# Iterative Mining of Resource-Releasing Specifications

Qian Wu[1], Guangtai Liang[1], Qianxiang Wang[1], Tao Xie[2], Hong Mei[1]
[1]Institute of Software, School of Electronics Engineering and Computer Science
Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education
Peking University, Beijing, 100871, China
{wuqian08, lianggt08, wqx, meih}@sei.pku.edu.cn
[2]Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA
xie@csc.ncsu.edu

*Abstract*— **Software systems commonly use resources such as network connections or external file handles. Once finish using the resources, the software systems must release these resources by explicitly calling specific resource-releasing API methods. Failing to release resources properly could result in resource leaks or even outright system failures. Existing verification techniques could analyze software systems to detect defects related to failing to release resources. However, these techniques require resource-releasing specifications for specifying which API method acquires/releases certain resources, and such specifications are not well documented in practice, due to the large amount of manual effort required to document them. To address this issue, we propose an iterative mining approach, called *RRFinder*, to automatically mining resource-releasing specifications for API libraries in the form of (*resource-acquiring*, *resource-releasing*) API method pairs. *RRFinder* first identifies resource-releasing API methods, for which *RRFinder* then identifies the corresponding resource-acquiring API methods. To identify resource-releasing API methods, *RRFinder* performs an iterative process including three steps: model-based prediction, call-graph-based propagation, and class-hierarchy-based propagation. From heterogeneous information (e.g., source code, natural language), the model-based prediction employs a classification model to predict the likelihood that an API method is a resource-releasing method. The call-graph-based and class-hierarchy-based propagation propagates the likelihood information across methods. We evaluated *RRFinder* on eight open source libraries, and the results show that *RRFinder* achieved an average recall of 94.0% with precision of 86.6% in mining resource-releasing specifications, and the mined specifications are useful in detecting resource leak defects.**

***Keywords- resource-releasing specification; resource leak detection; specification mining***

## I. Introduction

Software systems commonly use resources such as network connections or external file handles. Once finish using the resources, the software systems must release these resources by explicitly calling specific resource-releasing API methods. Failing to release resources appropriately could cause resource leaks. As a result, the system at runtime slowly depletes the limited supply of system resources, leading to performance degradation and even system crashes [1]. Although programming languages such as Java provide garbage collection to free programmers from the responsibility of memory management, the mechanism does not address the problem of resource management: programs for a system must return the acquired resources by explicitly calling a resource-releasing API method. Such tasks are error-prone in practice. For example, Sun's guide to Persistent Connections [4] gets it wrong in code that is claimed to be exemplary.

To assist programmers in resource management, automatic approaches for resource leak detection [1-3] report code locations where invocations of proper resource-releasing API methods are absent. These approaches require formal resource-releasing specifications for specifying which API method acquires/releases certain resources. A common type of such specifications is in the form of (*resource-acquiring*, *resource-releasing*) API method pairs, denoting the programming constraint that, if the program acquires resources by calling the *resource-acquiring* method, it should eventually call the corresponding *resource-releasing* method to release the resources and perform necessary clean-up actions. For example, a typical resource-releasing specification is (*new FileInputStream()*, *FileInputStream.close()*). Unfortunately, these formal specifications are often missing, due to the large amount of time and energy that must be invested to manually creating them.

To address these issues, we propose an iterative mining approach, called *RRFinder*, to automatically mining resource-releasing specifications for Java API libraries. *RRFinder* takes as input the source code (Java source or bytecode) and the API documents (Javadoc[1]) of the library, and produces a set of resource-releasing specifications in the form of (*resource-acquiring*, *resource-releasing*) API method pairs. Some existing approaches [12, 13] also mine such specifications by exploiting exceptional paths in API client programs. However, because client programs tend to include mistakes especially in resource management, these approaches usually suffer from high false positive rates, based on only statistical analysis of API usage information. In addition, API client programs may not be available or many enough, e.g., for newly developed libraries.

To mine such specifications from the source code and code documents for an API library, our insight behind *RRFinder* is that high-level resources are often wrappers of low-level resources. For example, database connections (high-level resources) are established through socket connections (low-level resources). Intuitively, given a set of known low-level resource-releasing specifications (*lowRA*, *lowRR*), a high level specification (*highRA*, *highRR*) is found when

[1]Javadoc documents are automatically generated from the code comments in Java source code. They are often provided along with the library distribution.

This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 60821003; and ARO Grant No. W911NF-08-1-0443.

*highRA* invokes *lowRA* methods, and *highRR* invokes *lowRR* methods. In other words, *RRFinder* conducts propagations based on method-calling relationships, starting from known basic specifications concerning low-level resources.

Our *RRFinder* approach addresses three main challenges to mine resource-releasing specifications from API libraries. First, our preliminary study (details of the study can be found at http://sa.seforge.org/RRFinder/) shows that relying on propagations alone is not sufficient to mine precise specifications. For example, although the method *acceptFrom()* shown in Figure 1 invokes the *lowRR* method *InputStream.close()* and performs some resource-releasing actions (the statements in bold), *acceptFrom()* cannot be regarded as a *highRR* method, because its main functionality is to accept a connection from a specified host.

```
public void acceptFrom(InetSocketAddress saddr) throws IOException
{
      socksBind(saddr);
      int i = cmdIn.read();//cmdIn is a field variable of type InputStream
      SocketException ex = null;
      switch (i) {
            case REQUEST_OK:
               // success, do some operations...
               break;
            case FAILURE:
               ex = new SocketException("SOCKS server failure");
               break;
            ......
      }
      if (ex != null) {
            cmdIn.close();
            cmdIn=null;
            throw ex;
      }
      //perform some functional operations...
}
```

Figure 1. Code snippets adapted from *java.net.SocksSocketImpl*.

