and a third experiment.

3.2 Second experiments

The objectives of the second experiment are to clarify whether there is an influence on the way intentional errors are introduced, covering well-known algorithm implementations. For this purpose, we selected 15 algorithms, from very simple ones like the greatest common divisor (GCD) to more complex ones like RSA encryption and decryption. See Table 9 for the list of implementations and their corresponding statistics. We provided the implementations to four students and asked them to place one error into the source code, not giving them a lot of input to avoid introducing bias. Note that the implementations were equally distributed among the students. The only requirement was that the error still lead to the execution of the program without any exceptions. The selected students come from different fields; where two are from software engineering, one from mathematics, and another one is not from any science discipline at all. Table 10 depicts the changes to the implementations introduced by the students.

After running the second experiment, we obtained the debugging outcome of SFL and the hybrid method that we depict in tables 11 and 12, respectively. The debugging methods performed better in this experiment than the initial one. This is probably because the programs of the second experimental evaluation are simpler. The improvements can be observed both for the Tarantula coefficient and the hybrid SFL method. However, Tarantula still performed worse than the other SFL coefficient. The already well-performing Sarhan-Beszedes coefficient performed even better than Ochiai, which performed the same as Sarhan-Beszedes for the initial setup.

Table 10 Changed code of programs used for the second experiments.

Program	LineNr.	Original code	Altered code
Armstrong	34	int r = temp % 10;	int r = temp / 10;
Bubblesort	36	return arr;	return res;
ChineseRemainder	26	if (x%num[j] != rem[j])	if (x%num[j] == rem[j])
Factorial	7	int res = 1;	int res = 0;
GCD	12	if (a % result == 0 && b % result == 0) {	if (a / result == 0 && b % result == 0) {
InverseCounter	12	for (int j = i + 1; j < n; j++) {	for (int j = i + 1; j < n - 1; j++) {
IsPrime	10	for (int i = 2; i < n; i++)	for (int i = 2; i <= n; i++)
LCM	10	return gcd(b % a, a);	return gcd(b % a, b);
LogExp	12	pow = pow * b;	pow = pow % b;
Minimax	48	return minimax(0, 0, true, scores, h);	return -minimax(0, 0, true, scores, h);
ModInverse	30	return modInverse(A, M);	return modInverse(M, M);
Mult	17	return -multiply(x, -y);	return multiply(x, -y);
RSA	14	z = (p - 1) * (q - 1);	z = (p - 1) * (q + 1);
RussianPeasant	19	res = res + a;	res = res % a;
Sqrt	17	return res - 1;	return res - res;

Table 11 Results of the second experiments only considering SFL without slicing. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
Armstrong	1	4.5	1	4.5	1	4.5
BubbleSort	1	3.0	1	7.5	1	3.0
ChineseRemainder	1	3.5	1	5.0	1	3.5
Factorial	1	2.0	1	2.5	1	2.0
GCD	1	0.5	1	0.5	1	0.5
InverseCounter	2	4.5	2	4.5	1	3.5
Isprime	1	1.0	1	1.0	1	1.0
LCM	1	0.0	1	0.0	1	0.0
LogExp	1	0.0	1	0.0	1	0.0
Minimax	1	3.0	1	3.0	1	3.0
ModInverse	1	3.0	1	5.0	1	3.0
Mult	1	0.5	1	0.5	1	0.5
RSA	1	12.0	1	12.0	1	12.0
RussianPeasant	1	4.0	1	4.0	1	4.0
Sqrt	2	3.0	2	3.0	2	3.0

Table 12 Results of the second experiment considering the hybrid approach with dynamic slicing. An arrow up indicates an improvement of the result compared to the one given in Table 11, and an arrow down the opposite. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
Armstrong	1	5.0↓	2↓	7.0↓	1	5.0↓
BubbleSort	1	0.5↑	1	7.5	1	0.5↑
ChineseRemainder	3↓	7.0↓	1	5.0	3↓	7.0↓
Factorial	1	1.0↑	1	2.0↑	1	1.0↑
GCD	1	1.0↓	1	1.0↓	1	1.0↓
InverseCounter	1↑	2.0↑	1↑	2.0↑	2↓	5.0↓
Isprime	1	1.0	1	1.0	1	1.0
LCM	1	0.0	1	0.0	1	0.0
LogExp	1	0.0	1	0.0	1	0.0
Minimax	1	2.0↑	2↓	4.0↓	1	2.0↑
ModInverse	1	2.5↑	1	4.5↑	1	2.5↑
Mult	1	0.5	1	0.5	1	0.5
RSA	1	9.5↑	1	9.5↑	1	9.5↑
RussianPeasant	1	3.5↑	1	3.5↑	1	3.5↑
Sqrt	2	3.0	2	3.0	2	3.0

