Graph-based Mining of Multiple Object Usage Patterns

Tung Thanh Nguyen
tung@iastate.edu

Hoan Anh Nguyen
hoan@iastate.edu

Nam H. Pham
nampham@iastate.edu

Jafar M. Al-Kofahi
jafar@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Electrical and Computer Engineering Department
Iowa State University

ABSTRACT

The interplay of multiple objects in object-oriented programming often follows specific protocols, for example certain orders of method calls and/or control structure constraints among them that are parts of the intended object usages. Unfortunately, the information is not always documented. That creates long learning curve, and importantly, leads to subtle problems due to the misuse of objects.

In this paper, we propose *GrouMiner*, a novel graph-based approach for mining the *usage patterns* of one or multiple objects. GrouMiner approach includes a graph-based representation for multiple object usages, a pattern mining algorithm, and an anomaly detection technique that are efficient, accurate, and resilient to software changes. Our experiments on several real-world programs show that our prototype is able to find useful usage patterns with multiple objects and control structures, and to translate them into user-friendly code skeletons to assist developers in programming. It could also detect the usage anomalies that caused yet undiscovered defects and code smells in those programs.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Design, Reliability

1. INTRODUCTION

In object-oriented programming, developers must deal with multiple objects of the same or different classes. Objects interact with one another via their provided methods and fields. The interplay of several objects, which involves objects' fields/method calls and the control flow among them, often follows certain orders or control structure constraints that are parts of the intended usages of corresponding classes.

ESEC-FSE'09, August 24–28, 2009, Amsterdam, The Netherlands.

In team development, newly introduced program-specific APIs by one or more team members often lack of usage documentation due to busy schedules. Other developers have to look through new code to understand the programming usages. This is a very inefficient, confusing, and error-prone process. Developers often do not know where to start. Even worse, some of them do not properly use newly introduced classes, leading to errors. Moreover, specific orders and/or control flows of objects' method calls cannot be checked at compile time. As a consequence, errors could not be caught until testing and even go unnoticed for a long time. These also occur often in the case of general API usages.

In this paper, we propose *GrouMiner*, a new approach for mining the usage patterns of objects and classes using graph-based algorithms. In our approach, the usage of a *set of objects* in a scenario is represented as a labeled, directed acyclic graph (DAG), of which nodes represent objects' constructor calls, method calls, field accesses, and branching points of control structures, and edges represent *temporal usage orders* and *data dependencies* among them.

A usage pattern is considered as a (sub)graph that frequently "appears" in the object usage graphs extracted from all methods in the code base. Appearance here means that it is label-isomorphic to an *induced* subgraph of each object usage graph, i.e. satisfying all temporal orders and data dependencies between the corresponding nodes in that graph.

GrouMiner detects those patterns using a novel graph-based algorithm for mining the frequent induced subgraphs in a graph dataset. The patterns are generated increasingly by their sizes (i.e. the number of nodes). Each pattern Q of size $k+1$ is discovered from a pattern P of size k via extending the occurrence(s) of P in every method's graph G in the dataset with relevant nodes of G. The generated subgraphs are then compared to find isomorphic ones. To avoid the computational cost of graph isomorphism solutions, we use Exas [24], our efficient structural feature extraction method for graph-based structures to extract a characteristic vector for each subgraph. It is an occurrence-count vector of sequences of nodes and edges' labels. The generated subgraphs having the same vector are considered isomorphic and counted toward the frequency of the corresponding candidate. If it exceeds a threshold, the candidate is considered as a pattern and is used to discover the larger patterns.

After the patterns are mined, they could be translated into user-friendly code skeletons, which assist developers in object usages. The patterns that are confirmed by the developers as typical usages could also be used to automatically detect the locations in programs that deviate from them. A

portion of code is considered violating a pattern P if the corresponding object usage graph contains only an instance of a strict sub-pattern P, i.e., not all properties of P are satisfied. These locations are often referred to as *violations* and *rare violations* are considered as *usage anomalies*.

The *departure points* of GrouMiner from existing mining approaches for temporal object usages include three aspects. Firstly, the mined patterns and code skeletons provide more information to assist developers in the usage flows among objects including control structures (e.g. conditions, loops, etc). Existing object mining approaches are still limited to the patterns in the form of either (1) a set of pairs of method calls and in each pair, one call occurs before another, or (2) a partial order among method calls. The patterns do not contain control structures or conditions among them. In other words, their detected patterns correspond to the subset of edges in GrouMiner's pattern graphs. Secondly, GrouMiner's mined patterns are for both *common* and *program-specific* cases with *multiple interplaying/interacting objects*, without requiring external inputs except the program itself. Existing approaches discover patterns involving methods of a *single* object *without* control structures. Finally, GrouMiner's mining and detection of anomalies of object usages can work as software changes. That is, GrouMiner could take the changes from one revision of a system to another, and then update the mined patterns and detect the anomalies in the new revision.

The *main contributions* of this paper include

1. A graph-based representation of object usages and their patterns that captures the interactions among multiple objects and the temporal usage orders and data dependencies among objects' methods and fields. An extraction algorithm is provided.

2. An efficient and scalable graph-based mining algorithm for multiple object usage patterns.

3. An automatic, graph-based technique for detecting and ranking the degree of anomalies in object usages. Both of pattern discovery and anomaly detection algorithms are resilient to software changes.

4. An empirical study on real-world systems shows the benefits of our approach. The evaluation shows that our tool could efficiently detect a number of high-quality object usage patterns in several open source projects. GrouMiner is able to detect yet undiscovered defects and code smells caused by the misuse of the objects even in mature software.

Sections 2, 3, and 4 discuss GrouMiner in details. Section 5 describes our empirical evaluation of GrouMiner. Related work is given in Section 6. Conclusions appear last.

2. OBJECT USAGE REPRESENTATION

Object usages involve in the object instantiations, method calls, or data field accesses. Because creating an object is the invocation of its constructor and an access to a field could be considered to be equivalent to a call to a getter or a setter, we use the term *action* to denote all of such operations.

Figure 1 shows a real-world example (that GrouMiner mined from Columba 1.4), containing a usage scenario for reading and outputting a file. There are 5 objects: a `StringBuffer` (`strbuf`), a `BufferedReader` (`in`), a `FileReader`, and two `String`s (`str` and `file`). The usage is as follows. First, `strbuf` and the `FileReader` are created. Then, the latter is used in the creation of object `in`. `in`'s `readLine` is called to read a text line into `str` and `str` is added to the content of `strbuf`

```
StringBuffer strbuf = new StringBuffer();
BufferedReader in = new
         BufferedReader(new FileReader(file));
String str;
...
while ((str = in.readLine()) != null) {
   ...
   strbuf.append(str + "\n");
}  ...
...
if (strbuf.length() > 0)
   outputMessage(strbuf.toString(),...);
in.close();
```

Figure 1: An Illustrated Example

via its method `append`. These two actions are carried out in a `while` loop until no line is left. After the loop, if the content of `strbuf` is not empty (checked by its `length`), it is output via `toString`. Finally, `in` is closed.

