Alattin: Mining Alternative Patterns for Detecting Neglected Conditions

Suresh Thummalapenta
Department of Computer Science
North Carolina State University
Raleigh, USA
sthumma@ncsu.edu

Tao Xie
Department of Computer Science
North Carolina State University
Raleigh, USA
xie@csc.ncsu.edu

***Abstract*—To improve software quality, static or dynamic verification tools accept programming rules as input and detect their violations in software as defects. As these programming rules are often not well documented in practice, previous work developed various approaches that mine programming rules as frequent patterns from program source code. Then these approaches use static defect-detection techniques to detect pattern violations in source code under analysis. These existing approaches often produce many false positives due to various factors. To reduce false positives produced by these mining approaches, we develop a novel approach, called Alattin, that includes a new mining algorithm and a technique for detecting neglected conditions based on our mining algorithm. Our new mining algorithm mines alternative patterns in example form "P_1 or P_2", where P_1 and P_2 are alternative rules such as condition checks on method arguments or return values related to the same API method. We conduct two evaluations to show the effectiveness of our Alattin approach. Our evaluation results show that (1) alternative patterns reach more than 40% of all mined patterns for APIs provided by six open source libraries; (2) the mining of alternative patterns helps reduce nearly 28% of false positives among detected violations.**

***Keywords*-code search; frequent itemset mining; alternative patterns;**

I. Introduction

Programming rules serve as a basis for applying static or dynamic verification tools to detect rule violations as software defects and improve software quality. However, in practice, these programming rules are often not well documented for Application Programming Interfaces (APIs) due to various factors such as hard project delivery deadlines and limited resources in the software development process [1]. To tackle the issue of lacking documented programming rules, various approaches have been developed in the past decade to mine programming rules from program executions [2]–[4], individual versions [5]–[12], or version histories [13], [14] of program source code. A common methodology adopted by these approaches is to mine common patterns (e.g., frequent occurrences of pairs or sequences of API calls) across a sufficiently large number of data points (e.g., code examples). These common patterns often reflect programming rules that should be obeyed when programmers write code using API calls involved in these rules. Then, these approaches use static defect-detection techniques that accept mined patterns as input and detect pattern violations as potential defects in source code under analysis.

In our empirical investigation of using static defect-detection techniques based on code mining, we found that these techniques often produce a high number of false positives. These false positives are partially due to mining and applying each frequent pattern individually. For example, consider that an API call, say API_1, has two mined patterns: P_1 and P_2 (such as condition checks on method arguments or return values of API_1). It is necessary for code examples using API_1 to satisfy at least one of them. Using P_1 alone, a static defect-detection technique reports violations in code examples (using API_1) that do not include P_1, but include P_2. As those code examples include P_2, the reported violations turn out to be false positives.

To address this issue, we introduce the notion of alternative patterns, where we consider all frequent patterns of an API call together. For example, an alternative pattern that includes P_1 and P_2 is of example form "P_1 **or** P_2", where P_1 and P_2 are two alternatives of the pattern. In our context, we refer to the preceding kinds of patterns as *balanced* patterns, where all alternatives are frequent. Even with mining and applying balanced patterns, we found that static defect-detection techniques still produce false positives. Some of these false positives are due to the case of writing source code in different ways to achieve the same programming task, and programmers use certain ways much more frequently than others. To address this issue, we need to mine and apply patterns that include both frequent and infrequent alternatives. We refer to such patterns as *imbalanced* patterns in example forms as "P_1 **or** $\hat{P_2}$", where P_1 and P_2 are frequent and infrequent alternatives, respectively. In our pattern representation, we annotate infrequent alternatives with $\wedge$. In an imbalanced pattern, some alternatives (such as P_1) dominate other alternatives (such as P_2) of the pattern. These imbalanced patterns include additional programming rules that can be used for both program comprehension and defect detection (as shown in our empirical results in Section IV).

We next show an example of an imbalanced pattern related to the `next` method of the `Iterator` class in

Example 1:

```
00:String printEntries1(ArrayList<String>
          entries){
01: ...
02: Iterator it = entries.iterator();...
03: if (it.hasNext()) {
04:    String last = (String) it.next();... }}
```

Example 2:

```
00:String printEntries2(ArrayList<String>
          entries){
01: ...
02: if (entries.size() > 0) {
03:    Iterator it = entries.iterator();...
04:    String last = (String) it.next();... }}
```

Figure 1. Two code examples using the `next` method of the `Iterator` class

the Java Util package. Using this example, we show the importance of imbalanced patterns and also show that existing approaches [15], [16] cannot mine these imbalanced patterns. Figure 1 shows two code examples using the `next` method of the `Iterator` class. The `next` method throws `NoSuchElementException` when invoked on an `ArrayList` object without any elements. To prevent this exception, a frequent condition check is "P_1: `boolean-check` on `return` of `Iterator.hasNext` before `Iterator.next`" (shown in `printEntries1` from Example 1). Another infrequent condition check that can also help prevent the exception is "P_2: `boolean-check` on `return` of `ArrayList.size` before `Iterator.next`" (shown in `printEntries2` from Example 2). Both P_1 and P_2 are valid alternatives, which ensure that there are elements in the `ArrayList`. To show that the pattern "P_1 **or** $\hat{P}_2$" is an imbalanced pattern, we gathered $1,243$ code examples that use the `Iterator.next` method through Google code search [17]. We found that P_1 and P_2 exist in $1,218$ and 6 code examples (with support values 0.92 and 0.0048), respectively. Although both P_1 and P_2 are valid, these support values show that the frequent alternative P_1 dominates the infrequent alternative P_2. Therefore, existing mining approaches such as frequent itemset mining [15] can mine *only* P_1 but not P_2. However, the alternative P_2 is important when these mined patterns are used for defect detection. For example, consider that only P_1 is mined without the P_2 alternative. Using P_1, a static defect-detection technique reports a violation in `printEntries2` since the method does not satisfy P_1. However, the code example does not include any defect on using `Iterator.next` since `printEntries2` satisfies P_2; therefore, the detected violation is a false positive. This example shows that mining imbalanced patterns can help reduce false positives among detected violations. But mining imbalanced patterns such as "P_1 **or** $\hat{P}_2$" is challenging since alternatives such as P_2 are not frequent among an entire set of code examples used for mining. To the best of our knowledge, there are no existing data mining approaches that can mine imbalanced patterns.

In this paper, we propose a novel approach, called Alattin, that includes a new mining algorithm, called *ImMiner*, and a technique that detects neglected conditions based on our mining algorithm. Our ImMiner algorithm uses an iterative mining strategy to mine imbalanced patterns. ImMiner is based on the observation that alternatives such as P_2 (despite infrequent in an entire set of code examples) are frequent among the code examples that do not support P_1. We then apply our mining algorithm to address the problem of detecting neglected conditions. *Neglected conditions*, also referred to as missing paths, are known to be an important category of software defects and are considered to be one of the primary reasons for many fatal issues such as security or buffer overflow vulnerabilities [10]. As shown by a recent study [10], 66% (109/167) of defect fixes applied in the Mozilla Firefox project are due to neglected conditions. In particular, neglected conditions (related to an API call) refer to (1) missing conditions that check the arguments or receiver of the API call before the API call or (2) missing conditions that check the return values or receiver of the API call after the API call.

