Mining Sequential Patterns of Predicates for Fault Localization and Understanding

Zebao Gao, Zhenyu Chen, Yang Feng, Bin Luo
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Corresponding author: zychen@software.nju.edu.cn

Abstract—Fault localization has been widely recognized as one of the most costly activities in software engineering. Most of existing techniques target a single faulty entity as the root cause of a failure. However these techniques often fail to reveal the context of a failure which can be valuable for the developers and testers to understand and correct faults. Thus some tentative solutions have been proposed to localize faults as sequences of software entities. However, as far as we know, none of these pioneering works consistently handles execution data in a sequence-oriented way, i.e., they analyze suspiciousness of software entities separately before or after the construction of a faulty sequence.

In this paper, we establish a systematic framework based on sequential-pattern mining to assist fault localization. We model the executions of test cases as sequences of predicates. Our framework outputs sequential patterns which are more likely related to the actual faults based on a 3-stage procedure: a pre-processing stage to prune sequences of predicates, a mining stage to discover candidate sequential patterns based on the revised SPADE mining algorithm, and a ranking stage to obtain top K results according to our novel metrics. The obtained sequential patterns of predicates can not only provide information about the locations of faults, but also convey valuable context information for understanding the root causes of software failures. A preliminary experiment on some widely used benchmarks was conducted to evaluate the performance of our framework. The experimental results show that our technique is effective and efficient in revealing causes of failures.

I. INTRODUCTION

Localizing faults are widely known as one of the most time consuming activities in software engineering. Typically when an error occurs, the software developer has to painstakingly trace back from the point where the program crashes or where the error finally emerges on output. In most cases, the information available to the developers is just stack traces at the spot where program crashes. With such limited information, manual debugging is inevitably a tedious and error-prone process.

A plenty of automatic fault localization techniques are raised by researchers to save the efforts of software developers and testers. A large portion of the most well known automatic fault localization techniques use dynamic behaviors of softwares to find out the suspicious code which may lead to failures. In this approach, the subject program will be instrumented in order to record the runtime coverage of software entities such as statements, branches or functions/methods. Then the program is executed on a set of test cases, and the traces of failed/passed execution are collected.

The collected data can be used in various forms to help localizing faults. One distinguished group of techniques are called spectrum-based fault localization techniques that adopt statical methodologies to rank software entities according to how likely they are faulty. Although spectrum-based techniques adopt statical approaches that require lighter instrumentation and are more effective in performing computation on their metrics, there are some disadvantages with these techniques. One inherent shortage is that there is no reasoning in the context of failed execution since all software entities are computed as independent units. Thus we may lose the chance to reveal the causes of failures in the running contexts. However, this information can be very valuable for developer since many software failures are often triggered by some certain runtime contexts.

The spectrum-based techniques often assume “perfect understanding of faults”, but this assumption can be unrealistic in many cases. Due to the complexity of various modern softwares, it may be very difficult for the developers to understand the real reason of a software failure given only a list of software entities without any other supporting information. Even if given a faulty line of code, the software developer may not be able to understand it immediately by simply looking at the line of code in isolation. Aimed to provide more sufficient information which will assist the debugging process of software developers, we present an automatic software fault localization framework which reveals the failure-related context based on mining sequential patterns of predicates in this paper.

Software predicates are often used as predicator of faults in previous research on software analysis and fault localization [1] [2]. Predicates are also used in the well known statistical debugging infrastructure named Cooperative Bug Isolation (CBI) [3]. We use predicates in our technique to depict the execution of test case because predicates are directly related to crucially important branches in a software system and thus can show the control flow of the program. Using predicates, rather than statements as the software entity will also make our sequential pattern mining technique more scalable.

In our technique, the subject programs are firstly instrumented at the level of predicates to automatically collect the sequence of runtime values of executed predicates. For each test case in the test pool, we obtain a sequence of predicate values during its execution. When the test cases are executed on instrumented subject programs, the predicate sequences of both failed and passed tests are automatically recorded.

Then a pre-processing procedure is adopted to prune the
predicate sequences of failed executions. This will help reduce
the cost of sequence mining process and increase the precision
of mining results. Generally, we remove some duplicate edges
generated by looping in programs, and some less suspicious
edges through a preliminary estimation metrics. In this way,
we obtain a much more compact predicate sequence which
holds most parts of the bug signatures. Then we use the
sequential pattern mining algorithms to find out those fre-
quent sequential patterns of predicates among all the failed
sequences. Finally, we use our metrics to rank these sequential
patterns according to how likely they are related to faults.

The main contributions of this paper are:

- A novel systemic framework is established to auto-
matically mine bug signatures as sequential patterns of
predicates. The framework adopts 3 stages to mine bug
signatures effectively and efficiently.
- A preliminary experiment is conducted on some subject
programs to evaluate the performance of our technique
and compare with other techniques.
- We discuss how the fault localization technique can be
improved by different settings of the sequential pattern
mining algorithm. Some guidelines are provided to use
our technique in practice.

The rest of the paper is recognized as follows. Section 2
introduces the necessary background of our work. Section 3
gives an overview of our technique and illustrates its main
characteristics through a motivating example. Section 4 de-
scribes the framework and detail information of our technique.
The preliminary empirical study is presented in section 5 and
the threats to validity are described in section 6. Related work
is discussed in section 7. We conclude and make future plans
in section 8.