These improvements are also very well visible in the summary of the results, where we also consider the hit ratio and not only wasted effort, which is given in the following table:

	Ochiai			Tarantula			Sarhan-Beszedes		
	Hit@1	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**
SFL	0.8667	1.0000	2.9667	0.8667	1.0000	3.5333	0.9333	1.0000	2.9000
Hybrid	0.8667	1.0000	2.5667	0.8000	1.0000	3.3667	0.8000	1.0000	2.7667

Interestingly, while HitRatio@1 decreased with the hybrid method for Tarantula and Sarhan-Beszedes, the wasted effort improved slightly for both. For Ochiai, we see no change in the hit ratio, but the wasted effort also improved. Hence, what we can conclude is the following:

Finding 2: The hybrid method for SFL behaves slightly better considering the wasted effort for simple algorithm implementations, whereas the hit ratio is slightly worse.

Hence, we see a different outcome in the experiments where all underlying programs have only one output. Therefore, we are interested in investigating the effect of considering programs with more than one output on the debugging results. To answer this question, we carried out a third experimental evaluation.

3.3 Third experiments

To answer the question of whether the number of output variables impacts the ranking when comparing SFL with our hybrid method, we used the same example programs as in the first experiments, which are depicted in Table 5. But for most of the programs of the initial experiments, we used available variables in the source code as additional outputs for which we defined the expected values. For the programs and their variants, we have the following number of outputs in the third experimental setup: BMI (2), Expint (2), Fisher (4), Gammq (4), Luhn (1), Middle (1), Tcas (5). Note also that the increase of outputs (with

Table 13 Results of the third experiments only considering SFL without slicing. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
BMI	1	0.0	1	1.5	1	0.0
BMI2	1	2.5	4	7.5	1	2.5
BMI3	1	3.0	1	3.0	1	3.0
Expint	1	7.5	1	7.5	1	7.5
Expint2	1	7.5	1	7.5	1	7.5
Expint3	1	7.5	1	7.5	1	7.5
Fisher	1	8.0	1	13.5	1	8.0
Fisher2	1	0.5	1	0.5	1	0.5
Fisher3	2	4.5	2	4.5	2	4.5
Gammq	1	10.0	1	10.0	1	10.0
Gammq2	1	6.0	1	6.0	1	6.0
Gammq3	1	6.0	1	6.0	1	6.0
Luhn	1	0.0	2	2.0	1	0.0
Luhn2	1	6.5	1	6.5	1	6.5
Middle	2	1.5	2	1.5	2	1.5
Middle2	1	0.0	1	0.0	1	0.0
Tcas	4	16.5	4	16.5	2	11.5
Tcas2	1	0.0	1	0.0	1	0.0

the exception of Luhn and Middle) also leads to an increase in test cases because we have one JUnit test for each property specifying the expected behavior of one output. Hence, the results for SFL also might change, which is well visible when comparing Table 7 with Table 13.

When comparing the results of SFL and the hybrid approach, which are depicted in Table 13 and Table 14 respectively, we see an improvement. The hybrid approach provides a better wasted effort for almost all programs. Moreover, the ranking improved as well and is now the same or better for most of the coefficients. Considering more outputs even compensates for the problem with the slicer. In the following table, we summarize the findings for all programs, also considering the hit ratio:

	Ochiai			Tarantula			Sarhan-Beszedes		
	Hit@1	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**	**Hit@1**	**Hit@5**	**WE**
SFL	0.8333	1.0000	4.8611	0.7222	1.0000	5.6389	0.8333	1.0000	4.5833
Hybrid	0.8333	1.0000	2.1667	0.7778	1.0000	3.1389	0.8333	1.0000	2.1389

We see that with one exception for Tarantula, the hit ratio of the hybrid approach is always the same as the one for SFL alone. The wasted effort improves between around 5% and 15% on average for the different SFL coefficients. Hence, the third experiment confirms that adding slicing has the potential to improve the outcome, at least for the wasted effort, which is an important measure for the efficiency of debugging. This leads to the following third finding of our preliminary experimental evaluation:

Finding 3: For programs with more than one output, the hybrid approach, which integrates dynamic slicing into SFL, the wasted effort improves.

Table 14 Results of the third experiments considering the hybrid approach with dynamic slicing. An arrow up indicates an improvement of the result compared to the one given in Table 13, and an arrow down the opposite. WE stands for wasted effort.