To detect neglected conditions related to an API call in an application under analysis, Alattin extracts APIs reused by the application. Then, Alattin interacts with a Code Search Engine (CSE) such as Google code search [17] to gather relevant code examples for each API. Alattin analyzes these relevant code examples statically to generate pattern candidates suitable for applying our ImMiner algorithm. ImMiner mines both balanced and imbalanced patterns related to these APIs. These mined patterns describe programming rules that must be obeyed in reusing the APIs. Finally, Alattin uses mined patterns to detect violations in the application under analysis.

This paper makes the following main contributions:

- An empirical investigation of mined patterns to show the existence of alternative patterns and their further classification into balanced and imbalanced patterns. These balanced and imbalanced patterns include programming rules that can be used for tasks such as program comprehension and defect detection.
- A novel mining algorithm, called ImMiner, that mines alternative patterns in the form of balanced and imbalanced patterns.
- A technique that applies ImMiner for detecting neglected conditions around individual API calls in an application under analysis and an Eclipse plugin implemented for the technique.
- Two evaluations to demonstrate the effectiveness of our approach. Our evaluation results show that (1) alternative patterns reach nearly 40% of all mined patterns for APIs provided by six open source libraries; (2) the mining of alternative patterns helps reduce nearly 28% of false positives among detected violations.

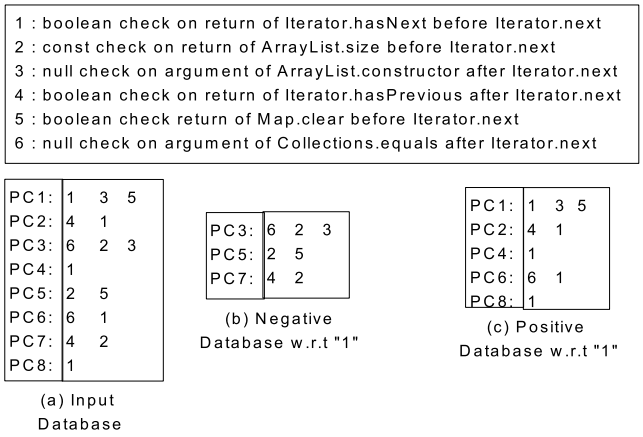

Figure 2. Example for the `Iterator.next` API method used to illustrate our mining algorithm

The rest of the paper is organized as follows. Section II presents the problem definition and solution of our ImMiner algorithm. Section III describes key aspects of the approach. Section IV presents evaluation results. Section V discusses threats to validity. Section VI discusses issues and future work of our approach. Section VII presents related work. Finally, Section VIII concludes.

II. ImMiner Algorithm

We next present an example to explain the major concepts of our ImMiner algorithm and then present a formal definition of the problem addressed by our mining algorithm.

A. Example

Figure 2a shows an input database for applying our mining algorithm, where each row includes the surrounding condition checks (i.e., condition checks before or after an API call) of the same API method from a code example. We refer to each row in the input database as a pattern candidate (PC). For example, PC1 describes that three condition checks are done before and after invoking `Iterator.next`. Given such input database, ImMiner uses two steps for mining alternative patterns of the form "P_1 **or** $\hat{P_2}$", where P_1 represents "`boolean check` on the return of `Iterator.hasNext` before `Iterator.next`" and P_2 represents "`const check` on the return of `ArrayList.size` before `Iterator.next`".

In Step 1, ImMiner applies frequent itemset mining [15] on the input database with a *min_sup* value, say 0.5. ImMiner identifies a frequent pattern as "P_1: `boolean check` on the return of `Iterator.hasNext` before `Iterator.next`". In Step 2, ImMiner splits the input database into two databases with respect to P_1: negative and positive. The negative database includes all pattern candidates that do not include P_1, whereas the positive database includes all pattern candidates that include P_1. Figures 2b and 2c show the positive and negative databases split with respect to P_1. ImMiner next applies frequent itemset mining on the negative database to identify frequent patterns. ImMiner identifies that the frequent pattern in the negative database is "P_2: `const check` on the return of `ArrayList.size` before `Iterator.next`". Then ImMiner combines these two patterns to construct an alternative pattern "P_1 **or** $\hat{P_2}$", where P_1 is a frequent alternative and P_2 is an infrequent alternative.

B. Problem Definition

We next present a formal definition of the mining problem targeted by our ImMiner algorithm based on *frequent itemset mining* [15]. Although we present our problem definition using frequent itemset mining, our ImMiner algorithm is general and can be used with other mining approaches (such as frequent subsequence mining [16]) as well.

Let $M = \{mi_1, mi_2, ..., mi_k\}$ be the set of all possible distinct items. Consider an ItemSet Database ISD as $\{is_1, is_2, ...,is_l\}$, where each itemset is_j includes different sets of elements such as $\{mi_1, mi_2, ..., mi_a\}$ from the set of all possible distinct elements. Consider that a frequent itemset mined from the ISD with a threshold value, say *min_sup*, is $FIS = \{fi_1, fi_2, ..., fi_b\}$ (FIS denotes Frequent ItemSet). Each fi_j is in the form of $\{mi_1, mi_2, ..., mi_x\}$. In our context, we refer to FIS as a dominating pattern. The objective of our ImMiner algorithm is to mine imbalanced patterns of the form "FIS **or** $\hat{AIS}$", where AIS is an infrequent alternative pattern of the form $\{ai_1, ai_2, ..., ai_c\}$ (AIS denotes Alternative ItemSet). For each item $fi_i \in FIS$, there can be multiple infrequent alternative items $\{ai_1, ai_2, ..., ai_d\} \in AIS$. The characteristics of each ai_j are such that ai_j is the most frequent in the Negative itemSet Database (NSD) and is infrequent in the Positive itemSet Database (PSD). Here, NSD represents all itemsets of ISD that do not support $fi_i \in FIS$. We consider that an itemset such as is_i does not support frequent item fi_i, if $fi_i \cap is_i == \emptyset$. In contrast, PSD represents all itemsets of ISD that support fi_i. We consider that an is_i supports fi_i, if $fi_i \cap is_i == fi_i$. Note that $PSD \cap NSD == \emptyset$. However, $PSD \cup NSD \neq ISD$ as there can be some other itemsets is_j that partially support fi_i. An is_j is considered as partially supporting fi_i, if $fi_i \cap is_i \neq \emptyset$ and $fi_i \cap is_i \neq fi_i$. In our algorithm, we discard such itemsets since these itemsets neither support nor reject the mined items fi_i.

In our problem definition, we represent imbalanced patterns as "FIS **or** $\hat{AIS}$". We next present a proof for our representation. Consider that $ai_j \in AIS$ is an infrequent alternative for $fi_i \in FIS$. We represent this pattern as "R1: fi_i **or** $\hat{ai_j}$". Actually, these imbalanced patterns should be of the form "R2: fi_i **or** ($\neg$ fi_i **and** $\hat{ai_j}$)", since $\hat{ai_j}$ is an infrequent alternative of fi_i. However, both representations R1 and R2 are equivalent based on the following proof by considering each alternative as a boolean variable.