II. BACKGROUND

A. Spectrum-based Fault Localization

A program spectrum is a collection of data reflecting the
dynamic behavior of software on how often each software
entity is executed on failed/passed test cases. The software
entities can be statements, blocks, branches, predicates, meth-
ods, or other software units. When an instrumented software
is executed on a set of test cases, four numbers are obtained for
each software entity. These four numbers are generally notated
as \( a_{np}, a_{nf}, a_{ef}, a_{ep} >\), where the first half of the subscripts
indicate whether the software entity is executed (e) or not (n),
and the second half of the subscripts indicate whether the test
fails (f) or passes (p). Thus, \( a_{np}\) of a software entity is the
number of passed test cases which do not execute the entity;
and \( a_{nf}\) of a software entity is the number of failed test cases
which execute the entity.

Based on the four parameters above, many metrics are
proposed to evaluate the suspiciousness of software entities
and rank them according to how likely they are faulty. The
basic idea is that the more an entity is executed by failed test
cases and the less it is executed by passed test cases, the more
likely it is faulty. Tarantula [4] is a typical example.

\[
susceptibility(c)_{\text{Tarantula}} = \frac{a_{ef}}{a_{ef} + a_{nf}} + \frac{a_{np}}{a_{np} + a_{ep}}
\]  

(1)

As the first technique to apply spectrum-base analysis in
software diagnosis, Tarantula is widely recognized and we
will evaluate the performance of our technique via comparing
with it. Although a plenty of metrics have been proposed in
recent years by researchers, there are some limitations in these
techniques.

The major problem is that these techniques handle software
entities as independent individuals and neglect the software
as a whole system. This problem is a innate disadvantage
and may contribute to some bad effects. Firstly, measuring
software entities separately may result in imprecise ranking
results due to the noises caused by natural connections among
software entities. Secondly, these techniques assume perfect
definition and understanding of faults, and target to find a
single suspicious software entity which causes the software
failure. Whereas providing a certain isolated software entities
can be inconvenient for the developers to understand without
a runtime context. Thirdly, due to the limited data dimen-
sions available in these techniques, the space available for
researchers to explore new metrics in this system is narrow.
Some recent research successfully proved that many existing
metrics actually produce the same rankings [5].

B. Sequential Pattern Mining

Motivated by the discussions above, researchers are ex-
ploring new techniques for automatic fault localization. Some
novel techniques use the probabilistic program dependence
graph [6], sequential pattern mining [7] and graph mining [8].
Most of the time, software entities work together in a whole
system at runtime. Each software entity is executed in a certain
context and is affected by the previous executed entities. Thus,
it is valuable for us to learn the behaviors of a software system
through analysis on executed sequences of it. This approach is
valuable for revealing and understanding of the reasons leading
to failed execution.

In this paper, we adopt sequence mining techniques to rec-
ognize the sequential pattern of software entities that leads to
software failures. The task of sequence mining is to discovered
the attributes shared across time among a given data set. One
of the most widely used technique is to discover frequent
sequences. The problem can be formally defined as follows.

Definition 1 (Sequence): A sequence is an ordered list of
events. An sequence \( S \) is donated as \( \{ e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n \} \),
where \( e_k (1 \leq n \leq k) \) is an event.

We call the sequence which contains k items an \( k \) –
sequence. The sequence \( \{ A \rightarrow B \} \) consisting two events
(each event contains a single item) is a 2-sequence. In this
sequence, the event \( A \) occurs before event \( B \), and we
denote it as \( A \prec B \).

Given a database \( D \) consisting of a set of sequences, the
task of frequent sequential pattern mining is to find the
subsequences that a certain number of input-sequences share.
We will give formal definitions to make the goal of sequential pattern mining concrete and precise.

**Definition 2 (Subsequence):** A sequence $S_1$ is a subsequence of another sequence $S_2$ if there exists a one-to-one order-preserving mapping function $f$ which can maps the events in $S_1$ to events in $S_2$ such that:
1. $e_k \subseteq f(e_k)$ for any $e_k \in S_1$, and
2. for any $e_k, e_j \in S_1$, if $e_k < e_j$, then $f(e_k) < f(e_j)$.

For example, the sequence $\{A \rightarrow BC\}$ is a subsequence of $\{AB \rightarrow E \rightarrow BC\}$. Conversely, we can also say the latter sequence contains the former one.

**Definition 3 (Support):** The support of a sequence $S$ in a database $D$ is the number of sequences in $D$ that contain $S$, donated as $\sigma(S, (D))$.

Given a database $D$ and the user-specified threshold of support (denoted as $min\_sup$), the problem of mining frequent sequential patterns is to find all sequences with a support greater than or equal to $min\_sup$, and these sequences are often called frequent sequences.

The SPADE [9] algorithm adopts the vertical data structure and efficient join operations to index and look up data faster. In this way, the algorithm is efficient and scalable.

### III. Overview

#### A. Basic Idea

According to the PIE model, there are three characteristics that should be taken into consideration when analyzing the relations between the execution of test case and the final output (pass or fail): (1) the probability that the faulty entity is executed, (2) the probability that the execution of the faulty entity affects the data state of the software system, and (3) the probability that the affected data state has an effect on the outputs of the software system. The latter two characteristics motivate that it is valuable to analyze the whole trace from the point where a certain data state is affected to the point where an effect is presented on the outputs. This is beneficial to revealing and understanding the nature of the software faults.

