Localising Faults in Test Execution Traces

Gulsher Laghari
gulsher.laghari@uantwerpen.be

Alessandro Murgia
alessandro.murgia@uantwerpen.be

Serge Demeyer
serge.demeyer@uantwerpen.be

ANSYMO
Universiteit Antwerpen — Middelheimlaan 1 — BE 2020 Antwerpen
http://www.uantwerpen.be/en/rg/ansymo/

ABSTRACT

With the advent of agile processes and their emphasis on continuous integration, automated tests became the prominent driver of the development process. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In this paper we propose a heuristic named SPEQTRA which mines the execution traces of a series of passing and failing tests, to localise the class which contains the fault. SPEQTRA produces ranking of classes that indicates the likelihood of classes to be at fault. We compare our spectrum based fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (*NanoXML*) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids, tracing*

General Terms

Algorithms

Keywords

Automated developer tests; spectrum based fault localisation; replication (different heuristic & same data)

1. INTRODUCTION

The quintessential principle of continuous integration declares that software engineers should merge their working copies with the main branch several times a day [7]. During each integration step, a continuous integration server builds the entire project, using a fully automated process involving compilation, unit tests, integration tests, code analysis, security checks, etc. When one of these steps fails, the build is said to be *broken*; development can then only proceed when the fault is repaired [11, 15]. The safety net on automated tests, encourages software engineers to write lots of tests — several reports indicate that there is more test code than application code [16, 5, 19]. Moreover, executing all these tests sometimes take several hours [14]. Hence, it is critical to quickly identify the location of the fault in the code. Not only does a broken build block all progress in the team, but more importantly the location of the fault serves as an indicator for the software engineer expected to repair the build.

In the simplest case, there is a one-to-one mapping between the failing test and the class containing the fault. However, for complex object interactions where objects must adhere to a certain protocol, illegal call sequences trigger faults which are notoriously hard to pinpoint to an exact location [13]. Faults induced by illegal method call sequences are real and hard to debug: a conservative estimation identified 115 faults related to missing method calls in the Eclipse bug repository [12]. For such faults the one-to-one mapping between the failing test and the class containing the fault does not hold and then software engineers resort to debugging [21].

Luckily, there is a class of heuristics —named *spectrum based fault localisation*— which give indications for the location of the fault. Such heuristics compare execution traces of passing tests against the ones from a failing test, assuming that the points where the traces differ are the most likely location of the fault [1]. The state of the art heuristic for class level fault localisation by analysing method call sequences in unit tests is proposed by Dallmeier et al. with a tool called AMPLE [4]. AMPLE traces method calls invoked by objects and collects call sequences of the corresponding classes by sliding a window over the executions traces. It then compares the execution trace of all passing tests against one trace with a failing test and deduces which class most likely contains the fault. AMPLE was shown to be quite effective on a small yet representative case (*NanoXML*): it could immediately pinpoint the faulty class in 36% of all test runs; while on average 21% of the executed classes (10% of all classes) must be inspected to find the location of the fault.

In this paper we report on a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. SPEQTRA addresses two shortcomings of the AMPLE heuristic: (a) it filters out repetitive method calls (e.g. contained in loops) and (b) avoids the sliding window and the arbitrary upper limit on the length of the call sequence it imposes. To address these shortcomings, SPEQ-

TRA uses a different algorithm (closed itemset mining) to characterise method call sequence and distinguish the faulty ones. Via the replication experiment we demonstrate that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE: we can immediately pinpoint the faulty class in 56% of all test runs; while on average 12% of the executed classes (5% of all classes) must be inspected to find the location of the fault. Moreover, for 70% of the faults, SPEQTRA has at most one false positive whereas for AMPLE this happens for 59% of the faults.

This paper is structured as follows. We explain the two heuristics under investigation in Section 2. Next, we describe the way we set up our replication experiment, including the particular details about the *NanoXML* case in Section 3. Then the bulk of the paper is contained within Section 4, where we show the results of the comparison including some anecdotal evidence from *NanoXML*. We list the related work, showing other research on dynamic analysis within a software evolution context in Section 5, followed by a discussion on the threats to validity in Section 6. Finally, Section 7 summarises our findings and lists the contributions.

2. HEURISTICS UNDER INVESTIGATION

In this section we give a detailed explanation of the two heuristics under investigation. First, we provide some background information regarding spectrum based fault localisation which is the basis for the two heuristics (Section 2.1). Then we contrast the two heuristics depicted in a Figure 1 showing where they are the same (i.e. collecting traces per object — Section 2.2) and where they differ (creating sequences of method calls — Section 2.3; the similarity coefficient used to rank the corresponding fault locations — Section 2.4).

