A Multiple Fault Localization Approach based on Multicriteria Analytical Hierarchy Process
Noureddine Aribi, Nadjib Lazaar, Yahia Lebbah, Samir Loudni, Mehdi Maamar

To cite this version:

HAL Id: lirmm-02089746
https://hal-lirmm.ccsd.cnrs.fr/lirmm-02089746
Submitted on 4 Apr 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.
A Multiple Fault Localization Approach based on Multicriteria Analytical Hierarchy Process

Noureddine Aribi*, Nadjib Lazaar†, Yahia Lebbah*, Samir Loudni‡, Mehdi Maamar*

*LITIO – University of Oran 1, 1524 El-M’Naouer, 31000 Oran – Algeria
†LIRMM – University of Montpellier, CNRS, 161 rue Ada, 34090 Montpellier – France
‡CNRS, UMR 6072 GREYC – University of Caen Normandy, 14032 Caen – France

Abstract—Fault localization problem is one of the most difficult processes in software debugging. Several spectrum-based ranking metrics have been proposed and none is shown to be empirically optimal. In this paper, we consider the fault localization problem as a multicriteria decision making problem. The proposed approach tackles the different metrics by aggregating them into a single metric using a weighted linear formulation. A learning step is used to maintain the right expected weights of criteria. This approach is based on Analytic Hierarchy Process (AHP), where a ranking is given to a statement in terms of suspiciousness according to a comparison of ranks given by the different metrics. Experiments performed on standard benchmark programs show that our approach enables to propose a more precise localization than existing spectrum-based metrics.

Index Terms—Fault Localization; Spectrum-based Fault Localization; Multiple Fault; Multicriteria decision making; AHP

I. INTRODUCTION

Developing software programs is universally acknowledged as an error-prone task. The major bottleneck in software debugging is how to identify where the bugs are [1], this is known as fault localization problem. Nonetheless, locating a fault is still an extremely time-consuming and tedious task. Over the last decade, several automated techniques have been proposed to tackle this problem.

Spectrum-based approaches. Spectrum-based fault localization (SBFL) (e.g. [2], [3]) is a class of popular fault localization approaches that take as input a set of failing and passing test case executions, and then highlight the suspicious program statements that are likely responsible for the failures. A ranking metric is used to compute a suspiciousness score for each program statement based on observations of passing and failing test case execution. The basic assumption is that statements with high scores, i.e. those executed more often by failed test cases but never or rarely by passing test cases, are more likely to be faulty. Several ranking metrics have been proposed to capture the notion of suspiciousness, such as TARANTULA [4], OCHIAI [5], and JACCARD [5]. The ultimate objective of SBFL is to have a metric able to always rank first the faulty statements. In practice, we are very far from this ideal [6]. SBFL metrics do not rely on a particular model of the program under test and thus, they are easy to use and practical in the presence of CPU time and memory resources constraints. SBFL metrics give different interpretation of suspiciousness degree. In addition, the semantics of statements and the dependencies are not taken into account. Thus, the accuracy of SBFL approaches is inherently limited.

Multiple fault programs. Most of current localization techniques are based on the single fault hypothesis. By dismissing such assumption, faults can be tightly dependent in a program, giving rise to numerous behaviours [7]. Thus, it makes the localization process difficult for multiple fault approaches [8]–[10]. The main idea of these approaches is to make a partition on test cases into fault-clusters where each one contains the test cases covering the same fault. The drawback is that a test case can cover many faults with overlapping clusters, which leads to a rough localization. Another idea consists in localizing one fault at a time [11]. Here, we start by locating a first fault, then correct it (which is an error-prone step), generate again new test cases, and so on until no fault remains in the program.

Artificial Intelligence based approaches. In the last decade, fault localization was abstracted as a data mining (DM) problem. Podgurski et al. present a method to automatically group faulty spectra with respect to the fault that leads to the failure [9]. This method relies on cluster analysis. Cellier et al. [12], [13] propose a combination of association rules and Formal Concept Analysis (FCA) to assist in fault localization. In [14], [15], the authors formalize the problem of fault localization as a closed pattern mining problem. A constraint programming model, with CLOSED PATTERN global constraint, is used to extract the k best patterns in terms of suspiciousness degree. Other approaches tackle fault localization as a supervised learning problem [16], [17].

