

2009

PACIFIC NW SOFTWARE  
QUALITY  
CONFERENCE



MOVING  
QUALITY  
FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE  
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

# An Empirical Study of Data Mining Code Defect Patterns in Large Software Repositories

Kingsum Chow, Xuezhi Xing, Zhongming Wu and Zhidong Yu  
Software and Services Group, Intel Corporation  
<mailto:{kingsum.chow, xuezhi.xing, zhongming.wu, zhidong.yu}@intel.com>

## Biography

Kingsum Chow is a principal engineer from the Intel Software and Services Group (SSG). He has been working for Intel since receiving his Ph.D. in Computer Science and Engineering from the University of Washington in 1996. He has published more than 30 technical papers and patents.

Xuezhi Xing is a software engineer intern. His research work includes software maintenance, software quality and software engineering paradigms. He is pursuing the Ph.D. degree in University of Science and Technology of China.

Zhongming Wu is a software engineer intern. He is pursuing a MS degree in Computer Science and Engineering from Shanghai Jiao Tong University. He is interested in information security and related research.

Zhidong Yu is a software engineer in the Intel SSG. Zhidong has been working on performance analysis of enterprise Java applications and server virtualization software. He received his MS degree from Shanghai Jiao Tong University in 2001.

## Abstract

There has been a growing interest in mining software code defect patterns and using this knowledge to identify potential problems [2, 8-15]. To understand the benefits of such methods, we applied them to several large software repositories. We learned the effectiveness and the limitations of applying these methods. These methods are called “static analysis” as they analyze the code for defects without execution. They are different from the traditional static analysis as they apply data mining in the analysis, while the traditional static analysis uses program analysis, such as data flow analysis and control flow analysis. There is a trend to combine the data mining methods and program analysis techniques to gain more effective results.

Many tools are available to detect software defects. But if the tools have no knowledge about what to check they can't find defects [5]. There are common defect patterns such as buffer overflow, null pointer dereference and several tools address these patterns such as Purify [19], FindBugs [20]. However, application specific defects can be difficult to find, probably because there are few common patterns among different applications. Therefore some approaches, such as DynaMine [2], try to explore the patterns of specific applications automatically, while others try to provide description based methods to describe these patterns as assertion statements. Examples of tools and methods that fall into this later category are contract programming, AOP (Aspect Oriented Programming), PQL [3], and Metal [4]. The automatic approaches such as DynaMine may have difficulty in offering a good solution to explore complex patterns. The description based approaches such as PQL are often powerful at describing patterns, but they require manually constructing the specifications. Such tasks can be overwhelming [5].

In this paper, we dive into industry-level projects such as Harmony [16] an open source virtual machine to analyze and summarize their defect patterns using different pattern analysis methods. We gain some insights into the classifications of common code defect patterns. We make use of data mining methods to extract usage patterns automatically from the source code of different projects. We also look into the issues tracking system and revision history to analyze and summarize patterns. We will apply these tools and methods to detect pattern-related defects in new source code as well. Several methods are evaluated for their effectiveness to detect defect patterns.

The contributions of this paper are: (1) an empirical study to evaluate the effectiveness of data mining software code defects using a few large software repositories, and (2) insights into the characteristics of the software systems and the common code defect patterns.

## 1 Introduction

Defect patterns are important for defect-finding tools. Without such patterns, defect-finding tools cannot find bugs [5]. The use of such tools is valuable for initial software development and maintenance. Recently much attention has been paid to find defect patterns, but defect patterns may be unique and complex. Often only the programmer, who has the knowledge and experience with the code, can summarize the defect patterns. This process is often costly and not scalable. Software code repositories can often reflect the programmer's knowledge and experience. Some researchers try to mine this knowledge or experience to aid software development and maintenance using data mining methods. Research in identification of defect patterns has emerged [2, 8, 9, 10, 11, 12, 13 14]. Still, there is a gap between the capability of data mining methods and the complexity of defect patterns. There are two questions: (1) how large is the gap between them, (2) what kinds of defect patterns can current data mining methods deal with.

To answer both questions, we downloaded 51 Java projects from Apache [22] to study the defect patterns. We picked one of the Apache projects, Harmony [16], to get some insight into the characteristics of software defect patterns. The aim of Harmony, an open source project, is to produce an independent and compatible implementation of the Java Standard Edition 5 JDK under the Apache License. It also includes a community-developed modular runtime architecture. It has more than 1.25 million lines of code. Using Harmony as a case study has an added advantage in that it contains both C/C++ and Java code for us to study. We classified code defects and evaluated some data mining methods, such as Apriori [6], that are used to detect them.