2.1 Spectrum Based Fault Localisation

AMPLE and SPEQTRA both are instances of the class of spectrum based fault localisation heuristics [1]. Such heuristics discover statistical coincidences between system failures and the activity of the different parts of a system. All these heuristics create a so-called *program spectrum*, which is a matrix where each column corresponds to a program entity (e.g. statement, block, sequence of method calls) and rows represent a particular test run. For each test run the corresponding column for program entity is marked as 1 (executed) or 0 (not executed). Alongside the program spectrum, the heuristic also creates an *error vector* which is a column where a cell is marked as 1 if the test run failed or 0 if the test was a success. Next, the error vector is compared against all columns in the program spectrum using a particular *similarity coefficient*; the column which is most similar to the error vector is then the program entity which most likely contains the fault.

2.2 Collecting Traces

AMPLE and SPEQTRA use sequences of method calls as the program entities which are represented in each column of the fault spectrum matrix. Both heuristics group the outgoing method calls according to the following scheme. Let $O = \{o_1, o_2, ..., o_n\}$ be the set of object instances of the class C and $T = \{t_1, t_2, ..., t_n\}$ be the set of object traces of class C, where t_i represents the trace of outgoing method calls by the object o_i. By outgoing method calls we mean an object calling a method of another object. For instance if we have an object o_1 with a method $m1()$ and hit method hosts a call to method $n1()$ belonging to an object o_2, the collected outgoing method call for o_1 is $m1()$.

Two objects o_1 and o_2 of a class C may have following traces of method calls (Equations 1 and 2):

$$t_1 = \left\{ \begin{array}{c} m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3 \end{array} \right\} \quad (1)$$

$$t_2 = \{m_1, m_1, m_1\} \quad (2)$$

AMPLE and SPEQTRA group all such object traces for the corresponding class C which has trace set T (Equation 3).

$$T = \left\{ \begin{array}{l} \{\, m_1, m_1, m_1, m_2, m_2, m_3, \\ m_1, m_1, m_2, m_2, m_3\}, \\ \{\, m_1, m_1, m_1\} \end{array} \right\} \quad (3)$$

2.3 From Traces to Class Sequences

Traces of outgoing method calls can grow to millions of method calls per object [3]. To reduce these traces AMPLE and SPEQTRA each apply a different technique to arrive at what we call *Class Sequences* for the remainder of the paper.

AMPLE — Sliding Window. AMPLE slides a window of fixed size over the trace to create a list of class sequences. From the previous example, if we fix the window size as 2 and slide it over the object traces in Equation 3 we obtain the set of class sequences in Equation 4

$$C_A = \left\{ \begin{array}{l} \{\{m_1, m_1\}, \{m_1, m_2\}, \{m_2, m_2\}, \\ \{m_2, m_3\}, \{m_3, m_1\}\} \end{array} \right\} \quad (4)$$

SPEQTRA — Frequent Sequences. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [20]. Given the set of object traces T of class C, we define:

- X —*itemset*— a set of method calls.
- $\sigma(X)$ —support of X— the number of traces of T that contain this itemset X.
- *minsup* —minimum support of X— a threshold used to tune the number of returned itemsets.
- *frequent itemset* — an itemset X is frequent when $\sigma(X) \geq minsup$.
- *closed itemset* — a frequent itemset X is closed if there exists no proper superset X' whose support is same as the support of X (i-e. $\sigma(X') = \sigma(X)$).

From now on, we refer a closed itemset X as a frequent sequence or simply a sequence of method calls. Adopting closed itemset mining in the context of fault localisation, we fix *minsup* to 1 because those classes which only create one object (and thus one trace) should be included in the program spectrum as well; this one call trace may be the one which triggers the fault. However, we tune the algorithm in another way. The mining algorithm also returns frequent sequences that comprise only one method call. Since we are looking for faults caused by complex object interactions where objects must adhere to a certain protocol, sequences should have at least a length of two.

From the previous example with input T (Equation 3) and $minsup = 1$ the generated set of frequent sequences is:

$$C_F = \{\{m_1\}, \{m_1, m_3, m_2\}\} \quad (5)$$

It can be observed that the sequences such as $\{m_2\}$, $\{m_3\}$, $\{m_1, m_2\}$, $\{m_1, m_3\}$, $\{m_2, m_3\}$ are not included in final set (Equation 5), since there exists a super sequence $\{m_1, m_3, m_2\}$

