Using Pattern Position Distribution for Software Failure Detection

Chunping Li, Ziniu Chen, Hao Du
School of Software, Tsinghua University
Beijing 100084, China
E-mail: {cli, czn09}@tsinghua.edu.cn

Hui Wang, George Wilkie, Juan C. Augusto, Jun Liu
Faculty of Computing and Engineering, University of Ulster
Jordanstown, Northern Ireland, BT37 0QB, UK
E-mail: {H.Wang,FG.Wilkie,JC.Augusto,J.Liu}@ulster.ac.uk

Abstract

In this paper, we present a novel approach to software failure detection based on pattern position distributions as features. In this approach, we divide an execution sequence into several sections and then compute a pattern distribution in each section. The distribution of all patterns is then used as a feature to train a classifier. This approach outperforms conventional frequency based methods by more effectively identifying software failures occurring through misused software patterns. Comparative experiments show the effectiveness of our approach.

Keywords: Sequential Patterns, Classification Algorithm, Software Failure, Anomaly Detection.

1. Introduction

As time goes by, computer software is playing an increasingly important role in our daily lives. However, it is difficult to validate the correctness of software. When bugs occur in practice, costs can be tremendous. Bugs can cause huge financial losses each year, in addition to privacy and security threats. According to the US NIST’s (National Institute of Standards and Technology) report, software bugs cost the US economy $59.5 billion annually [3].

To reduce the harm caused by software failure, hidden defects must be found as soon as possible before they cause damage. Unfortunately, traditional manual code review or software testing methods are time consuming, labor intensive and imprecise. These methods are difficult to apply to large-scale or market-sensitive software systems. As a result, many researchers and industry devote much effort to developing automatic software failure detection techniques. The pattern-based software failure detection approach is one of the most important topics in this area. Patterns which are found in software usually correspond to programming rules or usage patterns [1]. In software sizing activities, it is common to look for often required logic such as for ‘Adding’, ‘Deleting’, ‘Amending’, ‘Searching’ and ‘Listing’ data from a data store. There will be consequent patterns associated with these...
functions. These patterns are intuitive and commonly found in software documentation, such as: the Resource Locking Protocol: `<lock, unlock>` or the Java Transaction Architecture (JTA) Protocol [5]: `<TxManager.begin, TxManager.commit>`, `<TxManger.begin, TxManager.rollback>`, etc. Software Patterns have also been used as part of re-use strategies when developing software systems. The seminal work by Erich Gamma et al [27] introduces many software patterns including the ‘Singleton’, ‘Observer’ and ‘Façade’ patterns which have been widely adopted by industry.

These patterns, which reflect interesting program behavior, can be identified (or mined) by analyzing a set of program traces. Traces are an ordered list of events [4], where an event can correspond to the invocation of a method, or the execution of a program statement, etc. From the data mining viewpoint, each trace can be considered as a sequence. A pattern (e.g., `<lock, unlock>`) can appear multiple times within a sequence. Each pattern may be divided by an arbitrary number of unrelated intervening events (e.g., lock -> resource use -> … -> unlock) [1].

Pattern mining is found in a wide variety of application domains such as intrusion detection, failure detection, program comprehension [2], bioinformatics, weather prediction, and system health management [6]. Various pattern mining methods are proposed such as frequent itemset mining [10], sequential pattern mining [11], closed pattern mining [22, 23], episode mining [12], iterative pattern mining [2] and Closed Unique Pattern mining [1]. Recently there has been interest in developing discriminative pattern-based classifiers. In [7], Cheng et al. mine frequent itemsets for classifying transaction data. In [8, 9], frequent connected subgraphs are mined for classifying graph data. On a related front, Lo et al. proposed a novel method to extract Closed Unique Patterns for software failure detection [1].

Pattern-based software failure detection was inspired by the emerging area of dynamic analysis where program traces are analyzed in order to infer or mine temporal program properties or patterns of behavior [2]. In the dynamic analysis point of view, software can be viewed as a series of program execution traces which demonstrate a program’s behaviors. When a program executes, it produces a massive amount of execution traces corresponding to its various behaviors. Some behaviors are desirable, while some others are not. These undesirable behaviors are often referred to as failures. A set of execution traces can be collected to construct a sequence database which is the basis of our analysis.