Here are some important fundamental natures in our fault localization technique based on sequential pattern mining. Firstly, we treat the execution trace of a test case on the software as a whole inter-connected profile. We consider that the failure-related execution segments should be the common feature shared by most failed execution. Thus it is appropriate for us to obtain the fault signatures via sequential pattern mining. Secondly, since the data states of the software are affected by a certain relevant software entities intermittently, to obtain a compact and noise-free signature of a fault, we should try to pick only the failure-related entities in the sequence of executed entities. Thus it is proper to adopt this sequential pattern mining technique since it mines the sub-sequence without limitations on the intervals between software entities.

Now our goal is to fit the characteristics of sequential pattern mining technique into the domain of revealing bug signatures. In our technique, the value of an atomic or composite predicate used in a single statement is considered as a event. In the instrumentation stage, we will assign an ID to each predicate. When the instrumented predicate is executed, we will assign a sign to the recorded event to distinguish a true value from a false value. Each event is actually stored as a set in our mining algorithm, so that our technique can be easily extended to handle the different values of the atomic predicates within a composite predicate if necessary.

To collect the predicate sequence of a test case, we record the executed predicates as a list of events in chronological order. After executing a pool of test cases, we obtain a database containing a number of sequences. And these sequences will be used for mining the failure-related sequential patterns.

#### B. Example & Scenario

We illustrate the features of our technique in aiding debugging a snippet of sample code written in C (shown in Table I). The code snippet (method cal_sum_of_abs(int, int)) takes two integers as inputs, i.e., $a$ and $b$, and calculates the sum of the absolute values of the two numbers. The designer of this method adopts an 3-step procedure to fulfill the algorithm. Firstly, the program judges whether the signs of $a$ and $b$ are the same or different. Secondly, it calculates the sum of $a$ and $b$ if they have the same sign, or calculate difference of them otherwise. At last, if the previous step obtains a negative value, the program will get its opposite value. The result after the third step is considered the final output of this program.

<table>
<thead>
<tr>
<th>TABLE I: Buggy Code Snippet - An Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: <code>public int cal_sum_of_abs(int a, int b) {</code></td>
</tr>
<tr>
<td>2: <code>int result = 0;</code></td>
</tr>
<tr>
<td>3: <code>if(a * b &gt; 0) {</code></td>
</tr>
<tr>
<td>4: <code>result = a + b;</code></td>
</tr>
<tr>
<td>5: <code>} else {</code></td>
</tr>
<tr>
<td>6: <code>result = a - b;</code></td>
</tr>
<tr>
<td>7: <code>}</code></td>
</tr>
<tr>
<td>8: <code>/*a bug in the following line: */</code></td>
</tr>
<tr>
<td>9: <code>/*should be: if(result &lt; 0)*/</code></td>
</tr>
<tr>
<td>10: <code>if(a - b &lt; 0) {</code></td>
</tr>
<tr>
<td>11: <code>result *= -1;</code></td>
</tr>
<tr>
<td>12: <code>}</code></td>
</tr>
<tr>
<td>13: <code>return result;</code></td>
</tr>
<tr>
<td>14: <code>}</code></td>
</tr>
</tbody>
</table>

The program contains a bug in line 10. Rather than checking the sign of `result` from step 2, it checks the sign of $a - b$ by mistake. So the program will never fail after the execution of the "else" branch (line 5 and 6), even though the faulty line (line 10) is executed. In this case, even the developer is provided with the position of the faulty line, he/she may not be able to recognize or understand the reason behind the failure. The program can produce incorrect output only when the first branch (line 4) is covered. Thus that providing the information "the program may fail when line 10 is executed following line 4" to the developer will be very helpful for understanding and correcting the fault.

Assume we have the following test cases. The traces of each test case are also listed in Table II.

### Table II: Test Cases and Their Traces

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Traces</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>Run 1: <code>public int cal_sum_of_abs(1, 2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 2: <code>public int cal_sum_of_abs(2, 2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 3: <code>public int cal_sum_of_abs(2, -2)</code></td>
</tr>
<tr>
<td>T2</td>
<td>Run 1: <code>public int cal_sum_of_abs(-1, 2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 2: <code>public int cal_sum_of_abs(2, -2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 3: <code>public int cal_sum_of_abs(-2, -2)</code></td>
</tr>
<tr>
<td>T3</td>
<td>Run 1: <code>public int cal_sum_of_abs(-1, -2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 2: <code>public int cal_sum_of_abs(-2, -2)</code></td>
</tr>
<tr>
<td></td>
<td>Run 3: <code>public int cal_sum_of_abs(-2, 2)</code></td>
</tr>
</tbody>
</table>

The program fails in test case T3. The developer may fail to find the bug in line 10 due to the failure. This is a very common phenomenon in software development. The technique proposed in our paper can help the developer to understand and correct such bugs.
TABLE II: Sample Tests and their Execution Traces of Statements

<table>
<thead>
<tr>
<th>Number</th>
<th>Inputs</th>
<th>Trace</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2</td>
<td>{1 → 2 → 3 → 4 → 10 → 11}</td>
<td>Fail</td>
</tr>
<tr>
<td>2</td>
<td>-1 -2</td>
<td>{1 → 2 → 3 → 4 → 10 → 11}</td>
<td>Fail</td>
</tr>
<tr>
<td>3</td>
<td>-2 -1</td>
<td>{1 → 2 → 3 → 4 → 10 → 11}</td>
<td>Pass</td>
</tr>
<tr>
<td>4</td>
<td>1 -2</td>
<td>{1 → 2 → 3 → 6 → 10 → 13}</td>
<td>Pass</td>
</tr>
<tr>
<td>5</td>
<td>-2 1</td>
<td>{1 → 2 → 3 → 6 → 10 → 11}</td>
<td>Pass</td>
</tr>
<tr>
<td>6</td>
<td>1 -1</td>
<td>{1 → 2 → 3 → 6 → 10 → 13}</td>
<td>Pass</td>
</tr>
</tbody>
</table>

To better illustrate the characteristics of our sequential pattern mining technique, we will use the spectrum-based approach Tarantula as a comparison.