While we provide a detailed defect classification for the Harmony project, the manual classification process was not repeated for the other projects. Instead, we developed a tool set to apply data mining methods to all the Java projects. By combining both stages of our work, we found the following:

1. Project configuration and condition checking related defects constitute about half of all the defects;
2. Current data mining methods may only detect a small portion of the defects
3. More complicated defect patterns can be detected by combining data mining methods and program analysis techniques.

## 2 Software Defect Classification and Analysis

We manually analyzed the code defects for the Apache Harmony project. We found both the orthogonal defect classification [25] and the root cause defect analysis [26] helpful. However, we could not employ these methods directly because we did not have access to the entire development process. In our manual analysis, we found the following defect classes.

- Requirements Engineering (RM)
- Value Overflow (VO)
- Project Configuration (PC)
- Conditions Checking (CC)
  - Neglected Conditions (NC)
- Function Calls
  - Deprecated Functions (DF)
  - Missed Functions (MF)
    - ◆ Function Call Pairs (CP)
    - ◆ Function Call Sequence (CSq)
    - ◆ Function Call Structures (CSt)
  - Extra Functions (EF)
- Data Usage
  - Value block (VB)
  - Value concurrency (VC)
  - Value error(VE)

### 2.1 Requirements Engineering (RM)

Some defects are caused by misunderstanding of requirements, or a change of requirements. These kinds of defects by nature cannot be easily avoided by code-based approaches alone.

## 2.2 Value Overflow (VO)

A value overflow includes integer overflow and buffer overflow. Insufficient checking of this may cause bugs. It may be impractical to check every possibility of a value overflow because of the potential performance impact. An example containing the problem and its fix (e.g., Harmony issue #6204 [16], is given in Figure 1:

```
private int computeElementArraySize() {  
-   return (int) (((long) threshold * 10000) / loadFactor) * 2;  
+   int arraySize = (int) (((long) threshold * 10000) / loadFactor) * 2;  
+   return arraySize < 0 ? -arraySize : arraySize;  
}
```

Figure 1 A value overflow defect example: Harmony Issue 6204

The symbol “-“ means a deleted line in patch and “+“ means an added line in patch. Examples in the rest of this paper follow this notation. The above cast from `long` to `int` type in deleted line leads to overflow and a negative `arraySize` if threshold is too big. In Harmony, if `NegativeArraySizeException` occurs, it is often the result of this kind of defect. Just this defect class results in 15 defects in the Harmony project.

## 2.3 Project Configuration (PC)

Project management defects include files or libraries that are not synchronized with respect to a change. This kind of defect often involves non-source code files; such as ant build files, libs, jars, and dlls. For example in Harmony-795 [13], the build process is broken because of the omission of “`serializer.jar`” in the ant class path. In the Harmony VM module, this kind of defect accounts for 15% (3/19) of all bugs at the time of writing this paper.

## 2.4 Condition Checking (CC)

A condition checking defect occurs when some needed condition is not checked correctly. It can be described like this: “Always check conditions in set C before/after A”. A common example is the need to check the validity of a pointer before its use. An example containing the problem and its fix is given in Figure 2.

```
static void wait_finalization_end(void) {  
    hymutex_lock(&fin_thread_info->end_mutex);  
-   while(unsigned int fin_obj_num = vm_get_finalizable_objects_quantity()){  
+   unsigned int fin_obj_num = vm_get_finalizable_objects_quantity();  
+   while(fin_thread_info->working_thread_num || fin_obj_num){  
+       .....  
+       fin_obj_num = vm_get_finalizable_objects_quantity();  
    }  
    hymutex_unlock(&fin_thread_info->end_mutex);  
}
```

Figure 2 A condition checking defect example: Harmony Issue 3671

In Figure 2, before the wait action is performed, the conditions

```
(fin_thread_info->working_thread_num || fin_obj_num)
```

must be checked.

Neglected Conditions (NC) [8, 9] can be regarded as a subclass of this pattern. Neglected conditions refer to (1) missing conditions that check the receiver or arguments of an API call before the API call or (2) missing conditions that check the return values or receiver of an API call after the API call [8]. A recent

study conducted by Chang et al. [9] shows that 66% (109/167) of defect fixes applied in the Mozilla Firefox project are due to neglected conditions.

## 2.5 Function Call Usage

Function calls are basic elements in writing programs. There are often some specific patterns of function calls. If such patterns are violated, it often results in defect. Several examples are given below.

