Autom Softw Eng
DOI 10.1007/s10515-011-0086-z

Alattin: mining alternative patterns for defect detection

Suresh Thummalapenta · Tao Xie

Received: 14 June 2010 / Accepted: 28 March 2011

Abstract To improve software quality, static or dynamic defect-detection 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 or dynamic defect-detection techniques to detect pattern violations in source code under analysis. However, 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 new mining algorithms and a technique for detecting neglected conditions based on our mining algorithm. Our new mining algorithms mine patterns in four pattern formats: conjunctive, disjunctive, exclusive-disjunctive, and combinations of these patterns. We show the benefits and limitations of these four pattern formats with respect to false positives and false negatives among detected violations by applying those patterns to the problem of detecting neglected conditions.

This work was primarily done when the first author is at North Carolina State University.
This paper is an extended version of our previous work published at ASE 2009 (Thummalapenta and Xie 2009). Our previous work introduced the concept of balanced and imbalanced patterns that are expressed in the *Or* pattern format. In this work, we propose additional new pattern formats *Xor* and *Combo*. We also propose new mining algorithms for mining patterns in *Or*, *Xor*, and *Combo* pattern formats. Furthermore, we show the benefits and limitations of *And*, *Or*, *Xor*, and *Combo* pattern formats by applying the patterns mined using these formats to the problem of detecting neglected conditions in applications under analysis.

S. Thummalapenta (✉) · T. Xie
Department of Computer Science, North Carolina State University, Raleigh, NC, USA
e-mail: sthumma@ncsu.edu

T. Xie
e-mail: xie@csc.ncsu.edu

S. Thummalapenta
IBM Research, Bangalore, Karnataka, India

Published online: 08 April 2011

Keywords Alternative patterns · Static defect detection · Mining software engineering data · Code search engine

1 Introduction

Programming rules serve as a basis for applying static or dynamic defect-detection 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 (Lethbridge et al. 2003). 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 (Ernst et al. 2001; Ammons et al. 2002; Yang et al. 2006), individual versions (Engler et al. 2001; Li and Zhou 2005; Acharya et al. 2006; Ramanathan et al. 2007; Shoham et al. 2007; Chang et al. 2007; Acharya et al. 2007; Wasylkowski et al. 2007), or version histories (Livshits and Zimmermann 2005; Williams and Hollingsworth 2005) of program source code. A methodology, commonly referred to as mining software repositories (MSR) (Bibliography on mining software engineering data 2010), 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 or dynamic defect-detection techniques that accept mined patterns as inputs and detect pattern violations as potential defects in applications under analysis.

Since the inception of the MSR methodology, the major focus is to mine various types of patterns and use those patterns for detecting defects in applications under analysis. Although MSR has been shown to be effective in detecting defects in applications under analysis, a major drawback of MSR is that the violations detected by existing MSR-based approaches often include a large number of false positives. This phenomenon is reflected in the empirical evaluations of existing approaches (Engler et al. 2001; Li and Zhou 2005; Acharya et al. 2006; Chang et al. 2007; Wasylkowski et al. 2007) where majority of detected violations turn out to be false positives.

To illustrate how detected violations turn out to be false positives, we use two code examples (shown in Fig. 1) using the `next` method of the `Iterator` class. The `next` method throws `NoSuchElementException` when invoked on an `ArrayList` object without any elements. Programmers can avoid this exception by using either the condition check "P_1: `boolean-check` on `return` of `Iterator.hasNext` before `Iterator.next`" (shown in `printEntries1` from Example 1) or "P_2: `const-check` on `return` of `ArrayList.size` before `Iterator.next`" (shown in `printEntries2` from Example 2). In general, programmers use either P_1 or P_2 but not both, since using both P_1 and P_2 is redundant. Consider that a single pattern P_1 is mined from the data points. A static defect-detection

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();... }}
```

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

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 turns out to be a false positive.

In our empirical investigation of false positives generated by existing approaches (Engler et al. 2001; Li and Zhou 2005; Acharya et al. 2006; Chang et al. 2007; Wasylkowski et al. 2007), we identify that a major reason for such a large number of false positives is that the focus of MSR is to mine *single* patterns (such as P_1) or *conjunctive* patterns (such as $P_1 \wedge P_2$) (more details are presented in Sect. 5). The conjunctive pattern $P_1 \wedge P_2$ describes that both P_1 and P_2 often appear together among the data points (e.g., code examples). We identify that these single or conjunctive patterns alone cannot describe the *nearly complete* behavior among data points, resulting in false positives. The reason why single or conjunctive patterns are not sufficient is that programmers write source code in different ways to achieve the same programming task (as shown in Examples 1 and 2). For example, the pattern $P_1 \oplus P_2$[1] describes both the condition checks that can be used before the `next` method. Furthermore, using the pattern $P_1 \oplus P_2$ does not result in violations in `printEntries1` and `printEntries2`, thereby reducing false positives. We focus on mining patterns that describe nearly complete rather than complete behavior among code bases, since the patterns that describe complete behavior cannot help detect violations as deviations from those patterns, resulting in *false negatives*. For example, consider an API method such as the `next` method of the `Iterator` class. Consider that the code bases include four condition checks (P_1, P_2, P_3, and P_4) for the API method, where among these four condition checks, three condition checks P_1, P_2, and P_3 represent real properties, whereas P_4 is a false-positive property. A pattern that describes complete behavior comprising all condition checks is

[1]The symbol $\oplus$ represents the exclusive-or relationship.

"$P_1 \vee P_2 \vee P_3 \vee P_4$". However, the preceding pattern does not help detect violations in code examples that include only P_4, resulting in false negatives.

To reduce both false positives and false negatives among detected violations and to infer patterns that describe nearly complete behavior, in this paper, we propose a novel approach, called ***Alattin***, that includes new mining algorithms and a technique that detects neglected conditions (described next) using patterns mined by our mining algorithms. In particular, our algorithms mine patterns in four pattern formats: conjunctive (*And* or $\wedge$), disjunctive (*Or* or $\vee$), exclusive-disjunctive (*Xor* or $\oplus$), and combinations of these patterns (referred to as *Combo* patterns). We use *Alternative Patterns* to collectively refer to patterns of all four formats and refer to individual patterns such as P_1 and P_2 in $P_1 \wedge P_2$ as alternatives.

In general, mining *Or* and *Xor* patterns is more challenging than mining *And* patterns, since *Or* and *Xor* patterns do not follow the Apriori principle (Han and Kamber 2000) in data mining. Given an input database of itemsets for applying mining techniques, the Apriori principle states that if an itemset is frequent, then all its subsets should also be frequent. Existing mining techniques (Burdick et al. 2001) that target at mining *And* patterns use this principle for pruning the search space. For example, if an itemset P_1 is not frequent, then any super *And* itemset of P_1 such as $P_1 \wedge P_2$ cannot be frequent, and hence can be pruned. However, the Apriori principle does not hold for mining *Or* and *Xor* patterns. For example, although the itemset P_1 is not frequent, its super *Or* itemset such as $P_1 \vee P_2$ can be frequent, since $P_1 \vee P_2$ is supported by more itemsets in the input database compared to P_1 or P_2 individually.

In this paper, we show the benefits and limitations of these four pattern formats with respect to false positives and false negatives by applying these pattern formats to 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 (Chang et al. 2007). As shown by a recent study (Chang et al. 2007), 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. In our approach, we mine patterns that describe necessary condition checks related to an API call in these four formats and use those condition checks for detecting neglected conditions in applications under analysis.

This paper makes the following main contributions:

- An empirical investigation of four pattern formats: conjunctive (*And* or $\wedge$), disjunctive (*Or* or $\vee$), exclusive-disjunctive (*Xor* or $\oplus$), and combinations of these patterns (referred to as *combo* patterns) in software engineering data.
- New mining algorithms for efficiently mining patterns in *Or*, *Xor*, and *Combo* pattern formats.
- A technique that applies patterns of these four pattern formats for detecting neglected conditions around individual API calls in an application under analysis.
- Two evaluations to demonstrate the effectiveness of our approach. Our evaluation results show that the best pattern-mining approach (in terms of reducing both false

positives and false negatives) is to first mine *And* patterns for API methods and next mine *Combo* patterns. Among violations detected by *Combo* patterns, the best violation-detection approach is to assign higher priority to the violations of API methods with *And* patterns compared to the violations of API methods without *And* patterns. The primary reason is that the former violations are more likely to be real defects compared to the latter violations.

2 Example

We next use an illustrative example to describe our approach on how we collect the data that describes necessary condition checks around API calls. We also show how our proposed four pattern formats affect the number of false negatives and false positives among detected violations. Consider that an application under analysis uses the `Iterator.next` method as shown in the `printEntries2` method in Fig. 1.

Initially, we gather relevant code examples that invoke the `Iterator.next` method by constructing queries to Google code search (Google code search engine 2006). These relevant code examples are required for mining patterns that describe necessary condition checks around the `Iterator.next` method. Figure 2 shows a code example gathered from Google code search. We next construct control-flow graphs (CFG) for collected code examples and perform two traversals (backward and forward) of the CFG from the node corresponding to the `Iterator.next` method. In the backward traversal, we collect the condition checks on the receiver and argument objects preceding the call site of the `Iterator.next` method. Similarly, in the forward traversal, we collect the condition checks on the receiver and return objects succeeding the call site of the `Iterator.next` method. For the current code example, our backward and forward traversals gather the following condition checks.