Generally speaking, pattern-based software failure detection employs a three-step framework [1], first, mine a set of patterns from program execution traces; secondly, perform feature selection to extract discriminative patterns for the purpose of classification. These selected patterns are treated as features and their occurrence frequencies are treated as corresponding feature values. Thirdly, these features are used to train a classifier to detect failures. So, more specifically, pattern-based software failure detection is a pattern frequency-based method.

Existing research on pattern frequency based methods has produced promising results. [1, 7] demonstrated that this approach is much more discriminative than single event approaches. But it has a natural weakness in that the research neglects the pattern’s position within the sequence. For example, consider the login pattern P1 = `<login, passwd>` and the set of user command sequences S0-S4 as shown in Table 1. Sequences S0-S3 represent normal daily profiles of a user while the sequence S4 is anomalous - one can never do any other operations before logging into the system. Although S4 indicates an obvious failure, we are unable to distinguish S0-S3 from S4 when using the pattern frequency based method because the pattern P0 = `<login, passwd>` does occur once in each of S0-S4. It is very clear that pattern frequency based methods loose their discriminating power in this case.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Sequences of User Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0</td>
<td>login, passwd, mail, ssh, ... , mail, web, logout</td>
</tr>
<tr>
<td>S1</td>
<td>login, passwd, mail, web, ..., web, web, web, logout</td>
</tr>
<tr>
<td>S2</td>
<td>login, passwd, mail, ssh, ..., mail, web, logout</td>
</tr>
<tr>
<td>S3</td>
<td>login, passwd, web, mail, ssh, ..., web, mail, logout</td>
</tr>
<tr>
<td>S4</td>
<td>mail, ssh, web, ... , web, mail, login, passwd, logout</td>
</tr>
</tbody>
</table>

From this example, we see how a number of software failures could occur through misused software patterns and merely using the pattern’s frequency as feature cannot detect such kinds of failures. Notice that the login pattern P0 occurred in the tail of S4, but occurred in the head of S0-S3. So, patterns occurring in the different positions of a trace are likely to represent different meanings. A pattern’s position may imply...
some important semantic information or design
costs. In the example, it was: before we do any
other operations, we must login to the system. By using
the pattern position information, we can easily identify
abnormal sequences which contain misused patterns. So
it is appropriate to consider using positional information
to enhance the discriminating power of patterns.

In this paper, we propose a novel approach for using
the pattern positional distribution to detect software
failure instead of occurrence frequency, which is used in
traditional approaches. We present experiments using
both synthetic and real-world datasets to show that the
classification performance is improved significantly
compared with existing research. Our approach, with
the scheme of positional distribution, can be combined
with various pattern mining algorithms, which makes it
very flexible.

The organization of this paper is as follows. Section 2
introduces the concept definitions related to the pattern
position distribution. Section 3 describes our failure
detection method based on the pattern position
distribution. In Section 4, we provide our experimental
results and comparative study with existing published
research work. Section 5 then contains our concluding
remarks and ideas for future work.

2. Basic Concepts

This section provides the definitions for the following
four concepts:

(i) Pattern Instance;
(ii) Section;
(iii) Instance Position; and
(iv) Pattern Position Distribution.

In pattern mining, we denote a software execution
sequence S as it corresponds to a path which a program
takes when executing from its start to the end point
when it terminates [1]. Where each is an event, an event
in turn corresponds to a unit behavior of interest. This
can correspond to the execution of a statement, a
method call etc. The set of traces or sequence database
is denoted by TDB (Traces Database). An example
TDB is shown in Table 2.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>S₁</td>
<td>&lt;D, B, C, D, A, E, B, E, D, C, E, C, D, F, D, B, A&gt;</td>
</tr>
</tbody>
</table>

Table 2 Traces Database

**Definition 1: Pattern Instance** Given a pattern
\( P <e_0, e_1, \ldots, e_{n-1} > \), a substring \( f(f_0, f_1, \ldots, f_m-1) \) in a
sequence \( S \) in TDB (traces database) is an instance of \( P \)
iff it is of the following QRE expression
\( e_0 ; [ \ldots e_0, \ldots, e_{n-1}] * ; e_1 ; \ldots ; [ \ldots e_0, \ldots, e_{n-1}] * ; e_{n-1} \).