R2: fi_i **or** ($\neg$ fi_i **and** $\hat{ai_j}$)
$\Rightarrow$ (fi_i **or** $\neg$ fi_i) **and** (fi_i **or** $\hat{ai_j}$) by distributivity
$\Rightarrow$ `True` **and** (fi_i **or** $\hat{ai_j}$)
$\Rightarrow$ R1: fi_i **or** $\hat{ai_j}$

Our problem definition also includes a special category of alternative patterns with *only* one alternative in the pattern. This scenario happens when $|FIS|$ = 1 and $AIS = \emptyset$. In summary, our problem definition includes the following three categories of mined patterns defined based on "FIS **or** $\hat{AIS}$".

- *Balanced:* $AIS = \emptyset$ and $|FIS| > 1$
  Example: All alternatives are frequent such as "fi_1 **or** fi_2 **or** ... **or** fi_b".
- *Imbalanced:* $AIS \neq \emptyset$ and $|FIS| \geq 1$
  Example: Some alternatives are infrequent such as "fi_1 **or** fi_2 **or** ... **or** fi_b **or** $\hat{ai_1}$ **or** $\hat{ai_2}$ **or** ... **or** $\hat{ai_c}$".
- *Single*: $AIS = \emptyset$ and $|FIS|$ = 1
  Example: Mined pattern includes only one alternative such as "fi_i".

C. Solution

ImMiner mines all three categories of patterns in two steps. In Step 1, ImMiner mines balanced patterns, where all alternatives are frequent. In Step 2, for each frequent alternative, ImMiner mines infrequent alternatives.

Step 1. ImMiner applies frequent itemset mining such as MAFIA [15] on the input database ISD. ImMiner uses a minimum support threshold, say *min_sup*, for mining frequent patterns such as FIS. ImMiner assigns a support value given by frequent itemset mining to each mined pattern FIS, represented as $SUP(FIS)$.

Step 2. To mine imbalanced patterns for each frequent alternative fi_i, ImMiner partitions ISD into two groups with respect to the frequent alternative: the negative itemset database (NSD) and the positive itemset database (PSD). ImMiner partitions the ISD database in such a way that every itemset NIS of NSD does not include any of the items of fi_i, whereas every itemset PIS of PSD includes all items of fi_i[1]. ImMiner next applies frequent itemset mining on NSD to gather frequent patterns in NSD for each frequent alternative fi_i. Consider a mined pattern of NSD as ai_i. We next compute a final support value for each infrequent alternative, referred to as ABS, using the three formulas below. The notation $\psi(X, Y)$ represents the support of pattern X in database Y.

- $\psi(ai_i, NSD)$ = $\frac{\#\ of\ patterncandidates\ in\ NSD\ supporting\ ai_i}{Total\ \#\ of\ patterncandidates\ in\ NSD}$
- $\psi(ai_i, PSD)$ = $\frac{\#\ of\ patterncandidates\ in\ PSD\ supporting\ ai_i}{Total\ \#\ of\ patterncandidates\ in\ PSD}$
- $ABS(ai_i)$ = $\psi(ai_i, NSD)$ - $\psi(ai_i, PSD)$

Our formulas assign higher support to those alternative patterns that are frequent in NSD and are infrequent in PSD. The rationale behind these formulas is that *only* such patterns can be treated as infrequent alternative patterns for FIS. Similar to *min_sup* for mining FIS, we use another user-defined threshold such as *alt_sup* for mining infrequent alternative patterns. Based on the number of elements in FIS and AIS, ImMiner assigns category types to mined patterns.

[1]Recall that we discard itemsets that partially support fi_i

```
01:public Object evaluate(Object val) { ...
02:   if (val != null &&
                 val instanceof Collection) {
03:      Collection coll = (Collection) val;
04:      Iterator i = coll.iterator();
05:      if(!coll.isEmpty()) {
06:         for (; i.hasNext();) {
07:            Object obj = i.next();
08:            if(obj instanceof Node) {
09:               Node node = (Node) obj;
10:               //...
11:            } } } }
12:   return new Double(sum);
13:}
```

Figure 3. A code example using `Iterator.next` gathered from Google code search.

III. Alattin Approach

Our Alattin approach accepts an application under analysis and detects neglected conditions around APIs reused by the application. More specifically, Alattin scans the application and gathers APIs reused by the application. Alattin uses ImMiner to mine patterns that serve as programming rules in reusing those APIs. Then Alattin detects violations of these programming rules. In summary, Alattin includes four major phases. In Phase 1, Alattin gathers relevant code examples that reuse APIs. In Phase 2, Alattin analyzes gathered code examples to generate pattern candidates suitable for mining. In Phase 3, Alattin applies ImMiner on pattern candidates to mine patterns. In Phase 4, Alattin detects violations of mined patterns in the application under analysis. We next explain each phase in detail. We use notations C_i and F_i to denote a class or a method used by the application under analysis, respectively.

A. Phase 1: Gathering Code Examples

Our approach gathers code examples that include information on how to reuse classes and methods used by the application under analysis. Gathering code examples from a small number of project code bases often cannot surface out many programming rules as common patterns. The primary reason is that there are often too few data points in a small number of code bases to support the mining of desirable patterns. This phenomenon is reflected on empirical results reported by existing mining approaches [6], [10]: often a relatively small number of real programming rules mined from one or a few huge code bases.

To address the preceding issues of a limited number of code bases, Alattin collects code examples from existing

open source repositories through code search engines (CSE) such as Google code search [17] and Koders [18]. These code search engines are primarily used by programmers in searching for relevant code examples from available open source projects on the web. As these CSEs can serve as powerful resources of open source code, these CSEs can be exploited for other tasks such as detecting violations in applications that reuse existing open source projects. Our approach uses a CSE to gather relevant code examples and mines gathered code examples to detect violations in an application under analysis.

To collect code examples through a CSE, Alattin constructs queries for each C_i and F_i. For example, Alattin constructs query of the form "`lang:java Iterator next`" to collect code examples that invoke the `next` method of the `Iterator` class. More specifically, our queries include the names of the class and method along with the language type. Alattin stores gathered code examples in the local file system for further analysis. Alattin uses Google code search (GCSE) [17] for collecting relevant code examples with two main reasons: (1) GCSE provides client libraries that can be used by other tools to interact with and (2) GCSE has public forums that provide good support. However, our approach is independent of GCSE and can leverage any other CSE to gather relevant code examples.

B. Phase 2: Generating Pattern Candidates

In Phase 2, Alattin analyzes gathered code examples statically to generate pattern candidates suitable for mining. These pattern candidates include condition checks that are performed before and after invoking an F_i method. To identify these condition checks on method calls, Alattin has to associate condition checks in the conditional expressions of `If` or `While` statements with the related method calls. We use the `Iterator.next` method and its relevant code example in Figure 3 as a running example for explaining Phase 2.