Program	Ochiai		Tarantula		Sarhan-Beszedes	
	Rank	WE	Rank	WE	Rank	WE
BMI	1	0.0	1	1.5	1	0.0
BMI2	1	0.5↑	2↑	1.5↑	1	0.5↑
BMI3	1	0.5↑	1	0.5↑	1	0.5↑
Expint	1	0.5↑	1	0.5↑	1	0.5↑
Expint2	1	1.0↑	1	1.0↑	1	1.0↑
Expint3	1	5.5↑	1	7.0↑	1	5.5↑
Fisher	1	0.0↑	1	12.0↑	1	0.0↑
Fisher2	1	0.5	1	0.5	1	0.5
Fisher3	2	4.0↑	2	4.5	2	3.5↑
Gammq	1	6.5↑	1	7.0↑	1	6.5↑
Gammq2	1	3.5↑	1	4.0↑	1	3.5↑
Gammq3	1	0.0↑	1	0.0↑	1	0.0↑
Luhn	1	0.0	1↑	0.0↑	1	0.0
Luhn2	1	6.5	1	6.5	1	6.5
Middle	2	5.5↓	2	5.5↓	2	5.5↓
Middle2	1	0.0	1	0.0	1	0.0
Tcas	4	4.0↑	3↑	4.0↑	4↓	4.0↑
Tcas2	1	0.5↓	1	0.5↓	1	0.5↓

3.4 Discussion

The experimental evaluation's objective was to carry out experiments showing whether combining dynamic slicing with SFL has a positive impact on debugging, i.e., a better hit ratio or wasted effort. From the experiments, which also considered different SFL coefficients, we see not a big difference except when considering multiple output variables, which was somehow expected. What was not expected to see a slight decrease in the debugging efficiency of the hybrid method for some examples? A detailed analysis reveals that the slicer causes trouble in some cases. Hence, a result of this evaluation is that the slicer may have a huge impact depending on the program we want to debug.

As already mentioned, this experimental evaluation is an initial study. It uses small and at least partially simple example programs. Hence, the outcome may vary when using larger and more complex programs. However, the programs implement well-known algorithms and comprise ordinary data and control structures. They have not been selected for a particular purpose but used because they have already served as examples of different studies. Besides the selection of the programs, there are other threats to the validity. First, the implementation makes use of available tools and frameworks, which might be buggy, causing a bias. Second, we introduced faults manually. However, we did not introduce a fault having a certain result in mind. Third, the number of programs and their faulty variants are limited. The same holds for the test suites, which have also been manually generated. Considering more variants and different test suites might change the outcome. Finally, we used only one programming language, i.e., Java, for providing examples. Hence, there might also be an influence on the outcome. Therefore, further studies have to be carried out, including replications of the experiments, to prove or disprove the obtained findings.

4 Conclusions

In this paper, we present a first experimental evaluation that compares ordinary spectrum-based fault localization with a variant that utilizes dynamic slicing to overcome some limitations, like handling data dependencies. The experimental evaluation relied on several smaller Java programs where we introduced faults. The evaluation revealed a huge impact of the implementation, and in particular, the use of a dynamic slicer, on the outcome. Furthermore, we showed that for programs having more output, the hybrid method performs better in terms of wasted effort. Hence, the outcome of the study is promising. Howewver, further experiments are required for a final judgment of the combined debugging methodology. This includes identifying the influence of test suites or the complexity of programs on the outcome. Moreover, more faulty variants and larger programs should be used for the evaluation. The latter is of particular interest to show whether additional computational complexity of the computationally more demanding hybrid approach really pays off, leading to a substantially improved debugging outcome. All these open issues we want to consider in our future research.

References

1 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. doi:10.1016/j.jss.2009.06.035.

2 Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. On the accuracy of spectrum-based fault localization. In *Proceedings TAIC PART'07*, pages 89–98. IEEE, 2006.

3 Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-based multiple fault localization. In *Proc. IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 88–99, 2009. doi:10.1109/ASE.2009.25.

4 Khaled Ahmed, Mieszko Lis, and Julia Rubin. Slicer4J: A Dynamic Slicer for Java. In *The ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2021.

5 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)*, pages 175–182, 2017. doi:10.1109/ISSREW.2017.68.

6 Luca Console, Gerhard Friedrich, and Daniele Theseider Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings* 13^{th} *International Joint Conf. on Artificial Intelligence*, pages 1494–1499, Chambery, August 1993.

7 Johan de Kleer and Brian C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987. doi:10.1016/0004-3702(87)90063-4.

8 Richard A. DeMillo, Hsin Pan, and Eugene H. Spafford. Critical slicing for software fault localization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 121–134, 1996. doi:10.1145/229000.226310.