Second, to determine whether one API method is a resource-acquiring/releasing method, heterogeneous information should be considered together. For example, the resource-manipulation actions performed by an API method may be specified clearly in its code comments; therefore, including comment analysis could serve as a complement for the code analysis.

Third, through our manual investigation, although resource-releasing methods exhibit certain common features, such as conventions for method naming, resource-acquiring methods do not exhibit common features (these features are described in Section IV-A); therefore, it is difficult to extract resource-acquiring methods directly.

*RRFinder* addresses the preceding challenges and mines precise resource-releasing specifications effectively. Figure 2 gives an overview of *RRFinder*. The API library under analysis first goes through two preparatory analyses (Section III). Then to mine specifications, *RRFinder* starts by identifying resource-releasing API methods (Section IV), because these methods normally exhibit certain common features. Next, for these identified resource-releasing API methods, *RRFinder* searches for their corresponding resource-acquiring API methods (Section V), which acquire the resources released in the resource-releasing methods.

To identify resource-releasing API methods, *RRFinder* iteratively produces effective results by interleaving the step of model-based prediction (Section IV-A) with the steps of call-graph-based and class-hierarchy-based propagations (Section IV-B). The model-based prediction employs a classification model to predict whether an API method is a resource-releasing method. The model exploits heterogeneous information, ranging from static structural information to dynamic behavioral information, and from API library source code (Java source or bytecode) to code comments (Javadoc). The call-graph-based propagation explores method-calling relationships to detect callers of known resource-releasing methods. Such information is then used to update certain features of the detected caller methods, driving the model-based prediction step to identify new resource-releasing methods from the caller methods. Via the call-graph-based propagation, identification of high-level resource-releasing methods could benefit from identification of low-level ones. The class-hierarchy-based propagation identifies abstract resource-releasing methods by drawing conclusions from their overriding methods. Such propagation is important in object-oriented languages such as Java, where dynamic bindings are popularly used.

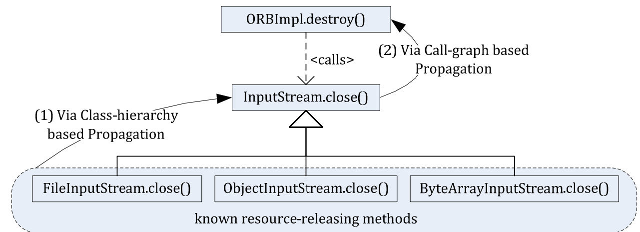


Figure 3. Example of propagations.

Figure 3 shows an example to demonstrate the propagation steps. *RRFinder* takes two steps to identify *ORBImpl.destroy()* as a resource-releasing method. First, *RRFinder* identifies *InputStream.close()* as a resource-releasing method, via class-hierarchy-based propagation.

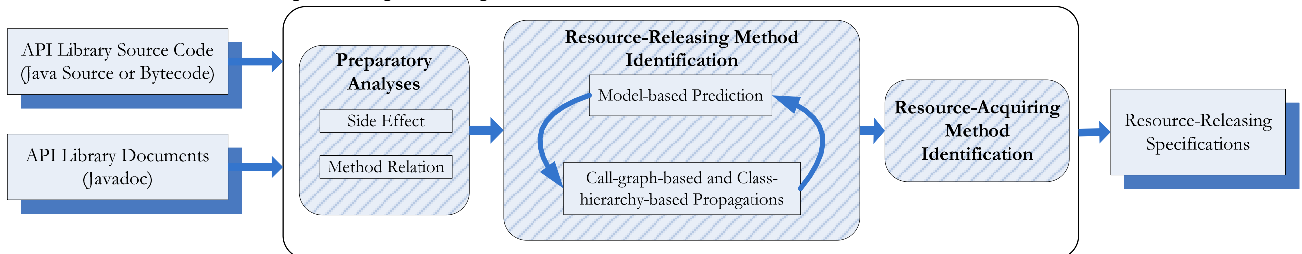


Figure 2. Overview of *RRFinder*.

Next, detecting *ORBImpl.destroy()* to have invoked *InputStream.close()* via call-graph-based propagation, *RRFinder* updates certain features of *destroy()* accordingly, and applies the classification model to finally predict *destroy()* to be a resource-releasing method. This example reflects the rationale for iteratively interleaving the model-based prediction with the propagations: the features of each API method rely on the relationships (including both calling and overriding relationships) between methods, and these features change dynamically with the propagations based on these relationships; in other words, the feedback from the propagations drives the model-based prediction to produce better mining results.

In summary, our iterative mining approach has several advantages owing to interleaving the three steps. First, the propagations greatly enhance the effectiveness of the model-based prediction to identify resource-releasing methods. Second, the model-based prediction mitigates the imprecision problem (discussed in the first challenge) of relying on propagations alone to identify resource-releasing methods. Intuitively, when a method is detected to invoke known resource-releasing methods via call-graph-based propagation, it will not be identified as a resource-releasing method immediately. Only when the heterogeneous features of the method satisfy the classification rules represented by the classification model, would this method be identified as a resource-releasing method.

This paper makes the following main contributions:

- A novel approach, called *RRFinder*, to automatically mining resource-releasing specifications for API libraries effectively. The evaluation on eight open source libraries shows that *RRFinder* identifies resource-releasing specifications with an average precision of 86.6% and recall of 94.0%.
- A set of measurable features for identifying resource-releasing methods.
- An iterative algorithm to identify precise resource-releasing methods, interleaving the step of model-based prediction with the steps of call-graph-based and class-hierarchy-based propagations.
- A technique to identify resource-acquiring methods for given resource-releasing methods.

The rest of this paper is organized as follows. Section II discusses related work. Section III introduces the required preparatory analyses. Section IV and Section V describe the identification of resource-releasing/acquiring methods, respectively. Section VI presents evaluation results. Section VII concludes.