From the example, we can see that the usages of multiple objects could be modeled with the following information: 1) The temporal order of their actions (e.g. `strbuf` must be created before its `append` can be used), 2) How their actions appear in control structures (e.g. `append` could be used repeatedly in a `while` loop), and 3) How multiple objects interact with one another (e.g. `strbuf`'s `append` is used after `in`'s `readLine`). In other words, the information describes the **usage orders** of objects' actions, i.e. whether an action is used before another, with involving **control structures** and **data dependencies** among them.

The usage order is not always exhibited in the textual order in source code. For example, the creation of the `FileReader` object occurs before that of `in` while the corresponding constructor appears after in the source code. It is not the order in execution traces either, where `append` could be executed before or after `readLine`. Therefore, we consider an action A to be *used before* another action B if A is *always generated before* B in the corresponding executable code.

To represent multiple object usages with all aforementioned information, in GrouMiner, we propose a novel graph-based representation called *graph-based object usage model* (**groum**). A groum is a labeled, directed acyclic graph representing a usage scenario for a single or multiple objects. Figure 2b) shows the groum representing the usage of the objects in the illustrated example. Let us explain in details.

2.1 Represent Usage Orders

To represent the usage order of objects' actions, GrouMiner uses *action nodes* in a groum. Each of them represents an action of an object and is assigned a label `C.m`, in which `C` is the class name of the object and `m` is the name of a method or a field. (In the context that the class name is clear, we use just the method name to identify the action node).

The directed edges of a groum are used to represent the usage orders. An edge from an action node A to an action node B means that in the usage scenario, A is used before B. This implies that B is used after A, i.e. there is no path from B to A. Therefore, a groum is a DAG.

For example, in Figure 2b), the nodes labeled `StringBuffer.<init>` and `StringBuffer.append` represent the object instantiation and the invocation of method `append` of a `StringBuffer` object, respectively. The edge connecting them shows the usage order, i.e. `<init>` is used before `append`.

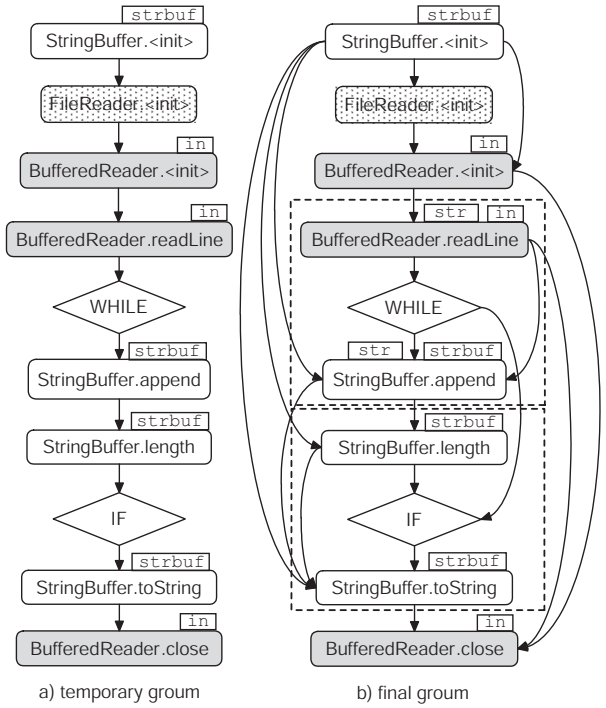

Figure 2: Groum: Graph-based Object Usage Model

2.2 Represent Control Flow Structures

To represent how developers use the objects within the control flow structures such as *conditions*, *branches*, or *loop* statements, GrouMiner uses *control nodes* in a groum. To conform to the use of edges for representing temporal orders, such control nodes are placed at the branching points (i.e. where the program selects an execution flow), rather than at the starting points of the corresponding statements. The edges between control nodes and the others including action nodes represent the usage orders as well.

For example, in Figure 2b), the control node labeled `WHILE` represents the `while` statement in the code in Figure 1, and the edge from the node `BufferedReader.readLine` to `WHILE` indicates that the invocation of `readLine` is *generated before* the branching point of that `while` loop.

To represent the scope of a control flow structure (e.g. the invocation of `readLine` is within the `while` loop), the list of all action nodes and control nodes within a control flow structure is stored as an attribute of its corresponding control node. In the Figure 2b), such scope information is illustrated as the dashed rectangles.

Note that there is no backward edge for a loop structure in a groum since it is a DAG. However, without backward edges, scope information is still sufficient to show that the actions in a loop could be invoked repeatedly.

2.3 Represent Multiple Interplaying Objects

To represent the usage of multiple interacting objects, a groum contains action nodes of not only one object, but also those of multiple objects. The edges connecting action nodes of multiple objects represent the usage orders as well. Moreover, to make a groum have more semantic information, such edges connect only the nodes that have data dependencies, e.g. the ones involving the same object(s) (Section 2.4.2). In Figure 2, action nodes of different objects are filled with different backgrounds. Let us describe how groums are built.

2.4 Extract Groum from Source Code

DEFINITION 1 (GROUM). *A groum is a DAG such that:*
1. Each node is an action node or a control node. An action node represents an ***invocation*** *of a constructor or a method, or an access to a field of one object. A control node represents the* ***branching point*** *of a control structure. Label of an action node is "C.m" with C is its class name and m is the method (or field) name. Label of a control node is the name of its corresponding control structure.*
2. A groum could involve ***multiple*** *objects.*
3. Each edge represents a ***(temporal) usage order*** *and a* ***data dependency****. An edge from node A to node B means that A is used before B, i.e. A is generated before B in executable code, and A and B have a data dependency. Edges have no label.*

DEFINITION 2. *Two groums are (semantically)* ***equivalent*** *if they are* ***label-isomorphic*** *[24].*

Compared to existing representations for object usages [1, 4], groums are able to handle **multiple** interplaying objects, with **usage orders**, **control structures**, and **data relations** among objects' actions. Groum is more compact and specialized toward usage patterns than Program Dependence Graph (PDG) and Control Flow Graph (CFG).

Our algorithm extracts groum from a portion of code of interest in the following steps: 1) Parse it into an AST, 2) Extract the action and control nodes with their partial usage orders from the AST into a temporary groum, and 3) Identify data dependencies and total usage orders between the nodes to build the final groum for the usage of all objects in the code portion. Step 1 is provided by the AST API from Eclipse. The other steps are discussed next.

2.4.1 Extract Temporary Groum

In this step, a temporary groum is extracted from the AST for each method. The extraction is processed bottom-up, building-up the groum of each structure from the groums of its sub-structures. For a simple structure such as a single method invocation or a field access, a groum with only one action node is created. For more complex structures such as expressions or statements, the groum is merged using two operations: sequential merge (denoted by $\Rightarrow$) and parallel merge (denoted by $\vee$). Of course, for a program structure having neither action nor control node, its groum is empty.

The merge operations are defined as follows. Let X and Y be two groums. $X \vee Y$ is a groum that contains all nodes and edges of X and Y and there is no edge between any nodes of X and Y. $X \Rightarrow Y$ is also a groum containing all nodes and edges of X and Y. However, there will be an edge from each sink node (i.e. node having no outgoing edge) of X to each source node (i.e. node having no incoming edge) of Y. Those edges represent the temporal usage order, i.e. all nodes of X are used before all nodes of Y. It could be checked that those two operations are associated; and parallel merge $\vee$ is symmetric but sequential merge $\Rightarrow$ is not.