9 EclEmma team. Jacoco java code coverage library. https://www.eclemma.org/jacoco/. Last accessed 20 January 2025. URL: https://www.eclemma.org/jacoco/.

10 Gerhard Friedrich, Markus Stumptner, and Franz Wotawa. Model-based diagnosis of hardware designs. *Artificial Intelligence*, 111(2):3–39, July 1999. doi:10.1016/S0004-3702(99)00034-X.

11 Birgit Gertraud Hofer and Franz Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In Luc de Raedt, editor, *ECAI 2012 - 20th European Conference on Artificial Intelligence.*, volume 242 of *ECAI*, pages 420–425, Netherlands, 2012. IOS Press. doi:10.3233/978-1-61499-098-7-420.

12 Jiajun Jiang, Ran Wang, Yingfei Xiong, Xiangping Chen, and Lu Zhang. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 502–514, 2019. doi:10.1109/ASE.2019.00054.

13 J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings ASE'05*, pages 273–282. ACM Press, 2005.

14 Roxane Koitz-Hristov, Thomas Sterner, Lukas Stracke, and Franz Wotawa. On the suitability of checked coverage and genetic parameter tuning in test suite reduction. *Journal of Software: Evolution and Process*, 36(8):e2656, 2024. doi:10.1002/smr.2656.

15 Roxane Koitz-Hristov, Lukas Stracke, and Franz Wotawa. Checked coverage for test suite reduction – is it worth the effort? In *2022 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 6–16, 2022. doi:10.1145/3524481.3527216.

16 Bogdan Korel and Janusz Laski. Dynamic Program Slicing. *Information Processing Letters*, 29:155–163, 1988. doi:10.1016/0020-0190(88)90054-3.

17 Adil Mukhtar, Birgit Hofer, Dietmar Jannach, Franz Wotawa, and Konstantin Schekotihin. Boosting spectrum-based fault localization for spreadsheets with product metrics in a learning approach. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*, pages 175:1–175:5. ACM, 2022. doi:10.1145/3551349.3559546.

18 Sofia Reis, Rui Abreu, and Marcelo d'Amorim. Demystifying the combination of dynamic slicing and spectrum-based fault localization. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, pages 4760–4766. International Joint Conferences on Artificial Intelligence Organization, July 2019. doi:10.24963/ijcai.2019/661.

19 Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987. doi:10.1016/0004-3702(87)90062-2.

20 Qusay Idrees Sarhan and Arpad Beszedes. Experimental evaluation of a new ranking formula for spectrum based fault localization. In *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 276–280, 2022. doi:10.1109/SCAM55253.2022.00038.

21 Ehud Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

22 Péter Attila Soha. On the efficiency of combination of program slicing and spectrum-based fault localization. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 499–501, 2023. doi:10.1109/ICST57152.2023.00061.

23 Lukas Stracke. Jsr - the java test suite reduction framework. https://github.com/Lms24/JSR?tab=readme-ov-file#jsr---the-java-test-suite-reduction-framework. Last accessed 20 January 2025. URL: https://github.com/Lms24/JSR?tab=readme-ov-file#jsr---the-java-test-suite-reduction-framework.

24 Raja Vallée-Rai and Laurie J. Hendren. Jimple: Simplifying java bytecode for analyses and transformations, 1998. URL: https://api.semanticscholar.org/CorpusID:10529361.

25 Mark Weiser. Programmers use slices when debugging. *Communications of the ACM*, 25(7):446–452, July 1982. doi:10.1145/358557.358577.

26 Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984. doi:10.1109/TSE.1984.5010248.

27 Wanzhi Wen, Bixin Li, Xiaobing Sun, and Jiakai Li. Program slicing spectrum-based software fault localization. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Miami Beach, USA, 2011.

28 W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016. doi:10.1109/TSE.2016.2521368.

29 Franz Wotawa. On the Relationship between Model-Based Debugging and Program Slicing. *Artificial Intelligence*, 135(1–2):124–143, 2002. doi:10.1016/S0004-3702(01)00161-8.

30 Franz Wotawa. Surveying and generalizing methods for combining dynamic slicing with spectrum-based fault localization. In *Proceedings of the 34th International Workshop on Principles of Diagnosis (DX)*, Loma Mar, CA, USA, 2023. URL: `https://dx-2023.sai.tugraz.at/DX23_Wotawa.pdf`.

31 Franz Wotawa, Mihai Nica, and Iulia Moraru. Automated debugging based on a constraint model of the program and a test case. *J. Log. Algebraic Methods Program.*, 81(4):390–407, 2012. `doi:10.1016/j.jlap.2012.03.002`.