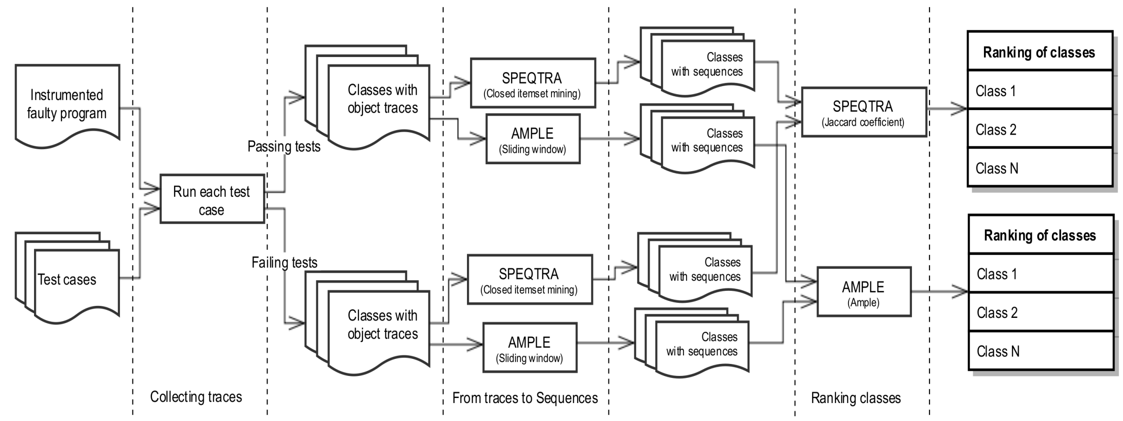

Figure 1: Overview of the two heuristics showing three steps (i) Collecting tracing, (ii) Collecting class sequences, and (iii) Ranking classes

with equal support. As SPEQTRA removes all frequent sequences of length 1, thus the sequence set C_F (5) is finally reduced into the set of Class Sequences C_S in (6).

$$C_S = \{\{m_1, m_3, m_2\}\} \tag{6}$$

2.4 Ranking Classes

Both AMPLE and SPEQTRA assign a weight $W(X)$ to each class sequence in C_A and C_S(Equation 4 and 6 respectively). Note that X refers to a sequence of method calls, but there is a difference in that X is a chunk of fixed size in AMPLE whereas X is set of frequent method calls in SPEQTRA.

AMPLE — AMPLE has defined its own weighting scheme based on a configuration of a single failing test and several passing tests. Sequences in AMPLE are assigned a weight between 0 and 1 using equation (7) [4].

$$W(X) = \begin{cases} \frac{k(X)}{n} & \text{if } X \text{ not in failing test} \\ 1 - \frac{k(X)}{n} & \text{if } X \text{ in failing test} \end{cases} \tag{7}$$

Where n is the number of passing tests and $k(X)$ is the number of passing tests that include the sequence X.

SPEQTRA — We tested several weighting schemes to rank the classes. Ultimately, in SPEQTRA, we opted for the Jaccard similarity coefficient (Equation 8) adopted from Chen et al. [2]:

$$W(X) = \frac{a_{11}(X)}{a_{11}(X) + a_{01}(X) + a_{10}(X)} \tag{8}$$

Where:

- $a_{11}(X)$ = Number of failing tests in which *sequenceX* is found.
- $a_{10}(X)$ = Number of passing tests in which *sequenceX* is found.
- $a_{01}(X)$ = Number of failing tests in which *sequenceX* is not found.

Weight per class. Both AMPLE and SPEQTRA take the average of all weights for all sequences of a class and assign this weight to class C, as defined in Equation 9:

$$W(C) = \frac{1}{n}\sum_{i=1}^{n} W(X_i) \tag{9}$$

where n is the number of sequences in the class and $W(X_i)$ is weight of a sequence as given by Equation 7 (AMPLE) or Equation 8 (SPEQTRA).

Finally, both heuristics rank all classes using their weights $W(C)$, where the one with the highest weight is the most likely location of the fault.

3. EXPERIMENTAL SETUP

This paper is set-up as a replication experiment (different heuristic & same data) where we compare a new fault localisation heuristic named SPEQTRA against the state of the art AMPLE. In what follows, we describe the way we set up our replication experiment, including the particular details about the *NanoXML* case (subsection 3.1). Next, we provide the necessary practical details about the mechanics of the experiment so that other researcher can replicate our findings (subsection 3.2)

3.1 Replication Case — NanoXML

The original paper proposing the AMPLE heuristic demonstrated its effectiveness on a small but representative project named *NanoXML* [4]. *NanoXML* is a non-validating XML parser written in Java. Its source code and documentation are available in the Software-artifact Infrastructure Repository[1] [6].

NanoXML has five development versions (V1 . . . V5) where the number of classes span from 16 to 23 (Table 1). With the exception of version V4, all others have documented faults that can be activated and exposed by the test suite. These versions (V1,V2,V3,V5) —the ones we use for the experiment— have 32 faults (cumulatively). Each version is

[1]http://sir.unl.edu/portal/index.php

Table 1: *NanoXML* version details

Version	# of classes	LOC	# of faults	# of tests
1	16	4334	7	214
2	19	5806	7	214
3	21	7185	10	216
5	23	7646	8	216

shipped along with tests and test drivers. A test driver is a class that sets up multiple tests by feeding them with the required input (e.g. read a XML file). The goal of these test drivers is to trigger one and only one feature of the project.