Alattin includes two sub-phases in Phase 2: CFG construction and traversal. In the CFG construction sub-phase, Alattin constructs CFGs for each code example with two kinds of nodes: control (CT) and non-control (NT) nodes. Control nodes represent control-flow statements such as *if*, *while*, and *for*, which control the flow of the program execution. Non-control nodes represent other statements such as method calls or type casts. For example, Statement 5 in the code example (Figure 3) is a control node and Statement 9 is a non-control node. When encountering a control node, say CT_i (i indicates the statement id), Alattin also extracts all variables, say $\{V_1, V_2, ..., V_n\}$, that participate in the conditional expression of that node and the condition checks on those variables. For example, the control node CT_2 includes the {(`val`, `null-check`), (`val`, `instance-check`)} pairs. If the control node includes comparisons with expressions such as method calls, our approach stores those method calls also as additional information within the control node. When encountering a non-control node such as a method call, Alattin extracts variables such as {`receiver`, `argument1`, ..., `argumentN`} associated with the method call.

In the CFG traversal sub-phase, Alattin associates gathered condition checks with their related method calls such as `Iterator.hasNext`. The traversal phase includes two kinds of traversals: backward and forward. Alattin performs a backward traversal from the call site such as NT_7 of the F_i method to collect condition checks on the receiver and argument objects preceding the call site. Similarly, Alattin performs a forward traversal to collect condition checks on the receiver and return objects after the call site of the F_i method. In each traversal, Alattin exploits program dependencies for associating condition checks with method calls. Failing to consider these program dependencies may result in programming rules that are not semantically related as shown in the limitations of the PR-Miner [6] and DynaMine [13] approaches. To exploit program dependencies, Alattin uses the concept of *dominance* with a combination of *control-flow* and *data-flow* dependencies.

Definition: *A node N* ***dominates*** *another node M in a control flow graph (represented as N dom M) if every path from the starting node of the CFG to M includes N.*

Initially, Alattin identifies the dominant CT_i nodes for each NT_k node. For example, the control node CT_6 dominates the non-control node NT_7. Alattin computes the intersection between the variable set associated with the CT_i node, say $\{V_1, V_2, ..., V_n\}$, and the receiver or argument variables of the NT_k node, say {`receiver`, `argument1`, ..., `argumentN`}. If the intersection $\{V_1, V_2, ..., V_n\} \cap$ {`receiver`, `argument1`, ..., `argumentN`} $\neq \emptyset$, Alattin checks whether the NT_k node is dependent on the CT_i node, i.e., whether there exists at least one variable of NT_k node involved in the CT_i node and is not redefined in the path between CT_i and NT_k nodes. If the NT_k node is dependent on the CT_i node, Alattin adds the condition check to the pattern candidate. For example, the extracted condition check for nodes CT_6 and NT_7 in the code example is "`boolean-check` on return of `Iterator.hasNext` before `Iterator.next`", which indicates that a `boolean-check` must be done on the return variable of the `hasNext` method before the call site of `Iterator.next`. In our experience, we found that there can be various code examples without any condition checks around an F_i method. Failing to consider these code examples can assign incorrect support values to mined patterns. To address this issue, we add an *Empty Pattern Candidate* to the input database ISD for each such code example.

C. Phase 3: Mining Alternative Patterns

In Phase 3, Alattin uses the ImMiner algorithm to mine both balanced and imbalanced patterns from pattern can-

Table I
SUBJECT APPLICATIONS AND THEIR CHARACTERISTICS.

Application	#Classes	#Methods	#Samples
Java Util APIs	19	144	49858
Java Transaction APIs	7	37	5555
Java SQL APIs	14	93	15052
BCEL	357	2691	9697
HsqlDB	143	1178	118610
Hibernate	478	4334	105549
Total	1018	8477	304321

didates. Alattin applies ImMiner on pattern candidates of each F_i method individually. The reason is that if we apply ImMiner on all pattern candidates together, the patterns related to an F_i method with a few pattern candidates can be missed due to patterns (related to other M_j methods) with a large number of pattern candidates.

We apply ImMiner on the pattern candidates of each F_i method in two steps. In Step 1 of mining, ImMiner computes frequent patterns by applying frequent itemset mining. We used a *min_sup* threshold value of 0.4. Consider that Step 1 resulted in a pattern "P_1 **or** P_2 **or** $\ldots$ **or** P_i". In Step 2, for each frequent alternative, ImMiner gathers the pattern candidates that do not support the alternative and mines infrequent alternative patterns. We use an *alt_sup* threshold value of 0.2. These two threshold values are based on our initial empirical experience presented in Section IV-E. At the end of Step 2, ImMiner generates patterns of the form "P_1 **or** P_2 **or** $\ldots$ **or** P_i **or** $\hat{A}_1$ **or** $\hat{A}_2$ **or** $\ldots$ **or** $\hat{A}_j$" for each F_i method. In this mined pattern, there are i frequent alternatives and j infrequent alternatives. In our approach, there is no restriction on the number of alternatives (either frequent or infrequent) within a pattern. However, each mined pattern includes at least one frequent alternative.

D. Phase 4: Detecting Neglected Conditions

In Phase 4, Alattin detects violations of mined patterns in an application under analysis statically. More specifically, Alattin gathers condition checks around each call site of an F_i method in the application under analysis. As each mined pattern contains several alternatives of using the F_i method, Alattin verifies whether a call site of the F_i method in the application under analysis satisfies at least one alternative of the mined pattern. If the gathered condition checks in the application under analysis do not satisfy any of the alternatives of the mined pattern, Alattin reports a violation. For each detected violation, Alattin assigns a support value as the same value as the support value of the associated mined pattern used to detect the violation.

IV. EVALUATIONS

We conducted two evaluations to assess the effectiveness of Alattin. Our empirical results show that there is a high percentage of *balanced* and *imbalanced* patterns in real applications. Our empirical results also show that Alattin effectively mines real rules from relevant code examples gathered through a CSE and is effective in detecting real defects. To show the significance of balanced and imbalanced patterns, we use the APIs provided by six open source libraries (three Java default API libraries and three popularly used open source libraries). For brevity, we refer to all subjects as applications. We also present our empirical evaluation for computing values for *min_sup* and *alt_sup* thresholds. The details of subjects and results of our evaluation are available at `https://sites.google.com/site/asergrp/projects/alattin/`. We next present research questions addressed in our evaluation.

A. Research Questions

In our evaluations, we address the following research questions.

- RQ1: How high percentage of balanced and imbalanced patterns exist in real applications?
- RQ2: How high percentage of false positives among detected violations are reduced by balanced and imbalanced patterns (while with no or low increase of false negatives)? As false positives are one of the common issues faced by existing static defect-detection techniques, this research question helps to show that more comprehensive patterns (such as balanced and imbalanced) help reduce the number of false positives.

B. Subject Applications

We next present subject applications used in our evaluations. In our evaluations, we used three Java default API libraries and three open source libraries. Table I shows the characteristics of the subject applications. Columns "Classes" and "Methods" show the number of classes and methods, respectively. Column "Samples" shows the number of code examples gathered from a CSE for mining patterns. For example, Alattin gathered and analyzed $49,858$ code examples for the Java Util package.

