ECAI 2012
Luc De Raedt et al. (Eds.)

doi:10.3233/978-1-61499-098-7-420

Spectrum Enhanced Dynamic Slicing for better Fault Localization

Birgit Hofer and **Franz Wotawa**[1]

Abstract. Debugging consumes a considerable amount of time in software engineering, but it is rarely automated. In this paper, we focus on improving existing fault localization techniques. Spectrum-based fault localization (SFL) and slicing-hitting-set-computation (SHSC) are two techniques based on program execution traces. Both techniques come with small computational overhead and aid programmers to faster identify possible locations of faults. However, they have disadvantages: SHSC results in an undesirable high ranking of statements which are executed in many test cases, such as constructors. SFL operates on block level. Therefore, it cannot provide fine-grained results. We combine SHSC with SFL in order to eliminate these disadvantages. Our objective is to improve the ranking of faulty statements so that they allow for better fault localization than when using the previously mentioned methods separately. We show empirically that the resulting approach reduces the number of statements a programmer needs to check manually. In particular, we gain improvements of about 50 % percent for SHSC and 25 % for SFL.

1 Introduction

There exist many techniques to automatically test software. These techniques are able to expose a huge amount of errors in software. Unfortunately, the next steps after fault detection, i.e., fault localization and correction, are rarely automated. Therefore, not all of the discovered errors can be corrected in an acceptable time. As a consequence debugging has been identified as a bottleneck for improving reliability [2].

In practice, there exist a considerable number of debugging tools, which should assist the programmer in localizing faults. However, most of these tools only enable programmers to execute a program step by step [8]. This step-by-step execution is very time consuming. Existing tools fail to lead the programmers to source code locations where the fault might be most likely located. Instead, a programmer has to narrow down the search space by introducing break points, comparing intermediate outcome of the program with the expectations, applying changes, and re-executing the program in order to validate these changes.

On the research side, automated debugging techniques and prototypes are available which help to narrow down possible fault locations. However, there is still room for improvements. In this paper, we borrow an idea from Mayer et al. [11] where the authors state that no single technique is able to deal with all types of faults. Consequently, a combination of different fault localization techniques is necessary to build more accurate and robust debugging tools because the strengths of the individual approaches complement each other. The approach we introduce in this paper follows this idea. In our Spectrum ENhanced DYnamic Slicing approach (SENDYS), we combine spectrum-based fault localization (SFL) [8] with slicing-hitting-set-computation (SHSC) [15]. The latter is a method based on program slicing [13]. In our approach, the SFL results are used as a-priori fault probabilities of single statements in the SHSC. This ensures that statements used by many passing test cases are lower ranked than statements, which are only used in failing test cases.

The advantage of combining SFL and SHSC lies in the use of the available information for ranking fault candidates. SENDYS makes use of the available dependence information like data and control dependences of programs. In contrast to SFL, which is not able to distinguish statements occurring in the same basic building block, our method allows for fault localization at the statement level. Moreover, SENDYS analyzes the execution information from both passing and failing test cases. This eliminates the weakness of SHSC: SHSC always ranks initialization statements high. Thus SENDYS helps to improve accuracy of fault localization compared to SHSC.

We illustrate the profitableness of SENDYS by means of an example. This example deals with transactions on a bank account and is a slight modified version of [15]. Figure 1 shows the source code of this example.

```
 1.public class BankAccount {
 2.  public long balance;
 3.  public long limit;
 4.  public BankAccount(long bal,long limit){
 5.    this.balance = bal;
 6.    this.limit = limit;
 7.  }
 8.  public void withdraw(long amount){
 9.    if((balance - amount) >= limit){
10.      balance = balance - amount;
11.    }
12.  }
13.  public void deposit(long amount){
14.    balance = balance + amount;
15.  }
16.  public void transferTo(BankAccount acc){
17.    long money = this.balance;
18.    if(money!=0){ //FAULT
19.      this.withdraw(money);
20.      acc.deposit(money);
21.    }
22.  }
23.}
```

Figure 1. The Bank Account Example - Line 18 contains a fault. The correct statement would be `if(money>0){`.

The balance of the account must never fall below the specified limit. The balance can only be transferred to another account if it is larger than zero. In this example program, we introduce a fault in Line 18. We extended the original example with a constructor in or-

[1] Institute for Software Technology, Graz University of Technology, Austria
email: {bhofer,wotawa}@ist.tugraz.at

Table 1. The Bank Account Example - An extended test suite.

