Using Pattern Position Distribution for Software Failure Detection[*]

Chunping Li, Ziniu Chen, Hao Du
Tsinghua National Laboratory for Information Science and Technology
School of Software, Tsinghua University
Beijing 100084, China
E-mail: {cli, czn09@tsinghua.edu.cn, duhao228@gmail.com}

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}

Received 9 April 2012

Accepted 12 September 2012

Abstract

We present a novel approach for using the pattern position distribution as features to detect software failure. In this approach, we divide an execution sequence into several sections and compute the pattern distribution in each section. The distribution of all patterns is then used as features 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 life. 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 industries 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',

[*]The work of this joint research project was facilitated through grant funding from the Natural Science Foundation in China and the Royal Society in UK.

'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, TxManger.rollback*>, etc. Software Patterns have also been used as part of reuse strategies when developing software systems. The seminal work in Ref. 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, bioinformatics, weather prediction, and system health management[6]. Various pattern mining methods are proposed such as frequent item set mining[10], sequential pattern mining[11], closed pattern mining[22, 23], episode mining[12], iterative pattern mining[2] and closed unique pattern mining[1],etc. Recently there has been interest in developing discriminative pattern-based classifiers. In Ref. 7, Cheng et al. mine frequent item sets for classifying transaction data. In Refs. 8 and 9, frequent connected sub-graphs are mined for classifying graph data. On a related front, Lo et al. propose a novel method to extract closed unique patterns for software failure detection[1].

Pattern-based software failure detection is 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 the 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. More specifically, pattern-based software failure detection is a kind of pattern frequency-based method.

Existing research on pattern frequency based methods has produced promising results. Refs. 1 and 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 P_1=<*login*, *passwd*> and the set of user command sequences S_0-S_4 as shown in Table 1. Sequences S_0-S_3 represent normal daily profiles of a user while the sequence S_4 is anomalous - one can never do any other operations before logging into the system. Although S_4 indicates an obvious failure, we are unable to distinguish S_0-S_3 from S_4 when using the pattern frequency based method because the pattern P_0 = <*login*, *passwd*> does occur once in each of S_0-S_4. It is very clear that pattern frequency based methods loose their discriminating power in this case.

Table 1. Sequences of user commands

S_0	*login, passwd, mail, ssh, …, mail, web, logout*
S_1	*login, passwd, mail, web, …, web, web, web, logout*
S_2	*login, passwd, mail, ssh, …, mail, web, web, logout*
S_3	*login, passwd, web, mail, ssh, …, web, mail, logout*
S_4	***mail, ssh, web, …, web, mail, login, passwd, logout***

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 P_0 occurred in the tail of S_4, but occurred in the head of S_0-S_3. 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 constraints. In the example, before we do any other operations, we must login into 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 position 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 position 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;
(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.

In order to obtain a pattern's position information, we need to define what we mean by a 'pattern instance'. This definition is given in DEFINITION 1 to follow. The pattern instance definition can be expressed as a Quantified Regular Expression (QRE). QRE is similar to the standard regular expression but with a semicolon denoting the concatenation operator, '[-]' denoting the exclusion operator (e.g., [-*P*, *S*] means any event except *P* and *S*), and '*' denoting 0 or more.

Table 2. Traces database

Identifier	Sequence
S_0	*<D, B, C, F, B, A, F, B, C, E>*
S_1	*<D, B, C, D, B, A, E, B, B, E, D, C, E, C, D, E, F, D, B, A>*

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*

$$e_0;[-e_0,\ldots,e_{n-1}]^*;e_1;\ldots;[-e_0,\ldots,e_{n-1}]^*;e_{n-1}.$$

An instance is denoted by a triplet (*seq-id*, *start-pos*, *end-pos*), where *seq-id* refers to the ID of a sequence *S* in the database while *start-pos* and *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 position information on their own because the length of sequences in TDB may not be equal. For example, consider a pattern P = <*A*, *B*> and two sequences S_0 and S_1 shown in Table 2. There are two instances *I*(0, 5, 7) and *J*(1, 5, 7) of pattern *P*. The length of S_0 is 10 and the length of S_1 is 20. Although *I* and *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 position information. To avoid the weakness of the absolute position, we use the relative position to represent the position information. In order to use relative position, we divide all sequences into *N* 'sections' separately, and then determine what a section or sections an instance belongs to. In this way, we can position an instance.