```
1: boolean-check on the return of Iterator.hasNext
       before Iterator.next
2: boolean-check on the return of Collection.isEmpty
       before Iterator.next
3: instance-check on the return of Iterator.next
       with org.w3c.dom.Node
4: boolean-check on the return of Iterator.hasNext
       after Iterator.next
```

Condition check "1" describes the condition check performed *before* the call site of the `next` method, whereas Condition check "4" describes the condition check performed *after* the call site of the `next` method. The reason for two condition checks is that the `next` method (Statement 7) is invoked in a `for` loop. Section 4 presents more details on how we exploit program dependencies while performing backward and forward traversals. The preceding set of condition checks collected from the code example forms an itemset in the itemset database *ISD*, used as input for mining patterns. We analyze all gathered code examples to generate various itemsets and use different mining algorithms for mining the patterns in formats: *And*, *Or*, *Xor*, and *Combo*. Section 3 presents more details on our mining algorithms.

```
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:}
```

Fig. 2 A code example using `Iterator.next` gathered from Google code search

IS1:	1 3 4
IS2:	4 1
IS3:	2 3
IS4:	1
IS5:	2
IS6:	4 1

1 : boolean check on return of Iterator.hasNext before Iterator.next
2 : const check on return of ArrayList.size before Iterator.next
3 : null check on argument of ArrayList.constructor after Iterator.next
4 : boolean check on return of Iterator.hasNext after Iterator.next

Input Database (ISD)

Fig. 3 An example input database *ISD*

Figure 3 shows a sample itemset database *ISD* with six itemsets. This *ISD* includes four distinct items labeled with IDs 1 through 4. The figure also shows the condition check corresponding to each item. We next apply our mining algorithms for mining different pattern formats. The patterns mined by our algorithms with a minimum support threshold value *min_sup* of 0.4 are shown below:

- *And Pattern:* "$1 \wedge 4$", support: 0.5
- *Or Pattern:* "$1 \vee 2 \vee 3 \vee 4$", support: 1.0
- *Xor Pattern:* "$1 \oplus 2$", support 1.0; "4", support: 0.5
- *Combo Pattern:* "$(1 \wedge 4) \oplus 2$", support: 0.83

Among the itemsets shown in ISD, Items 1 and 4 often appear together with an $\wedge$ relation, since the methods `hasNext` and `next` are often used in a loop (as shown in Statements 6 and 7 in Fig. 2). Although the *And* pattern captures this behavior, the *And* pattern cannot capture the relation with Item 2, resulting in false positives when applied on code examples such as `printEntries2` in Fig. 1. On the other hand, the *Or* pattern does not result in false positives, since the pattern includes all items. However, the *Or* pattern does not help in detecting violations, resulting in false negatives. Although the *Xor* pattern can perform better than the *Or* pattern, the *Xor* pattern may not detect violations in code examples, which include only Item 1 and do not include Item 4. As shown in this example, the *Combo* pattern describes the nearly complete behavior and also helps reduce both false positives and false negatives.

Along with the challenges faced while mining *Or* and *Xor* patterns, *Combo* patterns pose additional challenges in choosing the suitable operator while combining items. For example, the suitable operator for combining items "1" and "4" is $\wedge$, although "1 $\vee$ 4" results in a higher support value than "1 $\wedge$ 4". We describe these challenges in subsequent sections and present our algorithms for mining these patterns.

In summary, this example illustrates the existence of pattern formats *And*, *Or*, and *Xor*, and also shows that no single pattern format alone can help in describing the necessary condition checks around API calls.

3 Mining algorithms for alternative patterns

We next describe four pattern formats that we use in our empirical study. We first present formal definitions for the four pattern formats and next describe our algorithms for mining the pattern in four formats with illustrative examples.

3.1 Formal definitions

Let $M = \{m_1, m_2, \ldots, m_k\}$ be the set of all possible distinct *items*. For example, an m_j represents a condition check such as "`boolean-check` on return of `Iterator.hasNext` before `Iterator.next`". Consider an ItemSet Database *ISD* as $\{is_1, is_2, \ldots, is_l\}$, where each itemset is_j includes different sets of elements such as $\{m_1, m_2, \ldots, m_a\}$ from the set of all possible distinct elements.

Definition 1 (Pattern candidate) A pattern candidate pc is a single item $m_k \in M$ or a combination of two elements associated by a logical operator $op \in \{\wedge, \vee, \oplus\}$. Each element in the combination is an item $m_i \in M$ or another pattern candidate.

The preceding definition is a recursive definition, which defines that a pattern candidate can be either a simple or nested pattern candidate. For example, a simple pattern candidate $pc_k : m_i \wedge m_j$ is a combination of two items m_i and m_j with the operator $op \in \{\wedge\}$. On the other hand, a nested pattern candidate $pc_l : m_i \vee pc_k$ is a combination of an item m_i and the preceding pattern candidate pc_k with the operator $op \in \{\vee\}$. We use notations $pc_k.left$ and $pc_k.right$ to refer to the left and right *child* pattern candidates, respectively, and refer to pc_k as their *parent* pattern candidate. Furthermore, we use the notation $pc_k.op$ to refer to the operator op of the pattern candidate pc_k. We classify a pattern candidate by its format, especially using its operator.

Definition 2 (*And* pattern candidate) An *And* pattern candidate is a pattern candidate where the operator $op \in \{\wedge\}$ and all the child pattern candidates are also *And* pattern candidates.

Definition 3 (*Or* pattern candidate) An *Or* pattern candidate is a pattern candidate where the operator $op \in \{\vee\}$ and all the child pattern candidates are also *Or* pattern candidates.

Algorithm 1 IsSupportedBy (pc_k, is_j)

Require: PatternCandidate pc_k, ItemSet is_j
Ensure: `true`, if is_j supports pc_k
Ensure: `false`, if is_j does not support pc_k
1: **if** pc_k is a SingleItem **then**
2: **return** $pc_k \in is_j$
3: **else**
4: bool lefts = isSupportedBy($pc_k.left$, is_j)
5: bool rights = isSupportedBy($pc_k.right$, is_j)
6: **if** $pc_k.op == \vee$ **then**
7: **return** lefts $\vee$ rights
8: **end if**
9: **if** $pc_k.op == \wedge$ **then**
10: **return** lefts $\wedge$ rights
11: **end if**
12: **if** $pc_k.op == \oplus$ **then**
13: **return** lefts $\oplus$ rights
14: **end if**
15: **end if**

Definition 4 (*Xor* pattern candidate) A *Xor* pattern candidate is a pattern candidate where the operator $op \in \{\oplus\}$ and all the child pattern candidates are also *Xor* pattern candidates.

Definition 5 (*Combo* pattern candidate) A *Combo* pattern candidate is a pattern candidate where the operator $op \in \{\wedge, \vee, \oplus\}$.

The category of *Combo* pattern candidates subsumes the categories of *And*, *Or*, and *Xor* pattern candidates. An example combo pattern candidate is "$pc_1 \oplus pc_2$", where pc_1 and pc_2 are "$m_i \wedge m_j$" and "$m_k \wedge m_l$", respectively.

To compute frequent patterns among pattern candidates, we use a threshold value, referred to as *min_sup*, that describes the minimum support for a pattern candidate to be classified as a frequent pattern. Algorithm 1, `IsSupportedBy`, describes how we compute support values for the preceding pattern formats. In particular, `IsSupportedBy` accepts a pattern candidate pc_k and an itemset is_j, and returns `true`, if is_j supports pc_k, and otherwise returns `false`. Initially, `IsSupportedBy` checks whether pc_k is a single item and returns `true` or `false` based on whether pc_k is contained in is_j. Otherwise, `IsSupportedBy` recursively computes whether $pc_k.left$ and $pc_k.right$ of pc_k are supported by the itemset is_j, and uses the operator $pc_k.op$ for checking whether is_j supports pc_k (Lines 4–15). We compute the support value of a pattern candidate pc_k, referred to as `Support(`pc_k`)`, based on the number of the itemsets (in *ISD*) that return `true` for the algorithm `IsSupportedBy`.

Definition 6 (Frequent pattern (FP)) A pattern candidate pc_k is considered as a frequent pattern, if `Support(`pc_k`)` $\geq$ *min_sup*.

A frequent pattern fp_k is considered as an *And*, *Or*, *Xor*, or *Combo* pattern based on the operator $fp_k.op$.

3.2 Mining algorithms

We next present our algorithms for mining preceding pattern formats. In particular, we present algorithms for mining *Or*, *Xor*, and *Combo* patterns, since mining *And* patterns can be achieved by well-known approaches such as a frequent itemset miner (Burdick et al. 2001). We first explain our algorithm for mining *Or* and *Xor* patterns, and next describe the algorithm for mining *Combo* patterns.

3.2.1 Mining Or and Xor patterns

We next describe our greedy algorithm for mining *Or* and *Xor* patterns. Our algorithm is based on the following property for pruning the search space of patterns. This property is inspired by Nanavati et al. (2001) and is applicable to both *Or* and *Xor* patterns.

Property 1 The support of an *Or* or *Xor* pattern candidate pc_k, represented as `Support(`pc_k`)`, formed from two pattern candidates $pc_k.left$ and $pc_k.right$ should have higher value than `Support(`$pc_k.left$`)` and `Support(`$pc_k.right$`)`.

The rationale behind the preceding property is based on our objective to mine patterns that describe nearly complete behavior. For example, a pattern candidate $pc_k = pc_i \vee pc_j$, whose support value is less than `Support(`pc_i`)` and `Support(`pc_j`)` is not useful compared to individual pattern candidates in achieving our objective. Algorithm 2, `MineXorOr`, describes our greedy algorithm for mining *Or* and *Xor* patterns. `MineXorOr` accepts itemset database *ISD*, *min_sup*, and *ptype* $\in \{\vee, \oplus\}$ as inputs.

Initially, `MineXorOr` identifies all distinct items in the itemset database ISD using the function `ComputeDistinct` (Line 2). Among these distinct items, `MineXorOr` checks whether the support of any of these items is greater than *min_sup* and adds those items to *PCSet* (Lines 3 to 7). Next, `MineXorOr` uses various iterations, where pairwise combinations are computed using the function `ComputePairwise` from the elements of the previous iteration stored in *CurrSet* (Lines 9 to 31). For example, consider *Or* patterns. For the `CurrSet` $= \{pc_1, pc_2, pc_3\}$, `ComputePairwise` returns a set with three elements, i.e., `PWSet` $= \{pc_1 \vee pc_2, pc_1 \vee pc_3, pc_2 \vee pc_3\}$. The algorithm next identifies the elements in `PWSet`, whose support values are greater than *min_sup* and satisfy Property 1. `MineXorOr` next chooses pattern candidates (from *NextSet*) that participate in the next iteration by greedily choosing one parent pattern candidate in *NextSet* with the highest support value for each pattern candidate in *CurrSet* using the function `ApplyGreedy` (Line 24). The rationale behind our greedy approach is based on our observation that the real patterns describing necessary condition checks around API calls often have higher support values compared to other patterns. Finally, `MineXorOr` identifies those pattern candidates in *CurrSet* whose parent pattern

Algorithm 2 MineXorOr(*ISD*, *min_sup*, *ptype*)

Require: *ISD*, *min_sup*, *ptype*
Ensure: Set<PC> *PCSet*