Code	T1	T2	T3	T4	T5
`a1=new BA(0,-10)`		•		•	•
`a1=new BA(-1,-10)`	•				
`a1.withdraw(1)`		•		•	
`a2=new BA(1,0)`			•		
`a2=new BA(0,0)`	•			•	•
`a2.desposit(2)`			•	•	•
`a1.transferTo(a2)`	•				•
`a2.transferTo(a1)`				•	
Test oracle					
a1.balance	-1	-1	–	1	0
a2.balance	2	–	3	0	2

der to be able to highlight the advantage of our approach. The original bank account example contains only one test case (`T1`). Since our approach requires both passing and failing test cases, we extended the test suite. Table 1 shows a compressed version of the extended test suite. A test case is made up of the statements which are marked with •. In addition, we add statements to check if the computed results equal the expected results (test oracle).

We organize this paper as follows: In Sections 2 and 3, we explain the basic approaches, i.e., SHSC and SFL and highlight their advantages as well as their weaknesses. In addition, we demonstrate the usage of these approaches by means of the bank account example. We introduce SENDYS in Section 4, which combines these techniques, and apply SENDYS to the bank account example. In Section 5, we apply SENDYS to several programs with predefined faults. Furthermore, we compare the fault localization capabilities of SENDYS with those of the two other approaches. Finally, we review related work in Section 6 and conclude the paper in Section 7.

2 Fault Localization Based on Dynamic Slicing and Hitting-Set Computation

Wotawa [14] discussed the relationship of model-based debugging and program slicing. The basic idea of his technique is to combine slices of faulty variables so that they result in minimal diagnoses. Figure 2 describes the basic approach: For each variable x in a test case T where the expected value does not correspond to the actual value, a slice is computed. A slice is a subset of a program which behaves like the original program for a given set of variables. In principle, every type of slice can be computed, but by reasons of its precision and size a relevant slice [17] is favored over static [13] and dynamic slices [10]. From these slices, the minimal diagnoses Δ^S are computed by means of the corrected Reiter algorithm [5]. This computation is based on hitting sets. A hitting set is defined for a set of sets CO as follows: A set h is a hitting set if and only if for all $x \in CO$ there exist a non-empty intersection between x and h, i.e., $\forall x \in CO : x \cap h \neq \emptyset$. A hitting set h is said to be minimal if there exists no real subset of h that is itself a hitting set.

The approach works for one or more failing test cases. In case of a single failing test case with n statements in the slice, the approach delivers n single-fault diagnoses.

Single-fault diagnoses are a valuable support for programmers. However, in case of several faults, single-fault diagnoses miss to detect the real faults. In this case, multiple-fault diagnoses can be useful. Nevertheless, many multiple-fault diagnoses are confusing since a programmer might check the same statement for correctness several times. An extension of the approach [15] solves this problem by mapping back diagnoses to a summary slice. Figure 3 illustrates the computation of such a summary slice: First, the initial fault probabilities

Algorithm AllDiagnoses(Π, TC)
Require: program Π and test suite TC
Ensure: set of minimal diagnoses Δ^S
1: conflict set $CO = \{\}$
2: **for all** test cases $T \in TC$ **do**
3: **if** $test(\Pi, T) = FAIL$ **then**
4: **for all** wrong variables values x at position n **do**
5: $CO.add(Slice(\Pi, x, n, T))$
6: **end for**
7: **end if**
8: **end for**
9: **return** $HittingSets(CO, |\Pi|)$

Figure 2. *AllDiagnoses* algorithm for computing diagnoses from slices.

Algorithm HS-Slice (Π, Δ^S)
Require: program Π and minimal diagnoses Δ^S
Ensure: HS-slice S

1. Compute initial fault probability of all statements:
$$\forall s \in \Pi : p_F(s) = \frac{1}{|\Pi|}$$
2. Compute the fault probability for all diagnoses:
$$\forall \Delta_i \in \Delta^S : p(\Delta_i) = \prod_{s \in \Delta_i} p_F(s) \times \prod_{s' \in \Pi \setminus \Delta_i} (1 - p_F(s'))$$
3. Derive the probability that a statement s is faulty:
$$\forall s \in \Pi : p_{pred}(s) = \sum_{\Delta \in \Delta^S \wedge s \in \Delta} p(\Delta)$$
4. Normalize the fault probabilities:
$$\forall s \in \Pi : p'_F(s) = \frac{p_{pred}(s)}{\sum_{s' \in \Pi} p_{pred}(s')}$$
5. **return** statements s in descending order of $p'_F(s)$

Figure 3. *HS-Slice* algorithm for computing the fault candidates' ranking.