## II. RELEATED WORK

This section first discusses related work of our approach, and then discusses relevant industrial techniques, specifically, the Automatic Resource Management feature of Java7.

### A. Specification Mining

Our work is most related to specification mining approaches. According to their mining-data sources, these approaches fall into three categories.

The first category of approaches [11-13] mines frequent API usage patterns as specifications from API client programs. In particular, Weimer et al. [12] and Thummalapenta et al. [13] used exceptional paths to mine specifications. Because programmers usually perform clean-up actions when exceptions occur, a large proportion of their mined specifications are related to resource-releasing specifications. However, based on only statistical analysis of the usage information, these approaches usually suffer from high false positive rates. The second category of approaches directly synthesizes temporal specifications by analyzing API library source code. For example, Whaley et al. [14] proposed an approach to identify illegal pairs of method invocations that would cause exceptions for Java programs. Our technique of method relation analysis (a preparatory analysis in Section III-B) is primarily a refinement of their work. The third category of approaches [15, 16] extracts specifications from API library comments. In particular, Zhong et al. [16] proposed an approach to infer resource-manipulation specifications from Javadocs. A typical resource-manipulation specification pattern involves the creation, lock, operation, unlock, and closure of the resources. Compared to their approach, our approach exploits heterogeneous information and specifically focuses on specifications concerning only the creation and closure of resources. In our evaluation, 36.7% of the specifications for eight open source libraries lack method comments for describing their resource-acquiring/releasing actions. Moreover, the classification model for resource-releasing methods in Figure 13 also indicates that relying on method comments alone is not sufficient to extract precise specifications.

### B. Resource Usage/Cost Analysis

Another category of related approaches is resource usage analysis [17, 18]. These approaches collect resource usages of programs and check whether the usages are performed in a valid manner. Some approaches focus on resource leak problems and detect certain kinds of resource leaks [1-3]. These approaches require resource-releasing specifications, such as those produced by *RRFinder*. Other related approaches include program cost analysis [19-22]. These approaches estimate for programs the upper bounds of resource usage that program executions will cost. Traditional approaches focus on a reduced number of resources, such as execution steps, time, and memory. When involving higher-level, application dependent resources, these approaches require users to define the concerned resources and specify the relevant API methods [22]. *RRFinder* could aid the users in this process and help reduce much of the burden.

### C. Automatic Resource Management in Java7

One important feature of Java7 is the mechanism of Automatic Resource Management (ARM) [9], obviating the need for manual resource termination. Figure 4 shows an example. In Java7, resources declared in the "*try*" statement will be released automatically once the execution of the try-block terminates. The only requirement is that resources must implement the *java.lang.AutoCloseable* interface. This interface contains only one method with the signature "*public void close()*", which is expected to perform resource-

releasing actions. To maintain upward compatibility, all resource classes that implement *java.io.Closeable* in Java Development Toolkit (JDK) versions older than Java7 are also supported by this feature.

```
try (InputStream fis = new FileInputStream(src);
    OutputStream fos = new FileOutputStream(desc)){
        byte[] buf = new byte[8192];
        int i;
        while ((i = fis.read(buf)) != -1) {
          fos.write(buf, 0, i);
        }
}
catch (Exception e) {
        e.printStackTrace();
}
```

Figure 4. Code snippet to demonstrate the ARM Java7 feature.

From the perspective of mining resource-releasing specifications, on the one hand, our approach could work for all programs developed with JDK versions older than Java7; on the other hand, the advent of Java7 brings a new application scenario for our approach. For all off-the-shelf API libraries that are already developed (with JDK versions older than Java7), *RRFinder* analyzes these libraries and produces a list of possible resource-releasing API methods. These methods are the candidates that should be refactored to override the *close()* method in the *java.lang.AutoCloseable* interface to make the resources usable with the ARM feature. In this way, *RRFinder* has the potential to reduce the cost of the migration from older JDK versions to Java7 for existing API libraries.

## III. PREPARATORY ANALYSES

Before describing the key ideas of our approach, we first present the preparatory analyses for each given API library.

### A. Purity and Side-effect Analysis

The procedure of purity and side-effect analysis analyzes the side-effects of each method, and finds out the pure methods [6] that do not mutate states of libraries. For example, method *isClosed()* in class *java.net.Socket* is a pure method, which simply checks the current state of the socket, whereas method *close()* is an impure method, which changes the state of the socket into "closed".

This procedure provides information for our approach to filter out unwanted method declarations and method invocations. First, declarations of pure methods are not candidates of resource-releasing methods. Second, invocations of pure methods are in fact noise when *RRFinder* analyzes the behavior of a method to determine whether it mainly performs resource-releasing actions. Therefore, in the process of indentifying resource-releasing methods, *RRFinder* ignores all the pure methods.

### B. Method Relation Analysis

The procedure of method relation analysis analyzes the relationships between each method pair (*A*,*B*) for a given class, trying to determine whether there exists a "*Cause_Exp*" relationship: the execution of method *A* would cause exceptions being thrown from the execution of method *B* when these two methods are invoked in a row. This information is used to calculate an important feature "*forbiddenMtdPercent*" for model-based prediction as described in Section IV-A.

This procedure starts by computing under what conditions each method would throw exceptions, denoted as *etCondition*. Next, for a method *B* whose *etCondition* is computed, this procedure iteratively analyzes the side-effects of the other methods in the same class, and produces a "*Cause_Exp*" relationship between (*A*,*B*), if the execution of method *A* would cause the satisfaction of *etCondition* of method *B*. The side-effects of each method are computed in advance by our flow-sensitive, context-sensitive, inter-procedural analysis.

```
public void close() throws IOException {
      if (in == null)
          return;
      in.close();
      in = null;
}
/** Checks to make sure that the stream has not been closed */
private void ensureOpen() throws IOException {
      if (in == null)
        throw new IOException("Stream closed");
}
public int read() throws IOException {
      synchronized (lock) {
      ensureOpen();
          ......
      }
}
```

Figure 5. Code snippets of the class *java.io.BufferedReader*.

