AI for the Win: Improving Spectrum-based Fault Localization

Birgit Hofer
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/II
8010 Graz, Austria
bhofer@ist.tugraz.at

Franz Wotawa
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/II
8010 Graz, Austria
wotawa@ist.tugraz.at

Rui Abreu
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
rui@computer.org

ABSTRACT

A considerable amount of time in software engineering is spent in debugging. In practice, mainly debugging tools which allow for executing a program step-by-step and setting break points are used. This debugging method is however very time consuming and cumbersome. There is a need for tools which undertake the task of narrowing down the most likely fault locations. These tools must complete this task with as little user interaction as possible and the results computed must be beneficial so that such tools appeal to programmers. In order to come up with such tools, we present three variants of the well-known spectrum-based fault localization technique that are enhanced by using methods from Artificial Intelligence. Each of the three combined approaches outperforms the underlying basic method concerning diagnostic accuracy. Hence, the presented approaches support the hypothesis that combining techniques from different areas is beneficial. In addition to the introduction of these techniques, we perform an empirical evaluation, discuss open challenges of debugging and outline possible solutions.

Categories and Subject Descriptors

D.2 [**Software**]: Software Engineering; D.2.5 [**Software**]: Testing and Debugging—*Debugging aids, Diagnostics*

Keywords

Spectrum-based fault localization, Model-based diagnosis

1. INTRODUCTION

Artificial intelligence (AI) techniques have been successfully used in many areas of software engineering. For example, the raise of empirical software engineering in the past decade has been mainly pushed by the application of machine learning to software repositories in order to gain new knowledge from data. In this paper, we do not discuss all the influences between AI and software engineering. Instead, we focus on automated debugging, where the aim is to provide methods and techniques for localizing and possibly correcting faults in programs with minor human intervention. In particular, we contribute the following: (1) We discuss three existing approaches that make use of AI for improving spectrum-based fault localization. (2) We perform an empirical evaluation of these approaches and compare the results. (3) We discuss open research challenges in the domain of software debugging and propose possible solutions.

Why is research in automated debugging important? One important factor of software quality is the amount of faults that are contained in a software. There exist many automatic software testing tools and bug reporting tools. Unfortunately, the programmers are often not able to correct all the bugs reported by the testers and users. They often have to prioritize the reported bugs and only correct the most critical and important bugs. Therefore, we identify software debugging as one bottleneck.

In practice, there exist numerous debugging tools. The majority of these tools allow for executing a program step-by-step and setting break points. This kind of debugging is very time consuming, since the programmer has to manually narrow down the most likely fault locations. Moreover, setting the right break points or obtaining the right information from a program in order to speed up the debugging process, is a hard task. Even worse, in many cases a programmer does not fully understand a program and can hardly find such optimal decisions. Consequently, there is a strong need for tools supporting and guiding the programmer through the fault localization process.

The purpose of this paper is to show that AI techniques can be effectively used for improving automated debugging. We start with a brief discussion of three different approaches. Thus giving also a short overview on past research in this domain. The discussion of course is rather focused and far from being complete.

The first approach deals with algorithmic debugging. Shapiro [30] was one of the first authors who provided an algorithmic basis for program debugging. In his thesis, he outlined a meta interpreter for prolog that guides the user through the debugging process while avoiding unnecessary human interactions with the system. Ten years later, Console and colleagues [7] showed that the use of the AI technique model-based diagnosis [8] can help to further reduce the need for questioning a user in order to localize a fault.

The second very interesting area for automated debugging is the domain of tutoring systems, e.g., for teaching programming. Murray [25] and Johnson and Soloway [20, 19] are two examples where planning and search respectively are used for detecting and localizing a fault in programs written by students. Debugging programs written by novices is different to fault localization of programs written by experts. For the latter, only limited knowledge of the expected behavior and the structure is available. For programs written by novices as part of their study, all information including the intended structure and possible faults is available and can be used for debugging. Hence, knowledge-based techniques are well suited for the area of tutoring systems.

More recently, Weimar et al. [31], and Debroy and Wong [9] introduced a third debugging approach - genetic debugging. Their automated debugging approaches rely on genetic programming and mutations.

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

$$
\begin{array}{c} \\ N \text{ spectra} \end{array}
\begin{array}{c} \overset{M \text{ statements}}{\overbrace{}} \\ \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1M} \\ x_{21} & x_{22} & \dots & x_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N1} & x_{N2} & \dots & x_{NM}. \end{bmatrix} \end{array}
\quad
\begin{array}{c} \text{errors} \\ \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}
$$

Figure 1: Coverage matrix

These three examples show the successful use of AI techniques in the domain of automated debugging. However, there are various others. In this paper, we focus on spectrum-based debugging [21] and present three extensions coming from AI. Each extension improves the outcome of the original method in terms of the quality of the obtained results. By quality we understand the ability of a debugging approach to present the real fault of a program to the user with the smallest number of user interactions. In particular, we rank the statements according to a value stating the likelihood of being faulty. When assuming that a user will start investigating the highest ranked statements first, the ranking indicates the required user interactions.