$p_F(s)$ are computed for all statements $s \in \Pi$. It is assumed that each statement is equal likely to be faulty. The fault probabilities $p(\Delta_i)$ for all diagnoses $\Delta_i \in \Delta^S$ are computed. These fault probabilities are used to compute the fault probabilities $p_{pred}(s)$ of the statements. Finally, the statement probabilities are normalized ($p'_F(s)$) and the statements are sorted using their fault probabilities. The summary slice enhanced by the fault probabilities is called *HS-slice*. It consists of a set of pairs: $S = \{(s, p'_F(s)) | \exists \Delta \in \Delta^S : s \in \Delta\}$.

We illustrate the application of this approach using the example from Figure 1. There is one test case (`T1`) that reveals the bug. This test case has two variables with wrong values. Thus the slices for `a1.balance` ($S_1 = \{5, 6, 9, 10, 17, 18, 19\}$) and `a2.balance` ($S_2 = \{5, 14, 17, 18, 20\}$) are computed. The Reiter algorithm provides 11 minimal diagnoses. There are 3 single fault explanations ({5},{17} and {18}) and 8 double fault explanations ({6,14},{6,20},{9,14},{10,14},{14,19},{9,20},{10,20} and {19,20}). We set the initial fault probabilities to $p_F(s) = 1/9$. The fault probabilities for the single fault diagnoses are $p(\Delta_i) = 0.043$. Those of the double fault probabilities are $p(\Delta_j) = 0.005$. Table 2 shows the results. The last column of the table shows that the faulty statement is ranked at position 1, but there are 2 other statements with the same ranking. Thus 3 of 9 statements (33 %) must be investigated.

Table 2. The Bank Account Example - Ranking of the statements based on the slicing-hitting-set-approach. The faulty statement is marked with •.

Line s	$p_{pred}(s)$	$p'_F(s)$	Ranking
5	0.043	0.200	1
6	0.011	0.050	6
9	0.011	0.050	6
10	0.011	0.050	6
14	0.022	0.100	4
17	0.043	0.200	1
• 18	0.043	0.200	1
19	0.011	0.050	6
20	0.022	0.100	4

The slicing-hitting-set-approach (SHSC) has two major limitations: First, it is not able to localize faults which are caused by missing code. Second, it always ranks constructor statements high since they are part of every slice. The combined approach introduced in this paper is able to eliminate the second limitation.

3 Spectrum-based fault localization

Spectrum-based fault localization (SFL) is based on an observation matrix from which similarity coefficients are computed for each block. A block can be a component, a method or a compound statement. Blocks with the highest coefficients are most likely to be faulty.

The observation matrix consists of two parts: the program spectra and the error vector. A program spectrum is an abstraction of an execution trace. It maps only one specific view of the dynamic behavior of a program [2]. This could be e.g. the number of times the block was executed (block count spectrum) or more simple if the block was visited at all (block hit spectrum). A detailed overview of the different types of program spectra can be found in [7]. In this approach, block hit spectra are used. They only indicate which parts of a program have been executed during a run [8]. In the case of block hit spectra, the entries of the observation matrix are boolean values (covered / not covered). The error vector indicates whether the respective test case passes or fails. The information of the error matrix can be further compressed by computing the values from Eqs. 1-3 for each block.

$$a_{11}(j) = |\{i|x_{ij} = 1 \wedge e_i = 1\}| \quad (1)$$

$$a_{10}(j) = |\{i|x_{ij} = 1 \wedge e_i = 0\}| \quad (2)$$

$$a_{01}(j) = |\{i|x_{ij} = 0 \wedge e_i = 1\}| \quad (3)$$

Spectrum-based fault localization is based on the assumption that a high similarity of a block to the error vector indicates a high probability that a block is responsible for the error [3]. In principle, any type of similarity coefficient can be used. We have chosen the Ochiai coefficient since several experiments (e.g. [2, 3]) have shown that it outperforms other coefficients like Tarantula and Jaccard. The Ochiai coefficient is computed as described in Eq. 4.

$$s_O(i) = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}} \quad (4)$$

We demonstrate the use of Spectrum-based fault localization by means of our example from Figure 1. Table 3 shows the observation matrix obtained when executing the test cases on the faulty program. The rightmost columns show the computed coefficients and the resulting ranking when using the Ochiai coefficient. The faulty statement is ranked at third position together with 3 other statements. In this case, 6 of 9 lines of code (67 %) must be investigated.