Figure 5 presents an example to illustrate the process. The assignment made in method *close()* satisfies the *etCondition* of method *ensureOpen()*. Therefore, a "*Cause_Exp*" relationship is found between the method pair (*close()*, *ensureOpen()*). In addition, via inter-procedural analysis of the side-effects for each method, this procedure propagates the relationship to *read()*, which calls *ensureOpen()*. In total, we found that after the *close()* method is invoked, invocations of seven other public methods in the class would throw exceptions.

## IV. RESOURCE-RELEASING METHOD IDENTIFICATION

*RRFinder* identifies resource-releasing (denoted as *RR* in short) methods via an iterative algorithm interleaving the step of model-based prediction with the steps of call-graph-based propagation and class-hierarchy-based propagation. We first describe the core component, the model-based prediction step (Section IV-A), and then present the whole algorithm (Section IV-B).

### A. Resource-Releasing Method Classification Model

We aim to build a machine-learning-based classification model that automatically predicts whether an API method is an *RR* method. For each API method, a number of features are considered to build the model (Table I), described in detail below. We chose these features by drawing from our own experiences of manually identifying *RR* methods from all the 19,080 public methods in JDK. Because the ultimate goal of *RRFinder* is to assist programmers to write higher-

quality client code, we consider only methods that can be accessed publicly.

*1) Natural Language Information*

We consider the natural language information in method names and comments based on the observation that *RR* methods tend to follow common conventions for method naming and comment writing. For example, these methods are often named with the words like "close" or "dispose", and their comments often include phrases such as "release resources" and "garbage collection". This information provides an important hint to find *RR* methods. In our evaluation, we collect the list of *RR* related words (whose full list can be found at http://sa.seforge.org/RRFinder/) referring to the WordNet [7] dictionary, and the list also includes their synonyms appearing in the names and comments of the *RR* methods in JDK.

TABLE I. FEATURES OF *RR* METHOD CLASSIFICATION MODEL

| Features | Description |
|---|---|
| ***Natural Language Information*** | |
| hasReleaseWordsIn-NameOrComment | Whether the method name or the method comments contain words related to *RR*. |
| hasReleaseWordsIn-Comment | Whether the method comments contain words related to *RR*. |
| hasNoneReleaseWords-InName | Whether the method name contains words other than *RR* words. |
| ***Source Code Information*** | |
| ***Static Structural Information*** | |
| implCloseable | If the method is declared in a class that implements interface *java.io.Closeable*, and the method is the "*public void close()*" method, this feature is true. |
| overrideReleaseM | Whether this method overrides a known *RR* method. |
| ***Method Behavioral Information*** | |
| releaseStmtPercent | Of all the statements in the method, how high percentage (%) perform *RR* actions. |
| ***Method Relationship Information*** | |
| forbiddenMtdPercent | How high percentage (%) of other public impure methods in the class cannot be invoked safely (i.e., without throwing exceptions) after this method is invoked on the same object. |
| calledByFinalize | Whether this method is called by *finalize()*. |
| callersInClass | The number of methods in the same class that call this method. |
| calleesInClass | The number of methods in the same class that are called by this method. |

*2) Static Structural Information*

*RRFinder* uses the static information in class hierarchies to compute two features. The "*implCloseable*" feature denotes the conformation with Java7 standards [9]. As described in Section II-B, the mechanism of Automatic Resource Management works for only classes that implement the interface *java.lang.AutoCloseable*, whose corresponding interface in older JDK versions is *java.io.Closeable*, indicating that this interface is inherently designed for *RR* functionality. The rationale with the feature "*overrideReleaseM*" is that, if a method overrides an *RR* method, there is a good chance that it still performs *RR* actions.

*3) Method Behavioral Information*

For each method, *RRFinder* estimates whether its main purpose is to release resources by counting how high percentage of the method's statements perform *RR* actions, denoted as the feature "*releaseStmtPercent*". When counting the statements, it is important that *RRFinder* ignores trivial statements and focuses on critical operations. We calculate this feature following Formula (1). To calculate the total number of methods' statements, *RRFinder* considers only *NonPrimitiveAssignment* statements and *Method-Invocation* statements in a method, because these two types of statements could take the main responsibility for the side-effects caused by the method. Other statements such as *JUMP* statements are less related to functional operations and are ignored. Two types of statements are considered to perform *RR* actions: *NULLAssignment*, representing statements that assign a *NULL* value to a variable, and *RR-Invocation*, representing statements that invoke known *RR* API methods. In addition, when counting the total number of *Method-Invocation* statements, two types of invocations are regarded as noises and excluded. The first type is invocations of pure methods, which are filtered using the results of the Purity and Side-effect Analysis (Section III-A). The other type is auxiliary operations, such as logging operations or data-structure traversal operations, which are filtered based on a predefined list.

$$releaseStmtPercent = (NullAssignStmtCount + ReleaseIvkStmtCount)\ /(totalNonPmvAssignStmtCount + totalMethodIvkStmtCount) \quad (1)$$

```
private synchronized void flushAndClose() {
    writer.flush();
    writer.close();
    writer = null;
    output = null;
}
public synchronized void close() {
    flushAndClose();
}
```

Figure 6. Code snippets adapted from *java.util.logging.StreamHandler*.