### 2.5.1 Deprecated Functions (DF)

Some deprecated functions may cause defects. The designer wants to prevent this kind of function from being called. There are two patterns of this kind, 1) In situation S, call A rather than B; 2) always call A rather than B. Chow and Notkin [1] describe a semi-automatic approach to handling deprecated function calls effectively in the client programs if the library maintainer can specify the formal language proposed in that paper. In the defect fix patches, one-line changes with a function call often relate to this kind of pattern. An example is given in Figure 3.

```
Finref_Metadata *metadata = gc->finref_metadata;
- metadata->finalizable_obj_set = pool_get_entry(metadata->free_pool);
+ metadata->finalizable_obj_set = finref_get_free_block(gc);
```

Figure 3 A deprecated function defect example: Harmony Issue 4278

The function `pool_get_entry` is a stack-pop-like function. When this function is applied to the `free_pool` field of `Finref_Metadata` data, it should be replaced by the function `finref_get_free_block(gc)`. Besides getting an entry from

```
gc->finref_metadata->free_pool,
```

the new function does some extra work. This is a very common situation; at a function call, we find that the old function does not deal with some functionality needed by our program, and so we have to wrap or replace the old function call with a new function call. This produces the program pattern: when needing to call an old (deprecated) function, call the new function instead.

### 2.5.2 Missed Function Calls (MF)

In some specified situations, like handling an error, specific functions must be called. The pattern can be described like "In situation S, always call A". An example is given in Figure 4.

```
if (bytesRead == 0){
+   portLibrary->error_set_last_error (portLibrary, 0, 0);
   return -1;
}
```

Figure 4 A missed function call defect example: Harmony Issue 4017

The case "`bytesRead == 0`" is the error scenario; using the error process function is missed in the older version.

The recognition of a specified situation is the key step in characterizing this kind of defect, like above defect Harmony 4071. In the current literature, most of the data mining methods focus on the recognition of specified situations which involve function calls. For example: function call pairs, call sequence, and call structures are covered in the following sections.

#### 2.5.2.1 Function Call Pairs (CP)

Some function pairs, like `malloc` and `free`, need to be called together. Also some application-specific function pairs may exist, e.g., `addListener` and `removeListener` in DynaMine [2]. There may be defects if these pairs are not called together.

### 2.5.2.2 Function Call Sequence (CSq)

Some sequences of function calls need to be called in order. Some researchers [11, 12] have focused on mining such sequences. MAPO [11] mines the API call sequence pattern using BIDE [24] algorithms from open source projects. Mithun Acharya [12] uses partial order relation to mine the API sequence pattern.

### 2.5.2.3 Function Call Struc (CSt)

There are also more complicated patterns having to do with function calls and some structures are needed to address this kind of pattern. For example Huzefa Kagdi, et al. [13] take into account program's control constructs. They encode function calls with condition statement label. For example, if function **D** is called in an "if" condition statements as `if(D());`; this call is encoded by `<if><condition>D</condition></if>`. If function **A** is called inside the `if(D())` block, it's encoded by `<if_cond="D">A</if_cond="D">`. They developed *spinner* [13] to mine the frequent sequence pattern from the encoded function calls.

### 2.5.3 Extra Functions (EF)

Similar to missed function calls, some functions must not be called in some specific situations. It can be described like "In situation S, don't call A". An example is given in Figure 5.

```
public XMLDecoder(InputStream inputStream, Object owner, ExceptionListener
listener, ClassLoader cl) {
    .....
-   try {
-       SAXParserFactory.newInstance().newSAXParser().parse(inputStream,
           new SAXHandler());
-   } catch (Exception e) { this.listener.exceptionThrown(e); }
}
```

**Figure 5 An extra function defect example: Harmony Issue 6015**

The `parse` function call in Figure 5 is an extra function. The fix in `XMLDecoder`'s constructor is to remove it.

## 2.6 Data Usage

There are often some specific constraints between data, especially data members of the same object in Object Oriented programming languages. Such data needs to be modified in a specified pattern to uphold the constraints. If constraints are not implemented, defects often show up. An example is given in Figure 6.