C. Applying Spectrum-based Fault Localization Techniques

As introduced in previous section, to make use of the metrics of spectrum-based fault localization techniques, firstly we need to calculate the following four numbers for each statement: \( a_{sup}, a_{ef}, a_{ep}, a_{ef} \). According to the metrics introduced in the previous section, we can calculate the suspiciousness of each statement. The results show that line 4 owns the greatest suspiciousness (0.80) and line 11 follows (0.67). Line 6 is at the bottom with a suspicious value of 0. All other lines, including line 10 (the actual faulty line), share a suspiciousness value of 0.50.

D. Applying Sequential Pattern Mining Technique

We will firstly adopt a pre-processing stage to prune the sequences of failed test cases. In this stage, we remove some duplicate edges generated by structures such as loops in programs, and some less suspicious edges through a preliminary estimation metrics. In this way, we obtain a much more compact predicate sequence which holds most parts of the bug signatures.

There is no loop in this example. Because the edges \( (1 \rightarrow 2) \), \( (2 \rightarrow 3) \), \( (10 \rightarrow 13) \) and \( (11 \rightarrow 13) \) have the same chance to occur in failed and passed execution, thus they are considered less likely to be related to the fault in the program and will be removed. The statements 1, 2 and 13 will also be removed from the sequences since no edge corresponding to these statements exists any more.

Now, we obtain two pruned sequences of failed executions: \{3 → 4 → 10 → 11\} and \{3 → 4 → 10\}.

Then the sequential pattern mining algorithm will be applied to calculate the frequent subsequences to obtain the bug signatures. If we set the \( min_{sup} \) to be 2, we will get 4 frequent subsequences with lengths greater than 1. To confirm if these frequent sequences can stand for the significative features of failed executions, we calculate the support of these sequences in the passed sequences. The following table illustrates the ratio between supports of these sequences in failed and passed executions. Among them the last two with the greatest ratio value are considered most suspicious. Since sequence \{3 → 10\} is contained by \{(3 → 4 → 10)\}, So only the sequence \{3 → 4 → 10\} will be returned by our technique as a bug signature.

We can grasp a clear idea why our technique based on sequential pattern mining is better in assisting revealing and understanding faults by comparing it with the results of spectrum-based fault localization technique (Tarantula here).

In this motivating example, line 4 is a software entity which may trigger the real faulty line, i.e., line 10, to fail. Whereas Tarantula treats line 4 as the most suspicious line, and line 10 share the same ranking with many other correct lines. This result will bring misunderstanding to developers, since he will be confused by the rankings and have to pay much attention reviewing a lot of lines to finally find the faulty line.

On the contrary, when we treat the execution traces of each test case as a whole, and try to find the patterns of execution which are most likely to lead to failures of the program, much better results are obtained. The most suspicious sequential pattern provided by our technique is \{3 → 4 → 10\}. This result not only contains the faulty line in it, but also provides a context for the developer to understand why the failures could happen. When the developer sees that the program will fail when line 10 is executed following lines 3 and 4, he can probably understand the reason of the failure and fix it much easier.

IV. OUR APPROACH

A. Framework

Fig. 1: Framework of Our Bug Signature Mining Technique

The framework of our bug signature mining technique is illustrated in Figure 1. The very initial inputs of our framework include a subject program and a set of test data the program can run on. And our framework will finally output a list of sequential patterns of predicates ordered according to their suspiciousness.
B. Data Collection

The first stage is sequence data collection. We use our tool to automatically scan the source code of the subject program and detect the predicates contained in it. Then the tool will assign a unique ID to each predicate. When a test case is run on the instrumented program, the values of each executed predicate will be recorded in the chronological order to form a sequence of predicate values. In our approach, we assign a sign to each predicate to distinguish the value of predicate when it is executed. For example, if the predicates No. 15 (predicate_15) is executed after the predicate No. 6 (predicate_6) in a test case. And assume the value of predicate_6 is false when it is executed and the runtime value of predicate_15 is true. Then the predicate sequence built for this case will be \{-6 \rightarrow 15\}.

C. Pre-processing

The second stage performs pre-processing on these collected predicate sequences. This is an important and indispensable step in our technique as it is well known that long, redundant sequences will largely increase both the time and space costs in the process of sequential pattern mining. This technique may even fail due to the limit of execution time, memory space or disk space without these pre-processing mechanisms. As shown in Figure 1, two major steps are involved in this stage to remove equivalent edges and non-suspicious edges respectively.