Table 3. The Bank Account Example - The observation matrix, the resultant Ochiai coefficients and the subsequent ranking of the statements.

Line s	T1	T2	T3	T4	T5	Coeff.	Rank.
5	•	•	•	•	•	0.447	9
6	•	•	•	•	•	0.447	9
9	•	•		•		0.577	3
10	•	•		•		0.577	3
14	•		•	•	•	0.500	7
17	•			•	•	0.577	3
• 18	•			•	•	0.577	3
19	•			•		0.707	1
20	•			•		0.707	1
Error	•						

The major drawback of spectrum-based fault localization is its granularity. The finest granularity can only be a compound statement. This is due to the fact that it is not possible to distinguish between statements with identical execution patterns [8]. By using slices in our combined approach, this drawback is eliminated. Another handicap of SFL and many other debugging techniques is the lack of the ability to advise the programmer that the fault might be caused by missing code.

4 Combined approach

We combine the previously described approaches in SENDYS. Figure 4 illustrates this new approach: First, the observation matrix O and the similarity coefficients $sc_o(s)$ for all statements $s \in \Pi$ are computed and normalized ($sc_{norm}(s)$). Afterwards, the minimal diagnoses Δ^S are computed as described in Algorithm *AllDiagnoses* (Figure 2). Finally, Algorithm *HS-Slice* (Figure 3) is used with the normalized similarity coefficients $sc_{norm}(s)$ and the minimal diagnoses Δ^S as input. The resulting summary slice S' is returned.

Algorithm SENDYS(Π, TC)
Require: program Π and test suite TC
Ensure: HS-Slice S'

1. Compute observation matrix O for program Π and test suite TC:
$$O = observationMatrix(\Pi, TC)$$
2. Compute similarity coefficients $sc_o(s)$ for all statements $s \in \Pi$:
$$\forall s \in \Pi : sc_o(s) = ochiai(s, O)$$
3. Compute the normalized values of the similarity coefficients $sc_{norm}(s)$ for all statements $s \in \Pi$:
$$\forall s \in \Pi : sc_{norm}(s) = \frac{sc_o(s)}{\sum_{j=1}^{|\Pi|} sc_o(j)}$$
4. Compute the minimal diagnoses Δ^S:
$$\Delta^S = AllDiagnoses(\Pi, TC)$$
5. Compute the summary slice S' with Algorithm *HS-Slice*. Start with Step 2. Use $sc_{norm}(s)$ instead of $p_F(s)$.
6. **return** S'

Figure 4. SENDYS – The combined fault localization algorithm

We show the use of the combined algorithm for the running example. Table 4 shows the fault probabilities and the ranking when

using the normalized Ochiai coefficients as initial fault probabilities. In this case only 2 of 9 statements (22 %) must be investigated.

Table 4. The Bank Account Example - Ranking of the statements based on SENDYS. The faulty statement is marked with •.

Line s	$p_{pred}(s)$	$p'_F(s)$	Ranking
5	0.032	0.163	3
6	0.008	0.043	9
9	0.011	0.054	7
10	0.011	0.054	7
14	0.018	0.092	5
17	0.040	0.205	1
• 18	0.040	0.205	1
19	0.012	0.063	6
20	0.024	0.122	4

The combined approach performs better than the original slicing-hitting-set-approach when the fault is not part of the initialization of the program. One might think that it performs worse when the fault is located in the initialization, since it decreases the probabilities of these parts. However, this is not the case. In order to illustrate a fault in the initialization, we modify our example program from Figure 1 in the following way:

```
 5.    this.balance = balance;
18.    if(money>0){
```

Using Ochiai, the faulty statement is ranked at position 2 together with the other initialization statement. The union of all faulty execution traces comprises 5 statements. In the worst case, 3 of these 5 statements (60 %) must be investigated.