Multicriteria decision making. Many real-world decision problems involve several criteria. As soon as multiple conflicting criteria are considered in the evaluation of a decision, the notion of optimality is not workable, since no criterion is systematically better than all the others. In this context, the Multicriteria decision-making (MCDM) [18] provides a systematic approach to characterize and find the most-preferred trade-off solutions. While many preference models have been proposed in the literature, the additive preference model is the most commonly used in MCDM. It consists to aggregate additively the criteria into a single criterion, so as to take advantage of all advanced techniques in solving single-criterion optimization problems. A first difficult step in this direction consists to find the right weights. Fortunately, there are some existing methods that tackle, either directly or
indirectly, weights elicitation problem, in particular the AHP method described in Section III-B.

In this paper, we consider the fault localization problem as a multicriteria decision making problem. The proposed approach tackles the different metrics by aggregating them into a single metric using a weighted linear formulation. We propose a passive/active learning step to maintain the right expected weights of criteria. This approach is based on Analytic hierarchy Process (AHP), where a ranking is given to a statement in terms of suspiciousness according to a comparison of ranks given by the different metrics III-B. The approach is implemented in AHP-LOC. Experiments performed on standard benchmark programs show that our approach enables to propose a more precise localization than existing spectrum-based metrics.

This paper is organized as follows. Section 2 presents related work. Section 3 recalls preliminaries. Section 4 describes our approach. Section 5 illustrates the approach on a small program. Section 6 reports experimental results and a complete comparison with AHP-LOC and SBFL metrics. Finally, we conclude this work in Section 7.

II. RELATED WORK

To the best of our knowledge, Xuan and Monperrus propose in 2014 the first and unique work combining multiple ranking metrics [19]. ULTRIC is based on a passive learning process acting on multiple ranking metrics. ULTRIC consists of two phases: learning and ranking. It combines different ranking metrics in a weighted model. The learning phase is a passive one using a training set. The training set corresponds to a set of pairs of statements with their spectra. From each already dealt one using a training set. The training set corresponds to a set of suspiciousness metrics shares the same intuition: the more the statement is executed by passing test cases, the more suspicious the statement is considered. Fig.1 shows the suspiciousness spectrum of the different metrics according to the whole of suspiciousness metrics.

AHP-LOC has two distinguishing elements w.r.t. ULTRIC. Firstly, AHP-LOC is an adaptive approach based on an active learning makes it able to start with an empty training set. Secondly, AHP-LOC benefits from multicriteria AHP in the aggregation of different metrics.

A. Fault Localization

Let us consider a faulty program $P_i$ having $n_i$ lines, labeled $e_{i,1}$ to $e_{i,n_i}$. A test case $tc_{i,j}$ is a tuple $(D_{i,j}, O_{i,j})$, where $D_{i,j}$ is the input data and $O_{i,j}$ is the expected output. Let $(D_{i,j}, O_{i,j})$ a test case and $A_{i,j}$ be the current output returned by a program $P_i$ after the execution of its input $D_{i,j}$. If $A_{i,j} = O_{i,j}$, $tc_{i,j}$ is considered as a passing (i.e. positive), failing (i.e. negative) otherwise. A test suite $T_i = \{tc_{i,1}, tc_{i,2}, ..., tc_{i,m_i}\}$ is a set of $m_i$ test cases to check whether the program $P_i$ follows a given set of requirements.

A test case $tc_{i,j}$ and a program $P_i$, the set of executed (at least once) statements of $P$ with $tc_{i,j}$ is called a test case coverage $I_{i,j} = (I_{i,j,1}, ..., I_{i,j,n_i})$, where $I_{i,j,k} = 1$ if the $k$th statement is executed, 0 otherwise. $I_{i,j}$ indicates which parts of the program are active during a specific execution.

SBFL techniques assign suspiciousness scores for each of statements and rank them in a descending order of suspiciousness. Most of suspiciousness metrics are defined manually and analytically on the basis of multiple assumptions on programs, test cases and the introduced faults. Fig 1 lists the formula of three well-known metrics: TARANTULA [4], OCHIAI [5] and JACCARD [5]. Given a statement $e_{i,j}$, $\text{pass}(T_i)$ (resp. $\text{fail}(T_i)$) is the set of all passed (resp. all failed) test cases. $\text{pass}(e_{i,j})$ (resp. $\text{fail}(e_{i,j})$) is the set of passed (resp. failed) test cases covering $e_{i,j}$. The basic assumption is that the program fails when the faulty statement is executed. Moreover, the whole of suspiciousness metrics shares the same intuition: the more often a statement is executed by failing test cases, and the less often it is executed by passing test cases, the more suspicious the statement is considered. Fig.1 shows the suspiciousness spectrum of the different metrics.