Spectrum-based debugging is able to debug large programs because of its small computational requirements. Beside this fact, there is another reason for taking a closer look at the combination of spectrum-based debugging and AI techniques. We feel confident that only a combination of different fault localization techniques can finally lead to a completely automated debugger that can be used in practice. This firm conviction is motivated by the observation that there are dozens of debugging approaches reported in scientific literature but there is only limited impact in practice. Often the reason lies in the complexity of the approaches, which finally leads to a bad scalability and thus prevents from being used in an interactive fashion. This conviction is shared by colleagues. For example, Mayer et al. [1] state that no single technique is able to deal with all types of faults. In their paper, the authors encourage to combine different fault localization techniques to build more accurate and robust debugging tools.

In the reminder of the paper, we discuss spectrum-based fault localization in Section 2. Afterwards, we discuss three existing techniques that combine AI with program spectra in Section 3. In Section 4, we apply the combined fault localization approaches on an example program with predefined faults and compare the fault localization capabilities of the combined approaches with those of the basic approaches. We discuss open research problems in Section 5 and related work in Section 6. Finally, we conclude the paper in Section 7.

2. SPECTRUM-BASED SOFTWARE DIAGNOSIS

This section describes spectrum-based fault localization (SFL). SFL is a statistical technique that aims at guiding software developers to find faults quickly by offering a ranking of the most probable faulty components. This ranking is created through abstractions of program traces (also known as program spectra).

2.1 Program Spectra

A program spectrum [29] is a collection of data that provides a specific view on the dynamic behavior of software. This data, collected at run-time, typically consists of a number of counters or flags for the different *components* (statements in this paper) of a program. Many different forms of program spectra exist, see [15] for an overview. The so-called (statement) hit spectra is the most common for program debugging [5, 4].

2.2 Diagnosis

The hit spectra of N runs constitute a binary coverage matrix, whose columns correspond to M different statements of the program (see Figure 1). The information in which runs an error was detected constitutes another column vector, the *error vector*. This vector can be thought to represent a hypothetical statement of the program that is responsible for all observed errors. Spectrum-based fault localization essentially consists in identifying the statement whose column vector resembles the error vector most.

In the field of data clustering, resemblances between vectors of binary, nominally scaled data, such as the columns in our matrix of program spectra, are quantified by means of *similarity coefficients* (see, e.g., [18]). Many coefficients have been studied in the past in the fault localization domain. Ochiai, widely used in the molecular biology domain, was found to be amongst the most efficient ones for software fault localization [4]. For each component j, it is defined as follows

$$\textsc{Ochiai}(j) = \frac{n_{11}(j)}{\sqrt{(n_{11}(j) + n_{01}(j)) \times (n_{11}(j) + n_{10}(j))}}$$

where $n_{11}(j)$ is the number of failed runs in which component j is involved, $n_{10}(j)$ is the number of passed runs in which component j is involved, $n_{01}(j)$ is the number of failed runs in which component j is not involved, and $n_{00}(j)$ is the number of passed runs in which component j is not involved. Formally,

$$\begin{aligned} n_{00}(j) &= |\{i \mid x_{ij} = 0 \wedge e_i = 0\}| \\ n_{01}(j) &= |\{i \mid x_{ij} = 0 \wedge e_i = 1\}| \\ n_{10}(j) &= |\{i \mid x_{ij} = 1 \wedge e_i = 0\}| \\ n_{11}(j) &= |\{i \mid x_{ij} = 1 \wedge e_i = 1\}| \end{aligned}$$

Interested readers can refer to [4] for more information on the comparison of Ochiai with other techniques and similarity coefficients.

It is assumed that a high similarity to the error vector indicates a high probability that the corresponding component of the software causes the detected error. Under this assumption, the calculated similarity coefficients rank the parts of the program with respect to their likelihood of being faulty. This information is valuable to guide the developer to the root cause of observed failures.

2.3 Advantages/Disadvantages

While none of the benefits below are unique, their combination makes spectrum-based fault localization an attractive technique for diagnosing large, resource-constrained software:

- As opposed to many approaches that only analyze one failed execution, spectrum-based fault localization exploits valuable information from both passed and failed test cases.
- As a black-box diagnosis technique, it can be applied without any additional modeling effort. In addition, many of today's systems (e.g., concurrent systems) are difficult to model.

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

- The technique can easily be integrated with existing testing procedures, such as overnight playback of recorded usage scenarios.
- It requires less computation time than other fault localization approaches. The time complexity of spectrum-based fault diagnosis is $O(N)$ for executing N test cases, $O(N)$ to compute the similarity coefficient per component and thus for computing the similarity coefficients for all M statements $O(M \cdot N)$, and finally $O(M \cdot \log M)$ to rank the statements in the diagnostic report in order of their likelihood to be faulty. Therefore, the overall time complexity is $O(N + M \cdot N + M \cdot \log M)$.
- The space complexity is basically low with $O(M \cdot N)$ to store the coverage matrix. These storage costs can be further optimized.