### 2.6.1 Value Block (VB)

Some data values are closely related to each other. We call these data a value block. If one datum is not initialized or set correctly, there may be some errors. It is best illustrated by an example in Figure 6.

```
public int read() throws IOException {
    synchronized (lock) {
        .....
        if (pos < count || fillbuf() != -1)
            return buf[pos++];
+   markpos = -1;
        return -1;}
}
```

**Figure 6 A value block defect example: Harmony Issue 6110**

From the patch in Figure 6, we can see that the root cause of this defect is forgetting to set data `markpos` to -1. The data `pos` and `markpos` are two data members of the same class. If `pos` is greater or equal to `count`, `markpos` needs to be -1. The constraints on `pos` and the value returned by `fillbuf()` ensure that `markpos` is set correctly.

### 2.6.2 Value Concurrency (VC)

Unaware concurrency may cause related values to be inconsistent. Two kinds of this defect are unexpected concurrency on non-shared values and missing concurrency protection on shared values. An example is given in Figure 7.

```

- private static String name = null;
+ private String name = null;
```

**Figure 7 A value concurrency defect example: Harmony Issue 5978**

The value `name` is changed to be an instance attribute of a class. By doing this, unexpected concurrency on `name` can be avoided.

### 2.6.3 Value Error (VE)

An incorrect value causes a wrong behavior in the program. An example is given in Figure 8

```

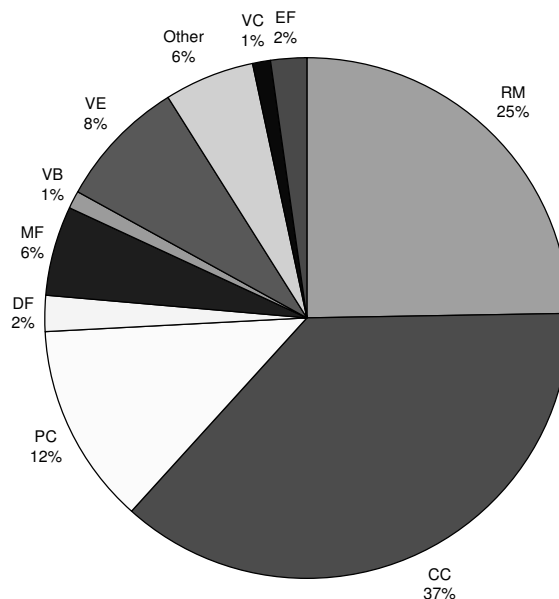
- factory.createURLStreamHandler(protocol);
+ factory.createURLStreamHandler("jar");
```

**Figure 8 A value error defect example: Harmony Issue 6059**

The value of parameter should be `jar`.

## 2.7 Analysis

We analyzed all the 89 closed Harmony defects that were documented between May 1, 2008 and May 1, 2009. The distribution and types of defects are presented in Figure 9.



**Figure 9 Distribution of defects in Harmony from May 2008 to May 2009.**

From Figure 9, we can see that the condition checking defects are most common. The project configuration related defects also have a serious impact, as they constitute 12% of all the defects. Project configuration and condition checking related defects make up nearly half of the total. Note that 25% of the defects are requirement defects and are beyond the scope of this paper.

90% of the defects are classified as being a major loss of functionality in the issue tracking system. All 89 defects cost 1947 days to fix. In terms of the total number of days to fix defects, the condition checking and project configuration defects constitute 38% and 14% respectively, roughly in proportion to the distribution of defects.

### 3 Mining Software Defects

As manual classification and detection of bugs does not scale to a large set of projects, we turn to data mining approaches such as the rule based analysis and the machine learning approach. In rule-based analysis [20, 23], programmers specify error patterns and coding rules manually, and then a software tool scans the source to check against them. The accuracy is comparably high and scanning speed is fast. However, as more libraries have been used, it has become difficult to specify rules because of the complexity and the quantity of the libraries. The machine learning approach [2] (MLA) uses data mining techniques to learn from the existing source code, and extract the knowledge in the form of coding patterns. This approach can characterize new patterns from newly created libraries automatically. However, MLA requires a large amount of training data.

Our approach mines usage patterns for specific library APIs (application programming interfaces) and locates the potential defects in software projects. An API in a library is designed to provide the client programs multiple working scenarios but an API used in unexpected scenarios may have unexpected results. Mining a huge amount of source code should enable us to extract the working scenarios, which we call the API usage patterns. As most APIs in released software are used correctly, the knowledge extracted is useful for detecting misused scenarios. This process is like extracting the knowledge from many experienced programmers to help the entire programming community. When pieces of source code violating the API usage patterns are found, the code should be checked for defects. We hope the process of detecting potential defects as described here can complement the current testing process practiced in the software assurance community.