```
1:  PCSet = φ
2:  Set<M> distinctItems = ComputeDistinct(ISD)
3:  for all m_i ∈ distinctItems do
4:    if Support(m_i, ISD) >= min_sup then
5:      PCSet+ = m_i
6:    end if
7:  end for
8:  Set<PC> CurrSet = distinctItems
9:  loop
10:   Set<PC> NextSet = φ
11:   Set<PC> ChildSet = φ
12:   Set<PC> PWSet = ComputePairwise(CurrSet, ptype)
13:   for all pc_j ∈ PWSet do
14:     sval = Support(pc_j, ptype)
15:     if (sval < min_sup || sval ≤ Support(pc_j.left) || sval ≤ Support(pc_j.right))
        then
16:       Continue
17:     end if
18:     NextSet+ = pc_j
19:     PCSet+ = pc_j
20:   end for
21:   if NextSet.size() ≤ 1 then
22:     break
23:   end if
24:   NextSet = ApplyGreedy(CurrSet, NextSet)
25:   for all pc_j ∈ CurrSet do
26:     if pc_j ∉ {pc_k.left, pc_k.right}, ∀pc_k ∈ NextSet then
27:       NextSet+ = pc_j
28:     end if
29:   end for
30:   CurrSet = NextSet
31: end loop
32: return PCSet
```

candidates do not belong to *NextSet* (Lines 25–29). `MineXorOr` adds such pattern candidates to *NextSet* as well, since these pattern candidates can still be helpful when combined with other pattern candidates in *NextSet*.

We next explain the algorithm in detail using the itemset database shown in Fig. 4 (the same as Fig. 3). The itemset database includes four distinct items. Figure 5 shows all possible pattern candidates that can be derived using the preceding four distinct items. This figure shows the complete search space of pattern candidates. We first explain mining *Xor* patterns and next explain mining *Or* patterns.

IS1:	1	3	4
IS2:	4	1	
IS3:	2	3	
IS4:	1		
IS5:	2		
IS6:	4	1	

1 : boolean check on return of Iterator.hasNext before Iterator.next
2 : const check on return of ArrayList.size before Iterator.next
3 : null check on argument of ArrayList.constructor after Iterator.next
4 : boolean check on return of Iterator.hasNext after Iterator.next

Input Database (ISD)

Fig. 4 An example input database *ISD*

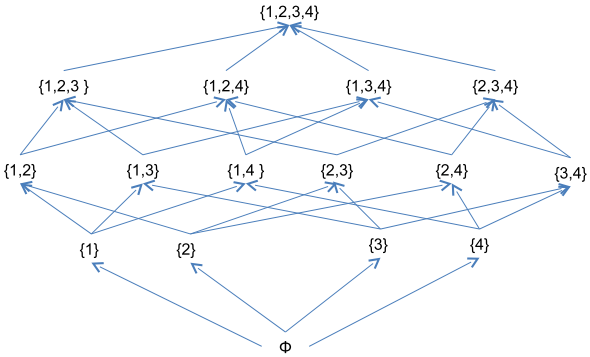

Fig. 5 All possible itemsets with four distinct items

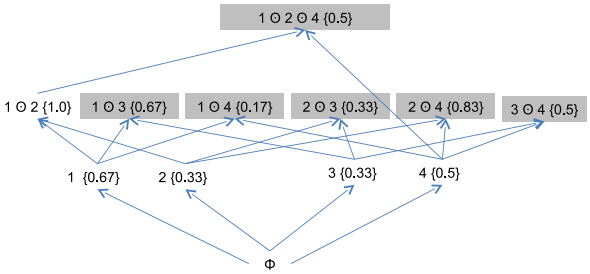

Fig. 6 Mining *Xor* patterns

Mining Xor patterns Figure 6 shows how `MineXorOr` prunes the search space and generates patterns "$1 \oplus 2$" (support 1.0) and "4" (support: 0.5). The value shown in braces next to each pattern candidate indicates support value of that pattern candidate. The pattern candidates shown in gray are pruned by `MineXorOr` due to three factors. First, the support value of pattern candidate is lower than *min_sup*. For example, $\texttt{Support}(2 \oplus 3) = 0.33$, which is lower than *min_sup*. Second, the pattern candidate does not satisfy Property 1. For example, the pattern candidate "$1 \oplus 3$" does not satisfy Property 1, since $\texttt{Support}(1 \oplus 3) \leq \texttt{Support}(1)$. Third, the pattern candidate is not the candidate with the highest support value among the parent pattern candidates of each child pattern candidate. For example, the pattern candidate "$2 \oplus 4$" is pruned, since the pattern candidate "$1 \oplus 2$" has a higher support value than "$2 \oplus 4$". Since all parent pattern candidates of "4" are pruned away, `MineXorOr` adds this candidate to *NextSet* for the next iteration and computes further pattern candidates such as "$1 \oplus 2 \oplus 4$".

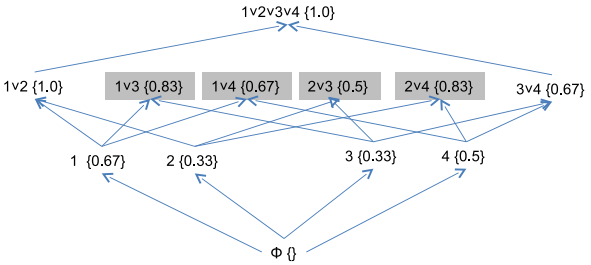

Fig. 7 Mining *Or* patterns

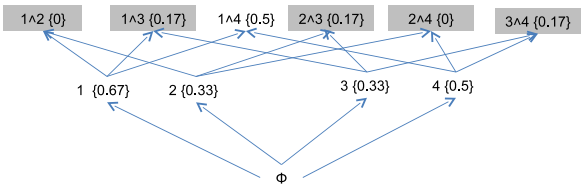

Fig. 8 Phase 1 of mining *Combo* patterns

Mining Or patterns Figure 7 shows how `MineXorOr` prunes the search space and generates the *Or* pattern "$1 \vee 2 \vee 3 \vee 4$" (support: 1.0). Similar to *Xor* patterns, the pattern candidates are pruned due to the same three factors.

3.2.2 Mining Combo patterns

We next describe how we mine combo patterns. The algorithm for mining *Combo* patterns includes two phases. Phase 1 mines *And* Patterns and Phase 2 mines *Combo* patterns using the output of Phase 1. We next explain each phase in detail.

Phase 1 Algorithm 3, `MineComboP1`, shows Phase 1 of mining *Combo* patterns. In particular, `MineComboP1` computes pairwise *And* combinations of all pattern candidates in *CurrSet* and checks whether new pattern candidates have higher support values than *min_sup* (Lines 9–11). If yes, `MineComboP1` computes the support of *Xor* combination of the two candidates, shown as $\texttt{Support}(pc_j.left \oplus pc_j.right)$ in Line 12. If the preceding support value is also higher than *min_sup*, then `MineComboP1` ignores the *And* combination. The rationale behind this decision is that, if both "$pc_j.left \wedge pc_j.right$" and "$pc_j.left \oplus pc_j.right$" have higher values than *min_sup*, then the suitable combination of $pc_j.left$ and $pc_j.right$ is "$pc_j.left \vee pc_j.right$".

Figure 8 shows the output of `MineComboP1` with the itemset database ISD shown in Fig. 3. As shown in the figure, Phase 1 produces three pattern candidates as output: "$1 \wedge 4$", "2", and "3", which are passed as inputs to Phase 2.

Phase 2 Phase 2 of mining *Combo* patterns is similar to Algorithm 2 for mining *Xor* and *Or* patterns. The major difference is to choose a suitable operator, $op \in \{\vee, \oplus\}$, when combining two pattern candidates during the computation of pairwise combinations. Given two pattern candidates pc_i and pc_j, Phase 2 chooses the $\vee$ operator if "$\texttt{Support}(pc_i \wedge pc_j) \geq min\_sup$ && $\texttt{Support}(pc_i \oplus pc_j) \geq min\_sup$"; otherwise, Phase 2 chooses the $\oplus$ operator. The rationale behind this decision is the same

Algorithm 3 MineComboP1(*ISD*,*min_sup*)

Require: *ISD*, *min_sup*, *ptype*
Ensure: Set<PC> *PCSet*
1: Set<M> *distinctItems* = ComputeDistinct(*ISD*)
2: Set<PC> *CurrSet* = *distinctItems*
3: *PCSet* = *distinctItems*
4: **loop**
5: Set<PC> *NextSet* = ϕ
6: Set<PC> *PWSet* = ComputePairwise(*CurrSet*, "And")
7: **for all** $pc_j \in PWSet$ **do**
8: $sval = Support(pc_j, ptype)$
9: **if** $sval < min\_sup$ **then**
10: **Continue**
11: **end if**