Despite the efforts to advance spectrum-based fault localization to efficiently aid programmers to pinpoint the root cause of observed failures, it still has a few drawbacks:

- Spectrum-based fault localization only exploits the topology of the software. Thus, low granularity components (e.g., statement level) will yield the best diagnostic performance, whereas coarser grain granularity may guide the developer to inspect healthy components.
- Due to its statistical foundation, the diagnostic accuracy is inherently limited: (1) the accuracy is rather dependent on the number and quality of the test cases, (2) it cannot reason over multiple faults, like model-based diagnosis approaches.
- The similarities are not probabilities: this hampers the analysis of the ranking using probabilistic methods (e.g., entropy).

3. HOW TO COMBINE AI WITH SFL

In this section, we answer the question how combinations of AI techniques and program spectra can help to improve fault localization. In particular, we discuss three existing approaches that combine spectrum-based fault localization with AI techniques and explain their underlying algorithms.

3.1 Refining Spectrum-based Rankings

Although spectrum-based fault localization (SFL) has been shown to be efficient and effective [4], its diagnostic accuracy is inherently limited, since the semantics of statements is not taken into account. In particular, greatly due to the statistical nature of the technique, statements that exhibit identical execution patterns cannot be distinguished amongst themselves. To enhance its diagnostic quality, in [1] spectrum-based fault localization was combined with a model-based debugging approach (MBSD) [24] within a framework coined DEPUTO. MBSD has its roots in AI and is based on abstract interpretation. Essentially, MBSD is used to refine the ranking obtained from the spectrum-based method. MBSD considers the program's semantics and can thus filter out those components that do not explain the observed failures.

Algorithm 1 outlines the combined approach. The algorithm executes in three stages, with the SFL approach used in the setup stage (lines 1 to 6), feeding into the subsequent model-based filtering stage (lines 7 to 16), followed by an optional best-first search stage (lines 17 to 24). This combination has significantly lower resource requirements than applying MBSD on the whole program and using SFL only to rank results as proposed in [23]. In the following paragraphs, we explain the algorithm step by step.

First the setup is performed. For this, the program $\mathcal{P}$ is split into a set of components $\mathcal{C}$ and the available test cases $\mathcal{T}$ are executed on $\mathcal{P}$ to obtain the participation matrix $\mathcal{M}$. Using $\mathcal{M}$, we split $\mathcal{T}$ into passing tests ($\mathcal{T}_P$) and failing ones ($\mathcal{T}_F$). From $\mathcal{M}$, a sorted list of components $\mathcal{R}$ in order of likelihood to be at fault is obtained using spectrum-based fault localization (line 4).

Next, the model-based filtering stage is performed. MBSD (line 9) is used to eliminate the top-ranked candidate explanations that are not considered as valid explanations for the fault. Instead of computing *all* explanations at once, an incremental strategy permits to stop early once a fault has been identified. First, the set of components $\widehat{\mathcal{C}}$ with the highest similarity coefficient in $\mathcal{R}$ are obtained using the RANKING_POP($\mathcal{R}$) function, which also removes all elements in $\widehat{\mathcal{C}}$ from $\mathcal{R}$. Second, the function MBSD($\widehat{\mathcal{C}}, \mathcal{T}_F$) returns a set of candidate explanations $\mathcal{D} \subseteq \widehat{\mathcal{C}}$ that explain observed failures in $\mathcal{T}_F$. Finally, if the fault is in the returned set, the algorithm stops; otherwise none of the candidates represent a valid explanation and other candidate explanations must be generated. The algorithm stops once no more explanations can be found or if none of the remaining components was executed in any failing test case. $\mathcal{S}$ is the set of components that are implicated by SFL but not by MBSD.

If no explanation is found in stage 2, a best-first search procedure is employed that traverses the program along dependencies between components with decreasing fault similarity. No explanation may be found if a fault in the program has a larger cardinality than the MBSD threshold or if the fault affects component inter-dependencies such that the fault assumptions and model abstraction can no longer represent the fault. In line 18, the set of components with maximum fault similarity that are connected to the previously explored components is returned. The function PDG_RANKING_POP($\mathcal{S}, \mathcal{I}$) returns the set of components in S with the highest similarity that are directly connected to the previously inspected set of component $\mathcal{I}$. If the component is confirmed to be (part of) a valid explanation, the search stops and the diagnosis is returned.

DEPUTO has been shown to outperform not only the individual approaches themselves, but also other state-of-the-art automated debugging techniques (e.g., Delta Debugging). In particular, 60 % diagnostic performance improvement over spectrum-based fault localization has been reported [1]. The major drawback of the approach is that, similar to model-based diagnosis approaches, it does not scale to large software systems because the model-based approach tries to analyze the whole source code.

