Institute for Software Technology
Graz University of Technology

Spectrum Enhanced Dynamic Slicing for Fault Localization

IST-DR-2012-01

Birgit Hofer, Franz Wotawa

IST Technical Report

February 2012

TU Graz

IST TECHNICAL REPORT
IST-DR-2012-01, FEBRUARY 2012

Spectrum Enhanced Dynamic Slicing for Fault Localization

Birgit Hofer, Franz Wotawa

Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/II
8010 Graz, Austria

Telephone +43 316 873 57 11, Facsimile +43 316 873 57 06
Email {bhofer,wotawa}@ist.tugraz.at

Abstract. Spectrum-based diagnosis (SFL) and Slicing-Hitting-Set-Computation (SHSC), a method using dynamic program slices, are two techniques for fault localization based on program execution traces. Both techniques come with small computational overhead and aid programmers to faster identify faults by determining their most likely locations. However, they have disadvantages: SHSC results in an undesirable high ranking of statements which occur in many test cases, such as constructors. SFL takes into consideration how often statements are executed in successful test cases. Thus it does not rank constructor statements that high. SFL operates on block level whereas SHSC operates on statement level and can therefore provide finer-grained results. In this technical report, we show how to combine SHSC with spectrum-based diagnosis. 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 slicing and 25 % for spectrum-based fault localization.

Contents

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 step after fault detection, i.e., fault localization and correction, is hardly supported. Moreover, debugging is still an activity which is hardly 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 [1].

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 [2], which is very time consuming. None of the tools 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 changes.

On the research side some automated debugging techniques and prototypes are available. However, there is still room for improvements. We borrow an idea from Mayer et al. [3] where the authors stated 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 are going to introduce follows this underlying idea. In our Spectrum ENhanced DYnamic Slicing approach (SENDYS), we combine spectrum-based fault localization (SFL) [4] with slicing-hitting-set-computation (SHSC) [5]. The latter is a method based on program slicing [6]. 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 such as constructors are lower ranked than statements which were only used in failing test cases.

The advantage of combining SFL and SHSC in SENDYS 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. This allows for fault localization at the statement level, which improves SFL that is not able to distinguish statements occurring in the same basic building block. Moreover, SENDYS also takes care of the execution information from both passing and failing test cases, which helps to improve accuracy of fault localization compared to SHSC.

In this paper we illustrate the profitableness of SENDYS by means of an example. The example deals with transactions on a bank account and is taken from [5] with some modifications. Figure 1 shows the source code of this example. The balance of the account must never fall below the specified limit and it can only be transferred to another account if the balance is larger than zero. In this example program we introduce a fault in Line 18. We extended the original example with a constructor in order to be able to highlight the advantage of our combined approach. We also added new test cases. The original bank account example [5] contains only one test case (`testTransfer1`). Since our approach requires both passing and failing test cases, we extended the test suite. Figure 2 shows the extended test suite.

In the following we will show that SENDYS improves fault localization using the bank account example. Whereas SHSC allows for reducing the search space to 3 statements, SFL reports 6 statements as potential root causes. In contrast SENDYS only returns 2 statements as result including the faulty statement.

There are three main contributions of the paper:

- We introduce SENDYS, a new approach for fault localization.
- We discuss the advantages of SENDYS compared to the used basic approaches using an empirical evaluation.
- When using the empirical evaluation results, we once more affirm that fault localization based on Ochiai yields better results than using other similarity coefficients.

We organize this technical report as follows: In Section 2 we introduce the basic approaches, i.e., the slicing-hitting-set-approach and spectrum-based fault localization. In addition we demonstrate the usage of these approaches by means of the bank account example. Afterwards we introduce SENDYS in Section 3, which combines these two techniques. In Section 4 we apply SENDYS to several programs with predefined faults and compare the fault localization capabilities of it with those of the two other approaches. Finally, we review related work in Section 5 and conclude the technical report in Section 6.

```
 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 taken from [5] and extended by a constructor - Line 18 contains a fault. The correct statement would be `if(money>0){`.

