An Approach to Mining Call-Usage Patterns with Syntactic Context

Huzefa Kagdi[1], Michael L. Collard[2], and Jonathan I. Maletic[1]

[1]Department of Computer Science
Kent State University
Kent Ohio 44242
{hkagdi, jmaletic}@cs.kent.edu

[2]Department of Computer Science
The University of Akron
Akron Ohio 44325
collard@cs.kent.edu

ABSTRACT

An approach to mine frequently appearing ordered sets of function-call usages, taking into account their proximal control constructs (e.g., if-statements), in the source code is presented. These ordered sets are termed as call-usage patterns. Additionally, variant usages, such as those with missing or out of order calls, are automatically identified along with their specific contextual location. The approach uses lightweight source code analysis and frequent sequential pattern mining. The hypothesis is that these call-usage patterns embody latent programming rules that developers commonly reuse, for example standard usages of API calls. The variants are an indicator of future changes such as the elimination of non-standard usages and/or bugs.

Categories and Subject Descriptors

D.2.7. **Software Engineering**: Distribution, Maintenance, and Enhancement – documentation, enhancement, extensibility, version control.

General Terms

Management, Measurement, Documentation

Keywords

Mining Software Repositories, Call Usage Patterns

1. INTRODUCTION

A function-call-usage pattern is an ordered list or a set of function calls as they appear in the program text (i.e., source code). A commonly occurring example is an *open();* followed by a *close();*. Although many of these call-usage patterns are quite intuitive and may be common knowledge to developers, they are typically not well documented. That is, they form latent programming rules that seldom exist outside the minds of developers. Such rules have been found to be potentially useful for tasks such as identification of standard library/API usages and fault location [4-6]. Violations of these rules can be very difficult to uncover, report to issue-tracking systems, and fix unless the rules are explicitly documented.

Recently, researchers [4-6] have applied data-mining techniques, specifically frequent-pattern mining algorithms, to the problem of uncovering/discovering call-usage patterns from the source code of large systems. These techniques uncover rules of frequently occurring sets [4, 5] or ordered lists of calls [3] and are quite effective; however they typically result in a large number of false positives (i.e., reporting potential violations where none actually occur). Obviously, large numbers of false positives tend to alienate the users of such tools. But to more accurately identify faults one must apply fairly complex, and computationally expensive, static/dynamic analysis techniques [5]. On very large software systems this may not be feasible or practical.

Here, to improve on this previous work, we take into consideration the syntactical context of where the calls occur in the source code. This additional information is fed into the mining algorithm and reflected in the resulting rules. A good example of where a syntactic context is useful is the case of a particular function call *strcat()* always being guarded by another call *strcmp()*. So just having the call *strcmp()* occurring before the call *strcat()* is not enough and a conditional construct is also necessary for correct usage.

To be utilized effectively by developers the rules, by themselves, are not enough. We need to also know the exact location, i.e., function, statement, and specific calls of the potential violations. Unfortunately, this level of detail with regards to location is missing in the results of previous work. Techniques such as item-set mining may be able to give the specific function where a violation occurs but if this function is more than 10 lines of code, hunting down the location of the exact violation typically proves quite time consuming.

Our approach not only identifies the rules but also the exact location and the context of violations of these rules in the source code. For example, the call *strcmp()* is missing from the condition of the *if*-statement guarding the third call *strcat()* in the function *foo*. This enables developers to quickly assess the potential rule violation in a timelier manner.

The rest of the paper is organized as follows. In Section 2, the associated concepts of call-usage rules are defined. Section 3 presents the mining approach with examples in Section 4. Relevant related work is discussed in Section 5. Finally, we conclude in Section 6.

2. CALL-USAGE PATTERNS

A *call-usage pattern* in source code is minimally a list of function calls. These patterns can additionally include the ordering of the calls along with the syntactic context in which each call occurs. The proximal control structure, such as a surrounding *if* or a *while* statement, to a specific call forms its syntactic context. We also consider the specific ordering of calls used in a function definition. Here, we focus on the call-usage patterns in the

ASE'07, November 5–9, 2007, Atlanta, Georgia, USA.
ACM 978-1-59593-882-4/07/0011.

function definitions of a procedural language, more specifically *C*. The ordering of calls primarily refers to the lexical position with a syntactic construct in source code, and does not necessarily imply a runtime order or control flow of call execution.