We collect the traces of all faults by injecting one fault at a time and subsequently running the test suite. Then for each test, we record the outgoing methods calls of the created objects. It is important to note that, for each test (passing or failing), a separate trace is maintained for each object. All the outgoing method calls of an object appear in their own trace. Thus, two objects o_1 and o_2 of the same class C have independent traces of method calls.

In the replication experiment we activate one fault at a time. Having 32 faults leads to 32 distinct variants of *NanoXML*. Among these variants we picked only the ones that generate at least one failing and one passing test for each test driver. Just like in the original AMPLE experiment, this ensures that failing and passing tests are all related to the same functionality. For each test driver, we group one failing test with all passing tests, the set-up that is needed to reflect the set-up of AMPLE experiment. We repeat this process for each failing test associated to that test driver. At the end of this process, we end up with 18 variants of *NanoXML* and 347 combinations of failing and passing tests used for our experiments.

Note that this set-up is not exactly the same as the one reported in the AMPLE paper because the version of *NanoXML* we downloaded from the Software-artifact Infrastructure Repository has been changed. In the latest version, one fault is removed from V5 with a note "since it is overly expressive it may not be representative of a *pseudo-real* fault". As a consequence, the fault matrix also differs from the previous version. This is an inherent risk with replication experiments and partially explains why we do not obtain the same results reported in the AMPLE experiment [4].

3.2 Replication Details

AMPLE replication.

When preparing for the replication experiment, we downloaded the original binary of the AMPLE implementation. Due to hardware constraints, we were unable to run this binary. Consequently, we implemented our own version of the algorithm as reported in the original AMPLE paper [4]. We used the optimal settings for the parameters of the heuristic, in particular we adopted a sliding window size of 8.

AspectJ. The object traces are collected by introducing logger functionality into the *NanoXML* code via AspectJ[2]. More specifically, we use a method call join point with a pointcut to pick out every call site. Each time a method call occurs, the aspect extracts the caller object and adds a method entry to the object's trace. The aspect is robust for different threads that may be running within the java project, although this was irrelevant for the *NanoXML* case. All object traces belonging to same class appear together in a HashMap maintained for each executed class.

Static Methods are Ignored. SPEQTRA, like AMPLE, also collects traces of method calls invoked by objects of a class. Calls to static methods are not captured and do not appear in the trace, hence cannot be identified as the location of the fault. For the particular replication of the *NanoXML* experiment this did not cause any problems however this limitation must be taken into account for future replication.

Single failing test. For this experiment, we inject one fault into the program which causes one or more tests to fail and several of them to pass. All these tests are executed with same test driver. In principle, SPEQTRA is able to rank fault locations using all these failing and related passing tests. However, since AMPLE is designed to work with only one failing test we replicated the set-up to include the trace of a single failing test and one or more passing tests.

Closed Itemset Mining. To avoid the arbitrary upper limit imposed by the size of the sliding window, SPEQTRA incorporates the frequently appearing sequences adopting an algorithm named *closed itemset mining* [20]. In particular, we used the implementation provided by the library SPMF[3].

Search Length. To compare the results of the two heuristics we use the so-called *search length* as defined in the AMPLE experiment [4]. The search length counts how many classes are placed atop of the faulty class in the ranking produced by the heuristic. In that sense, it represents how many classes the developer has to examine before finding the class containing the fault. The search length is zero whenever the faulty class is placed as the first item in the ranking.

4. RESULTS AND DISCUSSION

To compare our results with AMPLE, we replicated AMPLE with a sliding window size 8, which is the value for which AMPLE achieved the best performance. Following the experimental set-up explained in Section 3, we obtained rankings for all the 347 combinations both with our implementation of AMPLE and SPEQTRA. Below we discuss the results of both.

1. Our replication experiment confirmed the results reported in the original AMPLE paper. There were some minor changes in the results, but these can be attributed to the differences in the *NanoXML* version used in our experiment, in particular in the tests accompanying the project.
2. The average search length of all rankings in 347 test runs with both heuristics is reported in Table 2. Here SPEQTRA has less average search length than sliding window approach.
3. In 173 test runs out of 347, SPEQTRA outperforms AMPLE and has search length less than AMPLE. In 140 test runs both SPEQTRA and AMPLE have same search length, whereas in only 34 cases SPEQTRA has search length greater than AMPLE.
4. The plot of the cumulative of search length distribution in 347 test runs with both heuristics is given in Figure 2. With SPEQTRA, the search length of 0 covers 56% of faults, whereas with AMPLE it is only 40% of faults. Furthermore, the worst case search length with SPEQTRA is 6 whereas with AMPLE it is 8.