```
public void testTransfer1 { //T1
  BankAccount a1 = new BankAccount(-100,-1000);
  BankAccount a2 = new BankAccount(0,0);
  a2.deposit(200);
  a1.transferTo(a2);
  Assert.assertEquals(-100, a1.balance);
  Assert.assertEquals(200, a2.balance);
}
public void testWithdraw(){ //T2
  BankAccount a1 = new BankAccount(0,-1000);
  a1.withdraw(100);
  Assert.assertEquals(-100, a1.balance);
}
public void testDeposit() { //T3
  BankAccount a2 = new BankAccount(100,0);
  a2.deposit(200);
  Assert.assertEquals(300, a2.balance);
}
public void testTransfer2() { //T4
  BankAccount a1 = new BankAccount(0,-1000);
  a1.withdraw(100);
  BankAccount a2 = new BankAccount(0,0);
  a2.deposit(200);
  a2.transferTo(a1);
  Assert.assertEquals(100, a1.balance);
  Assert.assertEquals(0, a2.balance);
}
public void testTransfer3() { //T5
  BankAccount a1 = new BankAccount(0,-1000);
  BankAccount a2 = new BankAccount(0,0);
  a2.deposit(200);
  a1.transferTo(a2);
  Assert.assertEquals(0, a1.balance);
  Assert.assertEquals(200, a2.balance);
}
```

Figure 2: The Bank Account Example - An extended test suite.

2 Basic definitions

Our approach combines Spectrum-based fault localization with Slicing-Hitting-Set-Computation. Therefore we explain these techniques and highlight their advantages as well as weaknesses.

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

Wotawa discussed the relationship of model-based debugging and program slicing [7]. The basic idea of his technique is to combine slices of faulty variables so that they result in minimal diagnoses. Algorithm 1 describes the basic approach. First, for each variable x in a test case T where the expected value does not correspond to the actual value, a slice is computed. In principle every type of slice can be computed, but by reasons of its precision and size a relevant slice is favored over static and dynamic slices. The slices are handled as conflicts in model-based diagnosis. By means of the corrected Reiter algorithm [8, 9] the minimal diagnoses Δ^S are computed. This computation is based on hitting sets. A hitting set is defined for a set of sets CO as follows: A set $h \subseteq \bigcup_{x \in CO} x$ 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 is said to be minimal if there exist no real subset that is itself a hitting set. In the algorithm from Figure 1 the hitting sets are computed using the $HittingSets$ function.

Algorithm 1 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 ∈ TC do
3:    if test(Π, T) = FAIL then
4:      for all wrong variables values x at position n do
5:        conflict C = Slice(Π, x, n, T)
6:        CO.add(C)
7:      end for
8:    end if
9:  end for
10: sort CO in ascending cardinal order
11: return HittingSets(CO, |Π|)
```

The approach works for one or more failing test cases. In case of a single failing test case with n statements contained in the slice it delivers n single-fault diagnoses, one for each statement. Single-fault diagnoses are a valuable support for programmers. Too many multiple-fault diagnoses are confusing. An extension of the approach [10, 5] solves this problem by mapping back diagnoses to a summary slice.

Algorithm 2 illustrates the computation of such a summary slice. First for all statements $s \in \Pi$ the fault probability $p_F(s)$ is initialized. It is assumed that each statement is equal likely to be faulty. Afterwards the set of leading diagnoses LD is computed from the set of minimal diagnoses Δ^S. A leading diagnosis is a superset of at least one minimal diagnosis. The number of leading diagnoses grows exponential with the program size. A boundary value $B \geq 0$ is used to reduce the number of generated supersets. The computed leading diagnoses LD comprise all supersets of the minimal diagnoses Δ^S which have at most B elements more than the contained minimal diagnosis. The fault probabilities $p(\Delta_i)$ for all diagnoses $\Delta_i \in LD$ 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 and 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 by means of our example from Figure 1. There is one test case (T1) which 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 boundary value $B = 0$ so that the set of Leading Diagnoses LD is identical to the computed minimal diagnoses Δ^S in order to simplify the computation and to ease the traceability of this example. The results can change somewhat if B is set to a higher value, but it is assumed that the changes are not substantial. We set the initial fault probabilities to