#### 3.1 System Overview

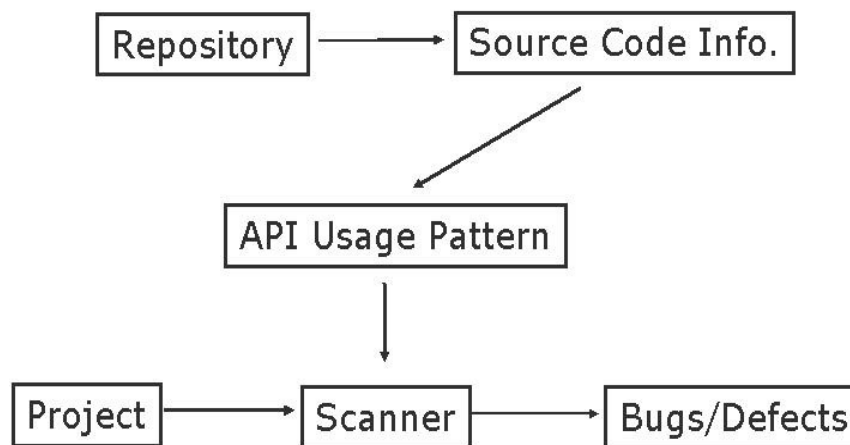


Figure 10 System Overview

Our approach in finding defects through API patterns discovery is given in Figure 10. First source files are extracted from a software repository. Then API usage patterns are examined by examining the method calls within a method. For example `java.util.Iterator.next()` and `java.util.iterator.hasNext()` appear in the same function together frequently. That implies that

these two methods being called together are considered to be a usage pattern. High confidence usage patterns are selected and we scan the source code with these patterns. The scanner flags potential defects when the source code violates the patterns.

By collecting data from publicly available source code repositories, e.g., sourceforge.net, we can decompose the code into 'transactions'. A usage pattern may hide in each transaction. Different kinds of learning approaches, like association rule learning and statistical classification, can be used to mine the API usage patterns from these projects. After scanning each transaction by usage pattern, we can locate the violations in the source code and inform the programmers.

### 3.2 Mining API Usage Patterns

Software developers are encouraged to reuse existing libraries through their public APIs. However, misunderstanding between the library documents and the programmers may lead to software defects. Some research [2, 11, 12, 15] has been done to understand the API usage and lower the risk of defects made by misusing the libraries. To evaluate the feasibility of data mining software defects, we mined a large number of Java projects and characterized their usage patterns. We adopted the association rule learning (ARL) in our approach as it is a well-established method for discovering the relationships between variables in a large set of transactions.

ARL was introduced by Agrawal [17] to mine the item relations in a database. The technique is composed of two steps. The first step is to find the item sets with a number of occurrences exceeding a threshold (called minimum support). The second step is to generate these association rules from these frequent item sets where the confidence exceeds the threshold. For example, if there are six transactions: {1, 2, 3, 4}, {2, 3, 4}, {2, 3, 4}, {1, 2, 4}, {2, 4} and {2, 3}, and the minimum support is 3, then we can find the following frequent sets: {2}, {3}, {4}, {2, 3}, {2, 4}, {3, 4} and {2, 3, 4}.

After frequent item sets are mined, the association rules can be generated. An association rule can be denoted as  $\mathbf{A} \Rightarrow \mathbf{B}$  with confidence  $c$  and support  $d$ , where  $\mathbf{A}$  and  $\mathbf{B}$  are mutually exclusive subsets from a frequent item set. The support  $d$  is the occurrence of  $\mathbf{A} \cup \mathbf{B}$ . The higher the support value, the more common the usage pattern is found in the source. The confidence  $c$  is the probability that item set  $\mathbf{B}$  occurs following the item set  $\mathbf{A}$ . Note that it is possible that a pattern with high support and low confidence, for example  $\mathbf{A}$  is a widely used API while  $\mathbf{B}$  isn't, the support of  $\mathbf{A} \Rightarrow \mathbf{B}$  may be high, but the probability that  $\mathbf{B}$  following  $\mathbf{A}$  is low.

In the association rule generation step, the confidence value is obtained by the ratio of  $\mathbf{A} \cup \mathbf{B}$ 's support and  $\mathbf{A}$ 's support ( $\text{conf}_{\mathbf{A} \Rightarrow \mathbf{B}} = \text{sup}(\mathbf{A} \cup \mathbf{B}) / \text{sup}(\mathbf{A})$ ). In the origin ARL algorithm such as Apriori [6], the size of suffix set in the rule is arbitrary. But we only generate those association rules  $\mathbf{A} \Rightarrow \mathbf{B}$  for which subset  $\mathbf{B}$  contains only one element. We can do that because the rule is used to detect violation in the form: " $\mathbf{A}$  occurs while  $\mathbf{B}$  doesn't". This improvement reduces the time complexity from  $2^n$  to  $n$ , where  $n$  is the size of a frequent set.