Consider the function *f1* shown in Figure 1. In this function, the calls *a, b,* and *d* occur in a specific order $d \rightarrow a \rightarrow b$. The symbol $\rightarrow$ defines the specific order in which the call occurs. In addition, the calls *a* and *b* are enclosed in an *if* statement with the call *d* in the conditional guard. We add a special notation to the pattern that reflects the syntactic context of each call. The call-usage pattern in the function *f1* is represented as:

$$\{\text{<if><cond>}\,\boldsymbol{d}\,\text{</cond></if>}\} \rightarrow \{\text{<if\_cond="d">}\,\boldsymbol{a}\,\text{</if\_cond="d">}\} \rightarrow \{\text{<if\_cond="d">}\,\boldsymbol{b}\,\text{</if\_cond="d">}\}.$$

In this pattern, the call *d* occurs in a condition of an *if*-statement. It also occurs before ($\rightarrow$) the call *a*. The call *a* occurs in the body of an *if*-statement that has the call *d* in the condition, likewise for the call *b*. The details of this notation and how it is derived from the source code is given in Section 3.1.

The goal of the mining approach is to uncover rules of call-usages from the patterns of call-usages found in functions. The basic premise is that these patterns are representatives of the standard usage rules that are prevalent in a software system. Once these call-usage rules are discovered then violations of the rule can easily be identified. A violation occurs when a rule is not completely followed, for example, due to a missing call.

Occurrence of calls in a particular pattern in a single function does not necessarily suggest a call-usage rule. To uncover standard call-usage rules, we need to mine for frequently occurring call-usage patterns in the source code. To identify violations of rules, we must also identify sub-patterns of frequently-occurring patterns. Specifically, a *variant* is an order preserving proper sub-pattern of a frequent pattern that occurs by itself in the system but in far fewer numbers (e.g., one or two times). A variant may occur due to a missing call, calls made out of order, or a differing syntactic context of a call. Note that not all sub-patterns of a rule are variants. For example, sub-patterns that are always subsumed in a larger pattern are not reported as variants.

A *violation* is an instance of a variant occurring in a specific function. We believe that this representation of a violation assists developers in two complementary ways. First, if a violation is examined and confirmed as a valid misusage in one function, the developer might take a similar remedy for other functions violating the same variant. Secondly, a developer maintaining, restructuring, or testing a particular function may be interested in all the possible misuses (i.e., variants) that it contains.

We now detail our overall approach to automatically mine call-usage patterns, and their variants and violations.

3. THE APPROACH

Our mining approach consists of two major components: 1) Extracting call occurrences with their ordering information and proximal control constructs from source code; 2) Applying a data mining technique to extract ordered patterns and their violations.

3.1. Extraction of Calls with Context

Call occurrences along with their ordering information and surrounding control constructs, i.e., call-usage patterns, are extracted from the functions in the source code. Ordering among calls and their proximal syntactic context is based on their lexical positions. Calls within expressions and argument lists where there is non-determinism are given partial ordering. In Figure 1, functions *f2* and *f3* have the same partially ordered pattern *{a c}→{b}* due to non-determinism in the occurrence of calls *a* and *c* in the expression *c()+a()*. Functions *f1* and *f4* form totally-ordered patterns.

```
void f1(){
 if(d())
   b(x+a());
 …
}

void f3(){
 y=c()+a();
 b(y);
 f();
}
```

```
void f2(){
 if(d())
   b(c()+a());
  e();
}

void f4(){
  if(d()){
   y=a();
   b(CONST+y);
  }
 }
```

Figure 1. An example of four function definitions demonstrating patterns, variants, and violations

The call-extraction tool, namely *callextractor*, is implemented on our srcML platform [1]. srcML is an XML representation of source code that embeds syntactic information into the text. Syntactic structures such as function definitions, expressions, and function calls are explicitly marked using XML elements.

Source code of the entire system is converted into a single srcML file using the *src2srcml* translator (see www.sdml.info). The tool *callextractor* produces lists of calls with ordering information, one list from each function (i.e., a *transaction* in data mining terms, defined in Section 3.2) from the srcML file. These lists not only include the calls, but also the relevant syntactic context. Currently, *if*-statements and *while*-statements that have a call in a condition and/or guarding other calls are supported. As unprocessed source code is used, the macro calls used in functions are also included. Figure 2 gives the transactions derived from the functions shown in Figure 1. Each transaction is uniquely identified by the complete path followed by the function name. For example in Figure 2, the transaction for the function *f1* is identified by the path `pt/f1.c#f1`.