12: **if** $Support(pc_j.left \oplus pc_j.right) \geq min\_sup$ **then**
13: **Continue**
14: **end if**
15: $NextSet += pc_j$
16: $PCSet -= pc_j.left$
17: $PCSet -= pc_j.right$
18: $PCSet += pc_j$
19: **end for**
20: **if** $NextSet.size() \leq 1$ **then**
21: break
22: **end if**
23: $CurrSet = NextSet$
24: **end loop**
25: **return** *PCSet*

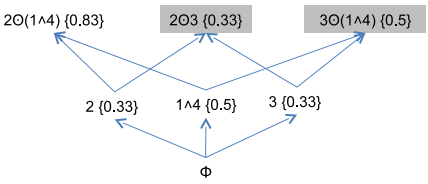

Fig. 9 Phase 2 of mining *Combo* patterns

as the reason given in Phase 1. Figure 9 shows the output of Phase 2 resulting in the pattern "$(1 \wedge 4) \oplus 2$" (support: 0.83).

4 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 our mining

algorithms 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 or application under analysis to generate pattern candidates suitable for mining. In Phase 3, Alattin applies mining algorithms 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.

4.1 Phase 1: gathering code examples

In Phase 1, Alattin gathers code examples from existing open source repositories through code search engines (CSE) such as Google code search (GCSE) (Google code search engine 2006) and Koders (The Koders source code search engine 2005). In general, these code search engines are used by programmers in searching for relevant code examples from available open source projects on the web. Since 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. Therefore, our approach uses a CSE to gather relevant code examples and mines gathered code examples to detect violations in an application under analysis. The primary reason for gathering code examples from a CSE is that gathering code examples from a small number of project code bases often cannot surface out many programming rules as common patterns. The 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 (Li and Zhou 2005; Chang et al. 2007): often a relatively small number of real programming rules mined from one or a few huge code bases.

In particular, Alattin gathers code examples by constructing queries for each method under analysis F_i. For example, to gather code examples from GCSE for the `next` method of the `Iterator` class, Alattin constructs the query of the form "`lang:java Iterator next`". More specifically, our queries include the names of the class and method along with the language type. These gathered code examples include information of how the `next` method is used by open source code available on the web. Alattin stores gathered code examples in the local file system for further analysis. Alattin uses GCSE for gathering 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.

4.2 Phase 2: generating pattern candidates

In Phase 2, Alattin analyzes gathered code examples or application under analysis 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 Fig. 2 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 (Fig. 2) 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 $\{V_1, V_2, \ldots, V_n\}$, 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 (Li and Zhou 2005) and DynaMine (Livshits and Zimmermann 2005) 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 $\{V_1, V_2, \ldots, V_n\}$, and the receiver or argument variables of the NT_k node, say {`receiver`, `argument1`, ..., `argumentN`}. If the intersection $\{V_1, V_2, \ldots, V_n\} \cap$ {`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.

4.3 Phase 3: mining alternative patterns

In Phase 3, Alattin uses our mining algorithms (described in Sect. 3) to mine patterns in all four pattern formats: *And*, *Or*, *Xor*, and *Combo* patterns. Alattin applies our mining algorithms on pattern candidates of each F_i method individually. The reason is that if we apply mining algorithms 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 F_j methods) with a large number of pattern candidates. For each F_i, Alattin mines patterns in all four pattern formats. We used a *min_sup* threshold value of 0.4 based on our empirical experience (Thummalapenta and Xie 2009).

4.4 Phase 4: detecting neglected conditions

In Phase 4, Alattin detects violations of mined patterns in the application under analysis statically. More specifically, Alattin gathers condition checks around each call site of an F_i method in the application under analysis. Alattin constructs an itemset is_j using gathered condition checks. For each mined pattern pc_k in all patterns of four formats, Alattin uses `IsSupportedBy` (Algorithm 1) to check whether the itemset is_j supports the mined pattern pc_k. If the itemset does not support 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.

5 Evaluations

We conducted two evaluations to assess the effectiveness of Alattin. We use the APIs provided by three Java default API libraries to show the existence of alternative patterns. We next use four popular applications to show the benefits and limitations of alternative patterns with respect to false positives and false negatives among detected violations. The details of subjects and results of our evaluations are available at https://sites.google.com/site/asergrp/projects/alattin/. We next present research questions addressed in our evaluations.

5.1 Research questions

In our evaluations, we address the following research questions.

RQ1: How high percentage of *And*, *Or*, *Xor*, and *Combo* patterns represent real programming rules, respectively? Since real programming rules are required for detecting violations in applications under analysis, this research question helps to show the pattern formats that are suitable for detecting violations.

RQ2: How low percentage of false negatives and false positives exist among violations detected using *And*, *Or*, *Xor*, and *Combo* patterns, respectively? Since false positives are one of the common issues faced by existing static defect-detection techniques, this research question helps to show that the patterns that describe nearly complete behavior (such as *Or* or *Combo*) help reduce the number of false positives with no or low increase of false negatives.

5.2 Subject applications