To enhance *RRFinder* and make it adaptive, the calculation of this feature would involve inter-procedural tracking at certain circumstances. Inter-procedural tracking works especially when a method *A* performs *RR* actions indirectly by calling a method *B*, which nevertheless would not be recognized as an *RR* method due to factors such as being a private method. In such cases, without tracking into method *B*, *RRFinder* would fail to identify the *RR* actions performed by method *A*. For example, as shown in Figure 6, the *RR* method *close()* calls only one method *flushAndClose()*, which in fact performs *RR* operations, but would never be identified as an *RR* method since it is a private method. To deal with this situation, *RRFinder* tracks into *flushAndClose()* and updates "*releaseStmtPercent*" for *close()* from 0 to 3/4. To avoid explosion, *RRFinder* tracks into a called method only when its method name contains releasing-related words, and updates "*releaseStmtPercent*" of the caller method only when its value can be increased.

By computing this feature, *RRFinder* could estimate the main functionality of a method. This strategy enables the exclusion of non-*RR* methods that involve *RR-Invocations*. For the method in Figure 1, the "*releaseStmtPercent*" is only 5.7% (many statements are omitted in Figure 1), too low to enable the method to be identified as an *RR* method.

This feature acts as the critical bond connecting the step of model-based prediction with the step of the call-graph-based propagation (described later in Section IV-B). Thanks to this feature, identification of higher-level *RR* methods could benefit from the identification of lower-level ones.

*4) Method Relationship Information*

*RRFinder* also exploits the relationships between methods in a class to identify *RR* methods. *RRFinder* includes the feature "*forbiddenMtdPercent*" based on the following observation: after the invocation of an *RR* method, invocations of all other public methods that mutate the object's state (i.e., impure methods) are usually forbidden on the same object; otherwise, exceptions would be thrown. For example, after the *RR* method "*close()*" is called on an object of "*java.io.BufferedReader*", all the seven other public impure methods cannot be invoked on the same object safely (i.e., without throwing exceptions). To compute the feature, given a method *A* from class *C*, *RRFinder* identifies public impure methods (of class *C*) that cannot be safely invoked after the execution of method *A*, based on the results of the Method Relation Analysis (Section III-B) and Purity Analysis (Section III-A). Suppose that *RRFinder* recognizes that *m* other public impure methods of class *B* could not be invoked afterwards, and class *B* includes totally *n* other public impure methods, then this feature is calculated as *m/n*.

The rationale with the feature "*calledByFinalize*" is that, in Java, the *finalize()* method is automatically invoked in the process of garbage collection, and this method should perform only *RR* actions [8]. For a method, *RRFinder* also exploits the number of caller methods and callee methods of this method in the same class, based on the observation that, in most cases, *RR* methods have very few interactions with the other methods in the same class.

In our evaluation, we manually prepared the training set using all the 19,080 public methods in JDK, and adopted the decision-tree classification algorithm [10] to build the model (details of the evaluation process are discussed in Section VI). After the model is built, for each API method under consideration, *RRFinder* first computes all its preceding features, and then applies the model to predict whether this method is an *RR* method.

## B. Propagation

In this section, we present the iterative algorithm of *RR* method identification, as shown in Figure 7. The algorithm takes as input a list of API methods under consideration, and identifies the *RR* methods among them.

This algorithm maintains two key data structures: *releaseMs* is a set of all identified *RR* methods, and *newFoundReleaseMs* is a first-in-first-out queue of newly detected *RR* methods, from which propagations would be started. In addition, *releaseMs* also serves as a guard condition to avoid redundant propagations (Lines 14 and 20).

The algorithm is mainly composed of three phases. Initially, the algorithm iteratively computes the features for each API method and uses the pre-built classification model to predict whether it is an *RR* method (Lines 3-7). Next, for each newly detected *RR* method, a call-graph-based propagation (Lines 10-16) and a class-hierarchy-based propagation (Lines 17-22) are started.

```
Algorithm identifyResourceReleasingMethods
Input unKwnMs a list of unknown API methods
Output releaseMs a list of resource-releasing API methods
Begin
1.  releaseMs←Ø;
2.  newFoundReleaseMs←Ø;
3.  foreach m in unKwnMs do
4.    label←classifyMethod(m);
5.    if label is "resource-releasing" then
6.      add m to the end of newFoundReleaseMs;
7.      add m to releaseMs;
8.  while newFoundReleaseMs is not empty do
9.    remove the head method releaseM of newFoundReleaseMs;
10.   foreach caller methods callerM of releaseM do
11.     recompute feature "releaseStmtPercent";
12.     label←classifyMethod(callerM);
13.     if label is "resource-releasing" then
14.       if callerM not in releaseMs then
15.         add callerM to the end of newFoundReleaseMs;
16.         add callerM to releaseMs;
17.   foreach overriden methods parentM of releaseM do
18.     update the releaseChildCount of parentM;
19.     if its releaseChildCount/ChildCount > inheritThresh then
20.       if parentM not in releaseMs then
21.         add parentM to the end of newFoundReleaseMs;
22.         add parentM to releaseMs;
End
```

Figure 7. Algorithm of *RR* method identification.

Via call-graph-based propagation, identification of higher-level *RR* methods could benefit from identification of lower-level ones. When a new *RR* method is detected, the value of feature "*releaseStmtPercent*" for all its caller methods is increased. Therefore, *RRFinder* performs such propagations by re-computing "*releaseStmtPercent*" (Line 11) for each caller method, and performs the prediction step again to see whether the caller method could be identified as an *RR* method (Line 12). This process is more effective if a list of known *RR* API methods (which the API library under analysis calls) is provided. For example, the method *AbstractPooledConnAdapter.close()* in library httpClient[1] calls the method *HttpConnection.close()* in library httpCore[2] (both are actually *RR* methods); due to the latter method being an *RR* method, the value of "*releaseStmtPercent*" is increased for the former method, and the former method's chance of being identified as an *RR* method is also increased. Normally, to find *RR* methods for any third-party API libraries, we provide the algorithm with the list of all the known *RR* methods in JDK.