An instance is denoted by a triplet \( (\text{seq-id}, \text{start-pos}, \text{end-pos}) \),
where \( \text{seq-id} \) refers to the ID of a sequence
\( S \) in the database while \( \text{start-pos} \) and \( \text{end-pos} \) refer to
the starting point and ending point of a substring in \( S \).
All indices start from 0.

The starting point and ending point can indicate the
absolute position of an instance but cannot represent the
whole positional information on their own because the
length of sequences in TDB may not be equal. For example,
consider a pattern \( P = <A, B> \) and the two
sequences \( S_0, S_1 \) shown in Table 2. There are two
instances \( I \) \((0, 5, 7), J \) \((1, 5, 7) \) of pattern \( P \). The length
of \( S_0 \) is 10 and the length of \( S_1 \) is 20. Although
\( I, J \) have the same absolute position, \( I \) appears in
the second half of \( S_0 \) while \( J \) appears in the first half of \( S_1 \).
So, the same absolute position may indicate different
positional information. To avoid the weakness of the
absolute position, we use the relative position to
represent the positional information. In order to use
relative position, we divide all sequences into \( N \)
'sections' separately, and then determine what section
or sections an instance belongs to. In this way, we can
position an instance.

**Definition 2: Section.** Divide a sequence
\( S_{\text{seq-id}} <e_0, e_1, e_2, \ldots, e_{n-1} > \) into \( N \) parts
\( s.t. \sum_{r=0}^{N-1} \text{part}_r = <e_0, e_1, e_2, \ldots, e_{n-1} > \) and \( \bigcap_{j=0}^{N-1} \text{part}_j = \emptyset \), this
Partition divide S_{seq-id} into N sections iff
∀i, j. 0 ≤ i, j ≤ N - 1, s.t. |part_i| - |part_j| ≤ ±1, where
part_i denotes the i-th part of the sequence and |part_i| denotes the number of the event in part_i.

After dividing a sequence into N sections, a sequence can be denoted by (section_{0S}, section_{1S}, ..., section_{NS}), and then we can determine the ‘instance position’ which is given in the following definition.

**Definition 3: Instance Position.** Given an instance \( i \) \( (\text{seq-id}, \text{start-pos}, \text{end-pos}) \), a sequence divides into N sections \( S_{seq-id} \) (section_{0S}, section_{1S}, ..., section_{NS}) that contains \( i \). The position of \( i \) is represented as (seq-id, start-section, end-section), where 'start-section' refers to the ID of the section s.t. start-pos_{section_{iS}} ≤ start-pos_{i} ≤ end-pos_{section_{iS}} and end-section refers to the ID of the section s.t. start-pos_{section_{iS}} ≤ end-pos_{i} ≤ end-pos_{section_{iS}}, where start-pos_{i}, and end-pos_{i}, refer to the starting point and ending point of \( i \). start-pos_{section_{iS}} and end-pos_{section_{iS}} refer to the starting point and ending point of section_{iS}.

When we have obtained all instance positions of pattern P, we can compute P’s position distribution.

**Definition 4: Pattern Position Distribution.** Pattern P’s position distribution in sequence S will be denoted by \( PD_{P,S} = \text{(count}_0, \text{count}_1, ..., \text{count}_{NS}) \), where \( PD_{P,S} \) means pattern P’s position distribution in sequence S, N refers to the number of sections, count_i refers to the number of P’s instances in the section, Instance I in the section \( k \) means
∀j. start-pos_{j} ≤ j ≤ end-pos_{j} s.t. start-pos_{section_{jS}} ≤ j ≤ end-pos_{section_{jS}}
A part of Instance I in the section \( k \) means
∃j. start-pos_{j} ≤ j ≤ end-pos_{j} s.t. start-pos_{section_{jS}} ≤ j ≤ end-pos_{section_{jS}} .