We next present subject applications used in our evaluations. In our evaluations, we used three Java default API libraries and four popular open source libraries. Table 1 shows the characteristics of the subject applications. Columns "Classes" and "Methods" show the number of classes and methods, respectively. For mining patterns of three Java default API libraries, we gathered 49858, 5555, and 15052 code examples for Java Util, Java Transaction, and Java SQL, respectively. Column "KLOC" shows the kilo lines of code in each subject application.

The Java Util package[2] includes the collections framework and other popular utilities used by many different applications. Java Transactions[3] and Java SQL[4] are industry standards for developing multi-tier server-side Java applications. Hibernate[5] and HsqlDB[6] abstract relational databases to use an object-oriented methodology. Columba[7] is an open source email-client application written in Java. Columba provides a user-friendly graphical interface and is suitable for internationalization support. The BCEL library[8], developed by Apache, is mainly used to analyze, create, and manipulate Java class files. We selected these applications, since these applications are popular open source applications and are used as subjects in evaluating previous related approaches (Weimer and Necula 2005; Thummalapenta and Xie 2009).

5.3 RQ1: alternative patterns

We next address the first research question of whether alternative patterns exist in real applications and how high percentage of those patterns represents real programming rules by mining patterns for three Java default API libraries. In general, Alattin accepts an application under analysis and mines patterns for API methods reused by

[2] http://java.sun.com/j2se/1.4.2/docs/api/java/util/package-summary.html.

[3] http://java.sun.com/javaee/technologies/jta/.

[4] http://java.sun.com/j2se/1.4.2/docs/api/java/sql/package-summary.html.

[5] http://www.hibernate.org/.

[6] http://hsqldb.org/.

[7] http://sourceforge.net/projects/columba/.

[8] http://jakarta.apache.org/bcel/.

Table 1 Subject applications and their characteristics

Application	#Classes	#Methods	KLOC
Java util APIs	19	144	–
Java transaction APIs	7	37	–
Java SQL APIs	14	93	–
Hibernate	478	4334	118
HsqlDB	143	1178	41
Columba	1500	7674	136
BCEL	357	2691	32
Total	2518	16151	327

Table 2 Alternative patterns mined by Alattin

Application	*And* patterns				*Or* patterns			
	Total	#RR	#PR	#FP	Total	#RR	#PR	#FP
Java Util	40	34	0	6	51	25	19	7
Java Sql	26	26	0	0	24	21	3	0
Java transaction	3	3	0	0	9	2	4	3
	Xor patterns				*Combo* patterns			
	Total	#RR	#PR	#FP	Total	#RR	#PR	#FP
Java Util	54	35	11	8	50	32	11	7
Java Sql	33	30	3	0	24	21	3	0
Java transaction	8	2	4	2	8	2	4	2

RR: Real Rules, PR: Partial Rules, FP: False Positives

the application under analysis. However, to address this research question, we configured Alattin to accept directly a set of API methods of Java default libraries rather than an application under analysis. In this mode of operation, Alattin mines patterns (programming rules) for the given API methods.

Table 2 shows the patterns mined in all four formats: *And*, *Or*, *Xor*, and *Combo*. For each pattern format, Columns "Total", "RR", "PR", and "FP" show the total number of mined patterns, real rules, partial rules, and false positives, respectively. *Real rules* describe properties that must be satisfied when using an API method. In real rules, all alternatives are real properties. In contrast to real rules, some alternatives in *Partial rules* do not represent real properties. Figure 10 shows a partial rule mined for the `size` method of the `java.util.List` class. In this partial rule, alternatives P_1 and P_3 represent real properties with respect to the `size` method. However, the alternative P_2 does not represent real property, since P_2 is not related to the `size` method. 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 alternatives among mined patterns. For example, consider the preceding partial rule of the form "$P_1 \vee P_2 \vee P_3$", where

Method: `java.util.List.size()`
Pattern: "$P_1 \vee P_2 \vee P_3$", SUP($P_1 \vee P_2 \vee P_3$): 0.79
P_1: "`boolean-check` on the return of `List.isEmpty()`"
P_2: "`boolean-check` on the return of `StringTokenizer.hasMoreTokens()`"
P_3: "`const-check` on the return of `List.size()` with 0"

Fig. 10 A partial rule mined for the `size` method of the `List` class

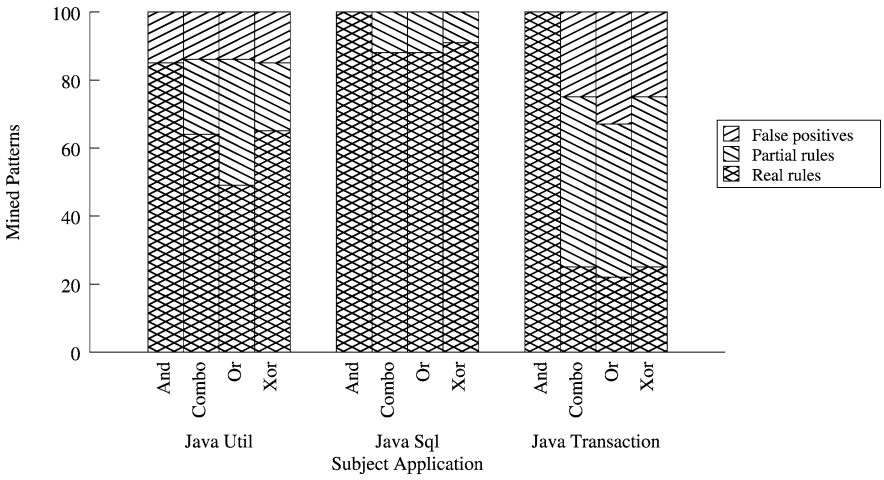

Fig. 11 Classification of mined patterns (shown in percentages)

alternatives P_1 and P_3 are real properties, and the property P_2 is a false-positive alternative. Consider that a code example includes only the P_2 alternative but not P_1 or P_3 alternatives. This code example includes a potential violation, since it does not include either P_1 or P_3 alternative. However, since the code example includes P_2, Alattin does not detect any violation in that code example, resulting in a false-negative defect. *False positives* represent mined patterns where none of the alternatives represents real properties. To mine patterns in all four pattern formats, Alattin took 13, 1, and 1 seconds for Java Util, Java Sql, and Java Transaction, respectively. All experiments were conducted on a machine with 2.2 GHz Intel processor and 3 GB RAM. We used available on-line documentations, JML specifications,[9] or source code of applications for classifying mined patterns into these three categories.

Figure 11 shows the percentages of real, partial, and false-positive rules among mined patterns of each pattern format. In summary, a high percentage of *And*, *Or*, *Xor*, and *Combo* patterns represent real rules. We also found that *Or*, *Xor*, and *Combo* patterns include new real rules that do not exist among *And* patterns (as shown in Table 2, by combining the number of rules in Columns "RR" and "PR" for each pattern format). Since existing approaches mine only *And* patterns, our results show that new

[9] http://www.eecs.ucf.edu/~leavens/JML/.