Via class-hierarchy-based propagation, *RRFinder* identifies abstract *RR* methods, which are important in object-oriented languages such as Java, where dynamic bindings are popularly used. To make such propagation, for each method, our algorithm counts the number of methods that override this method, denoted as "*ChildCount*", and also the number of *RR* methods among these methods, denoted as "*releaseChildCount*". Our algorithm considers a method to be an *RR* method if its *releaseChildCount*/*ChildCount* is higher

[1] httpClient: http://hc.apache.org/httpcomponents-client-ga/.

[2] httpCore: http://hc.apache.org/httpcomponents-core-ga/index.html.

than a predefined threshold *inheritThresh*[1]. Therefore, each time a new *RR* method is found, the number of "*releaseChildCount*" for each overridden method is increased, and then our algorithm checks whether this update produces a new *RR* method (Lines 17-22). The rationale with such propagation is that the common functionality of the overriding methods usually reflects the functionality of their commonly overridden methods.

## V. Resource-acquiring Method Identification

Both *resource-releasing* and *resource-acquiring* methods are needed to form specifications in the form of (*resource-acquiring*, *resource-releasing*) API method pairs. For each identified *RR* method, *RRFinder* next searches in its belonging class for the corresponding *resource-acquiring* (denoted as *RA* in short) method, which acquires the resources that are released in the *RR* method.

To find the corresponding *RA* method, *RRFinder* starts by considering the two types of *RR* statements in the *RR* method. For each *NULLAssignment* statement, *RRFinder* searches in the same class for the method that makes *non-NULL* assignment to the corresponding variable. For each *RR-Invocation* statement, *RRFinder* searches in the same class for the method that invokes the corresponding known *RA* method. (The definitions of these two types of *RR* statements are in Section IV-A-3.) Figure 8 presents an example, for which the known *RR* specification is (*new DatagramSocket(), DatagramSocket.close()*). *RRFinder* first identifies *SyslogWriter.close()* as an *RR* method, which contains one *RR-Invocation* statement. *RRFinder* then identifies the constructor to be the *RA* method by searching for the method that invokes the known *RA* method *new DatagramSocket()*.

```
public SyslogWriter(final String syslogHost){
        .......
        ds = new DatagramSocket();
        .......
}
public void close() {
        ds.close();
}
```

Figure 8. Code snippets of the class *org.apache.log4j.helpers.SyslogWriter*.

```
public ThreadSafeClientConnManager(HttpParams params,
                    SchemeRegistry schreg) {
        ......
     this.pool=(ConnPoolByRoute)createConnectionPool(params);
        ......
}
protected AbstractConnPool createConnectionPool(final HttpPa-
rams params) {
      return new ConnPoolByRoute(connOperator, params);
}
public void shutdown() {
      pool.shutdown();
}
```

Figure 9. Code snippets of *org.apache.http.impl.conn.tsccm.ThreadSafeClientConnManager*.

Because the *RA* method must also be a method that could be accessed publicly, a call-graph-based backward tracking (from callees to callers) is sometimes needed. Figure 9 shows an example where the known *RR* specification is (*new ConnPoolByRoute(), ConnPoolByRoute.shutdown()*). *RRFinder* first detects the method *createConnectionPool()*, which invokes the known *RA* method, and then identifies the public constructor to be the target *RA* method, which invokes *createConnectionPool()*.

Via backward tracking, the identified *RA* method may not necessarily reside in the same class with the *RR* method (the average percentage is 29.2% for the eight libraries in our evaluation.). For example, the identified *RR* method for class *org.hibernate.impl.StatelessSessionImpl* is *close()*, and the only method in the class that acquires the corresponding resources is the constructor but with a package accessibility (i.e., only accessible to methods or classes within the same package). The actual target *RA* method is *SessionFactoryImpl.openStatelessSession()*, which is a public method invoking the constructor of *StatelessSessionImpl*.

In addition, there are cases when several *RA* methods are identified for one *RR* method. In such situation, various resources are acquired in different methods in the class, and the *RR* method performs clean-up for all the resources that may be acquired by the methods in the class.

Finally, if there are no *NULLAssignment* or recognized *RR-Invocation* statements in the *RR* method, *RRFinder* assumes the public constructor of the class to be the *RA* method. (In our evaluation, constructors account for 77.3% of all the correctly identified *RA* methods.) A call-graph-based tracking may also be needed in case there are no public constructors available. In particular, if the corresponding *RR* method resides in an interface, a class-hierarchy-based propagation is also required to find the public *RA* method. Details of these techniques are omitted here due to the space limit.

## VI. Evaluation

We implemented a prototype tool for *RRFinder* and conducted an evaluation using it. To build the *RR* method classification model, we manually built the training set using 19,080 non-empty public methods in JDK in two weeks. We then applied *RRFinder* to mine specifications for eight open source libraries, detailed in Table II. In particular, column "#C" lists the number of classes, and column "#PM" lists the number of public methods exposed by the library. We selected these libraries because they are known to involve substantial manipulations of resources, such as external files, database, and network connections. To evaluate the effectiveness of *RRFinder*, we prepared a golden standard for each selected library. We invited four researchers from the Institute of Software at Peking University to manually identify *RR* specifications for the libraries. Initially, each library was examined by two researchers, and then, any disagreements between the two were inspected by a third researcher. We allocated the tasks based on the workload and also their familiarity with the subject libraries. Our evaluation was conducted on a 2.6GHz dual core machine. All the manually identified specifications for the eight libraries are available at http://sa.seforge.org/RRFinder/.