[2]AspectJ http://eclipse.org/aspectj/

[3]SPMF http://www.philippe-fournier-viger.com/spmf/

Table 2: Average Search Length in SPEQTRA and AMPLE

AMPLE	SPEQTRA
2.07	1.20

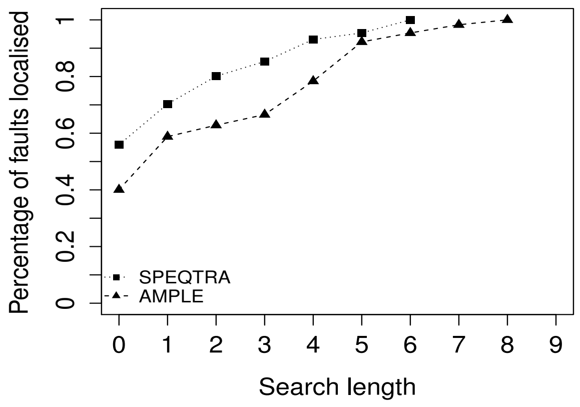

Figure 2: Search Length in SPEQTRA and AMPLE.

4.1 Anecdotal Evidence

From the experiment we collected some anecdotal evidence highlighting the main differences between the two approaches. Specifically the two differences, namely (i) the effect of the sliding window and (ii) the impact of repetitive method calls (e.g. contained in loops). In Listing 1, we see a piece of source code showing a fault in `XMLElement` class at line 7 which was exercised by several unit tests. This resulted in three `XMLElement` object traces generated by the failing test as shown in Listings 2, 3 and 4. Note that for brevity, method parameters are not shown.

From these object traces in failing test, SPEQTRA generated three sequences of method calls for class XMLElement. For brevity, we list the numbers in the sequences instead of method calls. These numbers in the sequences represents the line numbers in Listing 2, unless otherwise mentioned. The line number for the method is the very first entry of the method in the trace. The first sequence was s1 = {1, 4, 21, 24, 5, 15, 7}, the methods indicated by numbers 5 and 7 appear in Listing 3 at lines 5 and 7. The second sequence was s2 ={1, 4, 13, 21, 24, 15} and the third sequence was s3 = {1, 4, 21, 24, 15}. These sequences capture frequent method calls occurring in the traces and hence represent a good abstraction of the traces. The three sequences have length 7, 6 and 5 respectively and none of the sequence has repetitive method calls.

On the other hand, AMPLE generated 37 sequences for the class, each with repetitive method calls. If we slide a window of size 8 over the trace in Listing 2, the first sequence s1 = {1, 2, 3, 4 ,5, 6, 7, 8} contains 5 repetitions for method `XMLElement.findAttribute()` and three repetitions of method `XMLAttribute.getFullName()`. Likewise the second sequence s2 = {2, 3, 4 ,5, 6, 7, 8, 9} contains four times method `XMLElement.findAttribute()` and 4 times method `XMLAttribute.getFullName()`. As a consequence, the AMPLE heuristic fails to locate the fault and it ranked the faulty class on position 6 whereas SPEQTRA could pinpoint it exactly.

Listing 1: Code snippet with a fault in XMLElement class

```
 1  public Enumeration
        enumerateAttributeNames() {
 2   Vector result = new Vector();
 3   Enumeration _enum = this.attributes.
        elements();
 4   while (_enum.hasMoreElements()) {
 5    XMLAttribute attr = (XMLAttribute)
        _enum.nextElement();
 6    // The call should be to attr.getName()
 7    result.addElement(attr.getFullName());
 8   }
 9   return result.elements();
10  }
```

Listing 2: Failing trace of XMLElement object 1

```
 1  XMLElement.findAttribute(String)
 2  XMLElement.findAttribute(String)
 3  XMLElement.findAttribute(String)
 4  XMLAttribute.getFullName()
 5  XMLElement.findAttribute(String)
 6  XMLAttribute.getFullName()
 7  XMLElement.findAttribute(String)
 8  XMLAttribute.getFullName()
 9  XMLAttribute.getFullName()
10  XMLElement.findAttribute(String)
11  XMLAttribute.getFullName()
12  XMLAttribute.getFullName()
13  XMLElement.getName()
14  XMLElement.getName()
15  XMLAttribute.getName()
16  XMLAttribute.getName()
17  XMLAttribute.getName()
18  XMLAttribute.getName()
19  XMLAttribute.getName()
20  XMLAttribute.getName()
21  XMLElement.getAttribute()
22  XMLElement.findAttribute(String)
23  XMLAttribute.getFullName()
24  XMLAttribute.getValue()
25  XMLElement.getAttribute()
26  XMLElement.findAttribute(String)
27  XMLAttribute.getFullName()
28  XMLAttribute.getFullName()
29  XMLAttribute.getValue()
30  XMLElement.getAttribute()
31  XMLElement.findAttribute(String)
32  XMLAttribute.getFullName()
33  XMLAttribute.getFullName()
34  XMLAttribute.getFullName()
35  XMLAttribute.getValue()
```