Algorithm 2 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 set of Leading Diagnoses LD for all minimal diagnoses Δ^S:

$$LD(\Delta^S) = \{x | x \subseteq \Pi \wedge \exists \Delta \in \Delta^S : (x \supseteq \Delta \wedge (|x| - |\Delta|) \leq B)\}$$

3. Compute the fault probability for all diagnoses:

$$\forall \Delta_i \in LD : p(\Delta_i) = \prod_{s \in \Delta_i} p_F(s) \times \prod_{s' \in \Pi \setminus \Delta_i} (1 - p_F(s'))$$

4. Derive the probability that a statement s is faulty:

$$\forall s \in \Pi : p_{pred}(s) = \sum_{\Delta \in LD(\Delta^S) \wedge s \in \Delta} p(\Delta)$$

5. Normalize the fault probabilities:

$$\forall s \in \Pi : p'_F(s) = \frac{p_{pred}(s)}{\sum_{s' \in \Pi} p_{pred}(s')}$$

6. **return** statements s in descending order of $p'_F(s)$

$p_F(s) = 1/9$ for the computation of the fault probabilities of the diagnoses. 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 1 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 1: 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 has two major limitations: It is not able to localize faults which are caused by missing code. In addition, 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.

2.2 Spectrum-based fault localization

Spectrum-based fault localization is based on an observation matrix from which similarity coefficients are computed for each block. A block can be for example a component, a method or a compound statement. The blocks with the highest coefficients are most likely to be faulty. Figure 3 shows the structure of an observation matrix. It consists of two parts: The program spectra as row vectors and the error vector.

$$O = \left\{ \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1N} \\ x_{21} & x_{22} & \cdots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{M1} & x_{M2} & \cdots & x_{MN} \end{bmatrix}, \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} \right\}$$

Figure 3: Observation Matrix

A program spectrum is an abstraction of an execution trace of a program. It maps only one specific view of the dynamic behavior of a program [1]. This could be for example the number of times the block was executed (block count spectrum) or more simple if it was visited at all (block hit spectrum). Harrold et al. give an overview of the different types of program spectra and their subsumption hierarchy [11, 12].

In this approach block hit spectra are used. They only indicate which parts of a program have been executed during a run [2]. The $M \times N$ program spectra matrix from Figure 3 represents the information of M different test runs where N blocks were covered. In the case of block hit spectra the entries of this 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. 1-4 for each block.

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

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

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

$$a_{00}(j) = |\{i | x_{ij} = 0 \wedge e_i = 0\}| \tag{4}$$

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 [13]. In principle any type of similarity coefficient can be used. We have chosen the Ochiai coefficient since several experiments (e.g. [1, 13]) have shown that it outperforms other coefficients like Tarantula and Jaccard. The Ochiai coefficient is computed as described in Eq. 5.

$$s_O(i) = \frac{a_{11}(i)}{\sqrt{(a_{11}(i) + a_{01}(i)) * (a_{11}(i) + a_{10}(i))}} \tag{5}$$

We demonstrate the usage of Spectrum-based fault localization by means of our example from Figure 1. Table 2 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.

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 [4]. By using slices in our combined approach this drawback is eliminated.

Table 2: The Bank Account Example - The transposed observation matrix, the resultant Ochiai coefficients and the subsequent ranking of the statements. The faulty statement is marked with •.

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	•						

3 Combined approach

We have combined both approaches described in Section 2 in our approach called SENDYS. Algorithm 3 illustrates this new approach. First, the observation matrix O and the similarity coefficients $sc_{ochiai}(s)$ for all statements $s \in \Pi$ are computed. The computed coefficients are normalized ($sc_{norm}(s)$). The minimal diagnoses Δ^S are computed as described in Algorithm 1 AllDiagnoses(Π,TC). . Algorithm 2 HS-Slice(Π, Δ^S) 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.