3.2 Spectrum-based Reasoning

Model-based diagnosis approaches deduce component failure through logic reasoning using propositional models of component behavior [8, 34]. An inherent, strong point of model-based diagnosis is that it reasons in terms of multiple faults. In contrast to the simple component ranking of spectrum-based fault localization, the diagnostic report of model-based diagnosis approaches contains multiple-fault candidates. Thus, such a report provides more diagnostic information than a one-dimensional ranking. Furthermore, the ranking is determined in terms of (multiple) fault probabilities, a more solid basis for a candidate ranking than statistical similarity. While reasoning approaches are inherently more accurate than statistical approaches, they have two main disadvantages: (1) They need a model. These models are usually generated with the help of static analysis. However, static

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

Algorithm 2 BARINEL Algorithm

Input: Activity matrix A, error vector e,

Output: Diagnostic Report D

```
1: γ ← ϵ
2: D ← STACCATO((A, e))          ▷ Compute MHS
3: for all d_k ∈ D do
4:     expr ← GENERATEPR((A, e), d_k)
5:     i ← 0
6:     Pr[d_k]^i ← 0
7:     repeat
8:         i ← i + 1
9:         for all j ∈ d_k do
10:            g_j ← g_j + γ · ∇expr(g_j)
11:        end for
12:        Pr[d_k]^i ← EVALUATE(expr, ∀_{j∈d_k} g_j)
13:    until |Pr[d_k]^{i−1} − Pr[d_k]^i| ≤ ξ
14: end for
15: return SORT(D, Pr)
```

Algorithm 1 DEPUTO Algorithm

Input: Program $\mathcal{P}$, set of test cases $\mathcal{T}$

Output: Fault assumptions explaining failed test runs

```
1: C ← CREATECOMPONENTS(P)
2: M ← GETCOMPONENTMATRIX(C, P, T)
3: ⟨T_P, T_F⟩ ← PARTITION(M, T)
4: R ← SFL(M)                    ▷ Apply SFL
5: S ← ∅                         ▷ Skipped components
6: I ← ∅                         ▷ Inspected components
7: repeat
8:     Ĉ ← RANKING_POP(R)
9:     D ← MBSD(Ĉ, T_F)          ▷ Apply MBSD
10:    I ← I ∪ C
11:    if D_bug ∈ D is confirmed faulty then
12:        return D_bug
13:    else
14:        S ← S ∪ (Ĉ \ D)
15:    end if
16:    R ← R \ Ĉ
17: until R = ∅
18: while S ≠ ∅ do
19:    Ĉ ← PDG_RANKING_POP(S, I)
20:    I ← I ∪ Ĉ
21:    if C_bug ∈ Ĉ is confirmed faulty then
22:        return {¬h_bug}
23:    end if
24: end while
25: return ∅                     ▷ No explanation found
```

analysis is unable to capture dynamic data dependencies / conditional control flow. (2) The generation of diagnosis candidates comes with exponential costs. This typically prohibits the use of reasoning approaches for programs larger than a few hundred lines [24].

An approach called BARINEL [5] models program behavior in terms of program spectra. It employs a Bayesian approach to deduce multiple-fault candidates and their associated probabilities. Therefore, it yields a probabilistic, information-rich diagnostic report. BARINEL has been shown to be very effective, as it outperforms most state-of-the-art techniques for fault localization. Performance improvements up to 54 % have been observed [5].

BARINEL is detailed in Algorithm 2 and comprises three main phases. In the first phase, a list of candidates D is computed from an activity matrix A and an error vector e using STACCATO, a ultra-low cost algorithm for computing diagnosis candidates [2]. The required performance is achieved at the cost of completeness, because solutions are truncated at 100 candidates. Nevertheless, experiments [2, 3] have shown that no significant solution was ever missed.

In the second phase, $\Pr(d_k)$ (probability that a given candidate d_k is faulty) is computed for each candidate in D. First, GENERATEPR derives for every candidate d_k the formula to compute d_k's fault probability given the current set of observations (A, e). Subsequently, the algorithm estimates the *health* probability h_j (i.e., probability that a faulty component j still behaves as expected) such that the total health probability is maximized. To solve the maximization problem, we apply a simple gradient ascent procedure [6] bounded within the domain $0 \leq h_j \leq 1$ (the ∇ operator signifies the gradient computation). As the $\Pr(d_k)$ expressions that need to be maximized are simple, and the domain is bounded to $[0, 1]$, the gradient ascent procedure exhibits reasonably rapid convergence for all M and C.

In the third and final phase, for each d_k the diagnoses are ranked according to $\Pr(d_k)$, which is computed by the EVALUATE function based on the usual Bayesian update for each row. Note that this approach is independent of test case ordering. We refer interested readers to [5].

3.3 Spectrum Enhanced Dynamic Slicing

Another method that combines SFL with AI is Spectrum ENhanced DYnamic Slicing (SENDYS) [16]. It uses the similarity coefficients computed by spectrum-based fault localization as a-priori fault probabilities in model-based debugging. A lightweight model-based-debugging technique based on dynamic slicing, namely the slicing-hitting-set-approach (SHSC) [33], is applied in SENDYS. SHSC computes the dynamic slices of all faulty variables in all failed test cases, derives the minimal diagnoses from the slices and computes the fault probabilities of the single statements based on the size and the amount of the diagnoses which contain the statement. Since SHSC only considers negative test cases, it has a major disadvantage: Statements that are contained in many slices, e.g. constructor statements, are always ranked high.