As an example, consider a pattern \( P = \langle A, B \rangle \) and the TDB shown in Table 3, the set of instances of P denoted by \( Inst(P) \) are represented as: \( Inst(P) = \{(0,2,4), (0,5,7), (1,2,4), (1,7,8)\} \). Then we divide all sequences into 4 sections separately. For \( S_0 \), \( section_0 = \langle D, B, A \rangle \), \( section_1 = \langle F, B \rangle \), \( section_2 = \langle A, F, B \rangle \) and \( section_3 = \langle C, E \rangle \). For \( S_1 \), \( section_0 = \langle D, B, A \rangle \), \( section_1 = \langle D, B \rangle \), \( section_2 = \langle B, B \rangle \) and \( section_3 = \langle A, B \rangle \). Instance position for all instances belonging to \( Inst(P) \) will be represented as (0, 0, 1), (0, 2, 2), (1, 0, 1) and (1, 3, 3) separately. Pattern P’s position distribution in sequence \( S_0 \) is denoted by \( PD_{P,S_0} = (1, 1, 1, 0) \) and P’s position distribution in sequence \( S_1 \) is denoted by \( PD_{P,S_1} = (1, 1, 0, 1) \).

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_0 )</td>
<td>( \langle D, B, A, F, B, A, F, C, E \rangle )</td>
</tr>
<tr>
<td>( S_1 )</td>
<td>( \langle D, B, A, D, B, B, A, B \rangle )</td>
</tr>
</tbody>
</table>

### 3. Pattern Position Distribution Based Software Failure Detection

In this section, we present a four-step approach for the software failure detection based on pattern position distribution. First, we extract a set of patterns from a Traces Database (TDB). Secondly, pattern selection is performed to select discriminative patterns. Thirdly, we compute the position distribution for each selected pattern. This distribution will be used as the features. Finally, features are used to train a classifier to detect software failure.

#### 3.1. Pattern Mining

Creating a pattern mining algorithm is an essential component to building the pattern-based classifier. Our position distribution-based approach can be combined with various pattern mining algorithms. We use two different pattern mining algorithms separately. The first algorithm is the state of art Closed Unique Iterative Pattern mining algorithm proposed by David Lo et al [1]. This algorithm performs a depth-first traversal of the search space to grow patterns. It first computes frequent single events in the traces database (TDB). The frequent events are then grown in a depth-first fashion. Unique pattern detection [1] and InfixScan pruning strategies [2] are performed to cut the search space of non-closed patterns to get a compact set of patterns. The second algorithm is the classical FP-growth algorithm proposed by J. Han et al [26]. The FP-growth algorithm represents the transaction database as a prefix tree which is enhanced with links that organize the nodes into lists referring to the same item. The search is carried out by projecting the prefix tree, working recursively on the result, and pruning the original tree.
3.2. Pattern Selection

A large set of patterns will be mined from the set of failing and normal traces. Some of these patterns may be indiscriminative. To reduce the number of patterns and eliminate those that are indiscriminative, pattern selection is performed.

We employ the popularly used statistical measurement, e.g., Fisher score [14], this score is defined as follows.

$$F_r = \frac{\sum_{i=1}^{k} n_i (u_i - \mu)^2}{\sum_{i=1}^{k} n_i \sigma_i^2}$$  \hspace{1cm} (1)

where $n_i$ is the number of data samples in class $c_i$, $\mu_i$ is the average pattern value in class $c_i$, we treat a pattern’s instance number in a sequence $S$ as the corresponding pattern value. $\mu$ is the average pattern value in the whole dataset. $\sigma_i$ is the standard deviation of the pattern values in class $c_i$. $k$ is the number of classes. Assumed that $x_{ij}$ is the pattern value for the $j^{th}$ instance in class $c_i$, then $\mu$, $\mu_i$ and $\sigma_i$ are defined as

$$\mu = \frac{\sum_i \sum_j x_{ij}}{\sum_i n_i}, \quad \mu_i = \frac{\sum_j x_{ij}}{n_i}, \quad \sigma_i = \sqrt{\frac{\sum_j (x_{ij} - \mu_j)^2}{n_i}},$$

respectively. According to the formula, if a pattern has very similar values within the same class and very different values across different classes, the Fisher score becomes large, which means this pattern is very discriminative to differentiate instances from different classes. Otherwise, it is not discriminative.