The first step in the pre-processing stage is to remove equivalent edges inside each failed sequence. In a sequence \( S = \{p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_n\} \) there exists \( n - 1 \) edges: \( (p_1 \rightarrow p_2), (p_2 \rightarrow p_3), \ldots, (p_{n-1} \rightarrow p_n) \). Among them, some edges can be equivalent. We formally define equivalent edges as follows:

**Definition 4 (Equivalent Edges):** Given two edges \( e_1 = (p_1 \rightarrow p_2) \) and \( e_2 = (v_1 \rightarrow v_2) \), \( e_1 \) and \( e_2 \) are equivalent edges iff:
1. Vertices \( p_1 \) and \( v_1 \) correspond to the same predicate and have the same value (both true, or both false) when executed.
2. Vertices \( p_2 \) and \( v_2 \) correspond to the same predicate and have the same value (both true, or both false) when executed.

Equivalent edges can exist inside one sequence or between two different sequences. There are many cases that may lead to duplicate edges inside the same sequence. One major cause is the loops in the program, and many other program characteristics such as jump or invoke can also generate equivalent edges inside one execution. We will remove these equivalent edges because most cases the duplication in the final output will not be helpful. On the contrary, it may make the result difficult to understand. Another big advantage of removing these equivalent edges is that it well greatly decrease both the search space of the mining algorithm and the output volume of the sequential pattern mining technique.

The second step of the pre-processing stage aims to remove non-suspicious edges in the failed sequences. Thus we adopt this step to obtain a smaller and more precise search space before applying the sequential pattern mining algorithm. To identify suspicious edges in the failed sequences. We introduce the following concepts.

**Definition 5 (Frequency):** Given a set of sequences \( D = \{S_1, S_2, \ldots, S_n\} \) and an edge \( e \), the frequency of \( e \) in \( D \) is the number of sequences in \( D \) which contains edge \( e \), i.e., \( \text{frequency}(e, D) = \text{Card}(\{S|S \in D, e \in S\}) \).

We define suspicious edge as follows:

**Definition 6 (Suspicious Edge):** Given a set failed sequences \( D_{\text{fail}} \), a set of passed sequences \( D_{\text{pass}} \) and an edge \( e \), \( e \) is a suspicious edge iff:
\[
\text{frequency}(e, D_{\text{fail}}) > \frac{\text{Card}(D_{\text{fail}})}{\text{Card}(D_{\text{pass}})}.
\]

The above mechanism to identify suspicious edges will pick out those edges that are more likely to appear in the failed sequences because they are more likely to be related to faults in the software. When pre-processing the predicate sequences of failed test cases, only suspicious edges will be retained. And if both incoming and outgoing edges of an predicate are removed, the predicate will also be removed from the sequence.

D. Sequential Pattern Mining

Given a set of pre-processed predicate sequences of failed execution, we aim to find the frequent subsequences which are more likely to present the common execution patterns all these failed execution share, since these execution patterns are potentially related to the behaviors of failed execution. Intuitively, a subsequence is related to a fault with high probability if it appears frequently in the set of faulty execution. Formally, the problem is defined as follows.

Given a set of pre-processed predicate sequences of failed execution \( D_{\text{fail}} \) and a threshold support \( \min_{\text{sup}} \) for frequent subsequences, find the set of all predicate sequences \( \text{Freq}_{\text{Subseqs}} \), i.e., the set of sequences \( S \) such that:
1. \( \sigma(S, D_{\text{fail}}) > \min_{\text{sup}} \), and
2. not exists \( S' \in \text{Freq}_{\text{Subseqs}} \) such that \( S \subseteq S' \).

As introduced above, an efficient sequential pattern mining algorithm SPADE is adopted to mine frequent sequential patterns of predicates. SPADE adopts a lattice-based approach to divide the problem space into sub-lattices to make the algorithm applicable for large data sets. It also adopts a bottom-up approach to find the whole set of frequent subsequences using efficient joins.

The general mining process is the same as the original SPADE algorithm. The algorithm firstly converses the sequential data format to vertical format to efficiently index all items. The frequent items or frequent 1-sequences can be easily obtained from the vertical data storage. And breadth first search technique can be used to build all frequent 2-sequences. After that, the algorithm divides the data space into equivalence classes according to the prefix (i.e. the first several items) of sequences. Finally, SPADE will enumerate frequent sequences of each class, and join all the results to build the whole set of frequent sequences.
enumerate_freq_seq([X])
  for all Aᵢ ∈ [X] do
    enumerate_supp_top_k(Aᵢ);
    Tᵢ = ∅;
    for all Aᵢ ∈ [X] with j > i do
      if σ(R, D) ≥ min_supp then
        Tᵢ = Tᵢ ∪ {R};
      end if
    end for
    if (Tᵢ ≠ ∅) then
      enumerate_freq_seq(Tᵢ);
      topK.insert(freqSeq)
      suspiciousness(s) = suspiciousness(freqSeq)
    end if
  end for

insert_into_top_K(freqSeq)
  S = {s | s ∈ topK ∧ k ∧ suspiciousness(s) = suspiciousness(freqSeq)}
  for each sequence s ∈ S do
    if freqSeq ⊆ s then
      return;
    else if s ⊆ freqSeq
      topK.remove(s);
    end if
  end for
  topK.insert(freqSeq);
  if topK.size() > K then
    topK.remove_last_element();
  end if

Fig. 2: Pseudo-code of Enumerating Frequent Sequential Patterns of Predicates in an Equivalence Class

Fig. 3: Pseudo-code of Inserting a Frequent Sequential Pattern into the List of Top K Most Suspicious Patterns

The revise is made on the process of enumerating frequent sequential patterns of predicates inside an equivalent class (see Figure 2). When enumerating frequent sequential patterns of predicates in an equivalence class, we adopt a depth-first search approach. And we make a little modification on the original enumerating process to efficiently obtain the top K most frequent sequential patterns.