By the use of program spectra, this undesirable effect is eliminated. It is assumed that statements covered by many positive test cases are less likely to be faulty than those covered by negative test cases only. Similarity coefficients, e.g. Ochiai, condense the information of program spectra. Statements that are covered by both positive and negative test cases have a smaller similarity coefficient than statements covered by negative test cases only.

SENDYS is a simple technique to combine SHSC with program spectra. SENDYS computes the similarity coefficients from the program spectra, normalizes them and passes them to the SHSC approach as a-priori probabilities. Algorithm 3 illustrates this process in detail. First, the program $\mathcal{P}$ is split into its components $\mathcal{C}$. The coverage matrix $\mathcal{M}$ is computed by executing all test cases $\mathcal{T}$ on program $\mathcal{P}$. From $\mathcal{M}$ the similarity coefficients $\mathcal{R}$ are computed (line 3) and normalized. The normalized similarity coefficients are assigned to $\widehat{\mathcal{R}}$ (line 4). The function ALLDIAGNOSES$(\mathcal{P}, \mathcal{T})$ computes the slices for all faulty variables in all failing test cases.

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

Algorithm 3 SENDYS Algorithm

Input: Program $\mathcal{P}$, set of test cases $\mathcal{T}$

Output: List of possible fault locations sorted after their fault likelihood

1: $\mathcal{C} \leftarrow$ CREATECOMPONENTS($\mathcal{P}$)
2: $\mathcal{M} \leftarrow$ GETCOMPONENTMATRIX($\mathcal{C}, \mathcal{P}, \mathcal{T}$)
3: $\mathcal{R} \leftarrow$ SFL($\mathcal{M}$)
4: $\widehat{\mathcal{R}} \leftarrow$ NORMALIZE($\mathcal{R}$)
5: $\mathcal{D} \leftarrow$ ALLDIAGNOSES($\mathcal{P}, \mathcal{T}$)
6: **for** all $d \in \mathcal{D}$ **do**
7: $\quad pd[d] \leftarrow \prod_{s \in d} \widehat{\mathcal{R}}[s] \times \prod_{s' \in \mathcal{P} \setminus d}(1 - \widehat{\mathcal{R}}[s'])$
8: **end for**
9: **for** all $s \in \mathcal{P}$ **do**
10: $\quad ps[s] \leftarrow \sum_{d \in \mathcal{D} \wedge s \in d} pd[d]$
11: **end for**
12: $\overline{ps} \leftarrow$ SORT(NORMALIZE(ps))
13: **return** $\overline{ps}$

The resultant slices are combined to diagnoses by means of the corrected Reiter algorithm [13] and assigned to $\mathcal{D}$ (line 5). The diagnosis fault probabilities pd indicate for all diagnoses d the probability that d contains a fault. The value of $pd[d]$ arises from two products: first, the product of the fault probabilities $\widehat{\mathcal{R}}$ of the statements contained in d and second, the product of the counter probabilities $(1-\widehat{\mathcal{R}})$ of the statements that are not contained in d (line 7). In the next step, the statement fault probabilities are computed by mapping back the diagnosis fault probabilities pd to the statements. This is done by building the sum of all diagnosis fault probabilities which contain the statement (line 9). The statement fault probabilities ps are normalized, sorted after their size and assigned to $\overline{ps}$. Finally the normalized statement fault probabilities $\overline{ps}$ are returned.

This combination is simple, but very effective. It has been shown that the fault localization capabilities of Ochiai can be improved by 25 % and those of the slicing-hitting-set-approach by 50 % [16, 17].

4. EMPIRICAL RESULTS

We perform a small case study in order to show the advantage of combining spectrum-based fault localization with AI techniques. For this, we run the basic approaches and the combined approaches on Tcas. Tcas is an implementation of the traffic collision avoidance system and is taken from the Siemens Set [10]. Since the prototype implementations of the previously presented approaches are implemented for different programming languages, we use the original `C` version of Tcas for evaluating DEPUTO and BARINEL and a Java implementation for evaluating SENDYS. The `C` version comprises 138 lines of code (105 Non Commenting Source Statements, NCSS) and 1608 test cases. The Java version comprises 77 NCSS and 1545 test cases. Both program variants have the same faulty program versions. A fault consists of one to three faulty code lines.

There exist rank- [4, 21] and dependency-based [28, 24, 22] metrics in order to compare the quality of fault localization approaches. The former quantify the quality of a result based on the ranking position of the faulty component relative to all components. They are mainly used with techniques that rank components. In contrast, dependency-based measures are mainly applied to evaluate techniques that either do not rank components (for example MBSD) or do not rank all components of a program (such as SOBER [22]). Dependency-based measures typically operate on the program dependence graph (PDG). Essentially, starting with the set of blamed components, dependencies between components are traversed in breadth-first order until the fault has been reached. The quality of a diagnostic report is measured as the fraction of the PDG that is traversed. Both metrics quantify the percentage of a program that needs to be inspected in order to find the fault.