Definition 2. Section *Divide a sequence $S_{seq\text{-}id}$ <e_0, e_1, e_2,…,e_{n-1}> into N parts s.t.* $\bigcup_{i=0}^{N-1} part_i$ = <e_0, e_1, e_2,…,e_{n-1}> *and* $\bigcap_{i-0}^{N-1} part_i = \varnothing$ *, this partition divide* S_{seq-id} *into N sections iff* $\forall i, j.\ 0 \le i, j \le N-1,$ *s.t.* $|part_i| - |part_j| \le \pm 1$ *,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 (*$section_0$*, *$section_1$*,…, *$section_{N-1}$*), and then we can determine the 'instance position' which is given in the following definition.

Definition 3. Instance Position *Given an instance I(seq-id, start-pos, end-pos), a sequence divides into N sections $S_{seq\text{-}id}$($section_0$, $section_1$,…, $section_{N-1}$) 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.* $start-pos_{section_{ID}} \le start-pos_I \le end-pos_{section_{ID}}$ *and end-section refers to the ID of the section s.t.* $start-pos_{section_{ID}} \le end-pos_I \le end-pos_{section_{ID}}$ *, where* $start-pos_I$ *and* $end-pos_I$ *refer to the starting point and ending point of I,* $start-pos_{section_{ID}}$ *and* $end-pos_{section_{ID}}$ *refer to the starting point and ending point of $section_{ID}$.*

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 is denoted by $PD_{P,S}$=($count_1$, $count_2$, …, $count_{N-1}$) where $PD_{P,S}$ means pattern P's position distribution in sequence S, N refers to the number of sections, and $count_i$ refers to the number of P's instances in the $section_i$. Instance I appeared in the $section_k$ means:* $\forall j.\ start-pos_I \le j \le end-pos_I \text{ s.t. start-pos}_{section_k} \le \text{j} \le end-pos_{section_k}$. *A part of Instance I in the $section_k$ means* $\exists j.\ start-pos_I \le j \le end-pos_I \text{ s.t. start-pos}_{section_k} \le \text{j} \le end-pos_{section_k}$

.

As an example, consider a pattern *P* = <*A*, *B*> 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 four sections separately. For *S_0*, *$section_0$* =<*D*, *B*, *A*>, *$section_1$* =<*F*, *B*>, *$section_2$* =<*A*, *F*, *B*> and *$section_3$*=<*C*, *E*>. For *S_1*, *$section_0$*=<*D*, *B*, *A*>, *$section_1$*=<*D*, *B*>, *$section_2$*=<*B*, *B*> and *$section_3$*=<*A*, *B*>. 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 3. Traces database

Identifier	Sequence
S_0	*<D, B, A, F, B, A, F, B, C, E>*
S_1	*<D, B,A, D, B, B, B, A, B>*

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 traces database (TDB). Secondly, pattern selection is performed to select discriminative patterns. Thirdly, we compute the position distribution for each selected pattern. The distribution will be used as 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[1] proposed by David Lo et al. 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[26] proposed by J. Han et al. 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], which is defined as follows.

$$Fr = \frac{\sum_{i=1}^{k} n_i (u_i - u)^2}{\sum_{i=1}^{k} n_i \sigma_i^2} \qquad (1)$$

where n_i is the number of data samples in class c_i, and u_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. u is the average pattern value in the whole dataset. σ_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 u, u_i and σ_i are defined respectively as follows.

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

According to the formula (1), 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 Ref. 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 δ times have been processed.