To efficiently obtain the top K most suspicious sequential patterns among all frequent sequential patterns discovered by the mining algorithm, we integrate the top-k ranking algorithm into the mining process. Each time the mining algorithm SPADE discovers a frequent sequential pattern, we calculate its suspiciousness value and store this pattern in the top-k list if the value is big enough. During this process, we also check the contain relations among sequential patterns to ensure that among the sequential patterns with the same suspiciousness value, no sequential pattern is contained by any other pattern. The pseudo-code for this algorithm is shown in Figure 3.

E. Ranking Bug Signatures

Given a set of failed sequences of predicates, we can now obtain a set of frequent sequential patterns of predicates together with their supports after the process of previous stages. Before providing the results to the developer, we need a metrics to rank these sequential patterns of predicates according to how likely they are related to the faults in the program.

To obtain the suspiciousness of each sequential patterns S, we combine the information from the database of failed sequences D_FAIL and the database of passed sequences D_PASS to compose the metrics.

$$\text{Suspiciousness}(S) = \frac{\sigma(S, D_{\text{pass}})}{\sigma(S, D_{\text{pass}}) + \varepsilon}$$

During the stage of sequential pattern mining, we try to find a subsequence of predicates that appear often in the failed sequences based on the idea that the more likely a sequential pattern is related to a fault, the more frequently it appears in the database of failed sequences. In addition, we combine the information from the passed execution to make the final ranking more precise. We take the support of the sequential pattern in the database of passed sequences as the denominator because we are intuitively motivated that the less a sequential pattern appears in the database of passed sequences, the more likely it is related to faults. We add the support of the sequential pattern in the database of passed sequences by a small number $\varepsilon$ ($\varepsilon$ is equivalent to 0 in our current implementation), to solve the problem when $\sigma(S, D_{\text{pass}})$ is 0.

Please notice that the cost of this stage is much smaller than the previous mining stage, even though the supports of previously found frequent sequential patterns need to be calculated in this stage. The problem in the mining stage is a searching problem and we need to enumerate sequences of items in the database of failed sequences. Whereas in this step, we simply scan the database of passed sequences to count the support values of given sequences.

Finally, the ranked sequential patterns of predicates will be provided to the developer for localizing and understanding faults.

V. EMPIRICAL STUDY

In this section, we describe our experiments to evaluate the performance of our fault localization technique based on sequential pattern mining. We experiment with the widely used Siemens Suite developed by researchers of Siemens Corporation [10]. These data sets have been widely used by researchers in their previous work on fault localization [11][12][13]. The general information of our subject programs is shown in Table III.

<table>
<thead>
<tr>
<th>Program</th>
<th>versions</th>
<th>LOC</th>
<th>Predicates</th>
<th>Test Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>schedule</td>
<td>5</td>
<td>284</td>
<td>94</td>
<td>2650</td>
</tr>
<tr>
<td>schedule2</td>
<td>5</td>
<td>235</td>
<td>29</td>
<td>2710</td>
</tr>
<tr>
<td>punt_tokens</td>
<td>6</td>
<td>362</td>
<td>68</td>
<td>4113</td>
</tr>
<tr>
<td>replace</td>
<td>6</td>
<td>512</td>
<td>37</td>
<td>5542</td>
</tr>
</tbody>
</table>

TABLE III: Subject Programs

We built our tool to instrument these subject programs. The execution traces of predicates during runtime was recorded and used as inputs for our fault localization technique. We firstly pruned the failed sequences of predicates to remove equivalent edges and non-suspicious edges. Then we mined the frequent sequential patterns in the failed execution. All these sequential

patterns were then assigned suspiciousness values according to their appearance in failed and passed execution. Finally, we obtained top $K$ most suspicious sequential patterns for each faulty version of subjects.

A. Experimental Results

We propose the following research questions to better illustrate the performance of our technique. **RQ1.** Can the preprocessing stage significantly decrease the search space of our fault localization technique? **RQ2.** What is the performance of our technique in revealing faults? **RQ3.** How can the threshold of support, or $\min_{sup}$, affect the output of our technique?

**Addressing RQ1:** We evaluated the effects of the preprocessing stage by comparing the size of database before and after the pruning procedure. Since we are mining frequent sequential patterns on failed executions, we measured the lengths of the sequences of predicates in the database. In Table IV, we presented the $\max$, $\min$ and average lengths of sequences before and after the pruning process. As shown in the table, more than 85% of original redundant data was removed, and for print_tokens2 and replace, more than 95% of original data was pruned. Through this procedure, the search space of our fault localization technique based on sequential pattern mining is largely decreased, thus our technique becomes much more applicable.