We show the usage of the combined algorithm by means of our example. Table 3 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 3: 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. But this not necessarily is 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){
```

Table 4 shows the observation matrix, the computed Ochiai coefficients and the subsequent ranking. 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.

There are two slices ($S_1 = \{5, 17, 18\}$ and $S_2 = \{5, 14\}$) which result in 3 minimal diagnoses ({5}, {14,17} and {14,18}). Table 5 summarizes the results of the original slicing-hitting-set-approach and SENDYS. The faulty statement is ranked at position 1 for both approaches.

Algorithm 3 Combined approach (Π, TC)

Require: program Π and test suite TC
Ensure: HS-Slice S'

1. Compute the observation matrix O for program Π and test suite TC:

$$O = observationMatrix(\Pi, TC)$$

2. Compute the similarity coefficients $sc_{ochiai}(s)$ for all statements $s \in \Pi$:

$$\forall s \in \Pi : sc_{ochiai}(s) = ochiai(s, O)$$

3. Compute the sum of the coefficients sum_{sc}:

$$sum_{sc} = \sum_{j=1}^{|\Pi|} sc_{ochiai}(j)$$

4. Compute the normalized values of the similarity coefficients $sc_{norm}(s)$ for all statements $s \in \Pi$:

$$\forall s \in \Pi : sc_{norm}(s) = \frac{sc_{ochiai}(s)}{sum_{sc}}$$

5. Compute the minimal diagnoses Δ^S:

$$\Delta^S = AllDiagnoses(\Pi, TC)$$

6. Compute the summary slice S' with Algorithm 2. Start with Step 2. Use $sc_{norm}(s)$ instead of $p_F(s)$.

7. **return** S'

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.

The proposed approach eliminates disadvantages of both original approaches but it is not able to eliminate all of them. The fault location of errors, which are caused by missing code, can still not be localized similar to the other fault localization approaches. In addition, the introduced approach highly depends on the quality of the test suite, like the original approaches.

One might think that the combined approach is only valuable in case of single faults. But this is not true. As mentioned in [10] a list of diagnoses does not provide an overview to the programmer. There might be statements which are part of several slices. These 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 combined debugging method requires little run-time overhead when 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 a set of test cases of size M, all necessary information to construct the program spectrum in SFL can be computed within $O(M \cdot \Pi)$ time. Similarly, all execution traces necessary for computing all relevant slices can also be computed within $O(M \cdot \Pi)$ time. Given that the computation of slices also depends polynomial on the size of the execution trace, the worst case complexity of computing all relevant slices is $O(M \cdot \Pi)$. 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, or double faults we retain polynomial complexity. As a consequence the complexity of the SHSC approach is under the restricting assumptions $O(M \cdot \Pi)$. It is further worth noting that Algorithm 2 has $O(|\Pi|)$ complexity when assuming that $LD(.)$ is smaller or equal to $|\Pi|$. This assumption holds when only considering single 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 $O(M \cdot \Pi)$. When

Table 4: The modified Bank Account Example - The transposed observation matrix, the resultant Ochiai coefficients and the subsequent ranking of the statements.The faulty statement is marked with •.

Line s	**T1**	**T2**	**T3**	**T4**	**T5**	**Coeff.**	**Rank.**
• 5	•	•	•	•	•	0.632	2
6	•	•	•	•	•	0.632	2
14	•		•	•	•	0.707	1