Sequential merge is used where the code has an explicit generation order such as between statements within a block. Parallel merge is used where there is no explicit generation order such as between the branches of an `if-else` or a `switch`

Code Structure	Code Template	Groum
method invocation	o.m()	C.m
field access	o.f	C.f
parameters	o.m(X,Y,Z,...)	(X∨Y∨Z∨...)⇒ C.m
cascading call	X.m()	X⇒C.m
expression	X○Y	X∨Y
if statement	if (X) Y; else Z;	X⇒IF⇒(Y∨Z)
while statement	while (X) Y;	X⇒WHILE⇒Y
for statement	for (X;Y;Z) W;	X⇒Y⇒FOR⇒W⇒Z
block	{X;Y;Z;...}	X⇒Y⇒Z ⇒...

Table 1: Groum Composition Rules

```
StringBuffer aStringBuffer = new StringBuffer();
FileReader aFileReader = new FileReader(String);
BufferedReader aBufferedReader = new
                    BufferedReader(aFileReader);
 ...
while (aString = aBufferedReader.readLine()...){
    ...
    aStringBuffer.append(String);
    ...
}
 ...
if (aStringBuffer.length()...)
   outMessage(aStringBuffer.toString(),...);
aBufferedReader.close();
```

Figure 3: A Usage Skeleton

statement. With the use of parallel merge, a resulting groum is not affected by the writing order of some structures. E.g., two syntactically different expressions $X + Y$ and $Y + X$ have an identical groum, i.e. are considered as equivalent in usages. Thus, groum is well-suited for programming usages.

Table 1 shows the composition rules for groums in structures. Variable name symbols such as X, Y, Z, and W denote the structures (in the "Code Template" column) and their corresponding groums (in the "Groum" column). Other symbols o, m, f, and C denote the object, method, field, and class names, respectively. The table does not show rules for structures such as `try-catch`, `switch-break`, and `do-while` since they are processed similarly to two parallel blocks, an `if`, and an `while` structures, respectively.

In the groum extraction process, if there exists a consecutive sequence of the same method calls, they are contracted to a common node. For example, if multiple consecutive `append`s occur before a `toString`, they are contracted with a repetitive attribute "+". Thus, the groum is more concise and the usage representation captures better program semantics. Figure 2a) shows the extracted temporary groum of the illustrated example. Nodes in round rectangles are action nodes and those in diamonds are control nodes. The edges with solid lines represent the usage orders.

2.4.2 Build Final Groum

To build the final groum with total usage orders and data dependencies among all objects' actions, GrouMiner first needs to determine data dependencies between all the nodes in the extracted temporary groum G.

GrouMiner determines data dependency based on the following *intra-procedural* and explicit data analysis via shared variables. Firstly, for each node (including both action and control nodes), a list of involved variables is collected and stored as its attributes. Then, any two nodes that share at least a common variable in their lists are considered to have a data dependency. The rules to determine the list of involved variables for a node are as follows:

1) For an action node, its corresponding variable is considered as an involved variable.

2) For a control node, all variables processed in the corresponding control structure are regarded as involved ones.

3) For an action node representing a field assignment such as `o.f = E`, all variables processed in the evaluation of E are considered as involved variables.

4) For an action node representing an invocation of a method, all variables involving in the evaluation of the parameters of the method are considered as involved variables.

5) If an invocation is used for an assignment, e.g. `C x = new C()` or `x = o.m()`, the assigned variable `x` is also involved.

In Figure 2, involved variables of the action node `readLine` are `in` (rule 1) and `str` (rule 5). Those of `append` are `strbuf` (rule 1) and `str` (rule 4). Thus, those two nodes have a data dependency. For the control nodes `WHILE` and `IF`, by rule 2, the lists of involved variables are `{in,str,strbuf}` and `{strbuf}`, respectively. Thus, they have a data dependency.

This data analysis is only intra-procedural and explicit because GrouMiner focuses on the point of view of individual methods. (This individual method approach was shown to be scalable and to get comprehensive results [4].) To make a groum capture better the semantics of object usages, one could use inter-procedural analysis techniques to determine more complete data dependencies. Since those techniques are expensive, in our current implementation, we use a heuristic. That is, to increase the chance of connecting usages of objects having implicit data dependencies, each action node of an object will be connected to the *nearest* (downward) action node of any other object. For example, two nodes `StringBuffer.<init>` and `BufferedReader.<init>` in Figure 2b) are connected by this type of edge. This idea is based on the belief that the (implicitly) related objects tend to be used in near locations in code. In other words, these edges connect different parts of a method's groum where each part represents the usage of a different object.

This step also helps discriminating the usages of different objects of the same class with the same method call. In this case, their action nodes have the same labels, but the involved variables might be different, thus, have different edges (usage orders and data dependencies). E.g., assume that a scenario has two opened files: the first is for reading and the second for writing. If reading and writing involve a shared variable, the series of calls for two `File` objects would be connected as in a single usage. Otherwise, they would be identified as two separated usages of `File` objects.

2.5 Un-parse Groum to Code Skeleton

To make graph-based object usages and patterns more readable, GrouMiner could un-parse a groum of interest back into a *usage skeleton* using the scope information, the list of involved objects, and related code. E.g. with the groum in Figure 2b) and associated information, GrouMiner knows that 1) `BufferedReader.readLine` is in the scope of the `while` loop, 2) it is of the same object with `BufferedReader.<init>` and `BufferedReader.close`, and 3) it is assigned to variable `str`. GrouMiner will generate the corresponding usage skeleton as in Figure 3. This skeleton is useful for developers to understand the usage and re-use the pattern (if any).

To generalize a usage skeleton, GrouMiner names the objects based on their class names and uses indexes when there

are different objects of the same class. When the objects are not unique or un-determinable (such as in parameter expressions, cascading calls, constants, or static methods/fields), the class name (i.e. type of the expression) is used instead.

3. USAGE PATTERN MINING

This section describes our novel graph-based pattern mining algorithm for multiple object usages. Intuitively, an object usage is considered as a pattern if it frequently appears in source code. GrouMiner is interested only in the intra-procedural level of source code, therefore the groums are extracted from all methods. Each method is represented by a groum. In many cases, the object usages involve only some, but not all object action and control nodes of an extracted groum in a method. In addition, the usages must include all temporal and data properties of those nodes, i.e. *all* involving edges. Therefore, in a groum representing a method, an object usage is an *induced* subgraph of that groum, i.e. involving some nodes and all the edges of such nodes. Note that any induced subgraph of a groum is also a groum.

3.1 Formulation

DEFINITION 3. *A* **groum dataset** *is a set of all groums extracted from the code base, denoted by* $D = \{G_1, G_2, ..., G_n\}$.

DEFINITION 4. *An induced subgraph* X *of a groum* G_i *is called an* **occurrence** *of a groum* P *if* X *is equivalent to* P.

