Classification of Software Behaviors for Failure Detection: A Discriminative Pattern Mining Approach

David Lo
Singapore Management University
davidlo@smu.edu.sg

Hong Cheng[*]
Chinese University of Hong Kong
hcheng@se.cuhk.edu.hk

Jiawei Han
University of Illinois at Urbana-Champaign
hanj@cs.uiuc.edu

Siau-Cheng Khoo and Chengnian Sun
National University of Singapore
{khoosc,suncn}@comp.nus.edu.sg

ABSTRACT

Software is a ubiquitous component of our daily life. We often depend on the correct working of software systems. Due to the difficulty and complexity of software systems, bugs and anomalies are prevalent. Bugs have caused billions of dollars loss, in addition to privacy and security threats. In this work, we address software reliability issues by proposing a novel method to classify software behaviors based on past history or runs. With the technique, it is possible to generalize past known errors and mistakes to capture failures and anomalies. Our technique first mines a set of discriminative features capturing repetitive series of events from program execution traces. It then performs feature selection to select the best features for classification. These features are then used to train a classifier to detect failures. Experiments and case studies on traces of several benchmark software systems and a real-life concurrency bug from MySQL server show the utility of the technique in capturing failures and anomalies. On average, our pattern-based classification technique outperforms the baseline approach by 24.68% in accuracy[1].

Categories and Subject Descriptors

H.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation

1. INTRODUCTION AND MOTIVATION

In the information age, software is omnipresent in our daily life. Many of our daily activities are dependent on the correct working of software systems. Ensuring the reliability of systems throughout their lifetime is certainly a worthwhile goal.

If analyzed, software produces a massive amount of data corresponding to its various behaviors. Software behavior is the way a program executes. It corresponds to a path a program takes when executing from the start of the program till the end when it terminates. Some behaviors are desirable while a minority of others are not. Undesirable behaviors can correspond to bugs, malwares, intrusions, *etc.*. These undesirable behaviors are often referred to as failures.

Note that failures are often not easily identifiable. Some failures cause crashes (*e.g.*, blue screen in windows, unhandled exception thrown, *etc.*) which are noticeable. However, many others do not cause any "visible" effect. The software still runs to completion, while the computation result is wrong, data stored in the database is erroneous, or some system security constraints are violated. These "non-crashing" failures are often hard to identify and one might only have a few known samples of these. These failures are worthy of special concerns since they are not easily identifiable and might potentially pose serious security risks. US National Institute of Standards and Technology (NIST) reported that software bugs cost US economy 59.5 billions annually [25].

Can data mining help? In this paper, as a new step to address the reliability of software systems, we present a novel classification approach to predict software behaviors. Based on historical data of software and known failures, we construct a classifier to generalize the failures and to further detect other *unknown* failures. Generalizing past failures are useful in many situations as: programmers might miss related failures in the program, programmers could make similar mistakes in the future, programmers could make similar errors in usages of common libraries, programs could be re-infected by the same malware, *etc.*. This step of *failure detection* is the first step of the software quality assurance process. After a failure is detected, *fault localization* approaches, *e.g.*, [18], which assume that a set of relevant failures are known, can then be employed to localize the bug in

[*]This work was supported by the Chinese University of Hong Kong Direct Grant No. 2050446.

[1]For repeatability test, datasets and binary codes are available at [2].

KDD'09, June 28–July 1, 2009, Paris, France.

the code. Failure detection techniques can be applied periodically in the lifecycle of a software system to improve the reliability of the system. Furthermore, the classifier can aid other software engineering tasks, for example, by serving as a test oracle for test-suite augmentation (*i.e.*, adding new test cases to a test suite) [7].

A software behavior can be viewed as a series of events, where an event can correspond to the invocation of a method, the execution of a program statement, *etc.*. When recorded, this path or behavior is referred to as an execution trace. A set of these execution traces can be represented as a sequence database which is the basis of our analysis.

Recently there are active interests in developing discriminative pattern-based classifiers [10, 9, 29], especially for unstructured or semi-structured data including itemsets and graphs. A set of patterns are mined and used to represent multi-dimensional discriminative features from a set of data. These features are then further selected and used to build effective classifiers. On a related front, iterative pattern mining, based on the semantics of several software modeling languages, has been proposed to extract frequent repetitive series of events from program execution traces as candidate software specifications [21]. Iterative pattern mining considers ordering among events in the traces. It further takes into account repeated occurrences of patterns both *within a sequence* and *across multiple sequences* to address loops and repetitions in an execution trace.

Our proposed classification framework works in a three-step process. As the first step, we propose a new scalable algorithm to mine *closed unique iterative patterns* from program traces of known normal and failing executions (*i.e.*, failures). By capturing both the important temporal order and repetitions in a trace, iterative patterns are good features to characterize software behaviors and distinguish failing software traces from normal ones. Mining closed unique iterative patterns is much more scalable than mining the set of closed patterns. Feature selection is then applied to select highly discriminative patterns as classification features, which reveal important temporal information in software traces. A classifier is constructed based on the training traces with such pattern-based feature representation. The classifier can be used to classify unknown program behaviors corresponding to the current or a future version of a software system as similar errors might potentially be made in the future. The trained classifier can be updated several times during the life cycle of a software system with new known good and bad behaviors. It can be treated as a black box to predict and give warning signals in case the program is behaving in an undesirable way.

Our approach is not dependent on the availability of source code and precise documented specification. It follows a recent trend of dynamic analysis and specification mining [5, 21]. Dynamic analysis approaches analyze software execution traces which can be collected by various instrumentation methods, *c.f.*, [5, 20]. It is complementary to static analysis, *i.e.*, analysis of program code, and has the benefit of avoiding some problems faced by static analysis, *e.g.*, pointer aliasing and infeasible paths. Furthermore, dynamic analysis is crucial since (1) many Commercials-Off-The-Shelf (COTS) components and legacy systems come without any source code; and (2) many software systems are developed with poorly documented specifications [5].