- TARANTULA allows some tolerance for the fault to be executed by passing test cases (see Fig.1a). However, this metric is not able to differentiate between statements that are not executed by passing tests. For instance, consider two statements $e_{i,j}$ and $e_{i,k}$ with $|\text{pass}(e_{i,j})| = |\text{pass}(e_{i,k})| = 0$, $|\text{fail}(e_{i,j})| = 1$ and $|\text{fail}(e_{i,k})| = 1000$: $e_{i,j}$ and $e_{i,k}$ have the same suspiciousness degree according to TARANTULA.

- OCHIAI came originally from molecular biology. The specificity of this metric is that it attaches a particular importance of the presence to a statement in the failing test cases (see Fig.1b).

- JACCARD has been defined to find a proper balance between the impact of passing/failing test cases on the scoring measure [5] (see Fig.1c).

B. Analytical Hierarchy Process (AHP) [20]

AHP is a simple and easy to use structured process for organizing and eliciting criteria weights. It involves three main steps:

1) The criteria are subjectively compared in a pairwise manner according to their respective weight $w_i$. The comparison is organized into a square matrix $A = [1..m, 1..m]$, where $A[i, j]$ is the relative importance of criterion $i$ w.r.t. criterion $j$. The $i^{th}$ criterion is better than the $j^{th}$ criterion if $A[i, j] > 1$. In AHP, we have 9 degrees of dominance where $A[i, j]$ indicates

2) AHP assesses the criteria weighting vector \( w \) by solving the characteristic equation:

\[
A \cdot w = \lambda_{max} \cdot w
\]

where \( \lambda_{max} \) is the highest eigen value of \( A \).

3) Inconsistencies may occur in pairwise comparisons, because AHP does not enforce the preferences to be transitive. For this reason, a consistency check is conducted by calculating the consistency ratio (CR).

\[
CR = \frac{CI}{RI}, \quad CI = \frac{\lambda_{max} - n}{n - 1}
\]

where \( RI \) is a constant taken from the Random Consistency Index table of AHP. The weighting vector \( w \) is considered as reliable if \( CR < 0.1 \).

The AHP can be combined with a popular direct weights elicitation method ROC (Rank-Order Centroid) [21]. ROC produces an estimation of the weights that minimizes the maximum error of each weight. The ROC method assumes that the true weights are uniformly distributed on the simplex of rank-order weights. That is, ROC is a function based on the average of the corners in the polytope defined by the simplex \( S_w = w_1 > w_2 > \cdots > w_n, \sum w_i = 1, \) and \( w_i > 0, \) such that:

\[
w_i = \frac{1}{n} \sum_{k=i}^{n} \frac{1}{r_k}
\]

where \( r_i \) is the rank of the \( i^{th} \) criterion.

IV. AHP-LOC APPROACH

Let \( \{P_1, \ldots, P_n\} \) be a set of \( n \) faulty programs. A faulty program context is a triplet \( (P_i, L_i, T_i), i = 1..n, \) where \( P_i \) is a given faulty program, whose \( k \) faults are located at lines \( L_i = \{L_{i1}, \ldots, L_{ik}\}, \) and \( T_i \) is a test suite. Let \( (M_1, M_2, \ldots, M_m) \) be some SBFL ranking metrics (e.g., Tarantula, Ochiai, Jaccard, etc.).

The aim of this paper is to propose an approach aggregating SBFL ranking metrics in a single SBFL ranking metric that takes benefit of their localization effectiveness. This is achieved thanks to AHP technique. AHP requires a squared matrix \( A[1..m, 1..m] \), where \( A[i, j] \) scores how much criteria \( i \) is ranked better than criteria \( j \).

1) Scoring a single SBFL technique.