A usage could appear more than one time in a portion of code and in the whole code base, i.e. a groum P could have multiple occurrences in each groum G_i. We use $G_i(P)$ to denote the occurrence set of P in G_i and $D(P) = \{G_1(P), G_2(P), ..., G_n(P)\}$ to denote the set of all occurrences of P in the entire groum dataset. $G_i(P)$ is empty if P does not occur in G_i. If P occurs many times, only the *non-overlapping* occurrences are considered as different or independent.

DEFINITION 5. *The* **frequency** *of* P *in* G_i*, denoted by* $f_i(P)$*, is the maximum number of independent (i.e. non-overlapping) occurrences of* P *in* G_i.

The frequency of P *in the entire dataset,* $f(P)$*, is the sum of frequencies of* P *in all groums in the dataset.*

DEFINITION 6 (PATTERN). *A groum* P *is called a* pattern *if* $f(P) \geq \sigma$*, i.e.* P *has independently occurred at least* σ *times in the entire groum dataset.* σ *is a chosen threshold.*

DEFINITION 7 (PATTERN MINING PROBLEM). *Given* D *and* σ*, find the list* L *of all patterns.*

3.2 Algorithm Design Strategy

There have been many algorithms developed for mining frequent subgraphs on a graph dataset (i.e. multi-settings) or on a single graph. However, they are not applicable for this mining problem because (1) the existing mining algorithms for multi-settings count only one occurrence in each graph (i.e. the frequency of a candidate pattern is the number of graphs it occurs, which is different from our problem); and 2) mining algorithms on a *single* graph setting are developed for edge-oriented subgraphs, i.e. a subgraph is defined as a set of edges that form a weakly connected component. They are only efficient on *sparse* graphs while our patterns are the induced subgraphs of *dense* graphs [26].

```
1  function PattExplorer (D)
2    L ← {all patterns of size one}
3    for each P ∈ L do Explore(P,L,D)
4  return L
5
6  function Explore(P,L,D)
7    for each pattern of size one U ∈ L do
8      C ← P ⊕ U
9      for each Q ∈ patterns(C)
10       if f(Q) ≥ σ then
11         L ← L ∪ {Q}
12         Explore(Q,L,D)
```

Figure 4: PattExplorer Algorithm

We have developed a novel mining algorithm for our problem, named *PattExplorer*. The main design strategy of this algorithm is based on the following observation: isomorphic graphs also contain isomorphic (sub)graphs. Thus, subgraphs of frequent (sub)graphs (i.e. patterns) are also frequent. In other words, larger patterns must contain smaller patterns. Therefore, the large patterns could be discovered (i.e. generated) from the smaller patterns.

Based on this insight, PattExplorer mines the patterns increasingly by size (i.e the number of nodes): patterns of a larger size are recursively discovered by exploring the patterns of smaller sizes. During this process, the occurrences of candidate patterns of size $k+1$ are first generated from the occurrences of discovered patterns of size k and those of size one. Then, the generated occurrences are grouped into isomorphic groups, each of which represents a candidate pattern. The frequency of each candidate is evaluated and if it is larger than a threshold, the candidate is considered as a pattern and is used to recursively discover larger patterns.

Exact-matched graph isomorphism is highly expensive for dense graphs [24]. A state-of-the-art algorithm for checking graph isomorphism is *canonical labeling* [26], which works well with sparse graphs, but not with dense graphs. Our previous experiment [24] also confirmed this: it took 3,151 seconds to produce a unique canonical label for a graph with 388 nodes and 410 edges. Our algorithm employs an approximate vector-based approach. For each (sub)graph, we extract an Exas characteristic vector [24], an occurrence-counting vector of sequences of nodes and edges' labels. Graphs having the same vector are considered as isomorphic. Exas was shown to be highly accurate, efficient, and scalable. For example, it took about 1 second to produce the vector for the aforementioned graph. It is about 100% accurate for graphs with sizes less than 10, and 94% accurate for sizes in 10-30. In our evaluation of GrouMiner, most patterns are of size less than 10. Details on Exas are in [24].

3.3 Detailed Algorithm

The pseudo-code of PattExplorer is in Figure 4. First, the smallest patterns (i.e. patterns of size one) are collected into the list of patterns L (line 2). Then, each of such patterns is used as a starting point for PattExplorer to recursively discover larger patterns by function `Explore` (line 3). The main steps of exploring a pattern P (lines 6-12) are: 1) generating from P the occurrences of candidate patterns (line 8), 2) grouping those occurrences into isomorphic groups (i.e. function `patterns`) and considering each group to represent a candidate pattern (line 9); 3) evaluating the frequency of

each candidate pattern to find the true patterns and recursively discovering larger patterns from them (lines 10-12).

3.3.1 Generate Occurrences of Candidate Patterns

In the algorithm, each pattern P is represented by $D(P)$, the set of its occurrences in the whole graph dataset. Each of such occurrences X is a subgraph and it might be extended into a larger subgraph by adding a new node Y and all edges connecting Y and the nodes of X. Let us denote that graph $X+Y$. Since a large pattern must contain a smaller pattern, Y must be a frequent subgraph, i.e. an occurrence of a pattern U of size 1. This will help to avoid generating non-pattern subgraphs (i.e. cannot belong to any larger pattern).

The operation $\oplus$ is used to denote the process of *extending* and *generating* all occurrences of candidate patterns from all occurrences of such two patterns P and U:

$P \oplus U = \{X + Y | X \in G_i(P), Y \in G_i(U), i = 1..n\}$.

3.3.2 Find Candidate Patterns

To find candidate patterns, function `patterns` is applied on C, the set of all generated occurrences. It groups them into the sets of isomorphic subgraphs. Grouping criteria is based on Exas vectors. All subgraphs having the same vector are considered as isomorphic. Thus, they are the occurrences of the same candidate pattern and are collected into the same set. Then, for each of such candidate Q, the corresponding subgraphs are grouped by the graph that they belong to, i.e. are grouped into $G_1(Q), G_2(Q), ...G_n(Q)$, to identify its occurrence set in the whole graph dataset $D(Q)$.

3.3.3 Evaluate the Frequency

Function $f_i(Q)$ is to evaluate the frequency of Q in each graph G_i. In general, such evaluation is equivalent to the maximum independent set problem because it needs to identify the maximal set of non-overlapping subgraphs of $G_i(Q)$. However, for efficiency, we use a greedy technique to find a non-overlapping subset for $G_i(Q)$ with a size as large as possible. PattExplorer sorts the occurrences in $G_i(Q)$ descendingly by their numbers of nodes that could be added to them. As an occurrence is chosen in that order, its overlapping occurrences are removed. Thus, the resulting set contains only non-overlapping occurrences. Its size is assigned to $f_i(Q)$.

After all $f_i(Q)$ values are computed, the frequency of Q in the whole dataset is calculated: $f(Q) = f_1(Q) + f_2(Q) + ... + f_n(Q)$. If $f(Q) \geq \sigma$, Q is considered as a pattern and is used to recursively extend to discover larger patterns.

3.3.4 Disregard Occurrences of Discovered Patterns

Since the discovery process is recursive, occurrences of a discovered pattern could be generated more than once. (In fact, a sub-graph of size $k+1$ might be generated at most $k+1$ times from the sub-graphs of size k it contains.) To avoid this redundancy, when generating the occurrences of candidate patterns, `Explore` checks if a sub-graph is an occurrence of a discovered pattern. It does this by comparing Exas vector of the sub-graph to those of stored patterns in L. If the answer is true, the sub-graph is disregarded in $P \oplus U$.