To validate the utility of our proposed classifier, we performed controlled experiments and case studies on synthetic and real datasets. For comparison, we implemented a baseline classification model based on single events as features. A simulator proposed in [19] is used to generate synthetic traces following some real-world models. Real traces are also generated from programs belonging to the Siemens benchmark [15]. Furthermore, we experimented with a real-life data race concurrency bug from MySQL server [1].

Experiments demonstrate the effectiveness of our proposed discriminative iterative pattern-based classification for software failure detection. Over all tested datasets, our iterative pattern-based method is 24.68% more accurate on average than the baseline method. In the MySQL data race bug, a series of program statements are executed in a wrong order due to bad interleavings of threads. Iterative pattern-based features are able to characterize the MySQL data race, order-related bug with 100% classification accuracy and AUC = 1.0 while simple event-based features fail to classify this bug accurately (accuracy = 50%, AUC = 0.5).

Our proposed discriminative iterative pattern-based classification is not only confined to the classification of software behaviors, it is also potentially applicable to a range of other problems including user profile and behavior prediction based on web log analysis, and protein sequence classification based on motifs (patterns). In summary, our main contributions include the following:

- We propose a new framework for classifying sequential data based on discriminative iterative patterns. A feature selection algorithm based on Fisher score is proposed to identify highly discriminative patterns which distinguish the failing traces from normal ones. This enriches past studies on pattern-based classification working on itemsets and graphs [10, 9, 29].
- We design a new scalable algorithm for mining *closed unique iterative patterns* which repeat *within a trace* and *across multiple traces.* Closed unique pattern mining addresses the difficulty in mining sequential patterns from trace datasets with varied interleavings of noise or unrelated events (*e.g.*, *appendToXML()*, *hashCode()*, *toString()*) appearing with high frequency.
- We demonstrate through a comprehensive set of experiments and case studies the utility of our proposed framework to provide an effective solution for the classification of program traces.

The structure of this paper is as follows. In Section 2 we introduce preliminary concepts. Section 3 describes our proposed closed unique iterative pattern mining algorithm. In Section 4, we describe the iterative pattern-based classification framework. Experimental evaluation and case studies are presented in Section 5. Section 6 discusses related work, followed by conclusions and future work in Section 7.

2. PRELIMINARIES

In this section, we describe preliminary details on software execution traces and iterative patterns. We also present some notations and definitions.

2.1 Program Traces and Terminologies

A software behavior can be viewed as a series of events. An event in turn corresponds to a unit behavior of interest. This can correspond to the execution of a statement, a method call or a basic block in a Control Flow Graph

(CFG). When logged, a series of events corresponding to a software behavior forms an *execution trace.*

We denote a trace or sequence S as $\langle e_1, e_2, \ldots, e_{end}\rangle$ where each e_i is an event from an event set I. The set of input traces or sequence database under consideration is denoted by TDB.

A pattern P_1 ($\langle e_1, e_2, \ldots, e_n\rangle$) is considered a *subsequence* of another pattern P_2 ($\langle f_1, f_2, \ldots, f_m\rangle$) if there exist integers $1 \leq i_1 < \ldots < i_n \leq m$ where $e_1 = f_{i_1}$, $\cdots$, $e_n = f_{i_n}$. This relation is denoted as $P_1 \sqsubseteq P_2$. P_2 is a *super-sequence* of P_1. We denote the concatenation of two patterns with the $+\!\!+$ operator.

2.2 Iterative Patterns

Patterns which are found in software usually correspond to programming rules or usage patterns. Some software patterns are well documented: a well-known example is the set of commonly used software design patterns. The following are some examples of patterns in software documentations:

1. Resource Locking Protocol : $\langle lock, unlock\rangle$
2. Java Transaction Architecture Protocol (*c.f.*, [24]): $\langle TxManager.begin, TransactionImpl.enlistResource$, *TransactionImpl.delistResource*, *TxManager.commit*, *TransactionImpl.commitResources*$\rangle$

To mine frequent patterns in software traces, one needs to consider *repetitions both within a sequence and across multiple sequences*. Based on the above motivation of patterns in software, Lo *et al.* defined iterative patterns [21] by counting the frequency of a pattern within a sequence and across multiple sequences. Iterative pattern is defined based on the semantics of commonly used software modeling languages: Message Sequence Chart (MSC) (a standard of International Telecommunication Union (ITU)) [16] and its extension, Live Sequence Chart (LSC) [14].

Different from frequent itemset [3], iterative pattern takes into consideration temporal orders in a trace. The behavior "*allocate* followed by *consume*, followed by *release*" has a very different meaning from "*release* followed by *consume*, followed by *allocate*". In the latter, a resource is consumed before it is allocated. Different from traditional sequential pattern mining [4], iterative pattern takes into account repetitive behaviors within a trace. This is important due to loops and repetitive behaviors observed by a program. Also different from episode mining [23], iterative pattern mining considers a database of sequences rather than a single sequence and removes the restriction that related events must occur close together in an episode. Important patterns like $\langle lock, unlock\rangle$ can have their events separated by an arbitrary number of other events in the traces. This permits the pattern to be robust enough to characterize error cases that appear in different variants due to occurrences of many unrelated events.

The pattern instance definitioncould be expressed unambiguously in Quantified Regular Expression (QRE). Quantified regular expression is very similar to standard regular expression with ';' as the concatenation operator, '[-]' as the exclusion operator (*e.g.*, [-P,S] means any event except P and S), and '*' as the standard Kleene star.

Definition 2.1 (**Pattern Instance**). *Given a pattern P $\langle e_1, e_2, \ldots, e_n\rangle$, a consecutive series of events SB ($sb_1, sb_2, \ldots, sb_m$) in a sequence S in TDB is an instance of P iff it is of the following QRE expression*

$$e_1; [-e_1, \ldots, e_n]*; e_2; \ldots; [-e_1, \ldots, e_n]*; e_n.$$

An instance is denoted compactly by a triple (s_{idx}, i_{start}, i_{end}) where s_{idx} refers to the ID of a sequence S in the database while i_{start} and i_{end} refer to the starting point and ending point of a substring in S. By default, all indices start from 1.