Each event in a transaction is identified by a number corresponding to the lexical order of a set of calls. The function *f1* in the source code file `pt/f1.c` consists of three events identified with labels *1*, *2*, and *3*. The call to the function *d*, in the if-conditional forms the first event. The calls *a* and *b* represent

```
pt/f1.c#f1 1 <if><cond>d</cond></if>
pt/f1.c#f1 2 <if_cond="d"> a </if_cond="d">
pt/f1.c#f1 3 <if_cond="d"> b </if_cond="d">
pt/f1.c#f2 1 <if><cond>d</cond></if>
pt/f2.c#f2 2 <if_cond="d"> a c </if_cond="d">
pt/f2.c#f2 3 <if_cond="d"> b </if_cond="d">
pt/f2.c#f2 4 e
pt/f3.c#f3 1 c a
pt/f3.c#f3 2 b
pt/f3.c#f3 3 f
pt/f4.c#f4 1 <if><cond>d</cond></if>
pt/f4.c#f4 2 <if_cond="d"> a </if_cond="d">
pt/f4.c#f4 3 <if_cond="d"> b </if_cond="d">
```

Figure 2. Partially ordered patterns produced from functions in Figure 1 in the form of a transaction.

events *2* and *3*, and both are within the body of an *if*-statement with the call to the function *d*.

A call in an event is represented by the function call name. The notation for the syntactic context is more complex. We markup all calls in an *if*-statement with the syntactic context of the call, i.e., which calls occur in the condition of the *if*-statement, by wrapping the name of the call by a start and end tag. In Figure 2, the call *a* in the second event is guarded by the call *d* in the condition of the if-statement and is marked as *<if_cond="d">a</if_cond="d">*. Calls in conditions are marked with full elements. The call *d* in the first event is in a condition (guarding the call *a*) and is marked as *<if><cond>d</cond></if>*. The same notation is used for the *if* and *while*-statements as both serve as guards to the body.

3.2. Mining Call-Usage Patterns

The specific data mining technique used in our approach is sequential-pattern mining. Sequential-pattern mining takes a given set of sequences that are composed of items and finds all the frequently occurring subsequences, i.e., ordered patterns, that have at least a user-specified minimum support. The input data to frequent-pattern mining algorithms are in the form of transactions. A transaction refers to a group of items that share a common property or occur in the same event (e.g., customer baskets or items checked-out together in the case of market-basket analysis). The number of transactions in which a pattern occurs is known as its *support*. If the support of a pattern is at least a user-specified *minimum support* then it is considered a *frequent* pattern

For our call-usage pattern mining, an individual transaction corresponds to the extracted calls of a single function in the form of Figure 2. Once the ordered patterns are uncovered via mining, sequence rules can be generated to uncover variants. A sequence rule is formed between a pair of ordered patterns such that one of them is a (order-preserving) subset of the other. The *confidence* (among other metrics such as *lift*) of a sequence rule is used to determine its strength. A sequence rule with a very high confidence, however not the maximum value of 1.0, is likely to contain a variant. A sequence rule with the value of 1.0 simply suggests that none of the sub-patterns of a pattern are used in isolation, and are always used as a part of that longer pattern.

We have developed a sequential pattern-mining tool, namely *sqminer*, that is based on the Sequential Pattern Discovery Algorithm (SPADE) [9]. SPADE utilizes an efficient enumeration of ordered patterns based on common-prefix subsequences and division of search space using equivalence classes. With regards to computational cost, sequential-pattern mining has to potentially consider a combinatorial explosion in the search space of all possible patterns of the order of $\Theta(2^{nm})$ with *m* partially ordered components each consisting of an average of *n* calls. Most of the frequent-pattern mining algorithms only report the support of a pattern but not the transactions (functions) in which a pattern occurs. Both functions with patterns and their variants are recorded and reported by our tools. Additionally, Our approach produces only *closed* patterns (i.e., patterns without any sub-pattern with an equal support of any larger pattern).