To evaluate how well the localization accuracy is, a suspiciousness score, denoted as EXAM score [11], is assigned to every faulty version of each subject program. The score defines the percentage of statements that need to be examined before the one locating the fault: lower is better. When running a single SBFL technique \( M_j \) on some faulty program context \( (P_i, L_i, T_i) \), it can return a set of equivalent statements in terms of suspiciousness (i.e., with the same suspiciousness degree). In this case, the effectiveness depends on which statement is to check first. For that reason, we consider two EXAM scores, the optimistic and the pessimistic one, denoted respectively \( R_{i,j}^P \) and \( R_{i,j}^O \). We talk about optimistic EXAM (resp. pessimistic exam) when the first (resp. last) statement to be checked is in the set of equivalent statements is the faulty one. We also define a third metric, \( \Delta \ EXAM \) \( R_{i,j}^\Delta = (R_{i,j}^P - R_{i,j}^O) \), representing the margin of the EXAM score, and middle EXAM \( R_{i,j}^M = \frac{R_{i,j}^P + R_{i,j}^O}{2} \). In other words, \( \Delta \ EXAM \) (middle exam) represents the distance between the optimistic and the pessimistic scores (resp. the average between the optimistic and the pessimistic scores).

2) Comparing two SBFL techniques.

Let \( M_j \) and \( M_l \) be two SBFL techniques to be compared on some faulty program context \( (P_i, L_i, T_i) \), by considering their pessimistic and optimistic results. Computing their middle gap \( E_{i,j,l}^M = (R_{i,j}^M - R_{i,j}^L) \) enables to know how much \( M_j \) is better than \( M_l \); if \( E_{i,j,l}^M > 0 \), then \( M_l \) is better, else the converse. We can also consider their pessimistic or optimistic gaps, but the middle gap is preferred since it aggregates them.

Finally, when running two SBFL techniques \( M_j \) and \( M_k \) on all of the faulty program contexts \( (P_i, L_i, T_i), i = 1..n \), we can estimate the most efficient technique by averaging their middle gaps on all of the programs \( AVG_{i,l} = \frac{1}{n} \sum_{l=1..n} E_{i,j,l}^M \).

3) Computing AHP ranking matrix

![Fig. 1: Suspiciousness Degrees.](image-url)
Algorithm 1: Learning

1. Input $D = \{(P_i, L_i, T_i) | P_i$: faulty program, $L_i$: fault localizations, $T_i$: test cases\}, $M$ set of $m$ ranking metrics;
2. Output: A: AHP matrix;
3. foreach metric $M_j \in M$ do
4. \hspace{1em} $A[j, j] \leftarrow 1$
5. \hspace{1em} foreach $(P_i, T_i, L_i) \in D$ do
6. \hspace{2em} Compute $R^P_{ij}$ using $M_j$
7. \hspace{2em} Compute $R^O_{ij}$ using $M_j$
8. \hspace{2em} $n \leftarrow 0$; \hspace{1em} $AVG_{i,j} \leftarrow 0$
9. \hspace{1em} foreach pair of metrics $M_j, M_l \in M$ do
10. \hspace{2em} \hspace{1em} \hspace{1em} foreach $(P_i, T_i, L_i) \in D$ do
11. \hspace{3em} $E_{ij}^P \leftarrow R_{ij}^P - R_{lj}^P$
12. \hspace{3em} $E_{ij}^O \leftarrow R_{ij}^O - R_{lj}^O$
13. \hspace{3em} $AVG_{ij} \leftarrow \frac{n}{1 + n} AVG_{ij} + \frac{1}{n+1}(E_{ij}^P + E_{ij}^O)$
14. \hspace{2em} \hspace{1em} \hspace{1em} $n \leftarrow n + 1$;
15. \hspace{1em} $AVG_{i,j} \leftarrow AVG_{i,j}$;
16. \hspace{1em} Scale $[AVG_{i,j}]$ to $[1, 9]$
17. \hspace{1em} if $AVG_{i,j} < 0$ then
18. \hspace{2em} $A[i, j] \leftarrow AVG_{i,j}$;
19. \hspace{2em} else
20. \hspace{2em} $A[j, i] \leftarrow 1/AVG_{i,j}$;
21. return $A$;

Algorithm 1 generates the AHP matrix without any prior ranking of decision criteria, by exploiting pairs comparisons. It loops on all of the pairs of SBFL techniques: for all $j, l \in 1..m$, compute $AVG_{i,j}$. Then it computes a scaling of $m \times m$ comparisons $AVG_{i,j}$ to the permitted values of the AHP matrix $A$. As explained in section III-B, an indirect weight elicitation is performed with an indirect assessment of weights, which is obtained by computing the eigenvector $w$ associated to the highest eigenvalue $\lambda_{max}$. This technique is far superior to any of the direct techniques due to its ability to capture the decision maker’s trade-offs between criteria. The consistency ratio $CR$ (see formula 2) is checked to see in what extent the learned coefficients of $A$ are plausible.