For computing the ranking of the statements by the original slicing-hitting-set-approach and SENDYS, the dynamic slices for the test cases `T1` ($S_{a1.balance} = \{5, 17, 18\}$) and `T3` ($S_{a2.balance} = \{5, 14\}$) are computed. These slices result in 3 minimal diagnoses ($\{5\}$, $\{14,17\}$ and $\{14,18\}$). The faulty statement is ranked at position 1 for both approaches. This example demonstrates that the combined approach is able to detect faults in initialization statements with the same accuracy as the original slicing-hitting-set approach.

Similar to related approaches, our method is highly dependent on the quality of the test suite and cannot detect missing statements. However, our method eliminates the following disadvantages that are present in the original approaches: On the one hand it does not rank initialization statements high (compared to SHSC) and on the other hand it is finer grained than SFL.

One might think that the combined approach is only valuable in case of single faults, which is not true. As mentioned in [15], a probability-based slice is a comprehensive but compact representation. It provides a better overview than a list of diagnoses. There might be statements that are part of several slices. Such statements are investigated by the programmer several times when processing the diagnoses one after another. The summary slice *HS-slice* provides an overview of all statements and their fault probabilities. The programmer can process the statements in descending order of their fault probabilities. Thus each statement is investigated only once.

Our debugging method requires a marginal run-time overhead compared to the single approaches. Both approaches, i.e., SFL and SHSC, require a program Π to be executed, which can be performed in $O(\Pi)$. Given M test cases, the program must be executed M times. Thus the test execution requires $O(\Pi \cdot M)$ time. The observation matrix used in SFL can be computed within $O(\Pi \cdot M)$ time. The similarity coefficients are computed in $O(M \cdot N)$ time where N is the number of statements in Π. The statements are ranked in $O(N \cdot \log N)$ time. Thus SFL requires $O(\Pi \cdot M + M \cdot N + N \cdot \log N)$ time. For SHSC the relevant slices are required. The computation of slices depends on the size of the execution trace. In the worst case, the time complexity of computing all relevant slices is $O(\Pi \cdot M)$. In addition to the computation of the slices, it is necessary to compute hitting-sets, which is in the worst case exponential in the size of Π. However, when considering only single and double faults, we retain polynomial complexity. As a consequence the complexity of Algorithm *AllDiagnoses* (Figure 2) is $O(\Pi \cdot M + N^2)$. Since there exist at maximum N^2 diagnoses when considering only single and double faults, the fault probabilities of the diagnoses and statements in Algorithm *HS-Slice* (Figure 3) can be computed in $O(N^3)$ time. Thus Algorithm *HS-Slice* (Figure 3) has $O(N^3)$ time complexity for single and double fault diagnoses. Therefore, the overall run-time of SHSC, including all parts of the approach and under the given assumptions, has a time complexity of $O(\Pi \cdot M + N^3)$. When combining these time complexities, the SENDYS approach is still bounded by a complexity of $O(\Pi \cdot M + M \cdot N + N^3)$.

5 Evaluation

In this section, we first deal with the experimental setup by giving an overview of the implementation of the approach. Afterwards, we introduce the tested programs by quantitatively and qualitatively describing them. Finally, we show the advantage of SENDYS over the basic approaches through the results of the experiments.

Our implementation of SENDYS works with Java programs and JUnit test cases. It utilizes the JavaSlicer[2] for obtaining the execution traces and the dynamic slices. The spectra information is obtained from the execution traces.

Besides the fact that the slicer in use is only a dynamic slicer [10] and not a relevant slicer [17], it has some weaknesses [6], which influence the obtained results: First, data dependencies can vanish when a method is called by reflection or when native code is executed. Second, due to the restriction to dynamic slices, it is possible that the real fault is not part of a slice. This is why we have to exclude faults leading to incomplete slices from the case study. However, we are still able to proof our concept.

We investigated the 8 programs listed in Table 5. BankAccount, Mid and StaticExample are toy examples. The Traffic Light Example is borrowed from the JADE project[3]. Since this program does not have any JUnit test cases, we created our own. The source code of the previously mentioned programs and of ATMS is public available[4]. JTopas is taken from the Software Infrastructure Repository [4]. Tcas is a Java Implementation of the traffic collision avoidance system from the Siemens Set.