Listing 3: Failing trace of XMLElement object 2

```
 1  XMLElement.findAttribute(String)
 2  XMLElement.findAttribute(String)
 3  XMLElement.findAttribute(String)
 4  XMLAttribute.getFullName()
 5  XMLElement.findAttribute(String, String)
 6  XMLAttribute.getName()
 7  XMLAttribute.getNamespace()
 8  XMLAttribute.getName()
 9  XMLAttribute.getName()
10  XMLAttribute.getName()
11  XMLAttribute.getName()
12  XMLElement.getAttribute( )
13  XMLElement.findAttribute(String)
14  XMLAttribute.getFullName()
15  XMLAttribute.getValue()
16  XMLElement.getAttribute( )
17  XMLElement.findAttribute(String)
```

```
18  XMLAttribute.getFullName()
19  XMLAttribute.getValue()
```

Listing 4: Failing trace of XMLElement object 3

```
XMLElement.getName()
```

There are however 34 situations where AMPLE was more accurate than SPEQTRA. We use one fault injected in the class `NonValidator` to explain this difference. The failing test and several of the passing tests generated the same trace for the only object of `NonValidator`. In this case, SPEQTRA generated one sequence for the failing test and two sequences for the passing tests, one of which also appeared in the failing test. As a consequence, SPEQTRA ranked this class on position 4 whereas AMPLE could pinpoint it exactly. This can be explained as, the Jaccard similarity coefficient (or any other coefficient used in spectrum based fault localisation) assigns more weight to a sequence when it is present more in failing tests and less in passing tests. Hence the sequence only present in passing tests gets weight 0; the value 0 for numerator $a_{11}(X)$ in equation 8 evaluates the whole equation to 0. The other sequence presented in failing test gets a lower weight due to its presence in several passing tests; the higher value of denominator $a_{10}(X)$ in equation 8 decreases the value. Consequently, the weight of the class, which is average weight of the two sequences, is also less.

4.2 Discussion

Based on this replication experiment, we conclude that SPEQTRA is significantly better than AMPLE since:

1. The average ranking in SPEQTRA is lower than AMPLE. This average suggests that a developer has to search through, on average, 12% of 10.25 average executed classes or 5% of all 23 classes. This is significantly better than AMPLE, where 20% of executed classes or 9% of all classes need to be searched.
2. A faulty class is placed first in the ranking (search length 0) for 56% of faults by SPEQTRA whereas for AMPLE it happens only for 40% of the faults.
3. For 70% of faults, there is atmost one false positive (search length 1) with SPEQTRA whereas for AMPLE this happens for 59% of the faults.

5. RELATED WORK

In this section, we present related work on spectrum based fault localisation thus immediately relevant for this replication experiment. Moreover, we also give references to related work on dynamic analysis for program comprehension as this provides the broader context for our research.

5.1 Spectrum Based Fault Localisation

Spectrum based fault localisation is an automated fault diagnosis technique based on differences in program spectra of a program between passing and failing tests [1]. Spectrum based fault localisation techniques have been applied in many domains such as localising the fault and ranking program statements [10], blocks [1], failure related components [2] and —last but not least— classes [4].

- Jones et al. used statement-hit spectra to rank statements of C programs according to their likelihood to be at fault [10]. To visualize the ranking, they implemented a tool — Tarantula — able to mark statements with colors that span from red (statement likely at fault) to green (statement unlikely at fault).
- Abreu et al. used blocks-hit spectra to rank the blocks in order of their likelihood to be at fault [1]. Here the block is defined as C language statement where compound statement (statements inside curly brackets) counts as a single statement. They compared the performance of different similarity coefficients and their impact on diagnostic accuracy of spectrum based fault localisation technique.
- Chen et al. implemented the tool Pinpoint for tracing client requests in Internet service environments. Pinpoint records the components involved in the service and whether or not the request is satisfied [2]. The tool correlates the request failures to the components that most likely caused the failure.