As an example, consider a pattern P ($\langle A, B\rangle$) and a database consisting of two sequences:

Identifier	Sequence
$S1$	$\langle D, B, A, F, B, A, F, B, C, E\rangle$
$S2$	$\langle D, B, A, D, B, B, B, A, B\rangle$

The set of instances of P denoted as $Inst(P)$ is the set of triples {(1, 3, 5), (1, 6, 8), (2, 3, 5), (2, 8, 9)}. Note that multiple occurrences of an iterative pattern in the same sequence are taken into account to reflect loops and repetitions in an execution trace. We treat the frequency of pattern instances within a sequence and that across multiple sequences with an equal weight.

Definition 2.2 (**Frequent Iterative Pattern**). *For a trace (sequence) dataset TDB, an iterative pattern P is frequent if its instances occur above a certain threshold of min_sup in TDB, i.e., $|Inst(P, TDB)| \geq min\_sup$. The size of $Inst(P, TDB)$ is referred to as the support of the pattern and is denoted as $sup(P)$.*

Furthermore, an iterative pattern is closed if the following requirements stated in Definition 2.3 are satisfied.

Definition 2.3 (**Closed Iterative Pattern**). *A frequent iterative pattern P is* closed *if there exists no super-sequence Q s.t.:*

1. *P and Q have the same support;*
2. *Every instance of P corresponds to a unique instance of Q, denoted as Inst(P) $\approx$ Inst(Q).*

An instance of P ($seq_P, start_P, end_P$) corresponds to an instance of Q ($seq_Q, start_Q, end_Q$) iff $seq_P = seq_Q$ and $start_P \geq start_Q$ and $end_P \leq end_Q$.

3. ITERATIVE PATTERN MINING

Iterative patterns capture higher-order features from execution traces. Traces correspond to the various program behaviors a software system exhibits. An event in a trace, corresponding to an execution of a method, a statement, or a building block, can be treated as a feature of the behaviors. These features are not occurring in isolation. Rather, related features happening before or after a particular feature dictate whether the feature corresponds to a correct or failing behavior. Difficult-to-find bugs are often caused by interactions between multiple features. Single events occurring separately might be permissible, however, when they occur together in a particular order or context, they might cause a problem to the system.

Iterative pattern mining not only captures higher-order features but also their frequencies. To distinguish software behaviors, the frequency of an iterative pattern within a single trace is important. A program may work well for most cases, but fails on boundary cases. Errors might not occur when a program behavior repeats one or two times, but might crop out when a program behavior is repeated many times. An example is recursion, where after a certain number of recursions, system stack space might be exhausted. At times, a state-based system might run well when a behavior occurs once but becomes erroneous on a future repetition due to a wrong assignment made to the program state.

To reduce the number of iterative patterns, we tried to mine closed iterative patterns introduced in Definition 2.3

since they capture the frequency of all frequent iterative patterns without any loss of information. However, even with a closed definition, we may still generate a large number of iterative patterns, due to the combination between single events, especially if "noise" or unrelated events appear often in the trace. The "noise" can correspond to utility related events, for example, *appendToXML()*, *hashCode()*, *toString()* method calls, *etc.* that appear very frequently but carry little meaning. Consider for example the following database:

Identifier	Sequence
$S1$	$\langle A, C, A, A, A, C, A, A, A, C \rangle$
$S2$	$\langle A, A, A, A, C, A, A, A, A, C \rangle$

Given $min\_sup = 2$, patterns $\langle A, C \rangle$, $\langle A, A, C \rangle$, $\langle A, A, A, C \rangle$ and $\langle A, A, A, A, C \rangle$ will be reported. These four patterns have different support values and hence each one is not subsumed by the other. If we mine closed patterns, all four patterns will be reported. If the traces are reasonably long and the "noise" event A appears very often, the pattern set is likely to explode due to random pairings with the "noise". To avoid this problem and further reduce the number of patterns, we propose to mine a compact set of closed patterns that are composed of unique events. We define closed unique patterns as follows.

DEFINITION 3.1 (**Closed Unique Pattern**). *A frequent pattern P is a* closed unique *pattern if P contains no repeated constituent events, and there exists no super-sequence Q s.t.:*

1. *P and Q have the same support;*
2. *Every instance of P corresponds to a unique instance of Q;*
3. *Q contains no constituent events that repeat.*

As an example, consider a database with two sequences:

Identifier	Sequence
$S1$	$\langle A, B, B, B, B, C, E, D, A, B, B \rangle$
$S2$	$\langle C, E, D, A, B, B, B, B, B \rangle$

Assume $min\_sup = 2$. The pattern $\langle A, B \rangle$ is a closed unique pattern. It contains unique elements A and B, and there is no longer unique patterns having the same support as $\langle A, B \rangle$. Consider another pattern $\langle C, D \rangle$ which is unique. This pattern is not closed, as there exists a longer pattern $\langle C, E, D \rangle$ which is also unique and the two patterns have corresponding instances.

Although we do not prove it formally, instances of closed unique patterns are guaranteed to be non-overlapping. In our experiments, the set of closed unique patterns is *much less* than the set of closed patterns, and it is much more efficient and scalable to mine closed unique patterns.

Algorithm 1 presents the pseudocode for mining closed unique patterns. The algorithm performs a depth-first traversal of the search space to grow iterative patterns. It first computes frequent single events (Line 1). The frequent events are then grown in a depth-first fashion by performing recursive calls to the procedure *GrowRec*. At each recursive step, a check is performed to test whether the current pattern to-be-grown is closed and unique. If this is the case, we will output the current pattern (Line 6). If a pattern is not unique, all its extensions will not be unique either. Hence, we only grow the current pattern if it is composed of unique events (Line 7). *NPt* is a new pattern grown from *Pat* concatenated with a unique event f (Line 9). We use the InfixScan pruning property in [21] to cut the search space of non-closed patterns. We only grow the pattern *NPt* if it does not satisfy the pruning condition (Lines 10-11). We abstract the support computation process from the algorithm; to efficiently calculate support, we use the projected database operations defined in [21].