The Java Util package includes the collections framework and other popular utilities used by many different applications. Java Transactions and Java SQL are industry standards for developing multi-tier server-side Java applications. The BCEL library, developed by Apache, is mainly used to analyze, create, and manipulate Java class files. Hibernate and HsqlDB abstract relational databases into an object-oriented methodology. We selected these applications because these applications are used as subjects in evaluating previous related approaches [19], [20].

C. RQ1: Balanced and Imbalanced Patterns

We next address the first research question of whether balanced and imbalanced patterns often exist in real applications. To address this question, we configured Alattin, which by default accepts an application under analysis and mines patterns for third-party APIs, to accept a set of classes and methods directly. In this mode of operation, Alattin

Table II
PATTERNS MINED BY ALATTIN

Application	#Total Patterns	Categories of first 25 patterns			Subcategories of Real/Partial Rules			Time (in min.)
		#Real Rules	#Partial Rules	#False Positives	#Single	#Balanced	#Imbalanced	
Java Util APIs	40	21	2	2	8	1	14	9.75
Java Transaction APIs	3	2	1	0	0	0	3	1.67
Java SQL APIs	24	21	2	1	9	7	7	2.64
BCEL	64	19	2	4	16	3	2	12.02
HsqlDB	1	1	0	0	1	0	0	18.06
Hibernate	12	11	0	1	10	1	0	17.23
Total	144	75	7	8	44	12	26	61.37

mines patterns (programming rules) for the APIs of the given classes and methods. For each subject application, we analyzed the top 25 patterns mined by Alattin. The results of our evaluation are shown in Table II.

Column "Total Patterns" shows the total number of patterns mined for each subject application. Column "Categories of first 25 patterns" shows the manual classification of the top 25 patterns into three categories: real rules, partial rules, and false positives among mined patterns. *Real rules* describe properties that must be satisfied when using an API method. These real rules are of the form "P_1 **or** P_2 **or** ... **or** P_i **or** $\hat{A}_1$ **or** $\hat{A}_2$ **or** ... **or** $\hat{A}_j$", where all alternatives (both frequent and infrequent) represent real properties. In contrast to real rules, *partial rules* are of the form "P_1 **or** P_2 **or** ... **or** P_i **or** $\hat{A}_1$ **or** $\hat{A}_2$ **or** ... **or** $\hat{A}_j$", where all frequent alternatives such as P_1, P_2, ..., P_i represent real properties and some of the infrequent alternatives such as $\hat{A}_1$ and $\hat{A}_2$ do not represent real properties. The reason for introducing partial rules is that partial rules are as effective as real rules in reducing false-positive defects; however, partial rules can increase false-negative defects due to false-positive infrequent alternatives (among mined patterns) such as $\hat{A}_1$ and $\hat{A}_2$. We used available on-line documentations, JML specifications[2], or source code of applications for classifying mined patterns into these three categories.

Our results show that Alattin can mine a high percentage of patterns (ranging from 84% to 100%) that represent real or partial rules. These results show that Alattin can effectively mine real programming rules that can be used for tasks such as program comprehension and defect detection. To show that balanced and imbalanced patterns often exist, we further classified real and partial rules into three subcategories: balanced, imbalanced, and single patterns. The definitions of these subcategories are described in Section II-B. Column "Subcategories of Rules/Partial Rules" shows the results of our further classification. Figure 4 shows the distribution chart for balanced, imbalanced, and single patterns. In the distribution chart, x-axis shows subject applications and y-axis shows the percentages of Rules/Partial rules classified into three subcategories.

Our results show that imbalanced patterns range from 30% to 100% (on average 65%) among the Java default API

[2]`http://www.eecs.ucf.edu/ leavens/JML/`

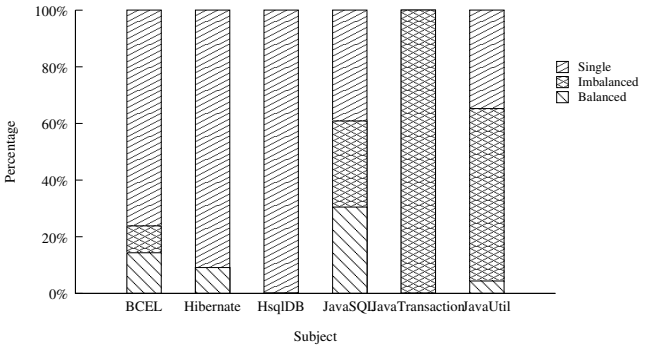

Figure 4. Classification of Real/Partial Rules into Balanced, Imbalanced, and Single patterns

libraries, whereas these patterns range from 0% to 9.5% (on average 5%) among the patterns of libraries BCEL, Hibernate, and HsqlDB. The difference in results between Java default libraries and other open source libraries with respect to imbalanced patterns could be that Java default API libraries provide different ways of writing code for achieving the same programming task, whereas other open source libraries such as BCEL or Hibernate often provide a single way for writing code. Furthermore, balanced patterns range from 0% to 30% (on average 9.69%) among all six subject applications. In summary, our results show that both a high percentage of balanced and imbalanced patterns exist among real applications.

We next present example balanced and imbalanced patterns mined by Alattin in Figure 5. For each pattern, we show the method name, all alternatives, and their support values (denoted by SUP for frequent alternatives and ABS for infrequent alternatives). We show the SUP value for frequent alternatives and both SUP and ABS values for infrequent alternatives. The balanced pattern shown in Figure 5A describes that there should be either a constant check such as greater than zero after the `read` method (P_1) or a `null-check` on the return of `getNextEntry` before the `read` method (P_2). Both alternative patterns are frequent as their support values are greater than the *min_sup* value. The imbalanced pattern shown in Figure 5B describes three different ways of using the `getString` method of the `java.sql.ResultSet` class. Among the alternatives of the pattern, P_1 is the only frequent alternative, whereas all the other alternatives are infrequent. The `ResultSet.getString` method throws `SQLException` if

***A*. Balanced pattern**

Method: `ZipInputStream.read (byte[], int, int)`

Pattern: "P_1 **or** P_2"

P_1:"`const-check` on the return of `ZipInputStream.read` with 0" (SUP(P_1): 0.79)

P_2:"`null-check` on the return of `ZipInputStream.getNextEntry` before `ZipInputStream.read`" (SUP(P_2): 0.55)

***B*.Imbalanced pattern**

Method: `ResultSet.getString (int)`

Pattern: "P_1 **or** $\hat{P}_2$ **or** $\hat{P}_3$"

P_1:"`boolean-check` on the return of `ResultSet.next` before `ResultSet.getString`" (SUP(P_1): 0.88)

P_2:"`boolean-check` on the return of `ResultSet.first` before `ResultSet.getString`" (ABS(P_2): 0.28, SUP(P_2): 0.01)

P_3:"`null-check` on the return of `ResultSet.getString`" (ABS(P_3): 0.4, SUP(P_3): 0.08)

Figure 5. Balanced and imbalanced patterns

the input column index is more than the number of columns in the database. The mined pattern presents different ways of ensuring that the index value is between zero and the number of columns. In this imbalanced pattern, the SUP value of P_2 alternative is 0.01, which is lower than min_sup. This pattern is chosen as infrequent alternative since its ABS value is 0.28, which is higher than the alt_sup value.