A. Passive version of AHP-Loc

Algorithm 1 enables to learn the AHP matrix from a training set composed of faulty program contexts $D = \{(P_1, L_1, T_1), ..., (P_n, L_n, T_n)\}$. The passive version of AHP-Loc consists of running Algorithm 1 on an already fixed set of faulty program contexts. The performances of this passive approach depends on the quality of the given training set.

B. Active version of AHP-Loc

The passive approach is straightforward and simple, but it is challenging to ensure that the given training set is sufficient to start an efficient localization. In addition, it is not taking benefit of current/future localizations. We propose the active version of AHP-Loc where the learning process is dynamically done. Its principle is the following: as long as the current learning dataset implies an AHP matrix less efficient than some SBFL ranking metric, a new faulty program context is added to the learning dataset $D$ of algo.1, updating the AHP matrix, and so on, until reaching an AHP matrix, which triggers the best localization matrix on the last added faulty program context.

C. AHP ranking score

Once the AHP matrix $A$ is learned in a passive or active way, we proceed in computing the metrics weighting vector $w$. For that, we solve the equation 1. Afterward, we compute the score of a given statement $e$ of a given faulty program using the following weighted aggregation function:

$$score_{AHP}(e) = \sum_{i=1}^{m} w_i \cdot scale(score_{M_i}(e)) \tag{4}$$

The scale function is used to adjusting score values of the different scales metrics to a $[0, 1]$ scale.

V. Running example

To illustrate our approach, we consider the Power program given in Fig.2. In this figure, we have six test cases where $tc_1$ to $tc_3$ are failing test cases, and $tc_4$ to $tc_6$ are passing test cases. According to the provided test cases, we report the suspiciousness ranking given by five ranking metrics ((1) AMPLE [22], (2) TARANTULA [4], (3) GP13 [23], (4) OCHIAI [5] and (5) JACCARD [5]) and our AHP-Loc approach. In this example, two faults are introduced at $e_3$ and $e_4$, where the correct statements are respectively "p = y;" and "p = y;".

AHP-Loc learns the AHP matrix from faulty program examples in a passive/active way. In this example, we asked an expert to provide us with preferences/dominances between the different pairs of the five ranking metrics:

$$\begin{pmatrix}
(1) & 2 & 3 & 9 & 9 \\
(2) & 1/2 & 1 & 2 & 3 & 7 \\
(3) & 1/3 & 1/2 & 1 & 2 & 4 \\
(4) & 1/9 & 1/3 & 1/2 & 1 & 2 \\
(5) & 1/9 & 1/7 & 1/4 & 1/2 & 1
\end{pmatrix} \tag{5}$$

It is important to stress here that: First, it is not always possible to obtain the opinion of an expert. Second, an expert will not be able to provide such matrix in an AHP context when the number of criteria exceeds 10 [20]. Third, the expert is facing a challenging task where its opinion heavily depends on the provided test cases. The learning step of AHP-Loc is proposed to bypass this limitation. We learn the AHP matrix from faulty programs contexts without the help of any expert.

Returning to our example, we compute the weighting vector $w$ according to the equation 1. This leads us to $w = (0.49, 0.25, 0.15, 0.07, 0.04)$. Now, for each statement we are able to give a ranking AHP score using equation 2.

The weighting vector $w_{AHP} = (0.49, 0.25, 0.15, 0.07, 0.04)$ was generated using the AHP matrix (5), with a consistency ratio $CR = 0.018 < 0.1$.

Fig.2 shows how the ranking metrics can be different with completely different rankings. The fault at $e_4$ is ranked first except for GP13 and OCHIAI. As they give a greater
prominence to the failing test cases, $e_4$ is ranked at the third position. The aggregation done by AHP-LOC ranks $e_4$ at the first position as well. For the fault introduced at $e_3$, TARANTULA, OCHIAI and GP13 rank it in the next-to-bottom place. The localization with AMPLÉ is able to rank it at the first position. But this localization is not accurate, since we have a big equivalent class. What is interesting here is that AHP aggregation ranks it at the second position. Here, AHP-LOC shows its best case by taking benefits of the difference that exists among the ranking metrics.