Algorithm 1 Mining Closed Unique Iterative Patterns

```
Procedure: Mine Closed Unique Patterns
Inputs: $TDB$: Trace database
        $min\_sup$: Minimum support threshold

1: Let $FqEv = \{p | (|p| = 1) \wedge (sup(p) \geq min\_sup)\}$
2: for every $e$ in $FqEv$
3:    Call GrowRec ($e$, $TDB$, $min\_sup$, $FqEv$)

Procedure GrowRec
Inputs: $Pat$: Pattern so far
        $TDB$: Trace database
        $min\_sup$: Minimum support threshold
        $FqEv$: Set of frequent events

4: Let $FqLoc = \{e \in FqEv | sup(Pat{+}{+}e) \geq min\_sup)\}$
5: if ($Pat$ is closed unique)
6:    Output $Pat$
7: if ($Pat$ is unique)
8:    for every $f \notin Pat$ in $FqLoc$
9:       Let $NPt = Pat{+}{+}f$
10:      if $NPt$ doesn't satisfy the InfixScan pruning
                                        condition in [21]
11:         Call GrowRec($NPt$, $TDB$, $min\_sup$, $FqEv$)
```

In Section 5, we show through experiments and case studies that closed unique patterns are powerful enough to be utilized as discriminative features and achieve satisfactory classification accuracy. We also show that Algorithm 1 is scalable and efficient even with low *min_sup* thresholds on datasets, while closed iterative pattern mining is unable to finish even at a high *min_sup* due to an explosion in the number of patterns. To abbreviate, after this point, unless otherwise stated, closed patterns refer to closed unique iterative patterns.

4. ITERATIVE PATTERN-BASED CLASSIFICATION OF SOFTWARE TRACES

Different from relational data, a program trace, which is composed of a sequence of single events, has no predefined feature vector. One could potentially use the set of events as a feature vector, however the single events by themselves are not discriminative to capture the temporal order in a trace. To solve this problem, we generate the set of closed patterns by Algorithm 1 from a set of program execution traces containing failing and normal traces, and use them as classification features. This is the core idea of our proposed iterative pattern-based classification approach. Existing studies which used frequent itemsets [9] and frequent subgraphs [10, 29] for classifying transaction data and graphs have demonstrated the effectiveness of the frequent pattern-based classification approach. Furthermore, Cheng *et al.* [9] derived a frequency upper bound of discriminative measures such as information gain and Fisher score, showing a close relationship between frequency and discriminative measures. The theoretical results demonstrate that most discrimina-

Algorithm 2 Feature Selection on Iterative Patterns

```
Procedure: Feature selection
Inputs: $\mathcal{F}$: A set of frequent iterative patterns
        $TDB$: Trace database
        $\delta$: Coverage threshold
Output: $\mathcal{F}_s$: A selected set of iterative patterns

Sort iterative patterns in $\mathcal{F}$ in decreasing order of
  Fisher score;
Start with the first pattern $f_0$ in $\mathcal{F}$;
while (true)
    Find the next pattern $f$;
    if $f$ covers at least one sequence in $TDB$
        $\mathcal{F}_s = \mathcal{F}_s \cup \{f\}$;
    $\mathcal{F} = \mathcal{F} - \{f\}$;
    if a sequence $S$ in $TDB$ is covered $\delta$ times
        $TDB = TDB - \{S\}$;
    if all sequences are covered $\delta$ times or $\mathcal{F} = \phi$
        break;
return $\mathcal{F}_s$
```

tive patterns likely fall into the high-quantile of frequency, *i.e.*, if we rank all iterative patterns according to their frequency, those discriminative patterns usually have a high rank. We name this phenomenon *frequency association.* Our proposed frequent iterative patterns will serve as discriminative features for distinguishing software behaviors.

The pattern-based classification method includes three major steps: iterative pattern mining which has been discussed in Section 3, feature selection and model learning. In the following, we will examine the feature selection and model learning issues.

4.1 Feature Selection

The set of closed iterative patterns mined from the set of failing and normal traces are considered as the initial set of features. Usually a large number of patterns will be generated from the mining step. Assume the initial feature set is $\mathcal{F} = \{f_1, f_2, ..., f_m\}$ where each iterative pattern f_i represents a feature. Given a software trace S and a feature set $\mathcal{F}$, $\mathbf{x}$ is the feature vector representation of S. Then,

$$x_i = \begin{cases} sup(f_i, S), \ if\ S\ contains\ f_i \\ 0,\ otherwise. \end{cases} \tag{1}$$

where $sup(f_i, S)$ is the support of f_i in the trace S. In other words, we treat an iterative pattern f_i as a feature and its occurrence frequency $sup(f_i, S)$ in a sequence S as the corresponding feature value.

To evaluate the discriminative power of a feature, the popularly used statistical measure of Fisher score [13] is adopted. This score is defined as:

$$Fr = \frac{\sum_{i=1}^{c} n_i(\mu_i - \mu)^2}{\sum_{i=1}^{c} n_i \sigma_i^2} \tag{2}$$

where n_i is the number of data samples in class i, μ_i is the average feature value in class i, σ_i is the standard deviation of the feature values in class i, and μ is the average feature value in the whole dataset. Assume x_{ij} is the attribute value for the jth instance in class i, then μ, μ_i and σ_i are defined as $\mu = \frac{\sum_i \sum_j x_{ij}}{\sum_i n_i}$, $\mu_i = \frac{\sum_j x_{ij}}{n_i}$, $\sigma_i = \sqrt{\frac{\sum_j (x_{ij} - \mu_i)^2}{n_i}}$, respectively. When μ is quite different from each μ_i, or when each σ_i is very small, Fisher score becomes large. A feature will have a very large Fisher score if it has very similar values within the same class and very different values across different classes. In this case, this feature is very discriminative to differentiate instances from different classes. Therefore, if the occurrence frequency of an iterative pattern in the failing traces is different from that in the normal traces, the pattern is discriminative with a large Fisher score. On the other hand, an iterative pattern with similar occurrence frequency in both failing and normal executions is not discriminative.