Our evaluation addresses the following research questions, which are organized in a top-down manner. *RQ1:* Could *RRFinder* mine specifications effectively? *RQ2:*

[1] The threshold *inheritThresh* was set to 0.3 in our evaluation.

Could *RRFinder* identify *RR* methods effectively? *RQ3:* How useful are the selected features (in Section IV-A) in predicting *RR* methods? *RQ4:* How much could *RRFinder* benefit from the propagations? *RQ5:* Are the mined specifications useful in defect detection? We do not discuss in detail the evaluation results of *RRFinder* in *RA* method identification, due to the space limit and its relatively good performance (as shown in Table III).

TABLE II. SUBJECT PROJECTS AND THEIR CHARACTERISTICS.

| Library | Version | #C | #PM | Description |
|---|---|---|---|---|
| Cayenne | 3.0.1 | 1361 | 7152 | object relational database |
| Hibernate | 3.6.0 | 2678 | 17651 | Java persistent framework |
| HttpClient | 4.1.1 | 272 | 1177 | HTTP client-side components |
| HttpCore | 4.1 | 311 | 1655 | HTTP transport components |
| Log4J | 1.2.16 | 308 | 1633 | logging framework |
| PDFBox | 1.5.0 | 504 | 3823 | Java PDF library |
| Xalan | 2.7.1 | 1130 | 7865 | XSLT processor |
| Xerces | 2.11.0 | 768 | 5729 | XML parser |

### A. RQ1:Effectiveness of Specification Extraction

We applied *RRFinder* to extract specifications for the eight open source libraries, and the results are shown in Table III. For the overall results (columns "Time" to "F"), column "Time" lists the time spent on each library in minutes; column "#T" lists the number of manually identified *RR* specifications; columns "P", "R", "F" give respectively the precision, recall, and F-Score of *RRFinder*. For the results of *RR* method identification (columns "#CAuto" to "RR"), column "#CAuto" lists the number of correctly identified methods by *RRFinder*; column "#Auto" lists the number of automatically identified methods by *RRFinder*; column "#Man" lists the number of manually classified methods; columns "PR" and "RR" list the precision and recall of *RR* method identification. For the results of *RA* method identification (the last two columns), columns "PA" and "RA" list respectively its precision and recall. In particular, to separate the impact of the *RR* method identification, the calculation of the precision/recall of *RA* method identification is confined to those *RA* methods whose corresponding *RR* methods are correctly identified by *RRFinder*.

From the results in Table III, we have the following observations. First, *RRFinder* achieves relatively high precisions, recalls, and F-scores on these libraries, with an average precision of 86.6% and recall of 94.0%. In particular, the low precisions for libraries *PDFBox* and *Xalan* are mainly caused by their corresponding low precisions of *RR* method identification, which we explain later in Section VI-B. Second, the time used to mine specifications is acceptable. Moreover, the time spent on each library is largely proportional to the size (considering the number of classes and methods) of the library, indicating that *RRFinder* is scalable. Third, compared with *RR* method identification, the *RA* method identification demonstrates a much better performance with an average precision of 95.9% and recall of 97.9%, indicating that the effectiveness of *RR* method identification is the critical factor in determining the overall performance of *RRFinder*. Finally, by comparing column "#T" with column "#Man", we found that there are 89 more *RR* specifications than *RR* methods, indicating that many *RR* methods (an percentage of 31.3%) have more than one corresponding *RA* method.

### B. RQ2:Effectiveness of RR Method Identification

We next explain the results of *RR* method identification shown in columns "#CAuto" to "RR" in Table III. In general, *RRFinder* identified 93.1% of the *RR* methods with the average precision 88.2%. The results indicate that, to identify *RR* methods for a library, *RRFinder* does not require the classification model to be trained using methods in the same library. Therefore, users of our approach could rely on a universally built model to identify *RR* methods.

In particular, the reason for the low precision of library *PDFBox* is that all the 13 wrongly identified *RR* methods are manually classified by us as *resource-releasing-utility* methods. Figure 10 shows an example method. This method looks much like an *RR* method, except that the releasing actions are conducted on the passed-in parameters. In contrast, in this paper, we focus on identifying resource-releasing methods that should be invoked when the tasks of their belonging classes are completed, and these methods perform releasing actions for their belonging classes. These *resource-releasing-utility* methods can be easily filtered by checking whether the *RR-Invocation* statements are conducted on the parameters. Similarly, among the wrongly identified methods, two of library *Xalan* and one of library *Hibernate* are all *resource-releasing-utility* methods. The other wrongly identified method of library *Xalan* is shown in Figure 11. This method is undoubtedly an *RR* method; however, as shown in the method comments, this method is automatically called and should not be invoked by client programs.

Figure 12 presents two *RR* methods that were not identified by *RRFinder*. The two methods demonstrate one challenge of *RR* method identification: some *RR* methods do not perform resource-releasing actions directly; instead, they fire certain events and send out signals, and the actual resource-releasing actions are accomplished via the cooperation of

TABLE III. RESULTS OF RESOURCE-RELEASING SPECIFICATION IDENTIFICATION.