A pattern selection algorithm is proposed in [1]. The algorithm ranks the patterns according to their Fisher Score and then select patterns in descending order until all data instances covered by at least $\delta$ times have been processed.

**Algorithm 1: Pattern Selection**

**Inputs:** Pattern set $P$, Trace Database TDB, Coverage Threshold $\delta$.

**Output:** A selected pattern set $P$.

1: for each pattern $Pat_i \in P$
2: compute Fisher score of $Pat_i$
3: sort $P$ in decreasing order of Fisher score;
4: for each pattern $Pat_i \in P$
5: if $Pat_i$ covers at least one sequence in TDB
6: add $Pat_i$ into $P$
7: remove $Pat_i$ from $P$
8: if a sequence $S$ in TDB is covered $\delta$ times
9: remove $S$ from TDB;
10: if all sequence are covered $\delta$ times or $P=\emptyset$:
11: break;
12: return $P$;

3.3. Position Distribution based Features

The conventional feature representation approach simply uses a pattern’s occurrence frequency as a feature value, this method is straightforward but imperfect. If a pattern’s frequency is the same in two different sequences, no matter what position the pattern instance appears in, in the viewpoint of this method, the two sequences are exactly the same. However, patterns occurring in different positions of a trace are likely to represent different meanings. For example, initialization patterns usually appear in the head of a normal sequence; data process patterns mainly in the middle and tail of a normal sequence etc. Patterns which do not appear in the “right” place usually indicate areas of potential software failure. Simple use of frequency as a feature would lose a lot of information and thereby reduce the discriminative power.

As discussed in Section 2, we use relative position to build positional information. For this, a program trace will be divided into $N$ sections. That is, a sequence is partitioned into $N$ nearly equal parts. There may be several ways to divide a sequence into $N$ sections. As an example, for a sequence $S=\langle D, B, A, F, B, A, F, B, C, E\rangle$, there are 6 ways to divide $S$ into 4 sections. All 6 solutions are show in Table 4. If each sequence in TDB randomly chooses its partition strategy, then different pattern position distributions may be deduced in repeated experiments and this would lead to unstable results. In order to unify partition strategies for each sequence, we use the following partition method to allocate every event into a corresponding section: for event $e$ at the position $i$ in sequence $S_{seq-id}$, we allocate $e$ into section $j$, where

$$j=\left\lfloor \frac{N \times \text{seqlen(seq-id)}}{\text{seqlen}} \right\rfloor$$  \hspace{1cm} (2)
N denotes the number of sections, \( \text{seqlen} (\text{seq} - \text{id}_j) \) denotes that the total number of events of the sequences whose ID is \( \text{seq} - \text{id}_j \). Using the above strategy, for the \( j \)th instance of pattern \( P_j \), we denote it by \( \text{Inst}(P_j) = (\text{seq} - \text{id}_j, \text{start} - \text{pos}_j, \text{end} - \text{pos}_j) \), the corresponding start-section is

\[
\text{start-section}_j = \left[ \text{start} - \text{pos}_j \times \frac{N}{\text{seqlen}(\text{seq} - \text{id}_j)} \right] \quad (3)
\]

Similarly, the corresponding end-section is

\[
\text{end-section}_i = \left[ \text{end} - \text{pos}_i \times \frac{N}{\text{seqlen}(\text{seq} - \text{id}_i)} \right] \quad (4)
\]

As \( \text{Inst}(P_j) \) across multiple sections from \( \text{start} - \text{section} \) to \( \text{end} - \text{section} \), the value between \( \text{count}_{\text{start} - \text{section}} \) and \( \text{count}_{\text{end} - \text{section}} \) all plus 1.