All previous papers detect the fault at different levels of granularity(e.g., statements, blocks). However, none of them uses spectrum based fault localisation to identify faults due to method call sequences. In the literature, the only two techniques able to localise faults related to method call sequence are AMPLE and MCA-E. AMPLE is a tool, created by Dallmeier et al., that traces method calls invoked by objects and collects call sequences of the corresponding classes [4]. The outcome of the tool is a list of classes ranked according to their likelihood to be at fault. MCA-E is a technique proposed by Tu et al. in order to improve the regular spectrum based fault localisation techniques adopted in AMPLE [17]. Its outcome is a list of statements ranked according to their likelihood to be at fault. In the first step, it computes the likelihood of classes to be at fault (suspiciousness) by taking into account the difference of their method call sequences between passing and failing tests. In the second step, the suspiciousness of classes is used together with their statements to generate ranking of statements. One of the limitations of AMPLE and MCA-E approaches is the adoption of a window of finite size that slides over the execution traces. Such sliding a window is not efficient for computing the sequences since method calls may stem from loops or may repeat in a trace resulting into sequences with repetitive method calls which add overhead with little extra information. Furthermore, the number of sequences linearly increases as the size of the trace increases. With window size w and n number of method calls in a trace, n^w sequences are possible. In this paper we address the shortcomings related to the sliding window by mining the frequent method call sequences. The sequence mining alleviates the arbitrary upper limit on the length of the call sequence. It also optimises the computational power required to obtain the ranking of classes as the mining algorithm limits the frequent sequences to closed ones: and also the SPEQTRA removes one-length sequences, the number of SPEQTRA sequences is far less than sliding a window over the trace.

It also optimises the computational power required to obtain the ranking of classes as it generates far less sequences than sliding a window over the trace.

5.2 Program Comprehension

Discovering program invariants and specifications such as legal method call sequences are common goals of research in program comprehension [8, 9, 13]. Such specifications, achieved by means of dynamic analysis, are used for purposes including documentation, learning the API's etc.

Ernst et al. implemented the tool Daikon to dynamically detect program invariants [8]. An invariant is defined as a property that is true at a particular program point or points. Daikon runs an instrumented program over a series of test runs and records program properties. At the invariant detection stage it starts with a list of hypothetical invariants comparing them across all the traced properties of the program for all test runs. It immediately discards the hypothetical invariant the moment it does not hold for a test run. Finally, all the invariants that are validated across all test runs are reported. Daikon can also be used to detect invariant violations in failing tests and as such may be used in a similar set-up as what we report here.

Gabel et al. implemented OCD, a tool which traces method calls and, using a predefined template as a model for specification inference, learns and enforces temporal specifications over method call sequence [9]. The algorithm suffers from two limitations: (1) the template limits the sequence to comprise only two method calls and (2) the sequences inferred from a limited window size. The efficiency of the algorithm critically depends on the window size. Experimenting with Eclipse and Ant, the tool detected a few anomalies as violations of inferred sequences, though the anomalies did not result in program crashes.

Pradel et al. proposed a dynamic analysis technique to infer specifications of correct method call sequences [13]. The technique focuses on object collaboration, namely objects and method calls used together in the execution of a single method. By running a software program, the technique traces method calls, computes object collaborations and identifies patterns among these collaborations. From these patterns, the technique infers the legal method call sequence in the form of finite state machines.

To certain extent, our research on SPEQTRA is complementary to the previous ones. We use method call sequences of a class from passing tests, which can be assumed as usage patterns of the program. On the other hand, the sequences in failing tests can be considered as deviant behaviour.

6. THREATS TO VALIDITY

Following the template for case studies in [18], we discuss the threats to validity that can affect our results.

Threats to **external validity** correspond to the generalizability of our experimental results. Our study is limited to the object oriented system *NanoXML*. Although *NanoXML* is small project, it represents a good testbed since it provides documented tests and faults for replicating our study. Moreover, by using the same case study of Dallmeier et al. [4], we were able to verify the impact of removing the sliding window and adopting different similarity coefficients for mining faults in stack traces. Nevertheless, it is desirable to replicate our findings using other projects.

Threats to **internal validity** concern confounding factors that can influence the obtained results. Our approach leverages on the fault's "ability" of changing the stack trace generated by software execution. In that sense, we localise faults by pointing out the class that has different method call sequences (in passing and failing tests) assuming that such deviation is due to the fault. This assumption is a key-element in spectrum based fault localisation techniques based on method call sequences [4, 17] and our results confirm its general validity. On the other hand, there are cases where it does not apply. In *NanoXML* there is (only) one faulty class that cannot be localised —with our approach— since the fault is caused by a variable accessed without any method call.

Threats to **construct validity** focus on how accurately the observations describe the phenomena of interest. Our experiment relies on the correct identification of fault responsible for test failure. From this point of view, we do not have threats to construct validity since we inject one fault at the time. When the fault is injected, otherwise the test passes.

Threats to **reliability validity** correspond to the degree to which the same data would lead to the same results when repeated. We describe all steps of our technique and provide references on any tool or library involved in the analysis. The case study we use is publicly available in the Software-artifact Infrastructure Repository[4], a repository created for supporting rigorous controlled experimentation with program analysis and software testing techniques [6].