Column "Time" presents the amount of time taken (in minutes) by Alattin for analyzing gathered code examples and mining patterns. The amount of processing time depends on the number of code examples gathered for an application under analysis. For example, Alattin took 9.75 minutes for mining patterns from $49,858$ code examples gathered for the Java Util package. All experiments were conducted on a machine with 3.0GHz Xeon processor and 4GB RAM.

D. RQ2: False Positives and False Negatives

We next address the second research question of whether balanced and imbalanced patterns help reduce a high percentage of false positives among detected violations. We also address whether these patterns introduce *only* no or a low percentage of false negatives among detected violations. To address this question, we applied mined patterns on gathered code examples themselves in three different modes. In each mode, we compute a Programming Rules Set, say PRS, from the patterns of single, balanced, and imbalanced categories in different ways and apply each programming rule individually for detecting violations. We use notations "P_1", "P_1 **or** ... **or** P_i", "P_1 **or** ... **or** P_i **or** $\hat{A}_1$ **or** ... **or** $\hat{A}_j$" for single, balanced, and imbalanced patterns, respectively.

Existing Mode (EM). EM reflects the methodology adopted by existing approaches [5]–[7] for detecting violations. This mode serves as a baseline to show the benefits of balanced and imbalanced patterns. For a single-category pattern, we add to PRS the corresponding rule for each alternative P_1 of the mined pattern. For a balanced-category pattern, we first split the alternatives into individual patterns and add them to PRS. We add to PRS i corresponding rules for the balanced pattern "P_1 **or** ... **or** P_i" with i alternatives. For an imbalanced-category pattern, we discard infrequent alternatives and split frequent alternatives. Similar to the way of handling a balanced pattern, we add to PRS i corresponding rules for the imbalanced pattern "P_1 **or** ... **or** P_i **or** $\hat{A}_1$ **or** ... **or** $\hat{A}_j$" with i frequent alternatives.

Balanced Mode (BM). BM helps show the benefits of balanced patterns in reducing false positives among detected violations. For a single- or balanced-category pattern, we add to PRS one corresponding rule for each mined pattern. For an imbalanced-category pattern, we discard infrequent alternatives in the pattern. Discarding infrequent alternatives of an imbalanced pattern transforms the pattern into a balanced pattern. We add to PRS one corresponding rule for each such transformed pattern.

(Im)balanced Mode (IM). IM helps show the benefits of imbalanced patterns. We add to PRS one corresponding rule for each mined pattern of all categories.

Table III shows violations detected in each mode for all applications. Column "Total" shows the total number of violations. Given the large number of detected violations in subjects such as the Java Util package, we present the violations detected by the top 10 patterns (shown in Columns EM, BM, and IM). Our results show that there is a reduction in the number of false positives by 15.17% in BM and 28.01% in IM. In summary, the results show that balanced and imbalanced patterns help effectively reduce the number of false positives among detected violations. Our results also show that the number of false negatives introduced in Balanced and (Im)balanced modes is quite minimal. These false negatives occur due to the partial rules shown in Table II.

To further show the effectiveness of our approach, we computed a metric called *accuracy* [21] that is based on four different factors: true positives (TP), false positives (FP), false negatives (FN), and true negatives (TN). In each mode, we consider the number of detected defects and the number of false positives as TP and FP, respectively. As EM is our baseline, we assume the number of FN and TN as zero. For mode BM, we compute false negatives as `Number of defects in mode EM - Number of defects in mode BM`. These false negatives show the number of defects missed in mode BM compared to mode EM due to partial rules mined by our approach. Similarly, we compute true negatives as `Number of false positives in mode EM -`

Table III
VIOLATIONS DETECTED BY ALATTIN IN OPEN SOURCE APPLICATIONS

Application	Total	Existing Mode (EM)		Balanced Mode (BM)				(Im)balanced Mode (IM)			
		Defects	FP	Defects	FN	FP	% Reduction in FP from EM mode	Defects	FN	FP	% Reduction in FP from EM Mode
Java Util APIs	974	37	104	37	0	104	0	36	1	74	28.85
Java Transaction APIs	156	51	105	51	0	105	0	47	4	76	27.62
Java SQL APIs	315	56	143	56	0	90	37.06	53	3	81	43.36
BCEL	160	2	14	2	0	8	42.86	2	0	6	57.14
HsqlDB	1	1	0	1	0	0	0	1	0	0	0
Hibernate	22	10	9	10	0	8	11.11	10	0	8	11.11
Average							**15.17**				**28.01**

```
00:public describeProcedureCall() throws
                    SQLException {
01:   PreparedStatement ps = null; ...
02:   ResultSet rs = ps.executeQuery();
03:   if(!rs.first())  {
04:      ...
05:      return;
06:   }
07:   ...
08:   do {
09:      String datatype = rs.getString(2);
10:      ...
11:   } while(rs.next()); ...
12:}
```

Figure 6. A code example gathered from the Debian tools source code.

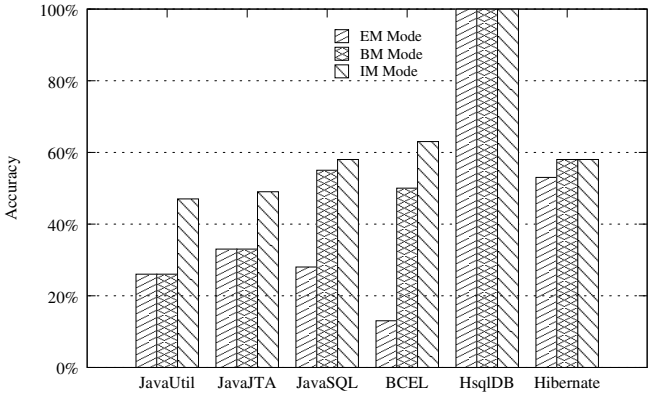

Figure 7. Accuracy metric values for Modes EM, BM, and IM.

`Number of false positives in mode BM`. These true negatives show the number of false positives reduced in mode BM compared to mode EM due to balanced and imbalanced patterns. We use the same way for computing these four factors for mode IM. We next compute the accuracy metric from these four factors using the formula shown below:

$$\text{Accuracy} = \frac{TP+TN}{TP+FP+FN+TN}$$

The rationale behind our accuracy metric is that higher accuracy metric values are assigned to those modes that reduce a higher number of both false positives and false negatives. Therefore, if our balanced and imbalanced patterns increase false negatives or do not decrease false positives, the corresponding accuracy metric values are also low. Figure 7 shows the computed accuracies for modes EM, BM, and IM. The figure shows that mode IM has higher accuracy metric values compared to modes EM and BM. Our results indicate that our balanced and imbalanced patterns help reduce false positives with *only* a minimal increase in false negatives.