```
00: ...
01: JarInputStream in;
02: ZipEntry ze; ...
03: while ((ze = in.getNextEntry()) != null){
04:  if(thePath.equals(zipEntry.getName())){
05:   ByteArrayOutputStream buffer =
          new ByteArrayOutputStream();
06:   byte[] bytes = new byte[2048];
07:   int bytesRead;
08:   while((bytesRead = in.read(bytes)) != -1){
09:    buffer.write(bytes, 0, bytesRead);
10:   }
11:   return new ByteArrayInputStream
          (buffer.toByteArray());
12:  }
13: } ...
```

Fig. 12 Alternative patterns mined for the `read` method of the `JarInputStream` class

defects can be detected using *Or*, *Xor*, and *Combo* patterns. The figure also shows that, except *And* patterns, all other pattern formats include a considerable percentage of partial rules. Therefore, although these three pattern formats can help reduce false positives, these three pattern formats can result in false negatives among detected violations. Furthermore, *Or* patterns have higher percentage of partial rules compared to *Xor* and *Combo* patterns, indicating that *Or* patterns result in *more* false negatives compared to *Xor* and *Combo* patterns.

We next present example patterns in each pattern format for the `read` method of the `java.util.jar.JarInputStream` class, which extends the `ZipInputStream` class. This class is used for reading the contents of a Jar file from any input stream such as `FileInputStream`. This class includes three methods: `getNextEntry`, `getNextJarEntry`, and `read`. The `getNextEntry` method reads the next Zip file entry, represented as an instance of the `ZipEntry` class, and positions the stream at the beginning of the entry data in the Jar file. On the other hand, the `getNextJarEntry` method reads the next Jar file entry, represented as an instance of the `JarEntry` class, and positions the stream at the beginning of the entry data. Indeed, the `JarEntry` class extends the `ZipEntry` class and includes additional methods such as `getAttributes` for reading the attributes specific to the Jar file. In general, programmers use either `getNextEntry` or `getNextJarEntry` for iterating through the entries in the Jar file and for reading the contents using the `read` method. Furthermore, if there is only one entry to read from the Jar file, the `read` method is used directly without using either `getNextEntry` or `getNextJarEntry`. Figure 12 shows an example usage of `getNextEntry` and `read` methods. This code example is extracted from Apache's Jakarta Cactus project.[10]

[10] http://jakarta.apache.org/cactus/.

Method: `JarInputStream.read (byte[], int, int)`
***A*. *And* Pattern**
Pattern: "P_1", SUP(P_1): 0.63
P_1: "`const-check` on the return of `JarInputStream.read` with -1"

***B*. *Or* Pattern**
Pattern: "$P_1 \vee P_2$", SUP($P_1 \vee P_2$): 0.67
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"

***C*. *Xor* Patterns**
Pattern: "P_1", SUP(P_1): 0.63
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
Pattern: "$P_2 \oplus P_3$", SUP($P_2 \oplus P_3$): 0.52
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"
P_3: "`null-check` on the return of `JarInputStream.getNextEntry()` before `JarInputStream.read`"

***D*. *Combo* Pattern**
Pattern: "$P_1 \vee (P_2 \oplus P_3)$ ", SUP($P_1 \vee (P_2 \oplus P_3)$): 0.67
P_1: "`const-check` on the return of `JarInputStream.read` with -1"
P_2: "`null-check` on the return of `JarInputStream.getNextJarEntry()` before `JarInputStream.read`"
P_3: "`null-check` on the return of `JarInputStream.getNextEntry()` before `JarInputStream.read`"

Fig. 13 Alternative patterns mined for the `read` method of the `JarInputStream` class

Figure 13 shows the patterns mined for the `read` method in all four formats. The *And* pattern includes only one alternative P_1, which describes that there should be a condition check with "-1" on the return value of the `read` method. Here, "-1" indicates that the end of the entry is reached. The *Or* Pattern includes two alternatives "$P_1 \vee P_2$", where P_2 indicates that there should be a `null-check` on the return of the `getNextJarEntry` method. The reason that the *And* pattern could not mine the alternative P_2 is that there are various scenarios where P_1 and P_2 are not used

together. For example, when P_2 is used, programmers often get the size of the buffer to be read using the `getSize` method of `JarEntry`, which is the return type of the `getNextJarEntry` method. Therefore, programmers often do not explicitly check the return value of the `read` method, when the `getNextJarEntry` method is used. Figure 13 also shows that there are two *Xor* patterns. Interestingly, the second pattern "$P_2 \oplus P_3$" shows that programmers often use either `getNextJarEntry` or `getNextEntry`, but not both together, since the related pattern "$P_2 \vee P_3$" is not being mined. However, the *Xor* pattern alone could not mine the relation among all alternatives P_1, P_2, and P_3. The *Combo* pattern addresses this issue via mining the pattern "$P_1 \vee (P_2 \oplus P_3)$", and shows the relation among all alternatives.

5.4 RQ2: false positives and false negatives

We next address the second research question of whether alternative patterns help reduce false positives among detected violations. We also address whether these patterns introduce no or a low percentage of false negatives among detected violations. To address this question, we used the four subject applications (Hibernate, Columba, BCEL, and HsqlDB) shown in Table 1. In particular, we mined patterns in all four formats from these applications under analysis and apply mined patterns on those applications to detect violations. We next inspected detected violations to classify violations as real defects or false positives based on available specifications such as JML and call sites of related API methods in source code of these subject applications. In our inspection, we ignored the violations related to API methods whose all pattern formats include only one alternative, since our objective is to show the benefits and limitations of alternative patterns. The primary reason is that such patterns do not help show benefits of alternative patterns, since those patterns have the same number of false positives or false negatives in all pattern formats. To compute false negatives, we need a baseline that shows the number of defects exist in subject applications. Since such a baseline does not exist for these applications, we identified all distinct real defects detected using patterns in all pattern formats and used those defects as a baseline for computing false negatives among violations detected using each pattern format.

Table 3 shows detected violations in all subject applications. Column "Real Defects" shows the total number of distinct defects detected using all pattern formats in each application. We used these defects as a baseline for computing the number of false negatives among violations detected using each pattern format. For each pattern format, Columns "Total", "RD", "FN", and "FP" show the total number of violations, real defects, false negatives (their percentage), and false positives (their percentage), respectively. We next summarize our findings for each pattern format with respect to real defects, false negatives, and false positives.

Real defects and false negatives Figure 14 shows comparison between real defects and false negatives for the four pattern formats in each subject application. The figure shows that *Or*, *Xor*, and *Combo* patterns helped detect new defects that are not detected using *And* patterns. For example, in the Columba application, *Or* patterns

Table 3 Analysis of violations detected in subject applications

Application	# Real defects	*And* patterns						*Or* patterns					
		Total	#RD	#FN	%	#FP	%	Total	#RD	#FN	%	#FP	%
Columba	49	117	26	23	47	91	78	113	41	8	16.3	72	63.7
Hibernate	22	93	14	8	36	71	76	177	17	5	22.7	160	90.4
Hsqldb	6	13	6	0	0	7	53.8	5	5	1	16.7	0	0
BCEL	1	2	0	1	100	2	100	13	1	0	0	12	92.3
		Xor patterns						*Combo* patterns					
		Total	#RD	#FN	%	#FP	%	Total	#RD	#FN	%	#FP	%
Columba	49	164	49	0	0	115	73	144	47	2	4	97	67
Hibernate	22	214	21	1	4.5	193	90.2	195	19	3	13.6	176	90.3
HsqlDB	6	11	6	0	0	5	45.5	10	6	0	0	4	40
BCEL	1	20	1	0	0	19	95	16	1	0	0	15	93.8

RD: Real Defects, FN: False Negatives, FP: False Positives