It was not possible to compute the possible fault locations for all available faulty program versions because of the following three reasons. First, no slices can be computed for test cases which produce endless loops. Second, the slicer in use is not able to compute correct slices for all faulty program versions due to its limitations. Third, JTopas includes predefined faults which are not detected by the available test cases. We excluded program versions from our case study for which any of these cases apply. Table 5 gives an overview of the number of faulty program versions used in the evaluation (see column Faults). The first number in the brackets indicates the number of program variants which were excluded from the case study because there were no failing test cases. The second number indicates

[2] `http://www.st.cs.uni-saarland.de/javaslicer/`
[3] `http://www.dbai.tuwien.ac.at/proj/Jade/`
[4] `http://dl.dropbox.com/u/38372651/Debugging/EP.zip`

Table 5. Description of the investigated programs including the version number (V), the Non Commenting Source Statements (NCSS), the number of test cases (TC) and the number of investigated faults, which is tripartite: faults with no failing test cases, excluded faults and investigated faults.

Program	Description	V	NCSS	TC	Faults
Bank Account	Demo program simulating a bank account	1	17	5	2(-/-/2)
Mid	Demo program returning the medial of three numbers	1	17	8	1(-/-/1)
Static Example	Demo program with static members and methods	1	16	8	1(-/-/1)
Traffic Light	Simulation system of different phases of traffic light	1	33	7	2(-/-/2)
ATMS	Assumption-based Truth Maintenance System	1	1573	14	3(-/1/2)
Reflec. Visitor	Implementation of the Visitor-Pattern	1	338	14	5(-/-/5)
JTopas	Text parser	1 2 3	1368 1485 3931	127 115 183	8(4/3/1) 12(11/-/1) 14(7/4/3)
Tcas	Traffic Collision Avoidance System	1	77	1545	39(-/15/24)

the number of variants which were excluded because of endless loops or limitations in the slicer. The third column indicates the number of program variants that were used in the evaluation.

When comparing the fault localization capabilities of SENDYS with those of the basic approaches, it turns out that SENDYS leads to huge savings in the number of statements that must be investigated. Table 6 compares the overall effectiveness of SENDYS to those of the basic approaches with respect to the mean value of statements that must be investigated in order to find the bug. Since the investigated programs considerably differ in size, we have normalized the basic values before computing the mean value: For each fault, instead of using the absolute number, we used the ratio of the statements that must be investigated and the total number of statements that were executed in the failing test cases of that program version. From this table it is obvious that SENDYS improves the fault localization capabilities of SHSC by 50 % and those of SFL by 25 %. The poor results of SHSC might result from the limitations of the slicer in use, since it was not possible to compute correct slices for all slicing criteria.

Table 6. Average percentage of statements that must be investigated in order to find the faulty one on basis of the total number of executed statements in the failing test cases for SHSC, SFL and SENDYS.

SHSC	SFL	SENDYS
63.9 %	43.5 %	31.7 %

An interesting question is if SENDYS always improves the results of the basic approaches. In 25 cases SENDYS improves the fault localization precision. In 14 cases it performs as good as the best of the original approaches. In 3 cases it performs worse than SFL.

Figure 5 graphically compares the fault localization capabilities of the three approaches for the 42 investigated faulty program versions. The figure compares SENDYS with its basic approaches in terms of the amount of code that must be investigated in order to find the faulty statement. The x-axis represents the percentage of code that is investigated. The y-axis represents the percentage of faults that are localized within that amount of code. This figure reads as follows: If you investigate the top 40 % ranked statements of the 42 investigated faulty program versions, SENDYS contains the faulty statement for 65 % of the program versions. SFL only contains the faulty statement for 45 % of the program versions, and SHSC for 18 % of the program versions. It can be seen that when using SENDYS faults can be earlier detected than when using one of the basic approaches.

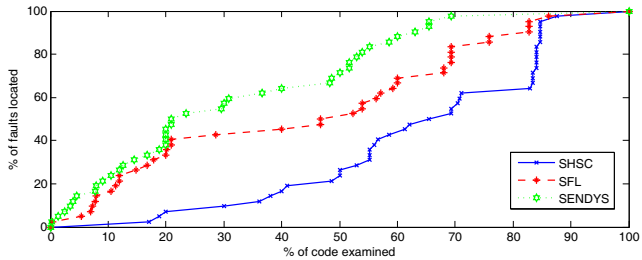

Figure 5. Comparison of SENDYS with SHSC and SFL (using Ochiai) in terms of the amount of code that must be investigated.