For example, suppose  $\mathbf{B} = \{\mathbf{b}_1, \mathbf{b}_2 \dots \mathbf{b}_n\}$ . Source code violates the rule  $\mathbf{A} \Rightarrow \mathbf{B}$  if and only if source code violates at least one of the following rules:  $\mathbf{A} \Rightarrow \{\mathbf{b}_1\}$ ,  $\mathbf{A} \Rightarrow \{\mathbf{b}_2\} \dots \mathbf{A} \Rightarrow \{\mathbf{b}_n\}$ . Thus detecting violation of the single-suffix rule is completeness.

Usually only association rules with high confidence are kept. However, considering that an API may have several usage patterns, the association rules with low confidence may be useful to our bug detecting. If a rule with the same prefix whose confidence sum is greater than a desired threshold is found, this rule is kept as well.

For example, both  $\mathbf{A} \Rightarrow \{\mathbf{b}_1\}$  and  $\mathbf{A} \Rightarrow \{\mathbf{b}_2\}$  have a confidence 0.45. The confidence sum of these two rules is 0.9, which is a high confidence. Then these two rules are kept in the form  $\mathbf{A} \Rightarrow \mathbf{b}_1 | \mathbf{b}_2$ . This rule can be interpreted as 'when  $\mathbf{A}$  occurs, at least one of  $\mathbf{b}_1$  and  $\mathbf{b}_2$  occurs', which means the two use a scenario of the API.

```

public void write(byte[] outText) throws IOException{
    File f=new File("foo.txt");
    System.out.println("foo.txt");
    if(!f.exists()){
        FileOutputStream out=new FileOutputStream(f);
        out.write(outText);
        out.close();
    }else{
        System.out.println("File exists");
    }
}
}

```

**Figure 11 A write method example in an ARL transaction.**

We consider each member function of a transaction considered in the ARL approach. We ignore the rest of the source code information such as control flow, invoke dependency and parameters of the API. For example, the `write()` method in Figure 11 is converted to the transaction in Table 1.

**Table 1 An ARL transaction derived from the write method in Figure 11**

ID	API
0	java.io.File.File()
1	java.io.File.exists()
2	java.io.FileOutputStream.FileOutputStream()
3	java.io.FileOutputStream.write()
4	java.io.FileOutputStream.close()
5	System.out.println()

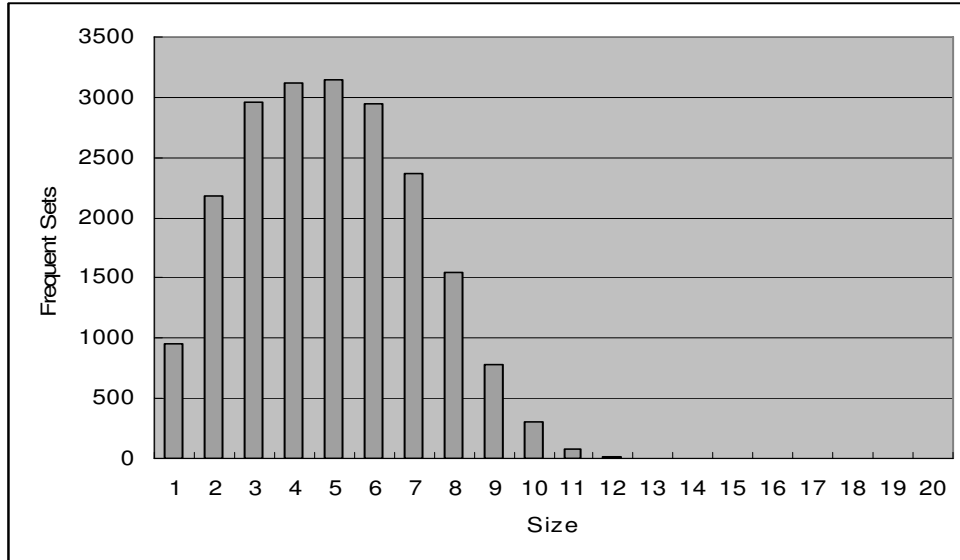
### 3.3 Experiment Results

We performed the experiments on the Apache software repository [22], a large open source application repository. A crawler was introduced to download 51 Java projects for our experiments. We arbitrarily divided them into two data sets, a small one containing 15 projects, e.g. Abdera, ActiveMQ and others and a large one containing 36 projects, e.g., Derby, Excalibur and others. We employed two data sets, as that would allow us to gain some insight into the impact from the data size. The number of lines of code (LOC), the number of source files, the number of Java classes, the total source file size in bytes and number of methods for each data set are given in Table 2. For the purpose within the scope of this paper, i.e., to discover patterns across projects, we focused on the library calls to `java.*` and `javax.*`. We discarded those methods that invoke `java.*` or `javax.*` just once.