VI. EXPERIMENTS

This section describes the experimental settings (including benchmark programs, protocol and implementation), the experimental results and comparison with eight SBFL ranking metrics. Experiments were performed on single and multiple fault programs.

A. Benchmark programs

We evaluate our approach by analyzing the performance of fault localization over 18 faulty programs coming from different real-world applications. As our approach does not require a specific programming language in order to be applied, we investigated two types of programming paradigms, namely C and Java Object Oriented programs.

Siemens and Space datasets. We have considered both Siemens and Space datasets. A complete description of Siemens suite and Space dataset can be found in [24], [25]. These C programs are the most common program suites used to evaluate software testing and fault localization approaches. The Siemens suite + Space are provided with eight C programs, each one has a correct version and a set of faulty versions (one fault per version). The suite is also provided with test suites for each faulty version.

Table I summarizes the 212 faulty programs. For each program, we report the number of faulty versions (single fault $1F$, two faults $2F$ and four faults $4F$), the size of the program with its lines of code (LOC) and lines of executable code (LEC), the number of test cases. We have 139 versions with single fault, 47 with two faults and 26 versions with four faults. The single fault versions are provided, where multiple fault versions are produced by combining randomly the provided faults.

For C programs and to know the statements that are covered by a given (passing/failing) test case, we used Gcov $^2$ profiler tool to trace and save the coverage information of test cases as a boolean matrix (e.g., see Fig.2). Then, each test case is classified as positive/negative w.r.t. the provided correct version.

Java Object-Oriented datasets We evaluate our approach on 25,386 faults in ten Java Open-Source softwares (see Table II).$^3$ We used the granularity of methods (rather than the more common statements, as we did in Siemens suite). In fact, methods have the advantage of giving a natural context of each fault, and of being the natural skip/step into units of contemporary debuggers (guided by the output of the fault locator) [26].

Table II shows the details of the different projects, including the number of faulty programs (single fault $1F$, two faults $2F$ and four faults $4F$), the number of methods (excluding JUnit test methods), the number of methods under test, and the number of test cases.

We have 5,386 single fault Java programs, $10^4$ programs with two faults and $10^4$ programs with four faults. All are provided by the distribution. The multiple faults are spread over different methods.

B. Experimental protocol

Some statements can be equivalent in terms of suspiciousness. Here, the accuracy may vary depending on which statement to check first. For such reason, we report the two exam scores, the optimistic and the pessimistic one, denoted respectively in this section O-EXAM and P-EXAM. We recall that we talk about O-EXAM (resp. P-EXAM) when the first (resp. last) statement to check in the set of equivalent statements is the faulty one. We also use $\Delta$-EXAM = O-EXAM – P-EXAM, representing the range of the EXAM score.

Our approach AHP-LOC is trained on faulty programs to learn the coefficients of the AHP matrix (training set). To evaluate how accurately the learned AHP matrix performs, the resulting model is validated on the remaining part of the faulty

<table>
<thead>
<tr>
<th>Program</th>
<th># Versions</th>
<th>LOC</th>
<th>LEC</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Replace</td>
<td>(2F, 2F, 4F)</td>
<td>514</td>
<td>245</td>
<td>5,542</td>
</tr>
<tr>
<td>PrintTokens2</td>
<td>(9, 6, 3)</td>
<td>358</td>
<td>200</td>
<td>4,056</td>
</tr>
<tr>
<td>PrintTokens</td>
<td>(4, 6, 3)</td>
<td>348</td>
<td>195</td>
<td>4,071</td>
</tr>
<tr>
<td>Scheduler</td>
<td>(5, 6, 3)</td>
<td>294</td>
<td>152</td>
<td>2,650</td>
</tr>
<tr>
<td>Scheduler2</td>
<td>(8, 6, 3)</td>
<td>265</td>
<td>128</td>
<td>2,680</td>
</tr>
<tr>
<td>TodoInfo</td>
<td>(19, 6, 3)</td>
<td>272</td>
<td>123</td>
<td>1,052</td>
</tr>
<tr>
<td>Tcas</td>
<td>(37, 6, 3)</td>
<td>135</td>
<td>65</td>
<td>1,578</td>
</tr>
<tr>
<td>Space</td>
<td>(28, 5, 5)</td>
<td>9,126</td>
<td>3,657</td>
<td>13,585</td>
</tr>
<tr>
<td>Total</td>
<td>(139, 47, 26)</td>
<td>11,312</td>
<td>4,765</td>
<td>35,214</td>
</tr>
</tbody>
</table>