We then rank the set of iterative patterns in the descending order of Fisher score. A feature selection algorithm is proposed in Algorithm 2 to filter indiscriminative patterns. The algorithm performs a sequential scan of the ranked iterative patterns. If a pattern covers some training instances, it will be selected. Any data instances covered by at least δ features will be removed from further consideration. The algorithm terminates if either all instances are covered by the selected features or the feature set becomes empty.

4.2 Iterative Pattern-based Classification

Our proposed iterative pattern-based classification framework can be divided into three stages:

1. Mine closed unique iterative patterns from a software trace database TDB using Algorithm 1.
2. Select discriminative iterative patterns from the pattern set in step 1 using Algorithm 2. Then represent the trace database TDB in the feature space of the selected iterative patterns.
3. Train a classifier from the trace database TDB. In this study, LIBSVM [8] with probability estimates is used as the classification model.

The iterative pattern-based classification framework is described in Algorithm 3.

Algorithm 3 Iterative Pattern-based Classification

```
Procedure: Model construction
Inputs: $TDB$: Trace database
        $min\_sup$: Minimum support threshold
        $\delta$: Coverage threshold
Output: $Classifier$: Software behavioral classifier

Let $\mathcal{F}$ = Mine_Closed_Unique_Pat($TDB, min\_sup$);
Let $\mathcal{F}_s$ = Feature_Selection($\mathcal{F}$, $TDB$, $\delta$);
Transform $TDB$ into the feature space of $\mathcal{F}_s$;
$Classifier$ = Train a classifier on $TDB$;
return $Classifier$;
```

4.3 Handling Skewed Class Distribution

The number of collected failing traces is usually much smaller than that of the normal traces. For example, from the *print_tokens* system described in Section 5, the portion of traces with anomalies is less than 10%. The skewed class distribution causes two problems in the iterative pattern-based classification process. First, when directly mining from the skewed dataset with both failing and normal traces, iterative patterns prevalent in the normal execution traces will dominate the result set while iterative patterns unique in the failing traces are overwhelmed. As a consequence, the failing traces will be insufficiently represented by iterative pattern features. Second, skewed class distribution also poses great challenges in the model learning phase. Traditional inductive learning methods would perform rather poorly on such software traces with skewed distribution, since the goal of those methods is to minimize classification error rate. As a result, the failing traces tend to be ignored and every instance is predicted as normal. In such cases, we could build a model with very high accuracy, but of very little use in practice.

To address the above two challenges caused by skewed class distribution, the failing traces in the training set are duplicated multiple times until the class distribution becomes balanced. Iterative pattern mining on the balanced training set will discover discriminative patterns from both classes. In addition, a classification model from the balanced training set avoids bias towards the majority class as well as improves recall on the minority class. Other techniques to handle the class imbalance issue are also applicable, including sampling [6] and ensemble [12]. But it is beyond the scope of this paper.

5. EXPERIMENTS AND CASE STUDIES

To validate the utility of our proposed classifier, we performed a set of controlled experiments on simulated trace data and case studies on real trace data. The experiments and case studies are designed to test whether the iterative pattern-based features are useful in detecting software failures. A characterization of failures based on how successful the pattern-based features are in training good classifiers is also discussed.

5.1 Evaluation Methodology

We test the performance of our proposed iterative pattern-based classification approach on nine datasets. For comparison, we design a baseline method, which uses individual events in a software trace as features. According to this definition, a sequence is represented based on a feature vector of unique single events. The corresponding feature value is the number of times a particular event occurs in that sequence. The baseline method simply measures the frequency distribution of single events in a sequence, but ignores the temporal order between them. For fair comparison, we apply the same feature selection procedure based on Fisher score to the baseline method as well. We denote the baseline method as **Evt** and our iterative pattern-based method as **Pat** in the following tables. Minimum support in iterative pattern mining is set to 0.25 unless stated otherwise. Parameter δ in Algorithm 2 is set to 5.

We use LIBSVM [8] with probability estimates as the classification model. Classification accuracy, defined as the percentage of test cases correctly classified, is used as one measure. Due to the skewed class distribution, the measure AUC which is the area under a ROC curve is also used. ROC curve shows the trade-off between true positive rate and false positive rate. The best possible classifier would generate an optimal AUC value of 1.0.

For each dataset, we perform 5-fold cross validation. In each fold, the training set is first rebalanced by duplicating the rare class if the class distribution is skewed. It is then used for iterative pattern mining, feature selection and model learning, while the test set with the original class distribution, is only used for prediction given a constructed classifier. Average performance over 5-fold cross validation is reported. Since both iterative pattern mining and feature selection are performed for each fold separately (on the training set in each fold), our evaluation guarantees that there is no information leak in classification.

5.2 Controlled Experiments

For the controlled experiments, we generated simulated data, *i.e.*, synthetic program traces, using the simulator QUARK proposed in [19]. QUARK provides a controlled environment for experiments. Given an input software component model in the form of a probabilistic finite state automaton, QUARK can generate traces that represent the model well following some coverage criteria. QUARK is also able to inject errors to the synthetic traces.

In this sub-section, we describe the models used to generate traces using our software behavior simulator. We then describe the generated synthetic datasets and the experimental results. Due to the space limitation, we move the graphical representation of the model to a technical report [2].

CVS Application. The first program we analyze is a Concurrent Versions System (CVS) application built on top of FTP library of Jakarta Commons Net [26]. This CVS functionality can be considered as a client of Jakarta Commons Net with a certain protocol pattern. The application is originally described in [19, 20].

There are six FTP interaction scenarios in our CVS implementation: initialization, multiple-file upload, download, and deletion, multiple-directory creation and deletion. All scenarios begin by connecting and logging in to the FTP server. They end by logging off and disconnecting from the FTP server. Each invocation of a method of `FTPClient` may generate exceptions, especially `FTPConnectionClosedException` and `IOException`. Hence the code accessing the `FTPClient` methods needs to be enclosed in a `try...catch...finally` block. Every time such an exception happens, the program simply logs out and disconnects from the FTP server. This is a bug as the corresponding CVS scenarios (*e.g.*, file upload, deletion, *etc.*) are not performed atomically and some method calls are omitted. It has been studied in [28] that programmers often make mistakes on the exceptional control flow path, *i.e.*, those involving error handler like `try...catch...finally`.