An additional benefit of mining closed patterns is the reduction in the number of rules. As the subsets of a pattern that have the same support are pruned, rules are only formed with sub-patterns that have higher support values. The confidence of such rules will be always less than one. Therefore, rules with confidence less than one are only formed and examined for inferring candidate variants of both ordered and unordered patterns. For the example in Figure 1, the singleton pattern *{a}* has a support of four as it occurs in all the functions *f1*, *f2*, *f3*, and *f4*. This pattern occurs in the same number of functions as the (larger) pattern *{a}→{b}* and is therefore pruned. However, the same pattern *{a}→{b}* that is a sub-pattern of *{a c}→{b}* is retained as their supports are different (four and two respectively).

4. EXAMPLES

We first demonstrate ordered pattern mining with the help of a synthetic example. Then we give specific examples uncovered from *Apache httpd* (v.2.0.55). Consider again a hypothetical system with four functions as shown Figure 1 and their extracted transactions in Figure 2. We will use a minimum support of two for a candidate pattern, i.e., at least two functions must contain the pattern. A minimum confidence of 0.75 is chosen for a sequence rule, i.e., at most 25% of the functions contain only the pattern variant. Sequential-pattern mining reports three ordered patterns:

1. *{a c}→{b},*
2. *{<if><cond>d</cond></if>}→{<if_cond="d">a</if_cond="d">}→{<if_cond="d"> b </if_cond="d">}*, and
3. *{a}→{b}*

The first, second, and third patterns occur with a corresponding support value of two, three, and four. The first pattern is partially ordered, whereas the other two are totally ordered. The lack of ordering in the first pattern is due to the calls *a* and *c* occurring in the same expression. The second pattern contains a combination of calls and a conditional construct, whereas, others contain only calls. From these patterns two sequence rules are possible,

1. *{a}→{b}⇒{<if><cond>d</cond></if>}→{<if_cond="d">a</if_cond="d">}→{<if_cond="d">b</if_cond="d">} with the confidence of 0.75*
2. *{a}→{b}⇒{a c}→{b} with the confidence of 0.5*

Only the first rule satisfies the required minimum confidence. As a result the pattern *{a}→{b}* is reported as a variant of the pattern *{<if><cond>d</cond></if>}→{<if_cond="d">a</if_cond="d">}→{<if_cond="d"> b </if_cond="d">}* in the function *f3*. Arguably, this case appears to make sense as the implementation of the function *f3* is very similar to that of *f2*. So we have the call-usage rule *{<if><cond>d</cond></if>}→{<if_cond="d">a</if_cond="d">}→{<if_cond="d"> b </if_cond="d">}* that is obeyed in the functions *f1*, *f2*, and *f4*, the variant *{a}→{b},* and the violation (*{a}→{b}, f3*). In this case, the second rule does not report the sub-pattern *{a}→{b}* as a variant of the pattern *{a c}→{b}*. However, this case leads to another interesting issue. Had the externally controlled value of minimum confidence been set to 0.5, the second rule would also report the sub-pattern *{a}→{b}* as a variant of the pattern *{a c}→{b}*. Thus, the sub-pattern would have been suggested as a variant of two rules. In our approach, variants and violations are ranked according to the descending order of confidences. Thus, the variant and violation of the first rule would have been reported before that of the second rule.

We have applied sequential-pattern mining on the *Apache httpd* v2.0.55 system. Below are some of the call-usage rules, variants, and violations with syntactic context uncovered from this system with a minimum support of 10 and minimum confidence of 0.9:

1. {<if><cond>*strncasecmp*</cond></if>} → {*apr_pstrcat*} occurs in 12 functions
2. {<if><cond>*apr_file_open*</cond></if>} → {*apr_file_close*} occurs in 27 functions
3. {*apr_brigade_create*} → {<if><cond>*APR_BUCKET_IS_EOS*</cond></if>} → {*apr_bucket_read*} occurs in 11 functions

Most of these patterns are self-explanatory and are representative of common programming practice or idioms.

The variant

{<if><cond>*ap_xml_parse_input*</cond></if>} → {*ap_log_rerror*} → {*dav_push_error*}

occurs due to the sequence rule

{<if><cond>*ap_xml_parse_input*</cond></if>}→{*ap_log_rerror*}→ *dav_push_error*} ⇒ {<if><cond>*ap_xml_parse_input*</cond></if>}→ {*ap_log_rerror*}→{*dav_handle_err*} → {*dav_push_error*}

with the confidence 0.90. The sub-pattern of this rule occurs as a variant only in the violating function *'modules/dav/main/mod_dav.c#dav_method_label'*, and as a part of the call-usage rule in all other functions.