Algorithm 1: pattern selection
Inputs: pattern set P, trace database TDB, coverage threshold δ.
Output: a selected pattern set P_s

```
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 $P_s$
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 $P = \varnothing$;
11:       break;
12: return $P_s$
```

3.3. *Position distribution based features*

The conventional feature representation approach simply uses pattern's occurrence frequency as the 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, and 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 the feature would lose a lot of information and thereby reduce the discriminative power.

As discussed in Section 2, we use relative position to build position 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<D, B, A, F, B, A, F, B, C, E>*, there are six ways to divide *S* into four sections. All of six solutions are shown 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 $S_{seq\text{-}id}$, we allocate e into $section_i$ where

$$j = \left| i \times \frac{N}{seqlen(seq-id)} \right| \qquad (2)$$

N denotes the number of sections, *Seqlen*(*seq-id$_i$*) denotes that the total number of events of the sequences whose ID is *seq-id$_i$*. Using the above strategy, for the *j*th instance of pattern *P$_i$*, we denote it by *Inst*(*P$_i$*)$_j$ = (*seq*-*d$_j$*, *start*-*pos$_j$*, *end*-*pos$_j$*), the corresponding start-section is

$$start-\text{section}_j = \left| start-pos_j \times \frac{N}{seqlen(seq-id)} \right| \qquad (3)$$

Similarity, the corresponding end-section is

$$end-\text{section}_i = \left| end-pos_i \times \frac{N}{seqlen(seq-id)} \right| \qquad (4)$$

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

Table 4. All solutions to divide *S* into four sections

Solutions	Section partition
Solution 1	<D, B, A, |F, B,| A, F, B,| C, E>
Solution 2	<D, B, A,| F, B, A,| F, B,| C, E>
Solution 3	<D, B, A,| F, B,| A, F,| B, C, E>
Solution 4	<D, B,| A, F, B,| A, F,| B, C, E>
Solution 5	<D, B, |A, F, B,| A, F, B,| C, E>
Solution 6	<D, B,| A, F,| B, A, F,| B, C, E>

In this way, we can determine the distribution of each pattern in the sequence, but we cannot use it directly as a feature vector. For instance, consider pattern *P* and its distribution in sequence $S_0 : PD_{P,S_0} = (5,\ 10,\ 5,\ 10)$ and its distribution in sequence $S_1 : PD_{P,S_1} = (55,\ 60,\ 55,\ 60)$. 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 (*count$_0$*, *count$_1$*,…,*count$_{N-1}$*) can be replaced by a normalized distribution (*count'$_0$*, *count'$_1$*,…, *count'$_{N-1}$*) using the following formula.

$$count_i^{'} = \frac{count_i - \mu_i}{\sigma_i} \qquad (5)$$

where μ_i is the mean value of the distribution (*count$_0$*, *count$_1$*, …, *count$_{N-1}$*) and σ_i is the standard deviation of (*count$_0$*, *count$_1$*, …, *count$_{N-1}$*). We use normalized pattern distribution as features. Each pattern's position distribution will be connected to generate the whole feature vector.

Algorithm 2: feature representation
Inputs: *A selected set of patterns P$_s$, number of sections N, trace database TDB*
Outputs: *Feature vector FV*

```
1: for each pattern $Pat_i \in P_s$
2:   Let $Inst(Pat_i)$ = all instances of $Pat_i$;
3:   for each instance $Inst(Pat_i)_j \in Inst(Pat_i)$
4:     Let $start-\text{sec}tion_j = \left\lfloor start-pos_j \times \frac{N}{seqlen(seq-id_j)} \right\rfloor$;
5:     Let $end-\text{sec}tion_j = \left\lfloor end-pos_j \times \frac{N}{seqlen(seq-id_j)} \right\rfloor$;
6:     for k = $start-\text{sec}tion_j$ to $end-\text{sec}tion_j$
7:       Let $FV[seq-id_j][i \times N + k]{+}{+}$;
8:   normalization
     ($FV[seq-id_j][i \times N + start-\text{sec}tion_j]$ to
     $FV[seq-id_j][i \times N + end-\text{sec}tion_j]$);
9: return $FV$;
```

As an example, consider the login pattern *P$_0$=<login, passwd>* and the traces database shown in Table 1. We divide each sequence into two sections, and then count pattern *P$_0$* position distribution. In this situation, *S$_0$*-*S$_3$* will be represented as $PD_{P_0,S_i} = (1,\ -1)\ (i = 0\ to\ 3)$ and *S$_4$* will be represented as $PD_{P_0,S_3} = (-1,\ 1)$. In this way, the differences between *S$_0$*-*S$_3$* and *S$_4$* 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.