X11 Windowing Protocol. Next, we experimented with a model describing an XLib and XToolkit intrinsic library usage protocol for X11 windowing system previously studied in [5, 20].

A common security concern is intrusion where valuable system resources are utilized by unauthorized system or party. An example of valuable system resources is privileged system call [27]. A system call can be "dangerous" when used inappropriately. We model a potential intrusion by adding an extra transition corresponding to a misuse of a privileged system call to the X11 Windowing Protocol model. The models of X11 windowing protocol with and without errors are available in the technical report [2].

Three Types of Injected Bugs. In the first program, we injected *omission* bugs. Some method calls do not get called when they should have been, which is a common type of bug. For the second program, we injected *additional* events resulting in failures. This is also a fairly common type of bug and could correspond to security concerns. The third type of bug is referred to as *ordering* bug where the order of events occurring is wrong. This bug occurs, for example, in the wrong usage of an Application Programming Interface (API). Furthermore, ordering bug has recently been reported as one common family of concurrency bug which is not addressed by existing bug detection algorithms [22]. To simulate this ordering bug on API, we simply reorder the traces from our model of CVS application.

Experiment Details and Results. We generated three sets of traces from the X11 and CVS models and attached labels to them. We compared our iterative pattern-based method with the single event-based method. The sizes of the datasets in terms of the number of traces are described in Table 1. Datasets "X11" and "CVS Omission" contain only addition and omission bugs respectively; "CVS Ordering" contains ordering bugs by permuting events in the traces of a CVS model; and "CVS Mix" contains a mixture of all three types of bugs. It is obvious that addition/omission bugs change the frequency distribution of single events in software traces while ordering bugs do not affect the frequency distribution but only change the order.

Table 1 also shows the classification accuracy, AUC and standard deviation on synthetic software traces using 5-fold cross validation. For these four datasets, our iterative pattern-based method selects 7, 16, 5 and 26 features, respectively. We observe that, by capturing the frequency distribution of single events, the single event-based method can effectively detect addition/omission bugs. Therefore, it achieves very high accuracy on the first two datasets. However, in the latter two datasets, ordering bugs are injected by permutation of single events without affecting the frequency distribution, thus the single event-based method fails in identifying the ordering bugs. As a result, in "CVS Ordering", all test traces are predicted as normal traces with an accuracy of 50% and AUC of 0.50. On the other hand, the iterative pattern-based method works well on all three types of addition, omission and ordering bugs.

5.3 Case Studies – Siemens & MySQL

To further evaluate our classification framework for software failure detection, we analyze different programs from Siemens Test Suite [15]. The test suite was originally used for research in test coverage adequacy and was developed by Siemens Corporation Research. We use the variant provided at www.cc.gatech.edu/aristotle/Tools/subjects/. The test suite contains several programs. Each program contains many different versions where each version has one bug. These bugs comprise a wide array of realistic bugs.

We take four largest programs in the test suite. They are referred to as: *replace*, *schedule*, *print_tokens* and *tot_info*. There are two variants of *schedule* and *print_tokens* programs. For this case study, we choose the first variant of each program. More information of the test suite is available in [15, 18].

To simulate real life situation where there are many bugs occurring together, we inject 3 bugs to each program and add 3 additional simulated ordering bugs to the execution traces. Running the instrumented program with an input produces a trace. We collect a set of traces by running a set of test cases provided by Siemens Test Suite. The test suite also allows us to compute the actual correct output. By comparing the output of the program with the correct actual output we can see if the program runs correctly or not. We label a trace as 0 if it corresponds to a correct execution or 1 if it corresponds to a failing execution. We run the test cases on each of the buggy programs, collect traces, and label them accordingly.

For *replace*, *schedule*, *print_tokens* and *tot_info* programs we run 5548, 2637, 4092 and 1067 test cases respectively. Some test cases cause the program to crash, and others produce the same trace as another test case. We remove the test cases that cause the program to crash and remove duplicate traces. There is no need to build a classifier for crashes as the crash itself is a sure evidence of a trace being faulty. Non-crashing failures are hard to identify and are the focus of our study. We also add some extra traces containing simulated ordering errors. The description of the datasets after the above processing is shown in Table 2.

From the four programs, we find that bug happens in only a minority of traces. In the first two datasets (*tot_info* and *schedule*), we introduce more simulated ordering errors to get balanced trace datasets. In the last two datasets (*print_tokens* and *replace*), we collect *unbalanced* datasets. We handle the skewedness issue by duplicating the failing traces multiple times to get a balanced training set in each fold, as described in Section 4.3. The numbers in Table 2 correspond to the number of traces before the duplication is performed.

We also analyze a data race concurrency bug from MySQL server [1]. A data race bug causes wrong ordering of statement executions. The bug causes a corrupted *mysqlbinlog*. Since the log is used to restore databases, inconsistency can result from the bug. This bug is rated as serious in MySQL bug database. We collected 102 traces, where 51 correspond to the case when the bug is manifested (*i.e.*, corrupted *binlog* and another 51 correspond to the case when the bug is not manifested (*i.e.*, *binlog* is not corrupted).

Table 2 shows the classification accuracy, AUC and standard deviation on the real software traces. For these five datasets, our iterative pattern-based method selects 65, 38, 49, 27 and 11 features, respectively. It is very clear that, according to both accuracy and AUC, the iterative pattern-based method outperforms the single event-based method significantly by preserving the important temporal information in the software traces. We find that our method works well on both balanced and unbalanced datasets (by applying the rebalance technique). For the Siemens dataset, we find that the lengths of patterns range between 1 and 10 with an average of 4.29. For the MySQL dataset, the lengths of patterns vary between 4 and 15 with an average of 9.63. The pattern set provides a rich combination of single features which contributes to the classification accuracy in classifying correct and failing traces. The results show that our method outperforms by up to 50% accuracy on the real-life MySQL bug that involves only wrong ordering of statement executions.

5.4 Varying min_sup and Database Size

In this sub-section, we describe an extended set of experiments to see the effect of varying minimum support on classification accuracy. We also investigate the scalability of our iterative pattern mining algorithm (which takes up the bulk of the time required for classification) on a range of support values and sizes of the trace databases.