7. CONCLUSION

In this paper we presented a novel spectrum based fault localisation heuristic (named SPEQTRA) which used closed itemset mining to identify the characteristic method call sequences and the Jaccard similarity coefficient to rank the classes according to the likelihood of containing the fault. We compare our fault localisation heuristic with the state of the art (AMPLE) and demonstrate on a small yet representative case (*NanoXML*) that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE. In particular, SPEQTRA can immediately pinpoint the faulty class in 56% of all test runs (against 40% for AMPLE); while on average 12% of the executed classes must be inspected to find the location of the fault (against 20% for AMPLE). From anecdotal evidence, we deduce that the main reason why SPEQTRA performs better than AMPLE is due to closed itemset mining: this filters out repetitive method calls (e.g. contained in loops) and avoids the arbitrary upper limit imposed by the sliding window. Nevertheless, for a few faults AMPLE provides a better ranking than SPEQTRA, caused by call sequences appearing in both failing and many passing tests which reduced the weight of sequences.

Over the course of this research, we have made the following contributions:

- *Replication Experiment.* We conducted a replication of the AMPLE experiment performed by Dallmeier et al. [4]. We used the same data (the *NanoXML* case provided in the Software-artifact Infrastructure Repository [6]) and confirmed the numbers provided in the original report.
- *Alternative Heuristic.* We proposed an alternative spectrum based fault localisation heuristic (named SPEQTRA). We compared it against the results from AMPLE. We demonstrate that the ranking of classes proposed by SPEQTRA is significantly better than the one of AMPLE.
- *Anecdotal Evidence.* We collected some anecdotal evidence from the *NanoXML* case interpreting the main differences between the two heuristics.

Fault localisation heuristics are particularly relevant in modern software engineering owing to the increasing popularity of continuous integration. Continuous integration states that software engineers should merge their working copies

[4]http://sir.unl.edu/portal/index.php

with the main branch several times a day using a suite of automated tests to verify the correctness of the build. When one of the thousands of tests fails, the corresponding fault should be localised as quickly as possible as development can only proceed when the fault is repaired. In that sense our work shows that while the state of the art is rapidly advancing, it is worthwhile to make improvements on research from a decade ago.

8. ACKNOWLEDGMENTS

This work is sponsored by (i) the Higher Education Comission of Pakistan under a project titled "Strengthening of Univeristy of Sindh (Faculty Development Program)"; (ii) the Institute for the Promotion of Innovation through Science and Technology in Flanders through a project entitled "Change-centric Quality Assurance (CHAQ)" with number 120028.

9. REFERENCES

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, Nov. 2009.

[2] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN '02, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.

[3] B. Cornelissen, A. Zaidman, D. Holten, L. Moonen, A. van Deursen, and J. J. van Wijk. Execution trace analysis through massive sequence and circular bundle views. *Journal of Systems and Software*, 81(12):2252 – 2268, 2008.

[4] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550, Berlin, Heidelberg, 2005. Springer-Verlag.

[5] B. Daniel, V. Jagannath, D. Dig, and D. Marinov. Reassert: Suggesting repairs for broken unit tests. In *Proceedings of the Int'l Conference on Automated Software Engineering (ASE)*, pages 433–444. IEEE CS, 2009.

[6] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[7] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. Mccamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, 2006.

[9] M. Gabel and Z. Su. Online inference and enforcement of temporal properties. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 15–24, New York, NY, USA, 2010. ACM.

[10] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[11] A. Miller. A hundred days of continuous integration. In *Agile, 2008. AGILE '08. Conference*, pages 289–293, Aug 2008.

[12] M. Monperrus and M. Mezini. Detecting missing method calls as violations of the majority rule. *ACM Trans. Softw. Eng. Methodol.*, 22(1):7:1–7:25, Mar. 2013.

[13] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382, Washington, DC, USA, 2009. IEEE Computer Society.

[14] P. Runeson. A survey of unit testing practices. *IEEE Software*, 23(4):22–29, 2006.

[15] D. Ståhl and J. Bosch. Modeling continuous integration practice differences in industry software development. *Journal of Systems and Software*, 87(0):48 — 59, 2014.

[16] N. Tillmann and W. Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4), 2006.

[17] J. Tu, L. Chen, Y. Zhou, J. Zhao, and B. Xu. Leveraging method call anomalies to improve the effectiveness of spectrum-based fault localization techniques for object-oriented programs. In *Quality Software (QSIC), 2012 12th International Conference on*, pages 1–8, Aug 2012.

[18] R. K. Yin. *Case study research: Design and methods*. Sage publications, 2013.

[19] A. Zaidman, B. V. Rompaey, van Arie van Deursen, and S. Demeyer. Studying the co-evolution of production and test code in open source and industrial developer test processes through repository mining. *Empirical Software Engineering*, 16(3):325–364, 2011.

[20] M. J. Zaki and C. J. Hsiao. CHARM: an efficient algorithm for closed itemset mining. In *Proceedings of the Second SIAM International Conference on Data Mining, Arlington, VA, USA, April 11-13, 2002*, pages 457–473, 2002.

[21] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann, 2005.