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

4. Experiment and Analysis

The experiment is carried out in two parts. Firstly, we compare our method with the state of art closed unique Iterative pattern's frequency based method[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 Ref. 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[26]. We perform 5-fold cross validation for each dataset.

In the first experiment, the datasets are a mixture of synthetic datasets and real-life datasets. The datasets correspond to traces databases (TDB). The synthetic datasets include CVS Application and X11 Windowing Protocol. Synthetic datasets are 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 are injected into the traces, i.e., addition bugs, omission bugs and ordering bugs. Table 5 explains the meaning of each type of bug. The correct execution traces are labeled as 0 and failing execution traces are labeled as 1.

Table 5. Three Types of Errors

Error Types	**Explanation**
Omission bugs	Missing method calls.
Addition bugs	Injection of additional events resulting in failures
Ordering bugs	The order of events occurring is wrong

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, is 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 for software failure classification in different failure types.

We continue 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 is 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, three bugs and three additional simulated ordering bugs are injected into each program execution trace. We select 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 Refs. 1,17 and 18. The comparative experimental results from the real-life datasets are shown in Table 7.

The results show that the position distribution based method outperforms the frequency-based method in all real-life data sets, and the standard deviation is also smaller than for the Pat-Fre method. The results further illustrate that the pattern position distribution based method is more discriminative and stable than the pattern frequency based method.

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

We use the FP-growth algorithm to generate frequent patterns and LIBSVM as the classification model. The support threshold is set at 0.88 and 119 patterns were mined. Sixty two patterns were selected. We perform 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 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

Table 6. Experiments 1: comparison results on synthetic datasets

Dataset	Correct(|traces|)	Error(|traces|)		Accuracy with standard deviation	
		Add/Omis	Order	*CUP-Fre*	***CUP-Pos-Dist***
X11	125	125	0	97.20 ± 3.35	• **100 ± 0**
CVS Omission	170	170	0	100 ± 0	**100 ± 0**
CVS Ordering	180	0	180	85.28 ± 2.71	**86.95 ± 2.22**
CVS Mix	180	90	90	93.89 ± 5.94	**96.39 ± 4.72**

Table 7. Experiments 1: results on real-life datasets

Dataset	Correct(|traces|)	Error(|traces|)		Accuracy with standard deviation	
		Add/Omis	Order	*CUP-Fre*	***CUP-Pos-Dist***
schedule	2140	289	1851	86.26 ± 14.90	**88.67 ± 10.79**
print_tokens	3108	187	187	99.94 ± 0.06	**100 ± 0**
replace	1259	269	269	90.84 ± 2.54	**93.24 ± 2.21**
MySQL	51	0	51	100 ± 0	**100 ± 0**

Table 8. Experiments 2: detailed information about *tot_info* dataset

Dataset	Correct(|traces|)	Error(|traces|)	
		Add/Omis	Order
tot_info	302	208	94

Table 9. Experiments 2: comparison results on *tot_info* dataset

5-flod cross validation	Accuracy with standard deviation	
	FP-Fre	***FP-Pos-Dist***
fold-1	70.83%	**93.33%**
flod-2	68.3%	**72.5%**
fold-3	**95.83%**	91.67%
fold-4	80.83%	**87.5%**
fold-5	63.33%	**74.17%**
summarized result	75.83 ± 12.87	**83.83 ± 9.84**

by catching the position information of patterns. This information implies that 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 under both 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 the pattern 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 algorithms, which makes it very flexible.