We make use of the SCORE metric in order to compare the previously described fault localization approaches. The SCORE is defined as

$$\text{SCORE} = \frac{|I|}{M} \cdot 100\%$$

where $|I|$ denotes the number of statements inspected and M denotes the number of components in total.

Figure 2 gives an overview of the fault localization capabilities of the discussed fault localization approaches. It plots the percentage of located faults in terms of the percentage of inspected code (i.e., effort to find the root cause). Since different Tcas versions (Java and `C`) were used, the percentage of inspected code has as basis the number of Non Commenting Source Statements (105 for the `C` version, 77 for the Java version). We only compare the Tcas variants where all approaches could produce results.

From this figure, it can be observed that the combined approaches largely outperform their basic approaches. In some cases, the basic techniques outperform the combined approaches. However, on average, the combined approaches perform better than their basic approaches. DEPUTO has the best fault localization capabilities. The basic approaches SFL (SCORE: 18.3 %) and SHSC (SCORE: 25.4 %) perform worst.

When using DEPUTO on average, only 5.6 % of the source code must be investigated until the fault location is found. When using MBSD, 10.9 % must be investigated. However, DEPUTO and MBSD are only suited for small programs, because they do not scale to large software systems.

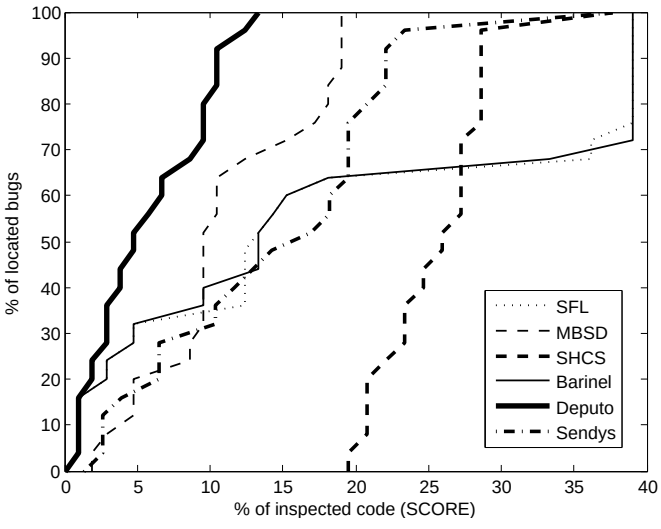

Figure 2: Comparison of Deputo, Barinel, and Sendys with their basic approaches in terms of the amount of code that must be investigated.

Due to their (comparable) time/space complexity, when debugging larger system, BARINEL and SENDYS are acceptable alternatives. BARINEL has an average SCORE of 18.1 %. BARINEL is

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

known to outperform related work when applied to several other - and larger - programs. For interested readers, refer to [5] for further details on the empirical evaluation's results.

SENDYS has an average SCORE of 14.3 %. SHSC is a low level variant of MBD which is in general known as computational complex. It comes with low computational costs, but is less precise than other MBD techniques. 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. We refer interested readers to [16] for more empirical evaluation results.

5. DISCUSSION

The previously presented techniques are a valuable support for programmers when debugging. However, the number of statements a programmer needs to inspect can still be prohibitive for real and large systems. Subsequently there is still room for improvements of the fault localization capabilities. In the following, we discuss the open challenges and propose solutions.

- *Computational complexity / scalability problem:* Many debugging approaches that potentially allow for improving debugging substantially come with a high computational complexity. Therefore, using such techniques is not feasible for larger programs. Approaches with a small computational overhead are less suitable for smaller programs or program fragments such as methods because their computed results are less accurate. Hence, a combination of these techniques is required which first tears down the search space and then switches to a more complex debugging approach at the proper point in time during the debugging process.

  As mentioned before, DEPUTO suffers from a scalability problem, which hampers its applicability to real systems. The following can however alleviate this rather important scalability problem. Being more lightweight than model-based approaches, spectrum-based fault localization can first be used at a, say, method/function level of granularity. Once the diagnostic ranking is computed, model-based diagnosis can be used to expand the ranking with the statements in the method body that should be inspected. For example, this can be done for the top 5 % of the components returned by spectrum-based fault localization.

- *Testing abort criteria:* Besides the important factor of reasoning in terms of multiple-faults, the BARINEL approach to spectrum-based reasoning, paves the way for several other opportunities. For instance, as the calculated ranking is based on Bayesian probabilities, an entropy-based confidence level can be computed. This confidence level can then be used to automatically decide whether one should continue testing (and eventually which tests give a higher information gain [12]) or if there is enough confidence that the ranking will indeed help developers to quickly find faults.