$^1$LOC: lines of code in the correct version

$^2$LEC: lines of executable code

$^3$MUT: Methods under test

$^4$The total number of methods, not counting JUnit test methods

<table>
<thead>
<tr>
<th>Program</th>
<th># Versions</th>
<th>Methods$^*$</th>
<th>MUT</th>
<th>Test cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daikon 4.6.4</td>
<td>(352, 10^4, 10^4)</td>
<td>14,387</td>
<td>1,936</td>
<td>157</td>
</tr>
<tr>
<td>Eventbus 1.4</td>
<td>(577, 10^4, 10^4)</td>
<td>859</td>
<td>338</td>
<td>91</td>
</tr>
<tr>
<td>Jaxen 1.1.5</td>
<td>(600, 10^3, 10^3)</td>
<td>1,689</td>
<td>961</td>
<td>695</td>
</tr>
<tr>
<td>Jester 1.37b</td>
<td>(411, 10^3, 10^3)</td>
<td>378</td>
<td>152</td>
<td>64</td>
</tr>
<tr>
<td>JExel 1.0.0b13</td>
<td>(537, 10^3, 10^3)</td>
<td>242</td>
<td>150</td>
<td>335</td>
</tr>
<tr>
<td>JParsec 2.0</td>
<td>(598, 10^3, 10^3)</td>
<td>1,011</td>
<td>893</td>
<td>510</td>
</tr>
<tr>
<td>AC Codec 1.3</td>
<td>(543, 10^3, 10^3)</td>
<td>265</td>
<td>229</td>
<td>188</td>
</tr>
<tr>
<td>AC Lang 3.0</td>
<td>(599, 10^3, 10^3)</td>
<td>5,373</td>
<td>2,075</td>
<td>1,666</td>
</tr>
<tr>
<td>Eclipse Draw2d 3.4.2</td>
<td>(570, 10^3, 10^3)</td>
<td>3,231</td>
<td>878</td>
<td>89</td>
</tr>
<tr>
<td>HTML Parser 1.6</td>
<td>(599, 10^3, 10^3)</td>
<td>1,925</td>
<td>785</td>
<td>600</td>
</tr>
<tr>
<td>Total</td>
<td>(5386, 10^3, 10^3)</td>
<td>29,368</td>
<td>8,397</td>
<td>4,395</td>
</tr>
</tbody>
</table>

$^*$MUT: Methods under test
programs (testing set). To avoid bias of random selection, we performed a 6-folds cross-validation of the data using different partitions. In each fold, we randomly select three faulty versions to form the training set and use the remaining 15 faulty versions for evaluation (testing set). We report the performance result using averaged EXAM score over the folds.

We compared our three AHP-LOC versions Active AHP-LOC, Passive AHP-LOC and ROC based AHP-LOC, with eight widely-studied spectrum-based ranking metrics, TARANTULA, OCHIAI, JACCARD, M2 and AMPLE, and three recent proposed ranking metrics including GP13, NAISH2 and ER1B, which are proved as the optimal under theoretical assumptions (cf. [27]).

C. Implementation

We have implemented our AHP-LOC versions (active, passive and ROC) in C++. For a fair comparison between our tool and the other approaches, we have implemented the SBFL metrics, GP13, NAISH2, AMPLE, ER1B, M2, TARANTULA, OCHIAI and JACCARD in C++ as presented in [4], [5]. Our experiments were conducted on an Intel® i5-2400 CPU at 3.10GHz x 4 with 8 GB RAM.

D. Single Fault Results

Table III reports an EXAM score comparison (O-EXAM, P-EXAM and $\Delta$-EXAM) between Active AHP-LOC, Passive AHP-LOC, ROC AHP-LOC and SBFL metrics on single fault programs. The first observation that we can draw is that, comparing our three versions, the Active AHP-LOC achieves the lowest O-EXAM and P-EXAM. We also observe that AHP-LOC (with its three versions) is more effective than SBFL approaches in most of the benchmarking instances.