17	•			•	•	0.408	4
18	•			•	•	0.408	4
Error	•		•				

Table 5: The modified Bank Account Example - Ranking of the statements based on SHSC and on SENDYS. The faulty statement is marked with •.

Line s	**SHSC**		**SENDYS**	
	$p'_F(s)$	**Ranking**	$p'_F(s)$	**Ranking**
• 5	0,429	1	0.476	1
6	0,000	5	0.000	5
14	0,286	2	0.262	2
17	0,143	3	0.131	3
18	0,143	3	0.131	3

combining these time complexities we are still bounded by a complexity of $O(M \cdot \Pi)$ for our combined approach.

4 Experimental Results

In order to illustrate the advantage of our proposed new approach over the basic approaches we perform a case study. Section 4.1 deals with the experimental setup by giving an overview of the implementation of the approach and introducing the subject programs by quantitatively and qualitatively describing them. Section 4.2 shows the results of the experiments.

4.1 Experimental Setup

4.1.1 Implementation

Our implementation of SENDYS works with Java programs and JUnit test cases (JUnit 3 or JUnit 4). It utilizes EMMA (`http://emma.sourceforge.net/`) for obtaining the spectra information and JavaSlicer (`http://www.st.cs.uni-saarland.de/javaslicer/`) for computing the dynamic slices. This combination is far from being perfect with regard to performance since two different approaches are used for instrumentation.

Besides the fact that the slicer in use is only a dynamic slicer [14] and not a relevant slicer [15], it has some weaknesses [16], which influence the obtained results. First of all, no slices can be computed for test cases which produce endless loops. Data dependencies can get lost when a method is called by reflection or when native code is executed. Moreover, due to the restriction to dynamic slices it is possible that the real fault is not part of a slice. However, since our tool is only for proof of concept and due to lack of a better slicer for Java, we used JavaSlicer and excluded faults leading to incomplete slices from our case study.

4.1.2 Subject programs

We investigated the 8 programs listed in Table 6. BankAccount, Mid and StaticExample are toy examples. The source code of theses programs and of ATMS is available on `http://dl.dropbox.com/u/38372651/Debugging/ExamplePrograms.zip`. The Traffic Light Example is from borrowed from the JADE project (`http://www.dbai.tuwien.ac.at/proj/Jade/`). Since this program does not have any JUnit test cases, we created some. The source code of this program is also available on `http://dl.dropbox.com/`

u/38372651/Debugging/ExamplePrograms.zip. JTopas is taken from the Software Infrastructure Repository[17]. Tcas is a Java Implementation of the aircraft collision avoidance system Tcas from the Siemens Set.

Table 6: Description of the investigated programs including the version number (V), the Non Commenting Source Statements (NCSS), the number of test cases (Test) and the number of investigated faults, which is tripartite: faults with no failing test cases, excluded faults and investigated faults.

Program	Description	V	NCSS	Test	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 dealing with static members/methods	1	16	8	1(-/-/1)
Traffic Light	Simulation system of the different phases of a traffic light	1	33	7	2(-/-/2)
ATMS	Assumption-based Truth Maintenance System	1	1573	14	3(-/1/2)
Reflection Vis.	Implementation of the Visitor-Pattern	1	338	14	5(-/-/5)
JTopas	Text parser	1	1368	127	8(4/3/1)
		2	1485	115	12(11/-/1)
		3	3931	183	14(7/4/3)
Tcas	Traffic Collision Avoidance System	1	77	1545	39(0/15/24)