4. USAGE ANOMALY DETECTION

4.1 Graph-based Anomaly Detection

The usage patterns can be used to automatically find the anomaly usages, i.e. locations in programs that deviate from the typical object usages. The definition of an anomaly usage is adapted from [4] for our graph-based representation.

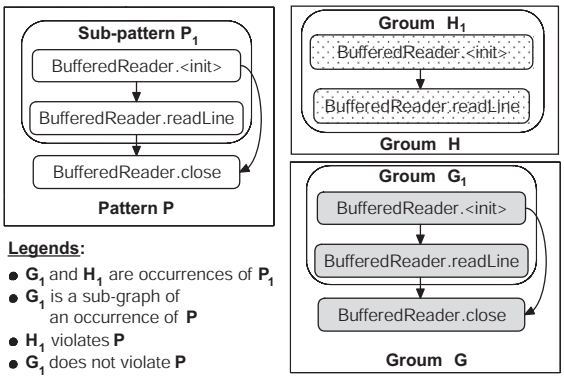

Figure 5: A Violation Example

Figure 5 shows an example where a `BufferedReader` is used without `close()`. P is a usage pattern with a `BufferedReader`. P_1 is a sub-pattern of P, containing only two action nodes `<init>` and `readLine`. A groum G contains an occurrence of P, thus contains also another occurrence G_1 of P_1 as a sub-graph of that occurrence of P. Another groum H contains an occurrence H_1 of P_1 but no occurrence of P. Since P_1 is a sub-pattern of P, H_1 is called an *inextensible* occurrence of P_1 (i.e. it could not extend to an occurrence of P), thus is considered to *violate* P. Because containing H_1, H is also considered to violate P. In contrast, G_1 is *extensible*, thus, G_1 and G do not violate P.

However, not all violations are considered as defects. For example, there might exist the occurrences of the usage `<init>-close()` (without `readLine`) that also violate P, but they are acceptable. A violation is considered as an anomaly when it is *too rare*. The rareness of the violations could be measured by the ratio $v(P_1, P)/f(P_1)$, with $v(P_1, P)$ is the number of *inextensible* occurrences of P_1 corresponding to P in the whole dataset. If rareness is smaller than a threshold, corresponding occurrences are considered as anomalies. The lower a rareness value is, the higher the anomaly is ranked.

DEFINITION 8. *A groum H is considered as a usage* **anomaly** *of a pattern P if H has an* inextensible *occurrence H_1 of a sub-pattern P_1 of P and the ratio $v(P_1, P)/f(P_1) < \delta$, with $v(P_1, P)$ is the number of such inextensible occurrences in the whole groum dataset and δ is a chosen threshold.*

GrouMiner provides anomaly detection in two cases: (1) Detecting anomalies in the currently mined project (by using mined groums) and (2) Detecting anomalies when the project changes, i.e., in the new revision.

In both cases, the main task of anomaly detection is to find the inextensible occurrences of all patterns P_1 corresponding to the detected patterns. In the first case, because storing the occurrence set $D(P_1)$, GrouMiner can check each occurrence of P_1 in $D(P_1)$: if it is inextensible to any occurrence of a detected pattern P generated from P_1, then it is a violation. Those violations are counted via $v(P_1, P)$. After checking all occurrences of P_1, the rareness value $v(P_1, P)/f(P_1)$ is computed. If it is smaller than the threshold δ, such a violation is reported as an anomaly. In the second case, GrouMiner must update the occurrence sets of detected patterns before finding the anomalies in the new version.

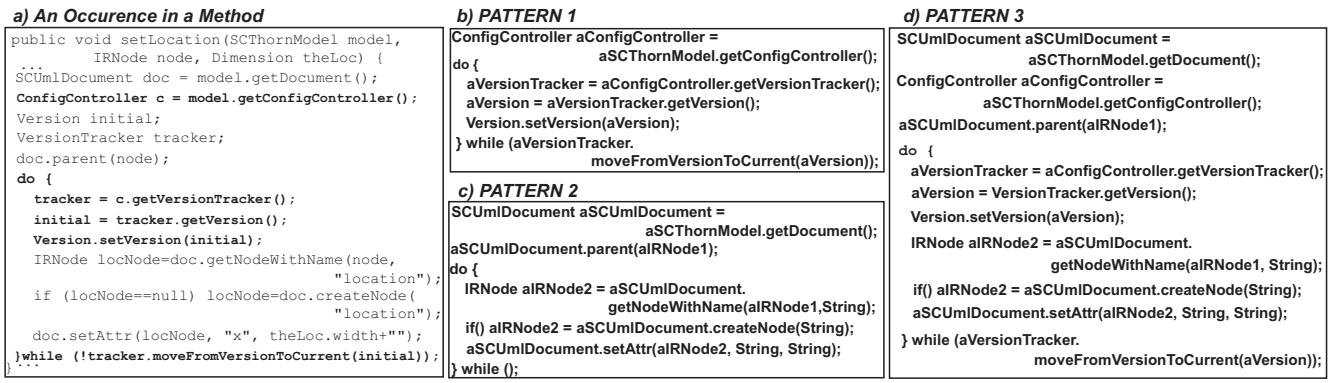

Figure 6: Usage Patterns Mined from Fluid

4.2 Pattern Updating

The problem of pattern updating is formulated as follows. Given D_+ and D_- as the set of added and deleted groums, respectively: update the occurrence sets of all discovered patterns in L and find new patterns occurring on D_+.

To solve this, GrouMiner first detects the deleted and added files when the code changes. Modified files are treated as the deletion of old files and the addition of new ones. This information is provided by a versioning system. Then, groums from added files are extracted into D_+ and groums of deleted files are collected to D_-. For each pattern P stored in L, the occurrences belonging to the groums in D_- are removed from its occurrences set $D(P)$.

PattExplorer is then applied on D_+ as in initial mining. This could detect new patterns occurring on D_+ and update occurrences sets of discovered patterns in L. When an occurrence X of a discovered pattern P is generated, X is added to $D(P)$ and P is considered as changed. P will be used to recursively discover other new or changed patterns.

For space efficiency, occurrence generating is applied only on D_+, i.e. new groums. Thus, it could not detect the new patterns that have occurrences in both old and new groums. (To be complete, all non-pattern subgraphs must be stored)

5. EMPIRICAL EVALUATION

To evaluate performance and effectiveness of GrouMiner, we have applied it to several Java projects (see Table 2). The experiments were carried out in a computer with WindowsXP, Intel Core 2 Duo 2Ghz, 3GB RAM.

5.1 Pattern Mining Evaluation

5.1.1 Case Studies

The information on the subject projects is given in Table 2. Let us examine the quality of resulting patterns.