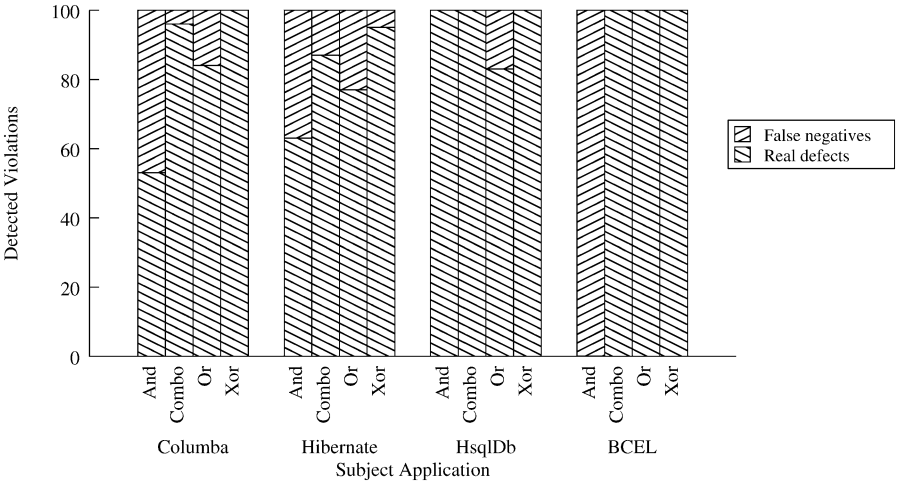

Fig. 14 Real defects and false negatives among detected violations (shown in percentages)

detected 41 real defects, whereas *And* patterns detected only 26 real defects. The reason for new defects is due to the new patterns mined using the *Or*, *Xor*, and *Combo* pattern formats.

Regarding false negatives, our results show that the violations detected using *And* patterns include a high percentage of false negatives in 3 out of 4 applications. Since *And* patterns represent patterns that can be mined by existing approaches, the results show the ineffectiveness of existing approaches in detecting defects in applications under analysis. Among *Or*, *Xor*, and *Combo* patterns, violations detected using *Or* patterns have a higher number of false negatives compared to the violations detected

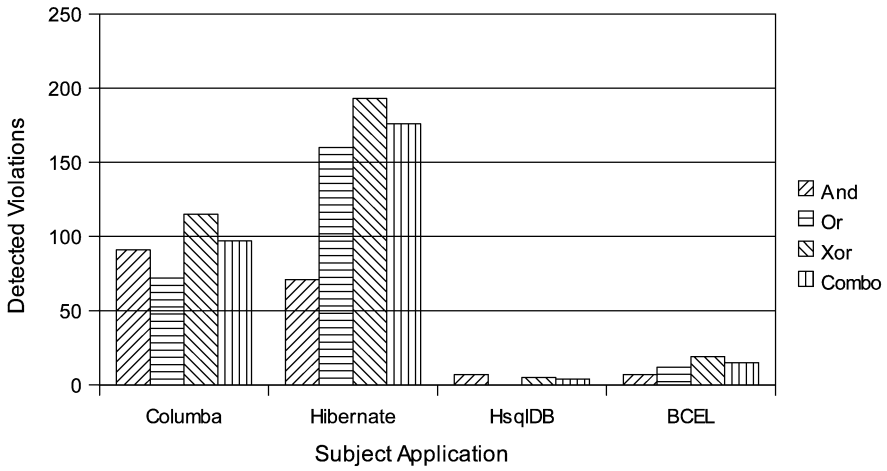

Fig. 15 False positives among detected violations

using *Xor* and *Combo* patterns. For example, in the Columba application, violations detected using *Or* patterns include 8 (16.3%) false negatives compared to 0 (0%) and 2 (4%) false negatives among violations detected using *Xor* and *Combo* patterns, respectively. The primary reason is that *Or* patterns often include partial rules (as shown in Fig. 11), resulting in false negatives among detected violations. The results show that *Xor* patterns are quite effective in detecting defects compared to all three other pattern formats: *And*, *Or*, and *Combo*. However, *Combo* patterns are also shown to be effective than *Or* patterns and have similar effectiveness as that of *Xor* patterns.

False positives Figure 15 shows the number of false positives among violations detected using patterns in each pattern format. Initially, we expected that *Or* and *Combo* patterns help reduce a high percentage of false positives among violations detected using *And* and *Xor* patterns. However, contrary to our expectation, the number of false positives is high among violations detected using *Or* and *Combo* patterns. For example, in the Hibernate application, the numbers of false positives are 71, 160, 193, and 176 among violations detected using *And*, *Or*, *Xor*, and *Combo* patterns, respectively.

In our manual analysis of these false positives, we identified an interesting phenomenon: the majority of false positives is related to the API methods that do not have any patterns mined using the *And* pattern format and have new patterns mined using one or more of the *Or*, *Xor*, and *Combo* pattern formats. To illustrate this scenario, we classified all false positives among detected violations into two categories: *FPAnd* and *FPWithOutAnd*. *FPAnd* includes all false positives detected using patterns (*Or*, *Xor*, and *Combo* patterns) related to API methods that have mined patterns using the *And* pattern format. In contrast, *FPWithOutAnd* includes all false positives detected using patterns (*Or*, *Xor*, and *Combo* patterns) related to API methods that *do not* have mined patterns using the *And* pattern format. Figures 16 and 17 show the classification of false positives for categories *FPAnd* and *FPWithOutAnd*, respectively. Figure 16 shows that *Or* patterns help significantly reduce the number of false

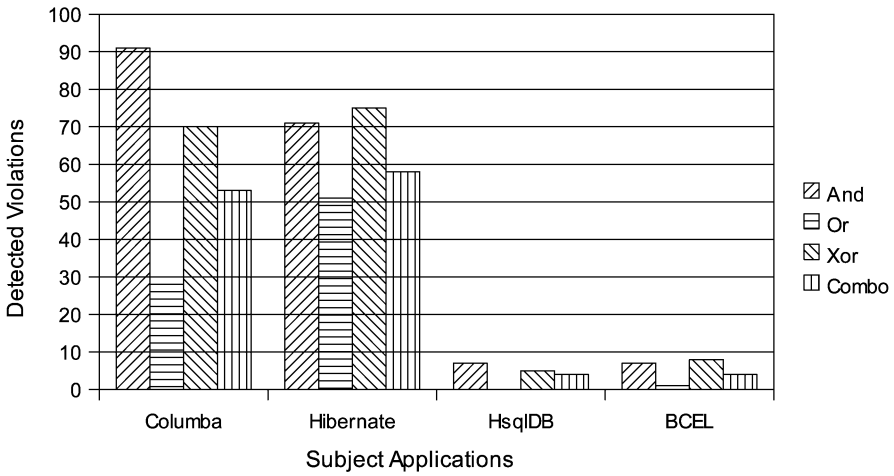

Fig. 16 False positives among detected violations related to API methods with *And* patterns

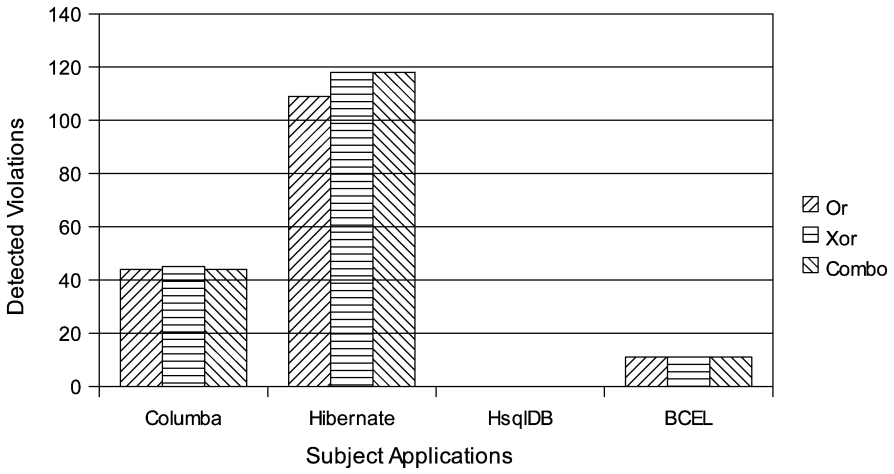

Fig. 17 False positives among detected violations related to API methods without *And* patterns

positives among detected violations compared to *And* and *Xor* patterns. Although *Combo* patterns help reduce false positives, these patterns are not as effective as *Or* patterns. The primary reason is that most of the *Combo* patterns are similar to *Xor* patterns based on our algorithm described in Sect. 3.

Figure 17 shows the classification of false positives for API methods without *And* patterns. As shown in the figure, neither *Or* nor *Combo* patterns help reduce false positives among violations detected using *Xor* patterns. The primary reason is that most of these mined patterns are false positives, resulting in false positives among their detected violations. We next summarize our findings.

5.5 Summary

In summary, based on our results, comparing to *Or*, *Xor*, and *Combo* patterns, *And* patterns are not effective in detecting defects and result in both false positives and false negatives among detected violations. Although *Xor* patterns are effective in detecting defects, these patterns result in a large number of false positives among detected violations. On the other hand, *Or* patterns are effective in reducing false positives, but, result in false negatives as shown in our results. *Combo* patterns can perform reasonably well with respect to both false positives and false negatives. However, *Or* or *Combo* patterns often result in false-positive patterns for those API methods without any *And* patterns. Therefore, based on our empirical results, the best pattern-mining approach (in terms of reducing both false positives and false negatives) is to first mine *And* patterns for API methods and next mine *Combo* patterns. Among violations detected using *Combo* patterns, the best violation-detection approach is to assign higher priority to the violations of API methods with *And* patterns compared to the violations of API methods without *And* patterns. The primary reason is that the former violations are more likely to be real defects compared to the latter violations.

6 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 or features that are not supported 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. In particular, to confirm specifications mined for Java Util, Java Sql, and Java Transaction API libraries, we inspected on-line documentations (Javadoc) and also available JML specifications. On the other hand, for confirming violations as defects, we inspected source code of subject applications to check whether the call sites of API methods include necessary condition checks described in mined rules.

7 Discussion

Since the inception of the Mining Software Engineering Data methodology, researchers have explored the mining of various pattern types ranging from simple itemsets that represent necessary condition checks around API method calls (Thummalapenta and Xie 2009) to complex graph-based pattern types that describe the usage patterns of one or multiple objects (Nguyen et al. 2009). In this paper, we focused

on mining itemset pattern types in four pattern formats and showed their benefits and limitations. In our future work, we plan to explore other pattern types such as sequences (Srikant and Agrawal 1996) and graph-based pattern types (Han and Kamber 2000) to develop new mining algorithms for mining those pattern types in various formats and show their benefits and limitations in defect detection.

In this paper, we proposed a greedy technique for mining patterns in *Or*, *Xor*, and *Combo* pattern formats. Our greedy technique (shown as the function `ApplyGreedy` in Algorithm 2) chooses one parent pattern candidate with the highest support value for each pattern candidate. The primary reason for adopting the greedy technique is to reduce the search space of pattern candidates. In future work, we plan to explore other techniques such as clustering techniques (Han and Kamber 2000). In particular, we plan to first group items (among itemsets in the input database) that are closely related to each other. We next apply our mining algorithms without the greedy technique on each cluster individually. We expect that the number of items in each cluster could be low and help reduce the search space of pattern candidates significantly. We also plan to adopt some properties used by Zhao et al. (2006) for handling search space of pattern candidates while mining patterns.