Table 3 shows the classification performance with standard deviation when *min_sup* is varied in the range of [0.05, 0.50] to mine the iterative patterns on traces of the *replace* program. As we increase *min_sup*, we get fewer number of iterative patterns and may miss some highly discriminative ones. As a result, the classification performance slightly degrades, but still maintains a good performance in general.

Table 1: Synthetic Datasets Description and Classification Performance

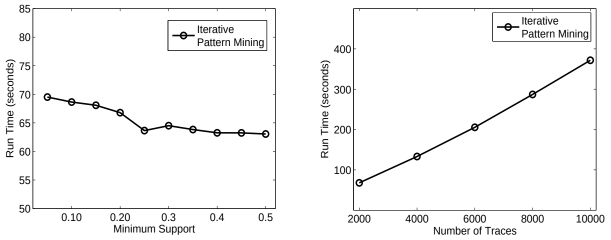
Dataset	Correct	Error (|traces|)		Accuracy		AUC	
	(|traces|)	Add/Omis.	Order	Evt	Pat	Evt	Pat
X11	125	125	0	96.40 ± 4.10	**97.20 ± 3.35**	0.97 ± 0.04	**1.00 ± 0.00**
CVS Omission	170	170	0	95.29 ± 1.61	**100.00 ± 0.00**	0.96 ± 0.03	**1.00 ± 0.00**
CVS Ordering	180	0	180	50.00 ± 0.00	**85.28 ± 2.71**	0.50 ± 0.00	**0.82 ± 0.08**
CVS Mix	180	90	90	66.39 ± 15.63	**93.89 ± 5.94**	0.65 ± 0.17	**0.95 ± 0.06**

Table 2: Siemens & MySQL Trace Datasets Description and Classification Performance

Dataset	Correct	Error (|traces|)		Accuracy		AUC	
	(|traces|)	Add/Omis	Order	Evt	Pat	Evt	Pat
tot_info	302	208	94	77.33 ± 2.31	**90.67 ± 5.82**	0.90 ± 0.03	**0.94 ± 0.03**
schedule	2140	289	1851	52.83 ± 19.27	**86.26 ± 14.90**	0.57 ± 0.25	**0.88 ± 0.16**
print_tokens	3108	187	187	72.60 ± 26.33	**99.94 ± 0.08**	0.64 ± 0.17	**1.00 ± 0.00**
replace	1259	269	269	61.12 ± 9.25	**90.84 ± 2.54**	0.63 ± 0.15	**0.93 ± 0.05**
MySQL	51	0	51	50.00 ± 0.00	**100.00 ± 0.00**	0.50 ± 0.00	**1.00 ± 0.00**

Table 3: Classification Performance vs. *min_sup*

min_sup	Accuracy	AUC
0.05	90.9497 ± 2.9203	0.9344 ± 0.0454
0.10	90.9497 ± 2.9203	0.9344 ± 0.0454
0.15	90.9004 ± 2.5949	0.9323 ± 0.0509
0.20	90.8939 ± 2.5949	0.9321 ± 0.0499
0.25	90.8380 ± 2.5402	0.9318 ± 0.0506
0.30	90.7263 ± 2.5555	0.9310 ± 0.0501
0.35	90.2794 ± 2.8650	0.9261 ± 0.0545
0.40	90.2794 ± 2.8650	0.9261 ± 0.0545
0.45	90.2794 ± 2.8650	0.9261 ± 0.0545
0.50	90.2794 ± 2.8650	0.9261 ± 0.0545



(a) Mining Time vs. *min_sup* (b) Mining Time vs. |Traces|

Figure 1: Efficiency and Scalability Tests

Figure 1 (a) shows the iterative pattern mining time on traces of the *replace* program as we vary *min_sup*[2]. We can observe from the figure that the run time of iterative pattern mining does not increase significantly as we lower down *min_sup*. One important factor that helps in ensuring scalability of our mining technique is our new definition of closed unique patterns. With the new definition, the number of mined patterns is less and the occurrences of "noise" events in the data do not cause the number of patterns to explode. Still with this reduced pattern set, the patterns are powerful enough to serve as classification features resulting in good classification accuracy. For comparison, we also tried mining closed iterative patterns which are allowed to have non-unique constituent events. Unfortunately, closed pattern mining throws an out-of-memory exception after running for more than 4 hours and consumes more than 1.7GB of memory even on a support level of 100%[2,3]. This shows the need for and scalability of mining closed unique patterns.

Figure 1 (b) shows the scalability test on iterative pattern mining when we increase the size of input trace database TDB[2]. We increase the number of traces of the *replace* program and run the mining algorithm. As shown in the figure, the mining algorithm scales linearly with the size of the sequence database.

6. RELATED WORK

Dickinson *et al.* detected program failures by performing hierarchical clustering on program traces [11]. They first obtain a set of profiles of interesting events (*e.g.*, branch decision, method calls) in program traces. The traces are then grouped into a pre-defined set of clusters in a hierarchical agglomerative fashion. The process stops when a desired number of clusters are found. Several distance metrics are used to measure similarity between two program traces. They found that small clusters are candidates of failures. Their approach considers only frequency of single events but not the ordering information of event executions. It also remains a question whether the generated clusters can cleanly separate correct behaviors from incorrect ones.

Bowring *et al.* proposed an active learning approach to build a classifier of program behaviors [7]. The input to their approach is a frequency profile of single events in the trace. They learn two sets of *first-order* Markov models, where each set characterizes correct and incorrect behaviors separately. A first-order Markov model (*c.f.*, [17]) is a state-based transition system, where the probability of the next state at time $t+1$ is determined by the current state at time t. Different from this work, our iterative patterns capture more than "first order" relationship, *i.e.*, it can relate that a state frequently leads to another state which is k distance away, where k can be arbitrarily large.