Example 1: Figure 6 shows example patterns that were mined by GrouMiner for Fluid project [11]. The goal of that code in Figure 6a) is to set up the Fluid version controller to track the changes to an UML element in a graphical editor. The particular type of changes to be tracked in that code is that of the element's location on screen. The location changing procedure involves the retrieval of an UML element object, a parent setting, the checking of the existence of "location" node, and the setting of the value for the location attribute. Since both *change-tracking* and *location-changing* routines occur frequently in Fluid code, GrouMiner detected them as two individual patterns (Figures 6b and 6c).

GrouMiner is able to detect both patterns even though in the code, they interleave with each other. Each pattern involves multiple objects interacting with one another and the flow involves a `while` control structure (Pattern 1 (4 objects): a `SCThornModel`, a `ConfigController`, a `VersionTracker`, and a `Version`, with 5 method calls; Pattern 2 (4 objects): a `SCThornModel`, a `SCUMLDocument`, and 2 `IRNodes`, with 5 calls). In addition to code interleaving, 2 patterns also have a common object `model` of type `SCThornModel`. Thus, there is a data dependency edge connecting subgraphs of two patterns.

Interestingly, the entire procedure of tracking changes to the location of UML elements was also detected as a pattern (Pattern 3 in Figure 6). The reason is that the procedure frequently occurs due to the needs to track changes to different types of UML elements in an editor. Since GrouMiner discovers the patterns from the smallest to the largest sizes, it is able to detect all three patterns (two smaller patterns connect via data dependency and usage order edges).

Example 2: Figure 7 shows another example mined from Ant 1.7.1. The piece of code on the left is the steps to test a mail server with a client-server paradigm. With similar reasons as Fluid's example, GrouMiner is able to detect three patterns. The first pattern is the steps to initiate a server thread, which involves two objects: a `ServerThread` and a `Thread`. The second pattern is the procedure to launch the client thread and to test the returned result. There are also two interplaying objects (a `ClientThread` and a `Thread`). Unlike in the Fluid's example, there is no intra-procedural data dependency between objects in two patterns. However, the temporal orders between method calls in an individual pattern and between calls in two patterns are important and captured as edges in a groum (e.g. a server thread is `started` before a client thread). These temporal properties are exhibited frequently as well. Thus, Pattern 3 is also detected. Moreover, this example shows that GrouMiner is able to handle multiple objects of the same type `Thread`.

Example 3: Figure 8 shows another pattern mined from AspectJ to illustrate a routine to convert a `Set` to a `String` using `StringBuffer` and `Iterator`. GrouMiner is able to detect this pattern with four interplaying objects and the control structures `for, if` among method calls. For object `iter`, JADET [4], a well-known object usage miner, would produce a pattern $P =$ `{hasNext() < hasNext(), hasNext() < next()}` (< means "occurs before"), thus, providing less information.

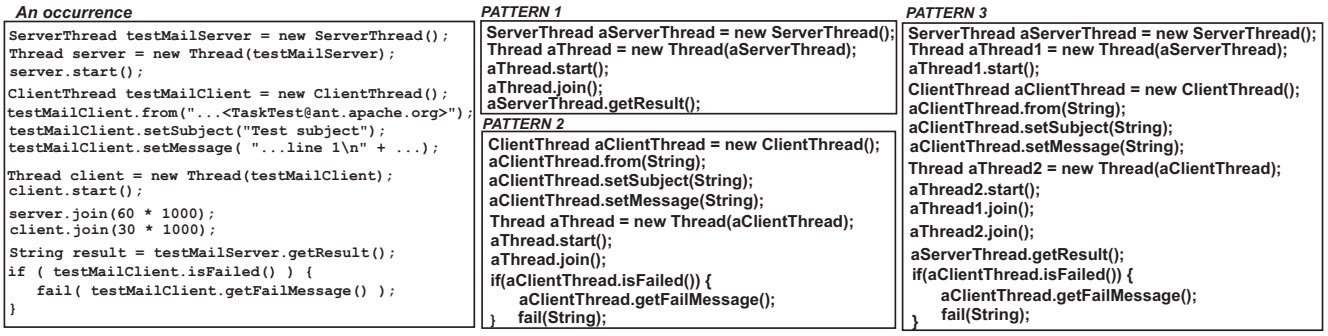

Figure 7: Usage Patterns Mined from Ant

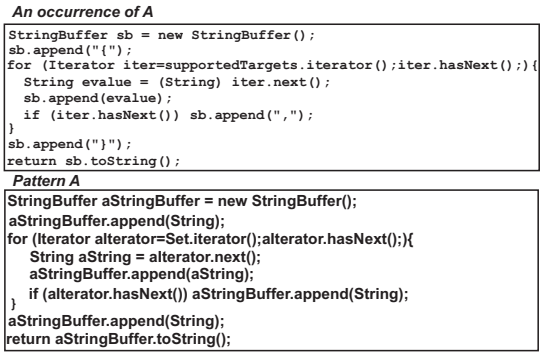

Figure 8: A Usage Pattern Mined from AspectJ

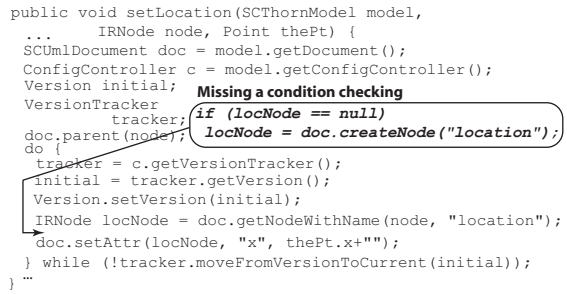

Figure 9: A Defect in Fluid: NullException

5.1.2 Other Experiments

Table 2 lists the results that GrouMiner ran on nine different open-source projects with the total of more than 3,840 patterns. It is impractical to examine all of them. We examined only a sample set of patterns and selected a set of interesting patterns as presented in Section 5.1.1.

The number of groums `#Gr` and the maximum groum sizes `Max_Gr` are very large. The number of method groums is smaller than that of methods due to abstract methods, interfaces, and the methods that do not involve objects. Table 2 shows that GrouMiner is quite efficient and can scale up to large graphs. The total size of graphs for AspectJ system is about 70,000 nodes. However, the pattern detection time is very reasonable (a few minutes for simple systems, to a half an hour and an hour for large/complex systems). The time depends more on the distribution nature of patterns and the graphs of each system, rather than its size. In Table 2, we counted the total number of distinct patterns and eliminated the patterns that are contained within others. The numbers of detected patterns with the sizes of 3 or more are about 44%-69% of the total numbers. This is also an advantage of GrouMiner over existing approaches, which focus on patterns of pairs or a set of pairs of method calls. Moreover, many GrouMiner's patterns are program-specific.

5.2 Anomaly Detection Evaluation

5.2.1 Case study: Fluid

The number of detected anomalies `#Ano` is in Table 2. The time reported in the table includes the time for mining the patterns, finding and ranking anomalies. We chose to examine all 64 reported anomalies for the Fluid project where we have the domain knowledge. We have found 5 defects that have not been yet discovered. Let us analyze them.

The first defect (Figure 9) is a violation of Pattern 2 in Figure 6. The defect occurs in `setLocation(SCThornModel,IRNode, Point)`. Developers did not check whether an `IRNode` with the name of "location" exists yet. If it does not, `setLocation` must create a new `IRNode` before setting the attribute values for it. GrouMiner detected this since Pattern 2 in Figure 6 contains an `if` statement after `SCUMLDocument.getNodeWithName`. The program crashed when it reached that method and no IRNode with the name of "location" existed yet.