JTopas includes predefined faults which are not detected by the available test cases. These faults are necessarily excluded from the case study. The slicer in use is not able to compute correct slices for all faulty program versions as already mentioned in Section 4.1.1. We excluded these program versions from our case study. Finally 42 faulty versions remain.

4.2 Results

4.2.1 SENDYS vs. basic approaches

Table 7 opposes the fault localization capabilities of SENDYS to those of the basic approaches. The FaultID is the unique identification number of the fault. In case of JTopas this number consists of the version number and the fault number. It is obvious that the combined approach performs at least as good as the best of the two basic approaches. In 25 cases the combined approach improves the fault localization precision. In 14 cases it performs as good as the better of the original approaches. Only in three cases it performs worse than SFL. 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.

Figure 4 graphically compares the fault localization capabilities of the three approaches for the 42 investigated faulty program versions. 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.

Table 8 compares the overall effectiveness of SENDYS to those of the basic approaches with respect to the median, the mean value and the standard deviation 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 median, mean value and standard deviation: 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 %.

4.2.2 SENDYS based on different similarity coefficients

We have chosen the Ochiai coefficient since studies [1, 13] have shown that it delivers better results than other coefficients. However, our approach works for other coefficients as well. Figure 5 illustrates the influence of the different similarity coefficients on the fault localization capability of SENDYS. It is obvious that SENDYS performs better than all of the basic SFL approaches. Table 9 affirms that using Ochiai in SENDYS improves its fault localization capabilites.

Table 7: Comparison of the number of statements that must be investigated using the slicing-hitting-set approach (SHSC), spectrum-based fault localization (SFL) and SENDYS. In addition, the table shows how many statements were executed in total (Stmt) in the failing test cases.

Program	**FaultID**	**Stmt**	**SHSC**	**SFL**	**SENDYS**
Bank Account	1	9	3	6	2
	2	5	1	3	1
Mid	1	6	5	1	1
Static Example	1	5	2	2	1
Traffic Light	1	25	14	2	1
	2	17	9	17	9
ATMS	1	318	226	91	46
	2	307	188	55	28
Reflection Visitor	1	35	17	20	17
	2	47	39	7	11
	3	119	43	24	15
	4	65	46	34	41
	5	48	42	5	5
JTopas	1_2	16	3	9	3
	2_3	41	7	3	1
	3_6	630	258	3	2
	3_7	665	253	36	9
	3_9	686	389	77	31
Tcas	1	25	21	3	2
	2	26	18	14	14
	3	29	18	24	17
	4	25	21	3	3
	5	29	16	22	15
	6	24	20	5	5
	9	26	22	18	13
	12	29	16	24	16
	15	29	16	22	14
	20	25	21	17	10
	21	25	21	17	15
	22	26	22	18	17
	23	26	22	18	18
	24	26	22	18	17
	25	26	22	2	2
	26	29	17	25	15
	28	30	15	14	11
	29	27	19	16	8
	30	26	18	14	8
	31	24	20	5	5
	32	25	21	5	5
	34	29	29	24	29
	35	30	15	14	9
	37	29	19	2	1

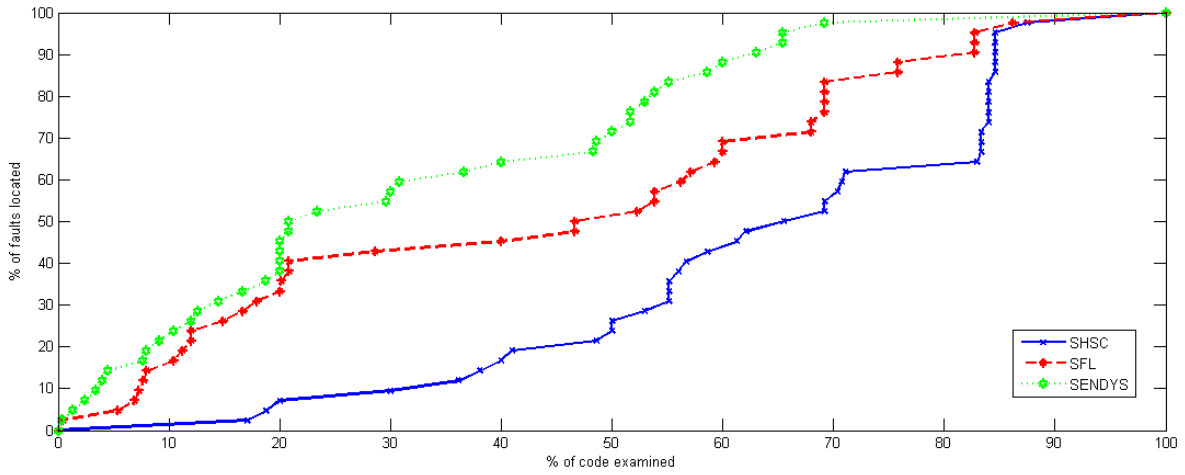

Figure 4: Comparision of SENDYS with the basic approaches with respect to their fault localization capabilities

Table 8: Overall effectiveness of SHCS, SFL and SENDYS. The total number of executed statements in the failing test cases in every faulty version act as the normalization basis.

	SHSC	**SFL**	**SENDYS**
median	67.37 %	49.49 %	22.12 %
mean	63.87 %	43.49 %	31.71 %
stdev	0.2135	0.2888	0.2400

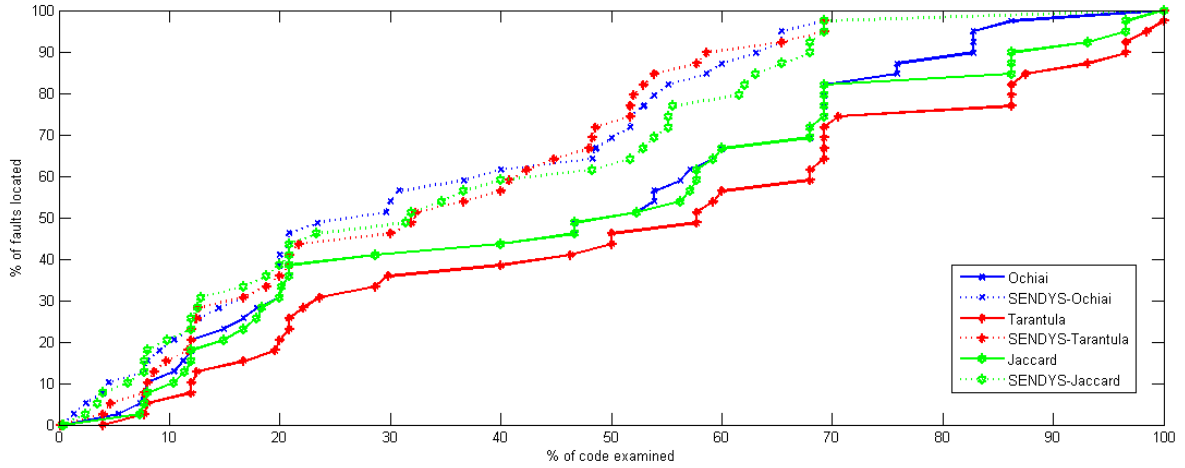

Figure 5: Comparison of SENDYS based on different similarity coefficients with the basic similarity coefficients with respect to their fault localization capabilities