### Table 4: all solutions to divide S into 4 sections

<table>
<thead>
<tr>
<th>Solutions</th>
<th>Section partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution1</td>
<td>(&lt;D, B, A, [F, B], A, F, B, C, E&gt;)</td>
</tr>
<tr>
<td>Solution2</td>
<td>(&lt;D, B, A, [F, B], A, F, B, C, E&gt;)</td>
</tr>
<tr>
<td>Solution3</td>
<td>(&lt;D, B, A, [F, B], A, F, B, C, E&gt;)</td>
</tr>
<tr>
<td>Solution4</td>
<td>(&lt;D, B, A, [F, B], A, F, B, C, E&gt;)</td>
</tr>
<tr>
<td>Solution5</td>
<td>(&lt;D, B, A F, B, A, F, B, C, E&gt;)</td>
</tr>
<tr>
<td>Solution6</td>
<td>(&lt;D, B, A, F, B, A, F, B, C, E&gt;)</td>
</tr>
</tbody>
</table>

In this way, we can determine the distribution of each pattern in the sequence, but we can’t use it directly as a feature vector. For instance, consider pattern \( P \) and its distribution in sequence \( S \): \( PD_{S,S} = (5, 10, 5, 10) \) and its distribution in sequence \( S' \): \( PD_{S',s} = (55, 60, 55, 60) \). It is easy to determine that these two distributions are very similar except for their baseline. For similarity analysis of distributions, we need to consider differences in the baseline and scale (or amplitude). A straightforward approach for solving the baseline and scale problem is to apply a normalization transformation [15]. For example, a distribution \( (\text{count}_0, \text{count}_1, \ldots, \text{count}_{N-1}) \) can be replaced by a normalized distribution \( (\text{count}_0, \text{count}_1, \ldots, \text{count}_{N-1}) \) using the following formula:

\[
\text{count}_i = \frac{\text{count}_i - \mu_i}{\sigma_i} \quad (5)
\]

where \( \mu_i \) is the mean value of the distribution \( (\text{count}_0, \text{count}_1, \ldots, \text{count}_{N-1}) \) and \( \sigma_i \) is the standard deviation of \( (\text{count}_0, \text{count}_1, \ldots, \text{count}_{N-1}) \). We use normalized pattern distribution as features. Each pattern’s position distribution will be connected to generate the whole feature vector.

As an example, consider the login pattern \( P_0 = \langle \text{login, passwd} \rangle \) and the trace database TDB shown in Table 1. We divide each sequence into two sections, and then count pattern \( P_0 \)’s position distribution. In this situation, \( S_0 - S_i \) will be represented as \( PD_{S_0, S_i} = (1, -1) (i = 0 \text{ to } 3) \) and \( S_i \) will be represented as \( PD_{S_i, S_i} = (-1, 1) \). In this way, the differences between \( S_0 - S_i \) and \( S_i \) are significant and the wrong sequence can be easily identified. From the example in Section 1, the frequency based method loses the discriminating power in this case, it is clear that pattern’s position distribution is more discriminating than frequency.

Algorithm 2 presents the pseudo code for Position Distribution Based Feature Representation.

### Algorithm 2: Feature Representation

**Inputs:** A selected set of patterns \( P \), Number of sections \( N \), Trace database TDB

**Outputs:** Feature Vector \( FV \)

1: for each patterns \( \text{Pat}_i \in P \)
2: \( \text{Let } \text{Inst}(\text{Pat}_i) = \text{all instance of } \text{Pat}_i \)
3: for each instance \( \text{Inst}(\text{Pat}_i) \in \text{Inst}(\text{Pat}_i) \)
4: \( \text{Let } \)
5: \( \text{start-section}_j = \left[ \text{start} - \text{pos}_j \times \frac{N}{\text{seqlen}(\text{seq} - \text{id}_j)} \right] \)
6: for \( k = \text{start-section}_j \) to \( \text{end-section}_j \)
7: \( \text{Let } FV[\text{seq} - \text{id}_j][i \times N + k]++ \)
8: normalization \( \langle FV[\text{seq} - \text{id}_j][i \times N + \text{start-section}_j] \rangle \) to \( FV[\text{seq} - \text{id}_j][i \times N + \text{end-section}_j] \)
9: return \( FV \)

It is also noteworthy that when \( N=1 \), the pattern distribution based method is exactly the same as the pattern frequency based method, this shows that pattern
position based method is more general than pattern frequency based one.

After generating the feature vectors, these features were used to train a classifier to detect software failure. When the classifier was built, suspicious program traces were processed in the same way, and then the feature vectors were put into the classifier, to test whether they contain failures or not. For the sake of comparison with a previous study, we used LIBSVM [16] as the classifier.