  Spectrum-based fault localization is rather dependent on the number and quality of the test cases in the test suite. One could imagine to automatically generate tests cases which yield maximal information gain (e.g., information gain would be used as the fitness function of a genetic algorithm). Such approach would reduce the costs of executing test cases, while improving the diagnostic quality of the fault localization technique. In addition, the automatic generation of test cases may increase the size of the test suite to the point that it becomes prohibitive to execute all test cases. Therefore, AI techniques (e.g., constraint satisfaction problem) can be used to reduce the size of the suite while still maintaining full coverage, and also prioritize the tests within the suite [12].

- *User interface:* Although, a fully automated debugger should not require user interactions, it is unlikely that such a tool can exist. There is always the need for asking about expectations regarding the program behavior. An automated oracle would require a formal specification of the whole program, which rarely will be available for all programs in the close future. Hence, a user interface to a debugger that minimizes user interactions and maximizes the benefit for the user is required. Research in this direction is desired.

- *Missing source code:* There exist faults, especially missing code, which cannot be detected by most of the debugging techniques. Spectrum-based fault localization combined with AI is able to point to areas which most likely contain the fault. However, it does not advise the programmer that the fault might be caused by missing code. A combination of the previously presented techniques with debugging based on mutations [31], [9] might solve this problem: First, the search space is narrowed down by means of Spectrum-based fault localization combined with AI. The highest ranked fault locations, e.g., the highest 5 % of the statements, are given to a mutation engine which randomly deletes, modifies and inserts statements. Each mutant is checked against the available test cases. If a mutant passes all test cases it is returned to the programmer as a possible fix of the fault. A fault localization at the best is an indispensable preprocessing step for mutation based debugging since the search space of debugging based on mutations would be otherwise too large in order to find a solution for large programs in an acceptable amount of time.

There is still a long way to go for providing the enabling theories and technologies for automated debugging. Combining AI methods with debugging methods seems to be a very fruitful path in the right direction.

6. RELATED WORK

Apart from the techniques detailed in this paper, many other techniques have been proposed in the past. Several systems employing dynamic analysis techniques for fault localization are present in the literature, most of them exploit the same reasoning as spectrum-based fault localization. Well-known examples of such approaches are Tarantula [21], Sober [22], CrossTab [32], and MKBC [36].

Machine learning techniques have been applied to programs [35] and their executions [26] to infer likely invariants that must hold at particular locations in a program. Violations can subsequently be used to detect potential errors. Model-based approaches, such as model-based software debugging, provide more reliable behavior than [26], since success of the trace analysis highly depends on the test runs and the type of invariants to be inferred.

Combining program execution and symbolic evaluation has been proposed to infer possible errors [11]. Similar to MBSD, a symbolic, under-constrained representation of a program execution and memory structures are built. Instead of using fault probabilities to guide diagnosis, only those candidate explanations that definitively imply a test failure are flagged. Hence, the tool complements our approaches by highlighting a subset of all provable

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784

faults in a program, while our approaches aim at identifying those program fragments that may contribute to a fault.

In model-based reasoning, the program model is typically generated from the source code, as opposed to the traditional application of model based diagnosis where the model is obtained from a formal specification of the (physical) system [27]. In [24] an overview of different models for model-based software debugging is given, concluding that the model generated by means of abstract interpretation (the one used in DEPUTO) leads to good results while not suffering from the computational complexity inherent to more precise analysis techniques [24]. Other approaches that fit into this category include `explain` and Δ-slicing [14], which are based on comparing execution traces of correct and failed runs using model checkers.

7. CONCLUSION

In this paper, we discussed the beneficial use of AI techniques in the domain of automated debugging. In particular, we showed how to improve spectrum-based fault localization. All of the mentioned improvements aim in providing a better ranking of fault candidates in order to further reduce the number of statements to be examined during debugging. All of the presented approaches make use of ideas originating from model-based diagnosis. In the empirical section, we showed that DEPUTO performs best. However, since DEPUTO does not scale to large programs, BARINEL and SENDYS are good alternatives. As a consequence, we strongly recommend the use of AI techniques for automated debugging. Moreover, we discussed open challenges of automated debugging and outlined possible solutions.

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).

8. REFERENCES

[1] R. Abreu, W. Mayer, M. Stumptner, and A. J. C. van Gemund. Refining spectrum-based fault localization rankings. In R. L. Wainwright and H. Haddad, editors, *Proceedings of the 24th Annual ACM Symposium on Applied Computing (SAC'09)*, pages 409–414, Honolulu, Hawaii, USA, 8 – 12 March 2009. ACM Press.

[2] R. Abreu and A. J. C. van Gemund. A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In V. Bulitko and J. C. Beck, editors, *Proceedings of the 8th Symposium on Abstraction, Reformulation and Approximation (SARA'09)*, Lake Arrowhead, California, USA, 8 – 10 July 2009. AAAI Press.

[3] R. Abreu and A. J. C. van Gemund. Diagnosing multiple intermittent failures using maximum likelihood estimation. *Artificial Intelligence*, 174(18):1481–1497, 2010.

[4] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems & Software (JSS)*, 2009.