We next present an example to show how imbalanced patterns help reduce the number of false positives. Figure 6 shows a code example of class `com.sap.dbtech.jdbc.Parseinfo` collected from the SVN repository of the Debian[3] tools. For simplicity, irrelevant code portions are omitted from the code example. The frequent alternative P_1 of the imbalanced pattern shown in Figure 5B describes that there should be a `boolean` check on the return of `ResultSet.next` before `ResultSet.getString`. With just the P_1 alternative, an existing mining approach detects a violation in this code example since the code example does not include a `boolean` check on the return of `ResultSet.next` preceding the first invocation of `ResultSet.getString`. However, the code example does not have a defect in using the `getString` method since the `if` condition in Statement 3 ensures that there is at least one element in `ResultSet` before reaching Statement 9. This code example illustrates the use of imbalanced patterns in reducing false positives among detected violations.

[3]`http://www.debian.org/`

E. Threshold Values

We conducted an empirical study with the Java Util package for computing values for *min_sup* and *alt_sup* thresholds. The *min_sup* threshold affects the number of real rules and false positives among mined patterns. For example, choosing a high value for the *min_sup* threshold leads to a low number of false positives but also leads to a low number of real rules among mined patterns. In contrast, the *alt_sup* threshold affects the number of imbalanced patterns among mined patterns. For example, choosing a low value for *alt_sup* increases the number of imbalanced patterns among mined patterns and also increases the number of partial rules that can result in false negatives among detected violations.

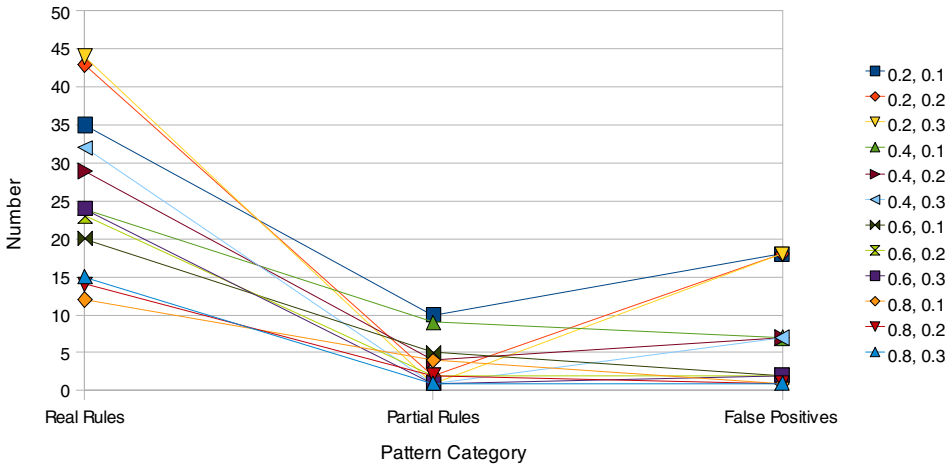

Figure 8. Distribution of rules, partial rules, and false positives.

To compute values for *min_sup* and *alt_sup* with reasonable tradeoffs, we applied our approach on the Java Util package with various values for these two thresholds. We classified mined patterns for each combination of *min_sup* and *alt_sup* values into real rules, partial rules, and false positives. We further classified the real and partial rules into balanced and imbalanced patterns. Figures 8 and 9 show the results of our evaluation. For our evaluations, we used *min_sup* as 0.4 and *alt_sup* as 0.2 since these two thresholds have reasonable tradeoffs.

V. Threats to Validity

The threats to external validity primarily include the degree to which the subject programs and used CSE are representative of true practice. The current subjects range from small-scale libraries such as Java SQL APIs to large-scale libraries such as BCEL and Hibernate. We used only one CSE, i.e., Google code search, which is a well-known CSE. These threats could be reduced by more experiments on wider types of subjects and by using other CSEs in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our Alattin prototype might cause such effects. There can be errors in our inspection of source code for confirming rules or defects. To reduce these threats, we inspected available specifications and also call sites in source code.

VI. Discussion

To mine imbalanced patterns, our ImMiner algorithm splits the input database into two groups for each frequent pattern mined in Phase 1. Although this solution is feasible for approaches (such as Alattin) that produce input databases with thousands of pattern candidates, our current solution might not be practical for large input databases with millions of pattern candidates. In future work, we plan to enhance our mining algorithm using support vector machines [22] so that we can mine imbalanced patterns without splitting the input database.

Our current implementation sometimes is not precise and cannot identify equivalent but syntactically different conditions. For example, our current implementation considers the conditions `a > 0` and `a ≥ 1` as different. In future work, we plan to address these issues using more precise static analysis that can identify equivalent conditions.

VII. Related Work

PR-Miner developed by Li and Zhou [6] uses frequent itemset mining to mine programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann [13] uses association rule mining to extract simple rules from software revision histories for Java code and detect defects related to rule violations. PR-Miner or DynaMine may suffer from issues of a high number of false positives since their rule elements are not necessarily associated with program dependencies. Furthermore, these approaches target at *only* frequent patterns, whereas Alattin can mine alternative patterns that include both frequent and infrequent alternatives.

Another related approach to our Alattin approach is the approach developed by Chang et al. [10] that applies frequent subgraph mining on C code to mine condition rules and to detect neglected conditions. Both Alattin and their approach target at the same type of defects: neglected conditions. Alattin significantly differs from Chang et al.'s approach in three main aspects. First, their approach cannot mine infrequent alternatives. Second, their approach is limited on a much smaller scale of code repositories (in fact, only one project code base) than Alattin, which exploits

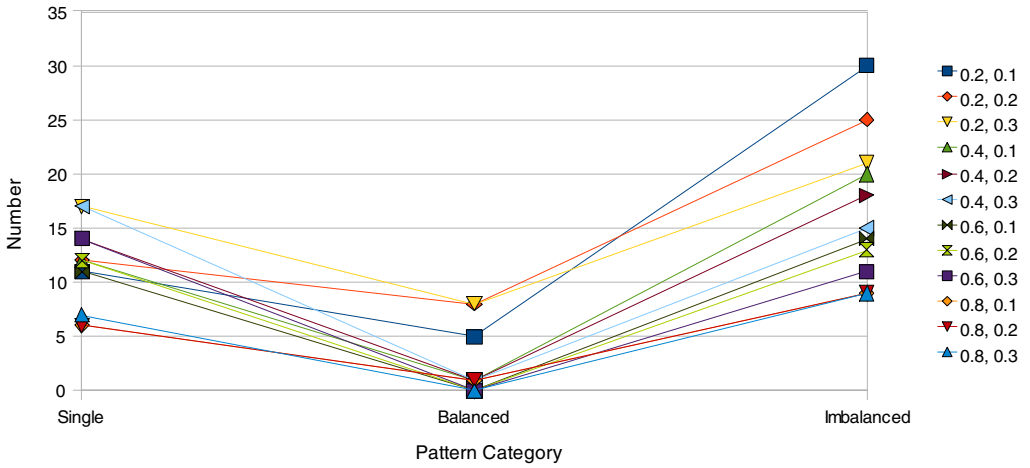

Figure 9. Distribution of single, balanced, and imbalanced patterns.

a CSE to search for relevant code examples from open source code available on the web. Third, the scalability of their approach is heavily limited by its underlying graph mining algorithms, which are known to suffer from scalability issues. In contrast, Alattin uses our new ImMiner algorithm based on frequent itemset mining, being much more scalable.