5. RELATED WORK

We discuss the work related to the problem of finding usage patterns via frequent-pattern mining techniques. This discussion is by no means exhaustive but does represent a number of different investigations.

Michail [6] presented an approach based on itemset and association-rule mining to uncover entities such as components, classes, and functions that occur frequently together in library usages. Similar to the work presented here, Li et al. [4] addresses the question of extracting rules and violations of typical usages of function calls in a system. Their approach is based on itemset mining. Their call extraction uses the *gcc* front end, whereas, our call-extracting mechanism is based on the language standards and decoupled from a specific compiler implementation.

Livshits and Zimmermann [5] present an approach based on itemset mining for discovering call-usage patterns from source-code versions. Williams et al. [7] analyzed usages of function-return values for detecting software bugs via static analysis of a single version and evolutionary changes. A number of researchers used a combination of static and dynamic analyses, and finite state automaton to infer usage patterns and program properties (refer to our previous work [3] for complete citations).

Xie et al. [8] used sequence mining to filter the results of a source-code search tool to report API-usage patterns in which a source-code entity is used. However, a sequence-mining approach has not been used before, with the exception of us, for function-call usage patterns discovery. However, we did not consider the syntactic context of calls then. Recently, Kagdi et al. [2] surveyed approaches, including those using frequent-pattern mining, to mining software repositories for various software evolution tasks.

6. CONCLUSIONS AND FUTURE WORK

The principal contribution of the work presented here is specifically the use of syntactic context of function calls in source code to automatically construct rules that embody latent programming idioms, practice, and function usage. No other work to date has used syntactic information in the mining or construction of sequence rules. This is a substantial step forward from our previous work on this topic.

In future, we plan to validate the usefulness of our mined call-usage patterns for software evolution tasks. One promising source for validation is the source code change (version) history. We are currently extending our approach to include other types of syntactic constructs for call context. This includes the detection of conditional guards to higher levels than the proximal statements. Another work in progress is mining call-usage rules from source cod in other languages such as C++ and Java.

7. REFERENCES

[1] Collard, M. L., Kagdi, H. H., and Maletic, J. I. An XML-Based Lightweight C++ Fact Extractor in Proceedings of 11th IEEE International Workshop on Program Comprehension (IWPC'03) (Portland, OR, May 10-11, 2003), 134-143.

[2] Kagdi, H., Collard, M. L., and Maletic , J. I. A Survey and Taxonomy of Approaches for Mining Software Repositories in the Context of Software Evolution. Journal of Software Maintenance and Evolution: Research and Practice, 19, 2 (March/April 2007), 77-131.

[3] Kagdi, H., Collard, M. L., and Maletic, J. I. Comparing Approaches to Mining Source Code for Call-Usage Patterns in Proceedings of 4th International Workshop on Mining Software Repositories (MSR'07) (Minneapolis, MN, May 19-20, 2007), 8 pages.

[4] Li, Z. and Zhou, Y. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code in Proceedings of 13th International Symposium on Foundations of Software Engineering (FSE'05) (Lisbon, Portugal, September 2005), 306-315

[5] Livshits, B. and Zimmermann, T. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories in Proceedings of 13th International Symposium on Foundations of Software Engineering (FSE'05) (Lisbon, Portugal, September, 2005), 296-305

[6] Michail, A. Data Mining Library Reuse Patterns Using Generalized Association Rules in Proceedings of 22nd International Conference on Software Engineering (ICSE'00) (Limerick, Ireland, June 4-11, 2000), 167-176.

[7] Williams, C. C. and Hollingsworth, J. K. Recovering System Specific Rules from Software Repositories in Proceedings of 2nd International Workshop on Mining Software Repositories (MSR'05) (St. Louis, Missouri 2005), 7-11

[8] Xie, T. and Pei, J. MAPO: Mining API Usages from Open Source Repositories in Proceedings of 3rd International Workshop on Mining Software Repositories (MSR'06) (Shanghai, China, May 22-23, 2006), 54-57.

[9] Zaki, M. J. SPADE: An Efficient Algorithm for Mining Frequent Sequences. Machine Learning, 42, 1-2 (January 2001), 31-60.