Our current implementation sometimes is not precise and cannot identify equivalent but syntactically different conditions. For example, our current implementation considers the conditions $\texttt{a} > \texttt{0}$ and $\texttt{a} \geq \texttt{1}$ as different. In future work, we plan to address these issues using more precise static analysis that can identify equivalent conditions. Furthermore, we use intra-procedural analysis for detecting violations. Therefore, in a few cases, some detected defects could be false positives with respect to the entire system point of view. However, this limitation does not affect the results of comparing the benefits and limitations of our four pattern formats, since we use the same analysis for all four pattern formats. In future work, we plan to address this limitation by using inter-procedural analysis.

8 Related work

Mining software repositories PR-Miner developed by Li and Zhou (2005) uses frequent itemset mining to mine programming rules from C code and detect their violations. DynaMine developed by Livshits and Zimmermann (2005) 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. (2007) 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 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 (2005) 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. (2006) 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 (2005), 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. (2006), 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. (2001) 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.

General data mining techniques Although ours is the first approach to propose new pattern formats such as disjunctive and exclusive-disjunctive pattern formats for mining software engineering data, a few approaches have been proposed (in the data mining research area) that target at mining patterns in these preceding formats for other applications such as market basket analysis (Agrawal et al. 1996). Zhao et al. (2006) proposed an approach, called BLOSOM, that targets at mining itemset patterns in four pattern formats: conjunctive, disjunctive, conjunction of disjunctive, and disjunction of conjunctive. In contrast to their BLOSOM approach, our Alattin approach additionally proposes the exclusive-disjunctive pattern format and includes a greedy technique for handling the search space of pattern candidates. In future work, we plan to adopt some properties used by their approach for handling the search space of pattern candidates while mining disjunctive patterns. Nanavati et al. (2001) proposed an approach for mining disjunctive association rules. Given a minimum confidence *min_conf*, their approach uses concepts from propositional logic for pruning the association rules that do not have confidence higher than *min_conf*. Shimizu and Miura (2005) proposed algorithms for mining disjunctive sequence patterns. In contrast to these two preceding approaches, our Alattin approach targets at mining itemset patterns in disjunctive and exclusive-disjunctive pattern formats.

Code search engines Finally, our previous approaches PARSEWeb (Thummalapenta and Xie 2007) and CAR-Miner (Thummalapenta and Xie 2009) 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.

9 Conclusion

To reduce false positives in static defect detection based on code mining, we have developed a novel approach, called Alattin, that includes new mining algorithms and a technique for detecting neglected conditions. Our new mining algorithms mine patterns in four pattern formats: *And*, *Or*, *Xor*, and *Combo*. In our evaluations, we show the benefits and limitations of these pattern formats with respect to false positives and false negatives among detected violations that represent neglected conditions in applications under analysis. Our evaluation results show that the best pattern-mining approach (in terms of reducing both false positives and false negatives) is to first mine *And* patterns for API methods and next mine *Combo* patterns. Among violations detected by *Combo* patterns, the best violation-detection approach is to assign higher priority to the violations of API methods with *And* patterns compared to the violations of API methods without *And* patterns. The primary reason is that the former violations are more likely to be real defects compared to the latter 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 (Thummalapenta and Xie 2009) 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 (Thummalapenta and Xie 2009), 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.

Acknowledgements This work is supported in part by NSF grants CCF-0725190, CCF-0845272, CNS-0958235, and CCF-0915400, ARO grant W911NF-08-1-0443, ARO grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI), and an NCSU CACC grant.

References

Acharya, M., Xie, T., Pei, J., Xu, J.: Mining API patterns as partial orders from source code: from usage scenarios to specifications. In: Proc. ESEC/FSE, pp. 25–34 (2007)

Acharya, M., Xie, T., Xu, J.: Mining interface specifications for generating checkable robustness properties. In: Proc. ISSRE, pp. 311–320 (2006)

Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., Verkamo, A.I.: Fast discovery of association rules. In: Advances in Knowledge Discovery and Data Mining, pp. 307–328 (1996)

Ammons, G., Bodik, R., Larus, J.R.: Mining specifications. In: Proc. POPL, pp. 4–16 (2002)

Burdick, D., Calimlim, M., Gehrke, J.: MAFIA: a maximal frequent itemset algorithm for transactional databases. In: Proc. ICDE, pp. 443–452 (2001)

Chang, R.-Y., Podgurski, A., Yang, J.: Finding what's not there: a new approach to revealing neglected conditions in software. In: Proc. ISSTA, pp. 163–173 (2007)

Bibliography on mining software engineering data. https://sites.google.com/site/asergrp/dmse/ (2010)

Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: a general approach to inferring errors in systems code. In: Proc. SOSP, pp. 57–72 (2001)

Ernst, M., Cockrell, J., Griswold, W., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Trans. Softw. Eng. **27**(2), 99–123 (2001)

Google code search engine. http://www.google.com/codesearch (2006)

Han, J., Kamber, M.: Data Mining: Concepts and Techniques. Morgan Kaufmann, San Mateo (2000)

The Koders source code search engine. http://www.koders.com (2005)

Lethbridge, T., Singer, J., Forward, A.: How software engineers use documentation: the state of the practice. In: IEEE Software, pp. 35–39 (2003)

Li, Z., Zhou, Y.: PR-Miner: Automatically extracting implicit programming rules and detecting violations in large software codes. In: Proc. FSE, pp. 306–315 (2005)

Livshits, V.B., Zimmermann, T.: Dynamine: finding common error patterns by mining software revision histories. In: Proc. ESEC/FSE, pp. 296–305 (2005)

Nanavati, A.A., Chitrapura, K.P., Joshi, S., Krishnapuram, R.: Mining generalised disjunctive association rules. In: Proc. CIKM, pp. 482–489 (2001)

Nguyen, T.T., Nguyen, H.A., Pham, N.H., Al-Kofahi, J.M., Nguyen, T.N.: Graph-based mining of multiple object usage patterns. In: Proc. ESEC/FSE, pp. 383–392 (2009)

Ramanathan, M.K., Grama, A., Jagannathan, S.: Path-sensitive inference of function precedence protocols. In: Proc. ICSE, pp. 240–250 (2007)

Shimizu, K., Miura, T.: Disjunctive sequential patterns on single data sequence and its anti-monotonicity. In: Proc. MLDM, pp. 376–383 (2005)

Shoham, S., Yahav, E., Fink, S., Pistoia, M.: Static specification mining using automata-based abstractions. In: Proc. ISSTA, pp. 174–184 (2007)

Srikant, R., Agrawal, R.: Mining sequential patterns: generalizations and performance improvements. In: Proc. EDBT, pp. 3–17 (1996)

Thummalapenta, S., Xie, T.: PARSEWeb: a programmer assistant for reusing open source code on the web. In: Proc. ASE, pp. 204–213 (2007)

Thummalapenta, S., Xie, T.: Alattin: mining alternative patterns for detecting neglected conditions. In: Proc. ASE, pp. 283–294 (2009)

Thummalapenta, S., Xie, T.: Mining exception-handling rules as sequence association rules. In: Proc. ICSE, pp. 496–506 (2009)

Wasylkowski, A., Zeller, A., Lindig, C.: Detecting object usage anomalies. In: Proc. ESEC/FSE, pp. 35–44 (2007)

Weimer, W., Necula, G.: Mining temporal specifications for error detection. In: Proc. TACAS, pp. 461–476 (2005)
Williams, C.C., Hollingsworth, J.K.: Recovering system specific rules from software repositories. In: Proc. MSR, pp. 1–5 (2005)
Yang, J., Evans, D., Bhardwaj, D., Bhat, T., Das, M.: Perracotta: mining temporal API rules from imperfect traces. In: Proc. ICSE, pp. 282–291 (2006)
Zhao, L., Zaki, M.J., Ramakrishnan, N.: BLOSOM: a framework for mining arbitrary boolean expressions. In: Proc. KDD, pp. 827–832 (2006)