Studies on fault localization (*e.g.*, [18]) are also related to fault detection in programs. They usually work in two phases. In the first phase, a set of labeled traces needs to be provided. Each input trace is labeled to indicate whether it is correct or erroneous. In the second phase, possible error locations are computed. Since it is expensive to assign labels to execution traces manually [11], our classification approach can complement studies in fault localization – a classifier constructed from a small set of labeled traces can be used to label a much larger set of unlabeled traces which, after being labeled, are then used for fault localization.

There are several recent studies on frequent pattern-based classification which use frequent itemsets [9] and frequent connected subgraphs [10, 29] for classifying transaction and graph data. In this work, we enrich the past studies by proposing a pattern-based classification utilizing iterative

[2] Performed on a 2.33 GHz Core 2 Duo desktop with 3.25 GB of RAM running Windows XP. Algorithm is written in C#.

[3] For iterative patterns, since a pattern can repeat multiple times in a sequence, a pattern can have a support value higher than 100% of the number of sequences in the dataset.

patterns. A program trace can alternatively be "coiled" to form a behavior graph [18]. In this work, we prefer to use iterative patterns as features as they capture *repetitions of patterns within a trace*. This information is lost when a trace is "coiled" to form a behavior graph. Also, different from nodes in a connected subgraph, *adjacent events in an iterative pattern not necessarily occur directly after another in a sequence* since gaps are allowed in mining iterative patterns. This permits a degree of flexibility in capturing discriminative patterns that appear in several variants with unrelated events appearing in between.

As to iterative pattern mining, there are many related work including sequential pattern mining [4] and episode mining [23]. Different from those studies, iterative patterns are patterns that repeat a substantial number of times both *within a sequence* and *across multiple sequences*. Constituent events of an iterative pattern can be separated by an arbitrary number of unrelated events in its instances in the execution traces – a pattern is not necessarily an episode occurring close together. A detailed discussion is available in [21]. There are other studies mining repetitive series of events, however since they are not used for pattern based classification, we omit references to them.

7. CONCLUSIONS

In this paper, we proposed a novel approach to mine closed unique iterative patterns for classifying sequential data, through an example of software trace analysis for failure detection. We address the issue of automating testing process by training a classifier based on discriminative iterative patterns and using it to find failures in program traces. A three-step framework is employed including feature generation using closed unique iterative pattern mining, feature selection based on Fisher score, and pattern-based model learning.

Experimental study has been performed on synthetic and real datasets. For the synthetic datasets, we generate traces from various models of existing systems and inject various types of errors: addition, omission and ordering. We also investigate standard programs from the Siemens benchmark and a real-life data race concurrency bug from MySQL. The iterative pattern-based classifier outperforms the single event-based method on all the datasets. On average, classification accuracy is improved by 24.68%. Furthermore, we are able to classify the real-life data race failures accurately while the baseline approach is not.

As a future work, we are looking into the possibility of direct mining of discriminative iterative patterns, applications of the classifier to other domains, and pipelining the proposed approach to existing fault localization techniques to enable both failure detection and fault localization.

8. REFERENCES

[1] Mysql atomicity violation. http://bugs.mysql.com/bug.php?id=169.

[2] *http://www.mysmu.edu/faculty/davidlo/kdd09.htm*, 2009.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB*, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*, 1995.

[5] G. Ammons, R. Bodik, and J. R. Larus. Mining specification. In *Proc. of SIGPLAN Symp. on Principles of Programming Languages*, 2002.

[6] G. E. A. P. A. Batista, R. C. Prati, and M. C. Monard. A study of the behavior of several methods for balancing machine learning training data. *SIGKDD Explorations*, 6(1):20–29, 2004.

[7] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proc. of Int. Symp. on Soft. Testing and Analysis*, 2004.

[8] C-C. Chang and C-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at http://www.csie.ntu.edu.tw/∼cjlin/libsvm.

[9] H. Cheng, X. Yan, J. Han, and C. Hsu. Discriminative frequent pattern analysis for effective classification. In *ICDE*, pages 716–725, 2007.

[10] M. Deshpande, M. Kuramochi, N. Wale, and G. Karypis. Frequent substructure-based approaches for classifying chemical compounds. *IEEE TKDE*, 17(8):1036–1050, 2005.

[11] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. of the Int. Conf. on Software Engineering*, 2001.

[12] T. G. Dietterich. Ensemble methods in machine learning. *Lecture Notes in Computer Science*, 1857:1–15, 2000.

[13] R. Duda, P. Hart, and D. Stork. *Pattern Classification*. Wiley Interscience, 2nd edition, 2000.

[14] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2003.

[15] 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.

[16] ITU-T. ITU-T Recommendation Z.120: Message Sequence Chart (MSC). 1999.

[17] D. Jurafsky and J.H. Martin. *Speech and Language Processing*. Prentice-Hall, 2000.

[18] C. Liu, X. Yan, H. Yu, J. Han, and P.S. Yu. Mining behavior graphs for "backtrace" of noncrashing bugs. In *SDM*, 2005.

[19] D. Lo and S-C. Khoo. QUARK: Empirical assessment of automaton-based specification miners. In *Proc. of Working Conf. on Reverse Engineering*, 2006.

[20] D. Lo and S-C. Khoo. SMArTIC: Toward building an accurate, robust and scalable specification miner. In *Proc. SIGSOFT Symp. on Foundations of Software Eng.*, 2006.

[21] D. Lo, S-C. Khoo, and C. Liu. Efficient mining of iterative patterns for software specification discovery. In *KDD*, 2007.

[22] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes – a comprehensive study on real world concurrency bug characteristics. In *Proc. of Int. Conf. on Archi. Support for Prog. Lang. and OS*, 2008.

[23] H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *DMKD*, 1:259–289, 1997.

[24] Java Trans. API Spec. java.sun.com/products/jta/.

[25] G. Tassey. The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology. Planning Report*, 2002.

[26] The Apache Software Foundation. Jakarta commons/net. *http://jakarta.apache.org/commons/net/*.

[27] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symp. on Security and Privacy*, 1999.

[28] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proc. of Int. Conf. on Tools and Algo. for the Const. and Ana. of Sys.*, 2005.

[29] X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by scalable leap search. In *SIGMOD*, pages 433–444, 2008.