Table 9: Overall effectiveness of different similarity coefficients, once standalone (B) and once as part of SENDYS (S). The total number of executed statements in the failing test cases in every faulty version act as the normalization basis.

	Ochiai		**Tarantula**		**Jaccard**	
	B	S	B	S	B	S
median	49.5 %	22.1 %	57.7 %	32.1 %	49.5 %	31.7 %
mean	43.5 %	31.7 %	52.3 %	33.7 %	46.0 %	34.8 %
stdev	0.289	0.240	0.315	0.231	0.307	0.256

5 Related Research

Weiser introduced the concept of static slicing [6]. He defined a slice as a program where zero or more statements are removed and, which 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. introduced the concept of dynamic slicing [14]. 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. But dynamic slices have one big disadvantage: They might 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 [15].

Many debugging techniques focus on the usage of failing test cases only. In contrast, spectrum-based fault localization considers both positive and negative test cases. This technique estimates the fault probability for parts of a program using the coverage information of several test cases and the result of the tests. There exist many spectrum-based approaches in literature, e.g. Tarantula [18, 19], Statistical Bug Isolation [20], Jaccard and Ochiai [21, 13]. The last-mentioned is borrowed from the molecular biology domain and it has been shown that it locates faults better than the others.

Mayer et al. [3, 4] combine spectrum-based fault localization (SFL) with model-based 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 [22] 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. We improve the results of SHSC by enhancing it with better a-priori probabilities at low additional computational cost. Our approach is similar to the previously described approach [3] 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 [23] 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 and 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. [24] 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.

There are also other approaches to automated debugging that are substantially distinct from ours because of the underlying methodology. Weimer et al. [25] and more recently Debroy et al.[26] describe the application of program mutations and genetic programming to software debugging. In order to avoid computing too many mutants the authors use focusing techniques that are based on dependences and spectrum-based methods respectively. Another very interesting approach is delta debugging [27], which can be used for reducing the size of test cases and also for identifying root causes. In contrast to delta debugging, we provide a ranked list of fault candidates instead of a single root cause candidate.

6 Conclusion

In this technical report 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 discuss the SENDYS approach in detail and compare 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 and thus 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 also 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. Moreover, the obtained empirical results also indicate that SFL provides better rankings than SHSC on average.

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 [22]. In addition we will investigate how our approach performs in the presence of multiple faults.

Acknowledgments

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] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in *PRDC '06: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 39–46.

[2] T. Janssen, R. Abreu, and A. J. 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*. New York, NY, USA: ACM, 2009, pp. 23–30.