Williams and Hollingsworth [14] incorporate an API call return value checker for C code, which checks that a value returned by an API call is checked before being used. This type of return-value checking before use falls into a subset of the types of rules being mined by Alattin. Different from their tool, Alattin does not require or rely on version histories, which may not include the types of defect fixing (required by their tool) related to the rules being mined. Acharya et al. [7] developed a tool to mine interface details (such as an API call's return values on success or failure and error flags) from model-checker traces for C code, and then mine interface robustness properties for defect detection. Similar to the tool of Williams and Hollingsworth [14], Acharya et al.'s tool mines only a subset of neglected conditions (e.g., return-value checking before use) mined by Alattin. In addition, as shown by Acharya et al. [7], only the interface details of 22 out of 60 POSIX API functions can be successfully mined by their tool, whereas Alattin exploits a CSE to alleviate the issue by collecting relevant API call usages from the web. Furthermore, these approaches cannot mine alternative patterns targeted by Alattin.

Engler et al. [5] proposed a general approach for detecting defects in C code by applying statistical analysis to rank deviations from programmer beliefs inferred from source code. Their approach allows users to define rule templates, which are not required by our approach. In addition, their approach also cannot mine infrequent alternatives targeted by our Alattin approach.

Finally, our previous approaches PARSEWeb [23] and CAR-Miner [20] also exploit code search engines for gathering relevant code samples. PARSEWeb accepts queries of the form "*Source* → *Destination*" and mines frequent method-invocation sequences that accept *Source* and produce *Destination*. Although Alattin uses code search engines for gathering relevant code examples, Alattin targets at mining patterns that describe programming rules that should be obeyed while reusing APIs. Unlike PARSEWeb, which mines frequent sequences, Alattin mines alternative patterns with both frequent and infrequent alternatives. CAR-Miner also incorporates a new mining algorithm for mining exception-handling rules in the form of sequence association rules. CAR-Miner and Alattin differ significantly in three major aspects. (1) CAR-Miner mines rules for detecting exception-handling-related defects, whereas Alattin mines rules for detecting neglected conditions. (2) Alattin is a more general approach compared to CAR-Miner and can be applied to enhance various existing mining-based approaches including CAR-Miner for detecting alternative rules. (3) CAR-Miner mines new kinds of patterns for reducing false negatives (i.e., detecting new kinds of exception-handling defects). In contrast, Alattin mines new kinds of patterns for reducing false positives.

VIII. CONCLUSION

To reduce false positives in static defect detection based on code mining, we have developed a novel approach, called Alattin, that includes a new mining algorithm and a technique for detecting neglected conditions based on the mining algorithm. Our new mining algorithm mines

alternative patterns classified into two categories: balanced and imbalanced. In balanced patterns, all alternatives are frequent, whereas in imbalanced patterns, some alternatives are infrequent. We conduct two evaluations to show the effectiveness of our Alattin approach. Our evaluation results show that (1) alternative patterns reach more than 40% of all mined patterns for APIs provided by six open source libraries; (2) the mining of alternative patterns helps reduce nearly 28% of false positives among detected violations.

In this paper, we follow a problem-driven methodology in advancing the field of mining software engineering data. Our current approach and previous approach [20] serve as examples in this direction. More specifically, in our approaches, we empirically investigate problems in the software engineering domain and identify required types of patterns for addressing those problems. We further develop new mining algorithms for mining these required types of patterns, rather than being constrained by available mining algorithms from the data mining community. Our approaches primarily target at reducing false negatives and false positives among detected violations. Our previous approach [20], which mines programming rules as sequence association rules, focuses on reducing false negatives by detecting new kinds of defects. In contrast, our current approach focuses on a new sub-direction of reducing false positives among detected violations. In future work, we plan to further expand our research by investigating broader types of problems, patterns, mining algorithms, and defects.

ACKNOWLEDGMENTS

This work is supported in part by NSF grant CCF-0725190, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI).

REFERENCES

[1] T. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice." in *IEEE Software*, 2003, pp. 35–39.

[2] M. Ernst, J. Cockrell, W. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Trans. Softw. Eng.*, vol. 27, no. 2, pp. 99–123, 2001.

[3] G. Ammons, R. Bodik, and J. R. Larus, "Mining specifications," in *Proc. POPL*, 2002, pp. 4–16.

[4] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das, "Perracotta: mining temporal API rules from imperfect traces," in *Proc. ICSE*, 2006, pp. 282–291.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf, "Bugs as deviant behavior: a general approach to inferring errors in systems code," in *Proc. SOSP*, 2001, pp. 57–72.

[6] Z. Li and Y. Zhou, "PR-Miner: automatically extracting implicit programming rules and detecting violations in large software codes," in *Proc. FSE*, 2005, pp. 306–315.

[7] M. Acharya, T. Xie, and J. Xu, "Mining interface specifications for generating checkable robustness properties," in *Proc. ISSRE*, 2006, pp. 311–320.

[8] M. K. Ramanathan, A. Grama, and S. Jagannathan, "Path-sensitive inference of function precedence protocols," in *Proc. ICSE*, 2007, pp. 240–250.

[9] S. Shoham, E. Yahav, S. Fink, and M. Pistoia, "Static specification mining using automata-based abstractions," in *Proc. ISSTA*, 2007, pp. 174–184.

[10] R.-Y. Chang, A. Podgurski, and J. Yang, "Finding what's not there: a new approach to revealing neglected conditions in software," in *Proc. ISSTA*, 2007, pp. 163–173.

[11] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: from usage scenarios to specifications," in *Proc. ESEC/FSE*, 2007, pp. 25–34.

[12] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *Proc. ESEC/FSE*, 2007, pp. 35–44.

[13] V. B. Livshits and T. Zimmermann, "DynaMine: finding common error patterns by mining software revision histories," in *Proc. ESEC/FSE*, 2005, pp. 296–305.

[14] C. C. Williams and J. K. Hollingsworth, "Recovering system specific rules from software repositories," in *Proc. MSR*, 2005, pp. 1–5.

[15] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: A maximal frequent itemset algorithm for transactional databases," in *Proc. ICDE*, 2001, pp. 443–452.

[16] J. Wang and J. Han, "BIDE: efficient mining of frequent closed sequences," in *Proc. ICDE*, 2004, pp. 79–80.

[17] "Google Code Search Engine," 2006, http://www.google.com/codesearch.

[18] "The Koders source code search engine," 2005, http://www.koders.com.

[19] W. Weimer and G. Necula, "Mining temporal specifications for error detection," in *Proc. TACAS*, 2005, pp. 461–476.

[20] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proc. ICSE*, May 2009, pp. 496–506.

[21] P. Tan, M. Steinbach, and V. Kumar, *Introduction to data mining*. Addison-Wesley 1st edition, 2006.

[22] C. Nello and S.-T. John, *An introduction to support vector machines and other kernel-based learning methods*. Addison-Wesley 1st edition, 2000.

[23] S. Thummalapenta and T. Xie, "PARSEWeb: A programmer assistant for reusing open source code on the web," in *Proc. ASE*, 2007, pp. 204–213.