Another defect occurs in `SCThornDiagramElementVersion.changeProperty` (Figure 10). The method violates the pattern of tracking the changes to the properties of a UML graphical element. It was supposed to check the existence of an `IRNode` with the name "Property" by calling `SCUMLDocument.getNodeWithName` before it called `createNode`. In this case, the defect did not cause a program to crash. However, it is harder to detect because document `doc` would have more than one property nodes, thus, creating a semantic error.

We also found three instances of the third defect in Fluid. They violate the following pattern: `if (IRNode.valueExists (IRAttr)) IRNode.getSlotValue(IRAttr)`. The pattern means that one must check the existence of an attribute before getting its value. Those three locations did not have the `if` expression and caused program errors.

In general, we had manually examined all 64 violations in Fluid and classified them into 1) defects (i.e. true bugs), 2) code smells (any program property that indicates something may go wrong), and 3) hints (i.e. code that could be improved for readability and understandability). We used the same classification as in JADET [4]. Among 64 anoma-

Project	#File	#Meth	#Gr	Max Gr	Avg Gr	# Pttn	Max Pttn	#Pattern at size 2	3-5	6-10	10+	# Ano	#Check	#Defect	#Cs	#Fapos	Time h:mm:ss
Ant 1.7.1	1123	12409	9573	153	6	697	17	317	315	62	3	145	15	1	0	14	0:22:14
Log4J 1.2.15	292	2479	1763	99	6	141	10	79	60	15	0	32	15	0	1	14	0:00:39
AspectJ 1.6.3	1500	14716	9818	332	7	1055	15	429	413	180	33	244	15	1	2	12	1:09:24
Axis 1.1	1127	7834	5355	425	8	614	16	251	258	100	5	145	15	0	2	13	0:12:23
Columba 1.4	799	5083	3024	185	7	219	7	118	94	7	0	40	15	1	0	14	0:00:33
jEdit 3.0	204	2274	1757	244	8	238	10	119	77	42	0	47	15	1	0	14	0:01:18
Jigsaw 2.0.5	701	6528	5073	152	6	443	11	197	204	41	1	115	15	1	1	13	0:26:34
Struts 1.2.6	365	3209	2412	107	5	198	8	62	114	22	0	33	15	0	0	15	0:01:19
Fluid VC12.05	229	3506	2477	115	6	236	14	92	94	46	4	64	64	5	8	40	0:08:43

Table 2: Details on Detected Patterns and Top-15 Anomalies of the Case Studies ($\sigma = 6$, $\delta = 0.1$)

```
public void changeProperty(SCThornModel model, ...){
 ...
 SCUmlDocument doc = model.getDocument();
 ...   IRNode propertyNode =
 ...      doc.getNodeWithName(node, "Property");
 ...   if (propertyNode == null)
 ...     propertyNode = doc.createNode("Property");
 do {
    tracker = c.getVersionTracker();
    initial = tracker.getVersion();
    version.setVersion(initial);   redundant object creation
    IRNode propertyNode = doc.createNode("Property");
    doc.setAttr(propertyNode, "name", name);
    doc.setAttr(propertyNode, "value", value);
    doc.addChild(node, propertyNode);
 } while (!tracker.moveFromVersionToCurrent(initial));
}...
```

Figure 10: A Semantic Error in Fluid

lies, there were 5 defects, 8 code smells (Cs), 11 hints, and 40 false positives. We confirmed the presented defects by running/testing the program. In this case study, the false positive rate is 62.5%. In [4], the reported false positive rate of JADET on AspectJ 1.5.3 was 87.8%. Currently, we use all discovered patterns for the detection. If they are presented to developers and only good patterns are kept, the false positive rate in GrouMiner will be even smaller. Among the top 10 anomalies in Fluid, 3 of them are defects, two are code smells, one is a hint, and 4 of them are false positives.

5.2.2 Other Experiments

In addition to Fluid, we also run anomaly detection on eight other systems (Table 2). We looked at Top 15 anomalies in each system and manually classified them. These case studies show that our graph-based ranking approach is successful. Among top 10 anomalies in Fluid, there are only 3 defects. But Top 15 anomalies contain all 5 defects. In addition to 5 defects found in Fluid, GrouMiner can reveal 5 more new defects in even mature software such as Ant, AspectJ, Columba, jEdit, and Jigsaw. All defects are both common and program-specific. Carefully examining those additional ones, we found that they are in the form of missing necessary steps in using the objects and missing condition and control structures. For example, in `PointcutRewriter.simplifyAnd()` in AspectJ, the use of `Iterator.next()` was not preceded by an `Iterator.hasNext()`.

Similarly, in the method `MapEntry.parseRestNCSA()` of Jigsaw 2.0.5, the call to a `StringTokenizer.nextToken()` was not preceded by a `StringTokenizer.hasNext()`. On the other hand, the usage of `ICloseableIterator` in the method `AbstractMessageFolder.recreateMessageFolderInfo` of Columba and `BufferReader` in the method `Registers.toString` of jEdit missed a `ICloseableIterator.close()` and a `BufferReader.close()`, respectively. Discovered patterns with all required steps enable the detection of those defects. They were all verified.

From	To	F+	F-	F*	O-	O+	Pat+	Ano	T(s)
-	r2020	220	0	0	-	-	205	54	44
r2020	r2030	1	0	30	1253	1316	9	28	9
r2030	r2040	0	0	13	386	380	3	3	2
r2040	r2050	6	0	40	1936	3227	16	38	20
r2050	r2060	6	1	25	2332	4072	12	30	25
r2060	r2070	6	1	27	2142	2806	18	25	15

Table 3: Pattern Update Result on jEdit revisions

5.3 Pattern Updating Evaluation

We run GrouMiner on several revisions of JEdit starting from revision 2020 (Table 3). The changes to the files such as the numbers of added (F+), deleted (F-), and modified files (F*) are provided by SVN repository. The changes to the occurrences (O+, O-), patterns (Pat+), and anomalies (Ano), and running time (T) are shown. The result shows that our tool can update new patterns and use them to detect anomalies in new revisions. The running time depends on the total number of changed files (i.e. F+, F-, and F*). We manually checked the new patterns and anomalies, and confirmed their high quality as in the separate executions.

6. RELATED WORK

There exist several methods for **mining temporal program behaviors**. The closest research to GrouMiner is JADET [4]. For each Java object in a method, JADET extracts a usage model in term of a finite state automaton (FSA) with anonymous states and transitions labeled with feasible method calls. The role of JADET's object usage model is similar in spirit to our *groum*. However, its model is built for a single object and does not contain control structures. GrouMiner's graphs represent the usage of multiple objects including their interactions, control flow and condition nodes among method calls. Another key difference is that GrouMiner performs frequent subgraph mining on object usage graphs to find graph-based patterns and then produce code skeletons. In contrast, from an FSA for a *single* object in a method, JADET uses frequent itemset mining to extracts a pattern in term of a set of pairs of method calls.