Now, when examining Fig. 3 we can see that Active AHP-LOC provides the narrowest $\Delta$-EXAM (i.e. the range of statements to be inspected in vain before reaching the faulty statement is minimal). Remarkably, the P-EXAM (4.91 %) of Active AHP-LOC is even lower that the O-EXAM of all the metrics. It is also clear from the same figure, that metrics shown in red (i.e. (5) TARANTULA, (7) JACCARD and (10) AMPLE) perform poorly, as they generate big class of suspicious statements (see also Tables III, $\Delta$-EXAM). Interestingly, Passive AHP-LOC, AHP-ROC, ER1B and NAISH2 almost achieve similar better accuracy.

<table>
<thead>
<tr>
<th>Test cases</th>
<th>O-EXAM</th>
<th>P-EXAM</th>
<th>$\Delta$-EXAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHP-LOC</td>
<td>5.07</td>
<td>4.91</td>
<td>4.91</td>
</tr>
<tr>
<td>Active</td>
<td>5.07</td>
<td>4.91</td>
<td>4.91</td>
</tr>
<tr>
<td>ER1B</td>
<td>5.07</td>
<td>4.91</td>
<td>4.91</td>
</tr>
<tr>
<td>NAISH2</td>
<td>5.07</td>
<td>4.91</td>
<td>4.91</td>
</tr>
</tbody>
</table>

E. Multiple Fault Results

In this section, we report the EXAM score comparison results on programs with multiple faults. The results for two and four faults are shown in Table III.

Let us start with the two faults programs. Our first observation is that the Active AHP-LOC is drastically more efficient than all SBFL metrics in terms of P-EXAM, O-EXAM and $\Delta$-EXAM. This is indicative of high effectiveness, and thus an improved localization on most subject programs (C and Java). However, the SBFL metrics, namely, GP13 and TARANTULA show the opposite behavior, which is indicative of a low effectiveness and large wasted debugging effort.


Fig. 4 shows a comparison on P-EXAM (fig.4a) and O-EXAM (fig.4b) between SBFL metrics, and AHP-LOC. When focusing on the exam score required to localize both faults ($f_1$ and $f_2$), we can see that Active AHP-LOC is the most accurate (quickly locates the last fault). Moreover, the Active AHP-LOC improve the localization accuracy by 45% for the O-EXAM and more than 47% for the P-EXAM compared to the best localization score given by ER1B metric. Here, we notice that the Passive AHP-LOC and AHP-ROC enable to better locate both of faults compared with SBFL metrics. Fig.4 also shows that, in some cases, the standard metric can be effective. For instance, the NAISH2 and ER1B report similar accuracy and are able to capture quickly the first fault compared to the remaining metrics.
Interestingly, the pessimistic EXAM score of Active AHP-LOC is nearly better than the optimistic score of all metrics (including the Passive AHP-LOC and AHP-ROC). In position two comes the AHP-based multicriteria approaches, with the best EXAM scores against the SBFL metrics.

For four-faults programs, here the results presented in Fig. 5 and Table III support our previous observations on the comparison between metrics and our AHP-based approaches. In fact, our approach significantly dominates all the SBFL approaches by capturing quickly all faults with high accuracy, especially according to the P-EXAM score. Regarding the O-EXAM score, this observation remains true for both the third and fourth faults (i.e. $f_3$ and $f_4$).

Concerning the first two faults, i.e., $f_1$ and $f_2$, the result depicted in Fig. 5c needs to be clarified and analyzed in depth. Indeed, standard SBFL metrics, namely, TARANTULA, OCHAIJ, and JACCARD seem to provide the best result. However, our approach is rather good in term of effectiveness, because those metrics give a large class of equivalent suspicious statements. To support this justification, we consider an important information given by the median of the box plots, which represent the average EXAM score of P-EXAM and O-EXAM.

Overall, by observing the global EXAM scores given in Table III, the Active AHP-LOC remains the best localization approach for programs with one, two and four faults.

VII. Conclusion

In this paper we have proposed a new fault localization method based on AHP (Analytic Hierarchy Process) multicriteria based approach, which aggregates spectrum-based ranking metrics. We propose a passive and active learning versions to maintain the right expected weights of criteria.

We have compared experimentally our approach with state of the art SBFL metrics on a set of multiple faults programs. The results we obtained show that our approach enables to outperform the best of the art SBFL metrics on a set of multiple faults programs.

REFERENCES