**TABLE IV: Lengths Before & After Pre-processing**

<table>
<thead>
<tr>
<th>Program</th>
<th>Len-Orig</th>
<th></th>
<th>Len-After</th>
<th></th>
<th>Pruned</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Max</td>
<td>Min</td>
<td>Avg</td>
<td>Max</td>
<td>Min</td>
</tr>
<tr>
<td>schedule</td>
<td>1516</td>
<td>35</td>
<td>605</td>
<td>104</td>
<td>31</td>
</tr>
<tr>
<td>schedule2</td>
<td>3425</td>
<td>128</td>
<td>1038</td>
<td>110</td>
<td>54</td>
</tr>
<tr>
<td>print_tokens2</td>
<td>27124</td>
<td>267</td>
<td>2090</td>
<td>263</td>
<td>150</td>
</tr>
<tr>
<td>replace</td>
<td>38426</td>
<td>22</td>
<td>2561</td>
<td>147</td>
<td>19</td>
</tr>
</tbody>
</table>

**Addressing RQ2:** The final output of our technique is a list of K most suspicious results that may be related to the faults in the software. When the ranked list is provided to developers, they will review the results in order. We evaluate RQ2 firstly by measuring the number of sequential patterns needed to be scanned/reviewed by the developers to find the cause of program failures. In this part, we adopt the widely recognized spectrum-based technique Tarantula as a comparison. We measure the number of predicates needed to be inspected by Tarantula to reach the faulty points of the programs. In this comparison, we compare the number of sequences and predicates in these two approaches because of their different output structures. Reviewing one sequential pattern could cost more than scanning one single predicate for a software developer, whereas through our experience, we found that the time cost was less when developers were reviewing sequences since the context information provided by the sequences could effectively reduce the difficulty of understanding.

The results are shown in Figure 6. The x-axis stands for the four different subject programs, and the y-axis stands for the number of results needed to be reviewed to finally find the faults. The results show that our technique based on sequential pattern mining outperforms Tarantula in a relatively large degree for most applications.

![Fig. 6: Number of Results Reviewed to Find Faults](image)

On the other hand, we evaluated the performance of our technique based on two widely used measures: precision and recall. Precision refers to the proportion of sequential patterns in the top-K lists which highlights the faults, and recall refers to the proportion of faults that can be highlighted by the output lists. To obtain a overlook of these two measures, we have $F - Measure$ as follows:

$$F - Measure = \frac{2 \cdot precision \cdot recall}{precision + recall}$$

(2)

In Figure 4, we show the precisions and recalls of each subject program respectively. Each figure in Figure 4 shows the performance of our technique on one program. The x-axis stands for the value of K, i.e., the number of most suspicious results returned by our technique. The y-axis stands for the values of precision, recall or F-Measure. These figures show that for most applications, our technique obtains a high recall (mostly greater than 80%) at a very early stage. Besides, our technique reaches a high precision when only a few most suspicious results are returned. It is common that precision will decline as the capacity of the returned results increases, but it has relatively light influence on the developers since they can stop the reviewing process early if the top results are precise.

In addition, we present the F-Measure values of all subject programs in a single plot in Figure 7 to provide an overview on the results of all these applications.

**Addressing RQ3:** Previous results are obtained when we set the threshold of support value ($\min_{sup}$) as 100% of the size of the database. It is an interesting question about how $\min_{sup}$ can affect the final results of technique. We

1The returned sequential patterns are contained by all sequences in the database.
conduct an empirical study on the subject schedule. The results are shown in Figure 5. Figure 5b to 5h shows the performance of our technique on schedule with 7 different values of \( \text{min}_\text{sup} \): 0.5, 0.6, 0.7, 0.8, 0.9, 0.95 and 1.0.

The curves in these figures are quite inconsonant as shown in the figures. This indicates that the \( \text{min}_\text{sup} \) will have a significant impact on the final results of our technique. To grasp an overview on the trends of the performance when the \( \text{min}_\text{sup} \) changes, we provide Figure 5a. For each \( \text{min}_\text{sup} \), we calculated the various F-Measure values as the capacities of returned lists changes in our experiments. Then we drew the box-plot when we have a set of F-Measure values for each different \( \text{min}_\text{sup} \) values. From the box-plot, we can see that our technique reaches its best performance when the \( \text{min}_\text{sup} \) is near 0.5 and 0.9. Resource to find out the concrete reasons of this phenomenon is very limited, and we propose that this may be caused by some general program characteristics. Most predicates lead to two branches, sometimes a failure occurs when one certain branch is covered (a \( \text{min}_\text{sup} \) of 0.9 will benefit in this case), while in some other cases, the failure

---

**Fig. 4: Precision, Recall and F-Measure of Each Subject Program**

![Fig. 4](image)

**Fig. 5: Precision, Recall and F-Measure of schedule with Different Values of \( \text{min}_\text{sup} \)**

![Fig. 5](image)
occurs either branch is covered (a min_sup of 0.5 will benefit in this case).

VI. Threats to Validity

The primary threats to the external validity come from the subjects and faults we use. All the four subjects are selected from the well known Siemens Test Suite which is extensively used not only in fault localization, but also in software analysis, test case selection and prioritization, etc. For each subject, we select a few faults instead of all the faults to evaluate our technique. It is mainly because that some faults are not revealed by any test case in the test pool. In addition, we omit the faults that can not be clearly ascribed to any certain predicates.

The threats to the internal validity are mainly due to the correctness of the implementation of our framework. We used some well-built components, such as SPADE [9]. We also try to minimize threats to internal validity by inspecting our codes carefully and conducting a thorough unit testing on our tools.