**Table 2 Summary of data set**

Data Set	LOC	Source files	Java Classes	File size (bytes)	Number of methods
Small	4,119,645	27,618	28,154	144M	17,937
Large	15,714,997	103,545	105,117	601M	82,879

We chose the Apriori algorithm [6], a data mining algorithm for retrieving frequent sets from a large amount of transactions, in our experiment. We first conducted an experiment to understand the impact of maximum item set size on rule discovery by looking at the actual item set sizes discovered while keeping the max size set to 20. The results of the experiment are given in Figure 12. By keeping the max size to 20, we essentially captured all the rules.



**Figure 12 Frequent set distribution for the small data set. The minimum support count is 10.**

We proceeded to the rule mining and verifying steps of the experiments using a Pentium D 3.0 GHz processor machine with 2GB memory. The rule-mining step took 35 minutes to extract 85,895 rules from small data set while it took 28 hours to extract 409619 rules from the large data set. The number of rules discovered by the selected minimum support, maximum rule size and confidence threshold are given in Table 3. The minimum support, the maximum rule size and the confidence threshold were chosen so a fair number of rules were discovered in each data set.

**Table 3 Summary of association rule**

Data Set	Minimum support	Maximum rule size	Confidence threshold	Number of rules
Small	8	20	0.85	85,898
Large	30	20	0.90	409,619

In the verifying step, we applied two sets of rules in ANT, which is one of the projects from Apache SVN. The results show that our method found 2664 and 2178 matched rules and 71 and 42 violated rules in ANT. It is difficult to classify if the violation is a defect or not. In order to measure the effectiveness of finding defects, we manually introduced several bugs by removing some method calls of `java.*` or `javax.*`, and applied rules again. The precision and recall of our method are listed in Table 4. The results show that using association rules learning under configuration above, about 29% and 27% of removed method calls are found, and 74% and 81% of new violations are caused by removed method call. While the results are promising, we agree that introducing bugs may not be the best approach. We hope to detect code defects by extracting code versions in the future.

**Table 4 Rule verifying on ANT**

Data Set	Matched rule	Violated rule	Inserted bugs	Recall (bugs)	Precision (bugs)
Small	2,664	71	696	0.29	0.74
Large	2,187	42	696	0.27	0.81

## 4 Discussion

Currently, item set mining [2, 6, 10] and sequence mining [7, 11, 12, 13] are the two primary kinds of data mining methods used to find software defect patterns. The items used as data sources for data mining methods are primarily function calls. Item set mining produces unordered patterns, i.e., sets of function calls, whereas, sequential pattern mining produces partially ordered patterns [14]. In the current literature,

most of the work in these data mining methods focuses on the missed function defect pattern described in section 2.5.2.

Two researchers [8, 9] tackle the issue of the neglected conditions. Both of them used some techniques of program analysis with data mining methods. Without program analysis, data mining methods suffer from issues of high false positives as their pattern elements are not necessarily associated with program dependencies [8]. There may be a trend combining data mining methods with program analysis techniques.

Table 5 shows the kind of defect patterns covered by current data mining methods. The first column is the defect classification used in this paper. The second column shows whether the corresponding pattern is covered by current data mining literature. The third column offers some typical work in the literature. The fourth column indicates whether program analysis techniques are used in the current literature to detect the corresponding defect pattern that is in the same row. And the last column is the defect ratio of each pattern in our Apache Harmony case study described in section 2.7.

From Table 5, we can see that current data mining methods can find missed function call patterns, which are mainly call pairs, call sequence and call structures. With the aid of program analysis, Neglected conditions can be addressed. Other kinds of patterns are not covered to the best of our knowledge in the literature.

In the Harmony's case, the missed function calls pattern related defects, which can be detected by stand-alone data mining methods, only constitute 6 percent. There are still large numbers of defects to be detected by data mining methods.