4. Experiment and Analysis

The experiment was carried out in two parts. Firstly, we compared our method with the state of art closed unique Iterative pattern’s frequency based method proposed in [1]. To make the experimental results more persuasive, for the datasets, all arguments of pattern mining, pattern selection and classifier are completely the same. Detailed arguments can be reviewed in [13]. Secondly, to further illustrate the strength and universality of our method, we compared our method with Frequent Pattern’s frequency base method. Frequent Patterns are mined using the FP-growth algorithm proposed in [26].

We performed 5-fold cross validation for each dataset. In the first experiment, the datasets were a mixture of synthetic datasets and real-life datasets. The datasets corresponded to traces databases (TDB). The synthetic datasets included CVS Application and X11 Windowing Protocol. Synthetic datasets were generated using the simulator QUARK [24]. Given a software component model in the form of a probabilistic finite state automaton as input, QUARK can generate traces that represent the model following some coverage criteria. QUARK is also able to inject errors into the synthetic traces. In this experiment, three types of errors were injected into the traces, they were: addition bugs, omission bugs and ordering bugs. Table 5 explains the meaning of each type of bug. The correct execution traces were labeled as 0 and failing execution traces were labeled as 1.

<table>
<thead>
<tr>
<th>Error Types</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Omission bugs</td>
<td>Missing method calls.</td>
</tr>
<tr>
<td>Addition bugs</td>
<td>Injection of additional events resulting in failures</td>
</tr>
<tr>
<td>Ordering bugs</td>
<td>The order of events occurring is wrong</td>
</tr>
</tbody>
</table>

Almost all of the real existing bugs belong to these three types, so the synthetic dataset can well simulate the real-life conditions. For the comparison experiments, argument N (number of sections) is the only adjustable argument, increasing N means divided program traces into more equal sections, and this would improve the veracity of the pattern’s position distribution but also generates more feature dimensions. As a compromise, we set N to 4, which means dividing the program traces into four equal sections. Comparative experimental results of synthetic datasets are shown in Table 6. Datasets “X11” and “CVS Omission” contain only ‘addition’ and ‘omission’ bugs respectively, “CVS Ordering” contains ordering bugs and “CVS Mix” contains a mixture of all three types of bugs. The number of correct and error traces is also shown in Table 6. We denote the closed unique Iterative pattern’s frequency based method as CUP-Pat-Fre and our closed unique Iterative pattern’s position distribution based method as CUP-Pos-Dist. “Add” refers to Addition bugs, “Omis” refers to Omission bugs, and “Order” refers to Ordering bugs. Classification accuracy, defined as the percentage of test cases correctly classified, was used as the performance metric.

From Table 6, our proposed position distribution method is better than the frequency-based method in all four synthetic datasets, which proves that additional position distribution information can help with software failure classification in different failure types.

We continued the first experiment by analyzing real-world datasets from the Siemens Test Suite [17] and a data race concurrency bug from MYSQL [19]. The Siemens Test Suite was originally used in testing coverage adequacy and error localization [25]. The test suite contains several programs. Each program contains several different versions where each version has one bug. To simulate the real-life situation where probably there are many bugs occurring in one program, 3 bugs and 3 additional simulated ordering bugs were injected into each program execution trace. We selected the three largest programs in the test suite. They are referred to as: schedule, print tokens and replace. A data race concurrency bug from MYSQL is also analyzed, this bug causes the wrong ordering of statement executions and can result in inconsistency of the database. The maintainers of MYSQL rate this bug as serious in their bug database. More information about the test suite and data race bug is available in [1, 17, and 18]. The comparative experimental results from the real-life datasets are show in Table 7.
The results show that the position distribution based method outperforms the frequency-based method in all real-life datasets, the standard deviation is also smaller than for the Pat-Fre method. The results further illustrate that the pattern’s position distribution based method is more discriminative and stable than the pattern’s frequency based method.

In the second experiment, we tested a real-life dataset - tot_info which comes from the Siemens Test Suite. Detailed information about the dataset is shown in Table 8.