The major threat to the construct validity is that we do not use previous techniques [14] [7] that output paths or sequences as well. The main reason is that we do not have sufficient resource to reproduce experimental results of these techniques. In addition, the previous sequence-mining based technique was proposed without further experimental study. And it brings more difficulty for us to provide a visual comparison since the previous technique is an interactive framework which involves manual inspections by the developer. The comparison between our approach and traditional spectrum-based approach could be unfair because it is difficult to prove that it will cost less time for a developer to scan a sequence than an isolated entity, even though the sequences provide a clear runtime context which are helpful for the developer to understand the cause of failures.

VII. Related Work

Automatic fault localization techniques have long been a hot research topic. Spectrum-based fault localization techniques adopt statistical methodologies to rank software entities according to how likely they are faulty. Many different metrics are proposed based on this framework [4] [15] [13] [3] [12]. In addition, Xie et al. [16] proposed the metamorphic slice technique to assist fault localization based on the spectrum-based fault localization framework. In their work, the set of covered statements are used as execution slices of test cases, and the metamorphic relations were introduced to solve the problem of fault localization without oracles. Though efficient, all these techniques measured the suspiciousness of software entities in isolation. They are much different from our technique based on sequential pattern mining since we regard each trace of execution as a interconnected sequence.

The CBI framework [3] serves as a good platform for predicate-based fault localization. Zhang et al. [17] firstly introduced a novel technique to handle the problem of short-circuit evaluations in composite predicates. They divided each composite predicate into a sequence of automatic predicate to refine previous predicate-based techniques. Their usage of sequences was confined to a composite predicate, whereas our technique make use of the sequence of all predicates executed in a test case. And our framework can be easily extend to adjust to mining sequences of composite predicates, because in the SPADE algorithm, each event is actually stored as a set of items.

Recently researchers started to aware the importance of context information in solving the problem of fault localization. And context-aware techniques were proposed. Some work just took the context information into consideration to assist localizing for one single software entity [18]. However, some other researchers, such as Jiang et al, proposed to include fault related context information into the outputs of fault localization techniques [14]. In Jiang’s work, machine learning technique was introduced: feature selection technique was used to select fault-related predicates as bug predictors and clustering technique was used to group correlated predicates. In the final stage, The static control flow graph of the subject program was traversed to build faulty control flow path. Machine learning techniques are used in both techniques of theirs and ours, but these two approaches are essentially different. In their work, suspiciousness of and correlations between predicates are measured at the statistical level. Instead, in our approach, each individual chronological sequence of predicates is treated as a interconnected unity to mine precise runtime context.

In recent years, Hsu et al proposed fault localization techniques named RAPID which is based on mining sequences of executed software entities[7]. And Cheng et al proposed techniques of identifying bug signatures using discriminative graph mining [8]. Since both RAPID and our technique are localizing faults through mining techniques on sequences, we will demonstrate how our technique is different from RAPID.

Firstly, RAPID set the Tarantula threshold at 0.6 (a human-assigned parameter) and collected sequences of predicates with suspiciousness above this point. This pruning strategy essentially treated all the events in the sequence as separated units and could lead to unexpected bias in the latter sequence mining stage. While in our technique, we prune edges using a parameter free metrics instead of events. We remove events
from the sequence only when no edge corresponding to these events exists.

Secondly, in the mining stage, RAPID used the BIDE[19] algorithm to mine for longest common subsequences (i.e., the support of each result has to be 100%), while in our approach, we mine for frequent subsequences. RAPID’s approach could be more efficient since less sequences have to be enumerated, whereas our approach obtains more sufficient candidate sequences for our framework to evaluate the suspiciousness of them based on a comprehensive metrics. In addition, we perform a preliminary study on the impact of different min_sup.

At last, the RAPID technique used the least suspiciousness value in the sequence to stand for the suspiciousness of the whole sequence. As a result, the shortest sequences were often provided to the developers for feedback. Whereas in our technique, frequencies of the signatures in databases of failed and passed execution are taken as the first consideration and we sort the results based on our metrics. We need to inspect the length and parent-child relationship among sequences only when they share the same suspiciousness.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper, we introduce an automatic fault localization technique based on mining sequential patterns of predicates. In our technique, firstly instrument the subject programs to automatically collect the sequence of executed predicates of each test case in the test pool. Then a pre-processing stage is adopted to prune the predicate sequences of failed executions. This procedure will help reduce the cost of the mining stage and increase the precision of mining results. In the mining stage, we use the sequential pattern mining algorithms to find out those frequent sequential patterns of predicates among all the failed sequences. Finally, in the ranking stage, we use our metrics to rank these sequential patterns of predicates according to how likely they are related to faults in the software system.

We evaluate the performance of our technique on some widely recognized and used benchmark data sets. The experimental results demonstrate the performance of our technique on how fast and precisely our technique can help reveal and understand faults. In addition, we conducted experiments to reveal the impacts of different setups of the mining algorithms.

Although our technique preforms well on current benchmark data sets. Some improvements are planned to be made in the future work. Firstly, the time and space costs of sequential pattern mining algorithms severely limit the scalability of our technique. We will try to use more instant pruning inside the process of sequential pattern mining to make this technique more applicable on large databases. Secondly, some limitations on the length of returned results could also be a way to save the cost of our technique. It is a challenge to draw a conclusion on the proper length of predicates to be properly fit for fault localization, and some in-depth survey on this needs to be performed. Finally, it is promising if we can apply this technique on programs containing multiple faults.

REFERENCES