[3] W. Mayer, R. Abreu, M. Stumptner, and A. 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, Sep. 2008.

[4] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund, "Refining spectrum-based fault localization rankings," *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 409–414, 2009.

[5] F. Wotawa, "Fault localization based on dynamic slicing and hitting-set computation," in *Proceedings of the 2010 10th International Conference on Quality Software*, ser. QSIC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 161–170. [Online]. Available: http://dx.doi.org/10.1109/QSIC.2010.51

[6] M. Weiser, "Programmers use slices when debugging," vol. 25, no. 7. New York, NY, USA: ACM, 1982, pp. 446–452.

[7] F. Wotawa, "On the relationship between model-based debugging and program slicing," *Artif. Intell.*, vol. 135, pp. 125–143, February 2002. [Online]. Available: http://portal.acm.org/citation.cfm?id=506694.506698

[8] R. Reiter, "A theory of diagnosis from first principles," *Artif. Intell.*, vol. 32, pp. 57–95, April 1987. [Online]. Available: http://portal.acm.org/citation.cfm?id=25667.25669

[9] R. Greiner, B. A. Smith, and R. W. Wilkerson, "A correction to the algorithm in reiters theory of diagnosis," *Artificial Intelligence*, 1989.

[10] F. Wotawa, "Bridging the gap between slicing and model-based diagnosis," in *SEKE*. Knowledge Systems Institute Graduate School, 2008, pp. 836–841.

[11] M. J. Harrold, G. Rothermel, R. Wu, and L. Yi, "An empirical investigation and comparison of program spectra," The Ohio State University, Tech. Rep., Nov. 1997, oSU-CISRC-11/97-TR-55.

[12] ——, "An empirical investigation of program spectra," in *Proceedings of the 1998 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, ser. PASTE '98. New York, NY, USA: ACM, 1998, pp. 83–90. [Online]. Available: http://doi.acm.org/10.1145/277631.277647

[13] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, vol. 82, no. 11, pp. 1780–1792, 2009.

[14] B. Korel and J. Laski, "Dynamic program slicing," *Inf. Process. Lett.*, vol. 29, pp. 155–163, October 1988. [Online]. Available: http://portal.acm.org/citation.cfm?id=56378.56386

[15] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," *SIGPLAN Not.*, vol. 42, no. 6, pp. 415–424, 2007.

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

[17] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[18] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*. New York, NY, USA: ACM, 2005, pp. 273–282.

[19] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 467–477.

[20] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 15–26.

[21] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An observation-based model for fault localization," in *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*. New York, NY, USA: ACM, 2008, pp. 64–70.

[22] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 128–137. [Online]. Available: http://dx.doi.org/10.1109/ASE.2008.23

[23] R. Abreu and A. J. C. van Gemund, "Diagnosing multiple intermittent failures using maximum likelihood estimation," *Artif. Intell.*, vol. 174, pp. 1481–1497, December 2010. [Online]. Available: http://dx.doi.org/10.1016/j.artint.2010.09.003

[24] J. Xu, W. Chan, Z. Zhang, T. Tse, and S. Li, "A dynamic fault localization technique with noise reduction for java programs," *Quality Software, International Conference on*, vol. 0, pp. 11–20, 2011.

[25] W. Weimer, T. V. Nguyen, C. L. Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *ACM/IEEE International Conference on Software Engineering (ICSE)*, 2009, pp. 512–521.

[26] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*. IEEE, 2010.

[27] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, feb 2002.