We used the FP-growth algorithm to generate frequent patterns and LIBSVM as the classification model. The support threshold was set at 0.88 and 119 patterns were mined. Sixty two patterns were selected. We performed 5-fold cross validation in this dataset. Comparison results in each fold and summarized results are shown in Table 9. “FP-Fre” refers to frequent pattern’s frequency based method, and “FP-Pos-Dist” refers to frequent pattern’s position distribution based method.

From Table 9, our method outperforms the frequency based method both in accuracy and standard deviation. It further confirms the strength of our method. It also demonstrates that our pattern position distribution method can be connected to other pattern mining algorithms, which makes it flexible.

The results from both synthetic and real-life datasets, indicate that our proposed position distribution based method can better distinguish normal and failing program traces than the pattern frequency based method by catching the positional information of patterns. This information implies that by getting the semantics/constraints between statement sets, enables us to obtain a more complete description of the software being analyzed, which helps improve the performance of software failure detection. Considering the data are collected both under the synthetic and real-world conditions, we can conclude that our method will be generally applicable to the detection of software failures.

5. Conclusions

In this paper, we present a novel method to use a pattern’s position distribution as features to detect software failure occurring through misused software patterns. This method can catch the semantics/constraints information between statement sets while the traditional pattern frequency based method cannot.

This method allows us to extract more complete information from program sequences and then to generalize more discriminative models. Comparative experiments show that our method outperforms the state of art pattern frequency based method. Our method can also be easily connected to any pattern mining algorithm, which makes it very flexible.

In future work, we are going to develop a new pattern presentation method, apply this method to other domains, such as malware detection, and attempt to utilize multi-classifiers to leverage classification performance.

References


23. X. Yan, J. Han, and R. Afhar, CloSpan: Mining closed sequential patterns in large datasets. *In Proc. SDM* (2003).


26. J. Han, H. Pei, and Y. Yin, Mining Frequent Patterns without Candidate Generation, *in Proc. SIGMOD* (2000) pp 1-12.

27. E. Gamma, R. Helm, R. Johnson and J. Vlissides, Design Patterns: elements of Reusable Object-Oriented Software (Addison-Wesley, 1995)

### Table 6. experiments 1: comparison results on synthetic datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Correct((traces))</th>
<th>Error((traces))</th>
<th>Accuracy with standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add/Omis</td>
<td>Order</td>
<td>CUP-Fre</td>
</tr>
<tr>
<td>X11</td>
<td>125</td>
<td>125</td>
<td>0</td>
</tr>
<tr>
<td>CVS Omission</td>
<td>170</td>
<td>170</td>
<td>0</td>
</tr>
<tr>
<td>CVS Ordering</td>
<td>180</td>
<td>0</td>
<td>180</td>
</tr>
<tr>
<td>CVS Mix</td>
<td>180</td>
<td>90</td>
<td>90</td>
</tr>
</tbody>
</table>

### Table 7 experiments 1: results on real-life datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Correct((traces))</th>
<th>Error((traces))</th>
<th>Accuracy with standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Add/Omis</td>
<td>Order</td>
<td>CUP-Fre</td>
</tr>
<tr>
<td>schedule</td>
<td>2140</td>
<td>289</td>
<td>1851</td>
</tr>
<tr>
<td>print_tokens</td>
<td>3108</td>
<td>187</td>
<td>187</td>
</tr>
<tr>
<td>replace</td>
<td>1259</td>
<td>269</td>
<td>269</td>
</tr>
<tr>
<td>MySQL</td>
<td>51</td>
<td>0</td>
<td>51</td>
</tr>
</tbody>
</table>

### Table 8 experiments 2: detailed information about tot_info dataset

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Correct((traces))</th>
<th>Error((traces))</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>tot_info</td>
<td>302</td>
<td>208</td>
<td>94</td>
<td></td>
</tr>
</tbody>
</table>

### Table 9 experiments 2: comparison results on tot_info dataset

<table>
<thead>
<tr>
<th>Accuracy with standard deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-fold cross validation</td>
</tr>
<tr>
<td>fold-1</td>
</tr>
<tr>
<td>fold-2</td>
</tr>
<tr>
<td>fold-3</td>
</tr>
<tr>
<td>fold-4</td>
</tr>
<tr>
<td>fold-5</td>
</tr>
<tr>
<td>summarized result</td>
</tr>
</tbody>
</table>