So far, our experiments have been performed using the Ochiai coefficient. We have chosen the Ochiai coefficient since studies [2, 3] have shown that it delivers better results than other coefficients. However, our approach works for other coefficients as well. Table 7 shows the amount of statements that must be investigated in percentage of the executed statements when using different similarity coefficients. This table affirms that Ochiai performs better than Tarantula and Jaccard. In addition, it shows that using SENDYS improves the fault localization capabilities. Still, SENDYS based on Ochiai performs best. Figure 6 illustrates the fault localization capabilities of SENDYS with different similarity coefficients. The figure compares SENDYS based on the different similarity coefficients with the basic coefficients in terms of the amount of code that must be investigated in order to find the faulty statement.

Table 7. Average percentage of statements that must be investigated in order to find the faulty one on basis of the total number of executed statements in the failing test cases for different similarity coefficients.

	Ochiai	Tarantula	Jaccard
Standalone	43.5 %	52.3 %	46.0 %
Part of SENDYS	31.7 %	33.7 %	34.8 %

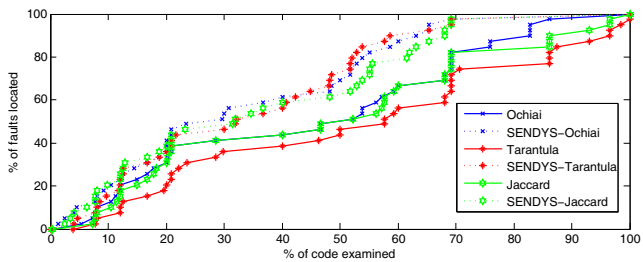

Figure 6. Comparison of SENDYS with different similarity coefficients in terms of the amount of code that must be investigated.

The computational overhead of SENDYS compared to the basic approaches is marginal. In this evaluation, the execution of the test cases absorbs the major part of the total computation time. For the larger programs (ATMS, JTopas, Reflection Visitor and Tcas), the spectra creation and coefficients computation requires approximately 10 % of the time required for the execution. The computation time of the slices and hitting sets ranges from 10 % to 25 % of the execution time. The computations of SENDYS (Slice and spectra computation, hitting sets, fault probabilities) account for 20 % to 35 % of the execution time.

6 Related Research

Weiser [13] introduced the concept of static slicing. He defined a slice as a program where zero or more statements are removed and the reduced program behaves like the original program for a given set of variables at a given location in the program. Since static slices tend to be rather large, Korel et al. [10] introduced the concept of dynamic slicing. Dynamic slices behave like the original program only for a given test case because they rely on a concrete program execution. Since dynamic slices are more restrictive, they yield smaller slices than their static counterpart. Occasionally, dynamic slices do not include statements which are responsible for a fault if the fault causes the non-execution of some parts of a program. This disadvantage is eliminated by the usage of relevant slicing [17].

Many debugging techniques focus on the usage of failing test cases only. In contrast, spectrum-based fault localization considers both passing and failing test cases. There exist many spectrum-based approaches in literature, e.g. Tarantula [9], Jaccard and Ochiai [3]. It has been shown that Ochiai locates faults better than the others.

Mayer et al. [11] combine spectrum-based fault localization (SFL) with model-based software debugging (MBSD) in order to eliminate the absence of a model in SFL and to assign a ranking to the diagnosis candidates. Mayer and Stumptner [12] provide an overview of model-based debugging techniques. They state that debugging with dependency-based models, such as SHSC, is faster than debugging with value-based models, abstraction-based models or bounded model checking, but it is less precise. Our approach is similar to the previously described approach [11] since both use SFL and the concepts of "reasoning from first principles". However, our approach has a lower computational complexity since SHSC is less complex than MBSD with more sophisticated models.

BARINEL [1] is a Bayesian framework that computes fault probabilities per statement using maximum likelihood estimation. In contrast to our approach, it relies only on the information which statements were covered. However, it does not make use of dependency information. Thus, it does not filter statements which are executed in faulty runs but do not contribute to the value of the faulty variable(s).

Xu et. al. [16] improve spectrum-based fault localization by adding a noise reduction term to the suspiciousness coefficient computation, called MINUS, and by using chains of key basic blocks (KBC - Key Block Chain) as program features. We differ from their approach by adding dynamic dependency information through slices to the data available by Spectrum-based Fault Localization.