Dynamine [21] looks at the set of methods that were inserted between versions of a software to mine usage patterns. Each pattern is a pair of method calls. Engler *et al.* [10]'s approach is also limited to patterns of pairs of method calls. Thus, each pattern corresponds to an edge in a GrouMiner's pattern. Acharya *et al.* [1] mine API call patterns using a frequent closed partial order mining algorithm and express them in term of partial orders of API method calls. Their patterns do not have controls and conditions and do not handle multiple object usages. Williams and Hollingsworth [31]

mine method usage patterns in which one function is directly called before another. Chang *et al.* [5] use a maximal frequent subgraph mining algorithm to find patterns on condition nodes on PDGs. They considered only a small set of nodes in PDGs, and the patterns are only control points in a program. FindBugs [14] also looks for specified bug patterns. LtRules [20] builds possible API usage orders determined by a predefined template for given APIs.

PR-Miner [19] uses the frequent itemset mining technique to find the functions, variables, data types that frequently appear in same methods. No order of method calls is considered as in GrouMiner. CP-Miner [18] uses frequent subsequence mining to detect clone-related bugs. Some clone detection approaches applied graph-based techniques, but are limited in scalability [17]. BugMem [15] mines patterns of defects and fixes from the version history.

Given an API sample, XSnippet [27] provides example code of that API. In contrast, GrouMiner does not require a sample as an input and it detects anomalies. Similar tools include Prospector [22] and MAPO [32]. PARSEWeb [29] takes queries of the form "from source object type to destination object type" as an input, and suggests relevant method-invocation sequences as potential solutions. CodeWeb [23] detects patterns in term of associate rules among classes.

Another line of related research is **temporal specification mining**. Ammons *et al.* [3] observe execution traces and mine usage patterns in term of probabilistic FSAs. Shoham *et al.* [28] applied static inter-procedural analysis for mining API specifications in term of FSAs. Both approaches require the alphabet of an FSA specification to be known.

Gabel *et al.* [12] mine temporal properties between method calls in execution traces and express a specification as an automaton. However, their approach does not distinguish methods from different objects. Yang *et al.* [33] find behavioral patterns that fit into user-provided templates. Chronicler [25] uses inter-procedural analysis to find and detect violations of function precedence protocols. Kremenek *et al.* [16] use a factor graph, a probabilistic model, to mine API method calls. Some other approaches take as input a *single type* and derive the valid usage patterns as an FSA using static analysis or model checking [2, 13, 20].

Dallmeier *el al.* [6] analyze method call sequences between successful and failing executions to detect defects. Similarly, Fatta *et al.* [8] find frequent subtrees in the graphs of calls in passing and failing runs. Dickinson *et al.* [9] cluster bugs based on their profiles to find error patterns. Fugue [7] allows users to specify object typestates and then checks for code conformance. Weimer *et al.* [30] mine method pairs from exception control paths. In brief, those runtime approaches for mining can complement well to our GrouMiner.

7. CONCLUSIONS

The information on specific protocols among method calls of multiple interplaying objects is not always documented. This paper introduces GrouMiner, a novel graph-based approach for mining usage patterns for multiple objects. The advantages of GrouMiner include useful detected patterns with control and condition structures among method calls of objects, change-resilient and scalable pattern discovery and anomaly detection, and useful usage skeletons. Our empirical evaluation shows that GrouMiner is able to find interesting patterns and to detect yet undiscovered defects.

Acknowledgment. This project was funded in part by a grant from Vietnam Education Foundation (VEF) for the first author and by Litton Professorship for the fifth author.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ESEC-FSE'07*, pages 25–34. ACM, 2007.

[2] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *POPL*, ACM, 2005.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. In *POPL'02*, pages 4–16. ACM, 2002.

[4] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *ESEC/FSE'07*, pages 35–44. ACM, 2007.

[5] R-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, 2008.

[6] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight defect localization for Java. In *ECOOP'05*. Springer Verlag, 2005.

[7] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP'04*, LNCS 3086, pages 465-490. Springer Verlag, 2004.

[8] G. Di Fatta, S. Leue, and E. Stegantova. Discriminative pattern mining in software fault detection. In *SOQUA'06*. ACM, 2006.

[9] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. *ICSE'01*, IEEE, 2001.

[10] 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 *SOSP'01*, pages 57–72. ACM, 2001.

[11] Fluid project. http://www.fluid.cs.cmu.edu:8080/Fluid.

[12] M. Gabel and Z. Su. Javert: fully automatic mining of general temporal properties from dynamic traces. *FSE'08*, ACM, 2008.

[13] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *ESEC/FSE'05*, pages 31–40. ACM, 2005.

[14] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[15] S. Kim, K. Pan, and E. E. J. Whitehead, Jr. Memories of bug fixes. In *FSE'06*, pages 35–45. ACM, 2006.

[16] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief:inferring the specification within.*OSDI'06*.

[17] J. Krinke. Identifying similar code with program dependence graphs. In *WCRE'01*, pages 301-309. IEEE CS, 2001.

[18] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, 2006.

[19] Z. Li and Y. Zhou. PR-Miner: automatically extracting implicit programming rules and detecting violations in large software code. In *ESEC/FSE'05*, pages 306–315. ACM, 2005.

[20] C. Liu, E. Ye, and D. Richardson. Softw. library usage pattern extraction using a softw. model checker. *ASE'06*, IEEE, 2006.

[21] B. Livshits and T. Zimmermann. DynaMine: finding common error patterns by mining software revision histories. In *ESEC/FSE'05*, pages 296–305, ACM, 2005.

[22] D. Mandelin, L. Xu, R. Bodík, D. Kimelman. Jungloid mining: helping to navigate the API jungle. *PLDI'05*, ACM, 2005.

[23] A. Michail. Data mining library reuse patterns using generalized association rules. *ICSE'00*, pp 167–176. ACM, 2000.

[24] H. A. Nguyen, T. T. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Accurate and efficient structural characteristic feature extraction for clone detection. *FASE'09*, Springer, 2009.

[25] M. K. Ramanathan, A. Grama, S. Jagannathan. Path-sensitive inference of function precedence protocols. *ICSE'07*,IEEE,2007.

[26] R. Read and D. Corneil. The graph isomorphism disease. Journal of Graph Theory **1** (1977) 339–363

[27] N. Sahavechaphan and K. Claypool. XSnippet: mining for sample code. In *OOPSLA'06*, pages 413–430. ACM, 2006.

[28] S. Shoham, E. Yahav, S. Fink, M. Pistoia. Static specification mining using automata-based abstractions. *ISSTA*, ACM, 2007.

[29] S. Thummalapenta and T. Xie. ParseWeb: a programmer assistant for reusing source code on the Web. *ASE*, ACM, 2007.

[30] W. Weimer and G. C. Necula. Mining temporal specifications for error detection. In *In TACAS*, pages 461–476, 2005.

[31] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. on Software Engineering*, 31(6):466–480, 2005.

[32] T. Xie and J. Pei. MAPO: mining API usages from open source repositories. In *MSR'06*, pages 54–57. ACM, 2006.

[33] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: mining temporal API rules from imperfect traces. In *ICSE'06*, pages 282–291. ACM, 2006.