**Table 5 Patterns covered by data mining methods**

Pattern	Covered by literature?	Example Work	Using Program Analysis?	Defect % in Harmony case study	
RM	No	-	-	25	
VO	No	-	-	0	
PC	No	-	-	12	
CC	NC	Yes	[8, 9]	Yes	37
	Other	No	-	-	
DF	No	-	-	2	
MF	CP	Yes	[2, 10]	No	6
	CSq	Yes	[11]	No	
	CSt	Yes	[13]	No	
	Other	No	-	-	
EF	No	-	-	2	
Data use	VB	No	-	-	1
	VC	No	-	-	1
	VE	No	-	-	8

## 5 Summary

We analyzed the characteristics of code defects from the Apache Harmony project and found six general defect patterns: requirements engineering, value overflow, project configuration, conditions checking, function call usage and data usage. Three quarters of the defects come from requirements engineering, project configurations and condition checking, which are not detected by the data mining approach we developed. However, we still have a strong interest in using data mining techniques to identify software defects. We have shown that data mining methods can detect the missed function pattern, which constitutes 6% of all the defects in Harmony's case study. We believe as we learn more, and with the aid of program analysis, data mining methods should be able to detect other patterns.

## Acknowledgments

The authors thank Bob Cohn, Ganesh Prabhala, Bob Scott and Khun Ban for their reviews that greatly improve the quality of this paper.

## References

1. Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In Proceedings of the 1996 international Conference on Software Maintenance (November 04 - 08, 1996). ICSM. IEEE Computer Society, Washington, DC, 359.
2. 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
3. M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In Object Oriented Programming, Systems, Languages and Applications (OOPSLA'05), pages 365–383. ACM, 2005.
4. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, pages1–16, 2000.
5. Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, Dawson Engler. From Uncertainty to Belief: Inferring the Specification Within. Proceedings of the 7th Symposium on Operating System Design and Implementation, 2006
6. R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, Sept. 1994
7. R. Agrawal and R. Srikant. Mining sequential patterns. In Proc. 1995 Int. Conf. Data Engineering, pages 3-14, Taipei, Taiwan, Mar. 1995
8. Suresh Thummalapenta, Tao Xie. NEGWeb: Detecting Neglected Conditions via Mining Programming Rules from Open Source Code. North Carolina State University Department of Computer Science Technical report TR-2007-24, September 16, 2007
9. Ray-Yang Chang, Andy Podgurski, Jiong Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In Proc. ISSTA, pages 163–173, 2007.
10. Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Codes. In Proc. FSE, pages 306-315, 2005
11. Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 54-57, May 2006
12. Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering ( ESEC/FSE 2007), Dubrovnik, Croatia, pp. 25-34, September, 2007
13. Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. An Approach to Mining Call-Usage Patterns with Syntactic Context. ASE'07, November 5-9, 2007, Atlanta, Georgia, USA
14. Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic; Comparing Approaches to Mining Source Code for Call-Usage Patterns. Fourth International Workshop on Mining Software Repositories (MSR'07)
15. Huzefa Kagdi, Shehnaaz Yusuf, Jonathan I. Maletic, Mining Sequences of Changed-files from Version Histories, MSR'06, May 22-23, 2006, Shanghai, China
16. Harmony <http://harmony.apache.org/>
17. R. Agrawal; T. Imielinski; A. Swami: Mining Association Rules Between Sets of Items in Large Databases, SIGMOD Conference 1993: 207-216
18. Geoffrey Phipps, Comparing observed bugs and productivity rates for Java and C. Software Practice and Experience, 1999, vol. 29. 345-358.
19. Purify <http://www.ibm.com/software/awdtools/purify/>
20. FindBugs. <http://findbugs.sourceforge.net>
21. Yoav Freund. An adaptive version of the boost by majority algorithm. In Proceedings of the 12th Annual Conference on Computational Learning Theory, 1999.
22. Apache SVN <http://svn.apache.org/repos/asf/>
23. PMD/Java. <http://pmd.sourceforge.net>
24. Jianyong Wang and Jiawei Han. BIDE: Efficient mining of frequent closed sequences. In Proc. 20th International Conference on Data Engineering, pages 79-90, 2004.
25. R. Chillarege et al: Orthogonal Defect Classification - A Concept for In-Process Measurements. IEEE Transactions on SW Engineering, vol. 18(11), 11/1992
26. Marek Leszak, Dewayne E. Perry, Dieter Stoll, A Case Study in Root Cause Defect Analysis. icse, pp.428, 22nd International Conference on Software Engineering (ICSE '00), 2000