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

References

1. D. Lo, H. Cheng, J. Han, S-C. Khoo, and C. Sun, Classification of software behaviors for failure detection: a discriminative pattern mining approach, in *Proc. KDD* (2009) pp. 557-566.
2. D. Lo, S-C. Khoo, and C. Liu, Efficient mining of iterative patterns for software specification discovery, in *Proc. KDD* (2007) pp. 460-469.
3. G. Tassey, The economic impacts of inadequate infrastructure for software testing, *Planning Report* (National Institute of Standards and Technology, USA, 2002).
4. Z. Xing, A brief survey on sequence classification, *J. ACM SIGKDD Explorations Newsletter* 12(1) (2010) 40-48.
5. Java Trans. API Spec. http://java.sun.com/products/jta.
6. V. Chandola, A. Banerjee, and V. Kumar, Anomaly detection for discrete sequences: a survey, *IEEE Transactions on Knowledge and Data Engineering* 99(2010) 1-19.
7. H. Cheng, X. Yan, J. Han, and C.Hsu, Discriminative frequent pattern analysis for effective classification, in *Proc. ICDE* (2007) pp.716-725.
8. M. Deshpande, M. Kuramochi, N.Wale, and G. Karypis, Frequent substructure-based approaches for classifying chemical compounds, *IEEE Transactions on Knowledge and Data Engineering* 17(8) (2005) 1036-1050.
9. X. Yan, H. Cheng, J. Han, and P-S. Yu, Mining significant graph patterns by scalable leap search, in *Proc. SIGMOD* (2008) pp. 433-444.
10. R. Agrawal and R. Srikant, Fast algorithms for mining association rules, in *Proc. VLDB* (1994) pp. 487-499.
11. R. Agrawal and R. Srikant, Mining sequential patterns, in *Proc. ICDE* (1995) pp. 3-14.
12. H. Mannila, H. Toivonen, and A.I. Verkamo, Discovery of frequent episodes in event sequences, *J. Data Mining and Knowledge Discovery* (1) (1997) 259-289.
13. Software failure detection: experimental dataset, http://www.mysmu.edu/faculty/davidlo/kdd09.htm, 2009
14. R. Duda, P. Hart, and D. Stork, *Pattern Classification* 2nd Edition (Wiley Interscience, 2000)
15. J. Han, M. Kamber, *Data Mining: Concepts and Techniques,* 2nd Edition (Elsevier, 2006).
16. C. Chang and C. Lin, LIBSVM: a library for support vector machines, 2001(Software available at http://www.csie.ntu.edu.tw/~cjlin/libsvm).
17. M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. in *Proc. of Int. Conf. on Software Engineering* (1994) pp. 191 -200.
18. C. Liu, X. Yan, H.Yu, J. Han, and P.S. Yu, Mining behavior graphs for “backtrace” of noncrashing bugs, in *Proc. SDM* (2005).
19. MYSQL atomicity violation, http://bugs.mysql.com
20. W. Dickinson, D. Leon, and A. Podguriski, Finding failures by cluster analysis of execution profiles, in *Proc. of Int. Conf. on Software Engineering* (2001) pp. 339-348.
21. J.F. Bowring, J.M. Rehg, and M.J. Harrold, Active learning for automatic classification of software behavior in *Proc. of Int. Symp. on Software Testing and Analysis* (2004) pp. 195-205.
22. J. Wang and J. Han, BIDE: mining of frequent closed sequences, in *Proc. ICDE* (2004) pp. 79-90.
23. X. Yan, J. Han, and R. Afhar, CloSpan: mining closed sequential patterns in large datasets, in *Proc. SDM* (2003) pp. 166-177.
24. D. Lo and S. Khoo, QUARK: Empirical assessment of automaton-based specification miners, in *Proc. of Working Conf. on Reverse Engineering* (2006) pp. 51-60.
25. C. Liu, X. Yan, L. Fei, J. Han, and S. Midkiff, SOBER: statistical model-based bug localization, in *Proc. SIGSOFT ESEC-FSE* (2005) pp. 286-295.

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)