7 Conclusion

In this paper, we introduced a novel approach to fault localization in programs, i.e., Spectrum Enhanced Dynamic Slicing (SENDYS). SENDYS combines spectrum-based fault localization (SFL) with slicing-hitting-set-computation (SHSC). The approach solves some disadvantages of SFL and SHSC that occur when executing them individually. We discussed the SENDYS approach in detail and compared its outcome with the individual approaches in an empirical study. The empirical study indicates that our combined approach outperforms SFL and SHSC. SENDYS provides an improved ranking of fault candidates. Thus SENDYS is a more valuable aid for programmers when debugging. In particular, SENDYS improves the fault localization capabilities of SHSC by 50 % and those of SFL by 25 %.

With respect to the SFL approach, we compared different coefficients used for ranking the statements. The empirical results show that Ochiai is performing best compared to Tarantula and Jaccard with respect to fault localization capabilities.

In future work we will compare the computation time and the diagnostic accuracy of our approach with those of more complex model-based software debugging approaches, e.g. [12].

ACKNOWLEDGEMENTS

The research herein is partially conducted within the competence network Softnet Austria II (www.soft-net.at, COMET K-Projekt) and funded by the Austrian Federal Ministry of Economy, Family and Youth (bmwfj), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT).

REFERENCES

[1] Rui Abreu and Arjan J. C. van Gemund, 'Diagnosing multiple intermittent failures using maximum likelihood estimation', *Artificial Intelligence*, **174**, 1481–1497, (December 2010).

[2] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund, 'An evaluation of similarity coefficients for software fault localization', in *PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, pp. 39–46, Washington, DC, USA, (2006). IEEE Computer Society.

[3] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan J. C. van Gemund, 'A practical evaluation of spectrum-based fault localization', *J. Syst. Softw.*, **82**(11), 1780–1792, (2009).

[4] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel, 'Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact.', *Empirical Software Engineering: An International Journal*, **10**(4), 405–435, (2005).

[5] Russell Greiner, Barbara A. Smith, and Ralph W. Wilkerson, 'A correction to the algorithm in Reiters theory of diagnosis', *Artificial Intelligence*, (1989).

[6] Clemens Hammacher. Design and implementation of an efficient dynamic slicer for Java. Bachelor's Thesis, November 2008.

[7] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi, 'An empirical investigation of program spectra', in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '98, pp. 83–90, NY, USA, (1998). ACM.

[8] Tom Janssen, Rui Abreu, and Arjan J.C. van Gemund, 'Zoltar: a spectrum-based fault localization tool', in *SINTER '09: Proceedings of the 2009 ESEC/FSE workshop on Software integration and evolution and runtime*, pp. 23–30, New York, NY, USA, (2009). ACM.

[9] James A. Jones and Mary Jean Harrold, 'Empirical evaluation of the tarantula automatic fault-localization technique', in *ASE '05: Proceedings of the 20th IEEE/ACM int. Conference on Automated software engineering*, pp. 273–282, New York, NY, USA, (2005). ACM.

[10] B. Korel and J. Laski, 'Dynamic program slicing', *Inf. Process. Lett.*, **29**, 155–163, (October 1988).

[11] Wolfgang Mayer, Rui Abreu, Markus Stumptner, and Arjan J. C. van Gemund, 'Prioritising model-based debugging diagnostic reports', in *Proceedings of the 19th International Workshop on Principles of Diagnosis*, Blue Mountains, Sydney, Australia, (September 2008).

[12] Wolfgang Mayer and Markus Stumptner, 'Evaluating models for model-based debugging', in *Proceedings of the 2008 23rd IEEE/ACM Int. Conference on Automated Software Engineering*, ASE '08, pp. 128–137, Washington, DC, USA, (2008). IEEE Computer Society.

[13] Mark Weiser, 'Programmers use slices when debugging', volume 25, pp. 446–452, New York, NY, USA, (1982). ACM.

[14] Franz Wotawa, 'On the relationship between model-based debugging and program slicing', *Artif. Intell.*, **135**, 125–143, (February 2002).

[15] 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, pp. 161–170, Washington, DC, USA, (2010). IEEE Computer Society.

[16] Jian Xu, W.K. Chan, Zhenyu Zhang, T.H. Tse, and Shanping Li, 'A dynamic fault localization technique with noise reduction for java programs', *Int. Conference on Quality Software*, **0**, 11–20, (2011).

[17] Xiangyu Zhang, Sriraman Tallam, Neelam Gupta, and Rajiv Gupta, 'Towards locating execution omission errors', *SIGPLAN Not.*, **42**(6), 415–424, (2007).