[5] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In G. Taentzer and M. Heimdahl, editors, *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*, Auckland, New Zealand, 16 – 20 November 2009. IEEE Computer Society, to appear.

[6] M. Avriel. *Nonlinear Programming: Analysis and Methods*. Dover Publishing, Mineola, New York, USA, 2003.

[7] L. Console, G. Friedrich, and D. T. Dupré. Model-based diagnosis meets error diagnosis in logic programs. In *Proc. of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1494–1499, Chambery, August 1993.

[8] J. de Kleer and B. C. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[9] 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.

[10] 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*, 10(4):405–435, 2005.

[11] D. Engler and D. Dunbar. Under-constrained execution: making automatic code destruction easy and scalable. In *Proceedings of the 2007 international symposium on Software testing and analysis*, ISSTA '07, pages 1–4, New York, NY, USA, 2007. ACM.

[12] A. González-Sanchez, R. Abreu, H.-G. Gross, and A. J. C. van Gemund. Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 83–92, 2011.

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

[14] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, June 2006.

[15] M. Harrold, G. Rothermel, R. Wu, and L. Yi. An empirical investigation of program spectra. *ACM SIGPLAN Notices*, 33(7), 1998.

[16] B. Hofer and F. Wotawa. Spectrum enhanced dynamic slicing for better fault localization. In *The 20th European Conference on Artificial Intelligence (accepted for publication)*, 2012.

[17] B. Hofer and F. Wotawa. Spectrum enhanced dynamic slicing for fault localization. Technical Report IST-DR-2012-01, Institute for Software Technology, Graz University of Technology, 2012.

[18] A. Jain and R. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[19] W. L. Johnson. *Intention-Based Diagnosis of Novice Programming Errors*. Pitman, London, 1986.

[20] W. L. Johnson and E. Soloway. PROUST: Knowledge-Based Program Understanding. *IEEE Transactions on Software Engineering*, 11(3):267–275, Mar. 1985.

[21] 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.

[22] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff. Statistical debugging: A hypothesis testing-based approach. *IEEE Transactions on Software Engineering (TSE)*, 32(10):831–848, 2006.

[23] W. Mayer, R. Abreu, M. Stumptner, and A. J. C. van Gemund. Prioritizing model-based debugging diagnostic reports. In A. Grastien, M. Stumptner, and W. Mayer, editors, *Proceedings of the 19th International Workshop on Principles of Diagnosis (DX'08)*, pages 127–134, Blue

Mountains, New South Wales, Australia, 22 – 24 September 2008.

[24] W. Mayer and M. Stumptner. Evaluating models for model-based debugging. In A. Ireland and W. Visser, editors, *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, pages 128–137, L'Aquila, Italy, 15 – 19 September 2008. ACM Press.

[25] W. R. Murray. *Automatic Program Debugging for Intelligent Tutoring Systems*. Pitman, London, 1988.

[26] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants: Integrating daikon and esc/java. *Electr. Notes Theor. Comput. Sci.*, 55(2):255–276, 2001.

[27] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, Apr. 1987.

[28] M. Renieris and S. P. Reiss. Fault localization with nearest neighbor queries. In J. Grundy and J. Penix, editors, *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE'03)*, pages 30–39, Montreal, Canada, 6 – 10 October 2003. IEEE Computer Society.

[29] T. W. Reps, T. Ball, M. Das, and J. R. Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In M. Jazayeri and H. Schauer, editors, *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC / SIGSOFT FSE)*, volume 1301 of *Lecture Notes in Computer Science*, pages 432–449, Zurich, Switzerland, 22 – 25 September 1997. Springer-Verlag.

[30] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[31] 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)*, pages 512–521, 2009.

[32] E. Wong, T. Wei, Y. Qi, and L. Zhao. A crosstab-based statistical method for effective fault localization. In *Proceedings of the 2008 International Conference on Software Testing, Verification, and Validation*, ICST '08, pages 42–51, Washington, DC, USA, 2008. IEEE Computer Society.

[33] F. Wotawa. Fault localization based on dynamic slicing and hitting-set computation. In *Proceedings of the 10th International Conference on Quality Software (QSIC 2010)*, pages 161–170, 2010.

[34] F. Wotawa, M. Stumptner, and W. Mayer. Model-based debugging or how to diagnose programs automatically. In T. Hendtlass and M. Ali, editors, *Proceedings of IAE/AIE 2002*, volume 2358 of *LNCS*, pages 746–757, Cairns, Australia, 17 – 20 June 2002. Springer-Verlag.

[35] Y. Xie and D. R. Engler. Using redundancies to find errors. *IEEE Trans. Soft. Eng.*, 29(10):915–928, 2003.

[36] J. Xu, W. K. Chan, Z. Zhang, T. H. Tse, and S. Li. A dynamic fault localization technique with noise reduction for java programs. In *Proceedings of the 11th Int. Conference on Quality Software (QSIC 2011)*, pages 11–20, 2011.

DOI: 10.1145/2382756.2382784 http://doi.acm.org/10.1145/2382756.2382784