| Library | Overall Results | | | | | RR Idtf. | | | | | RA Idtf. | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Time (m) | #T | P(%) | R(%) | F(%) | #CAuto | #Auto | #Man | PR(%) | RR(%) | PA(%) | RA(%) |
| Cayenne | 97.9 | 25 | 95.5 | 95.5 | 95.5 | 16 | 16 | 17 | 100.0 | 94.1 | 100.0 | 100.0 |
| Hibernate | 139.6 | 47 | 85.0 | 89.5 | 87.2 | 36 | 38 | 38 | 94.7 | 94.7 | 89.5 | 94.4 |
| HttpClient | 6.2 | 13 | 100.0 | 80.0 | 88.9 | 8 | 8 | 10 | 100.0 | 80.0 | 100.0 | 100.0 |
| HttpCore | 5.3 | 47 | 100.0 | 84.4 | 91.5 | 27 | 27 | 32 | 100.0 | 84.4 | 100.0 | 100.0 |
| Log4J | 5.9 | 22 | 100.0 | 95.2 | 97.5 | 19 | 19 | 19 | 100.0 | 100.0 | 100.0 | 95.2 |
| PDFBox | 13.8 | 22 | 40.7 | 100.0 | 57.9 | 11 | 24 | 11 | 45.8 | 100.0 | 84.6 | 100.0 |
| Xalan | 70.6 | 17 | 57.1 | 100.0 | 72.7 | 4 | 7 | 4 | 57.1 | 100.0 | 100.0 | 100.0 |
| Xerces | 28.1 | 40 | 86.7 | 100.0 | 92.9 | 13 | 13 | 13 | 100.0 | 100.0 | 100.0 | 100.0 |
| Total | 367.4 | 233 | 86.6 | 94.0 | 90.1 | 134 | 152 | 144 | 88.2 | 93.1 | 95.9 | 97.9 |

several different classes. We plan to address this challenge in future work.

```
// Close the document.
public void close( FDFDocument doc ) throws IOException
{
  if( doc != null )
  {
    doc.close();
  }
}
```

Figure 10. Code snippet of class *org.apache.pdfbox.ImportFDF*.

```
/* Automatically called when the HTML page containing the applet is
no longer on the screen. Stops execution of the applet thread. */
public void stop()
{
   if (null != m_trustedWorker){
      m_trustedWorker.stop();
      m_trustedWorker = null;
   }
   m_styleURLOfCached = null;
   m_documentURLOfCached = null;
}
```

Figure 11. Code snippet of class *org.apache.xalan.client. XSLTProcessorApplet*.

```
public void shutdown() {
    if (shutdown) {
        return;
    }
    shutdown = true;
    lock.lock();
    try {
        condition.signalAll();
    } finally {
        lock.unlock();
    }
}
```

```
// Stops event threads. After the EventManager is stopped, it can not be restarted and should be discarded.
public void shutdown() {
    stopped = true;
    for (DispatchThread thread : dispatchThreads) {
        thread.interrupt();
    }
}
```

Figure 12. Code snippets with the left one from class *org.apache.http. nio.util.SharedInputBuffer*, and the right one from class *org.apache. cayenne.event.EventManager*.

### C. RQ3: RR Method Classification Model

We manually built the training set using all the 19,080 non-empty public methods in JDK. We adopted the decision-tree classification algorithm [10] to build the model. In this algorithm, rules are organized as tree structures, where leaves represent classification results, and branches leading from the root to the leaves represent the conjunctions of the conditions of features leading to the classification results. We chose this algorithm because the classification model can be explicitly visualized, enabling us to evaluate the value of each feature and also the soundness of the training set. We present the built decision-tree model[1] in Figure 13. For simplicity, only branches leading to positive classifications (of an *RR* method) are shown. Five manners of the conjunctions of the feature conditions could lead to the prediction of an *RR* method (the corresponding leaves are in bold). For example, a method is classified as an *RR* method when (1) it does not override an *RR* method, (2) its method name contains only releasing-related words, and (3) its "*releaseStmtPercent*" is higher than 0.377967.

The model in Figure 13 justifies our intentions of choosing these features and it shows the predictive power of each feature quantitatively. Intuitively, the shorter the path leading from the root to the leave, and the closer the feature condition gets to the leave, the stronger the predictive power of the feature is. Therefore, feature "*releaseStmtPercent*" demonstrates the strongest predictive power. In particular, when calculating the features "*callersInClass*" and "*calleesInClass*" in Table I, we calculated both the number, denoted as "*callers*/*calleesInClassCount*", and the percentage, denoted as "*callers*/*calleesInClassPercent*" for a better prediction performance. However, the feature "*calleesInClassCount*" was dropped due to its relatively low predictive power, indicating that the *RR* methods may sometimes invoke other methods in the same class, but normally would not be invoked by the other methods in the same class. Through our manual inspection, we found that the *RR* methods are called only either by the *finalize()* method in the same class, or when unexpected behavior occurs during the operation in the other methods, which are forced to terminate and need immediate clean-up. In addition, the feature "*hasReleaseWordsInComment*" is also dropped. Although the comments for *RR* methods usually involve phrases such as "release resource" or "allow garbage collection", the comments are missing at certain occasions, partially due to the developers' assumption that they have already expressed their intention using the method names.

```
overrideReleaseM = yes
|  hasReleaseWordsInNameOrComment = yes
|  |  hasNoneReleaseWordsInName = no
|  |  |  callersInClassCount <= 2.5
|  |  |  |  callersInClassPercent <= 0.202041
|  |  |  |  |  calleesInClassPercent <= 0.188345: yes
overrideReleaseM = no
|  hasNoneReleaseWordsInName = no
|  |  releaseStmtPercent <=  0.377976
|  |  |  implCloseable = yes: yes
|  |  |  implCloseable = no
|  |  |  |  calledByFinalize = yes: yes
|  |  |  |  calledByFinalize = no
|  |  |  |  |  forbiddenMtdPercent > 0.33333
|  |  |  |  |  |  hasReleaseWordsInNameOrComment = yes
|  |  |  |  |  |  |  callersInClassPercent <= 0.052983
|  |  |  |  |  |  |  |  callersInClassCount <= 2.5
|  |  |  |  |  |  |  |  |  calleesInClassPercent <= 0.101282: yes
|  |  releaseStmtPercent >  0.377976: yes
```

Figure 13. The decision-tree classification model.

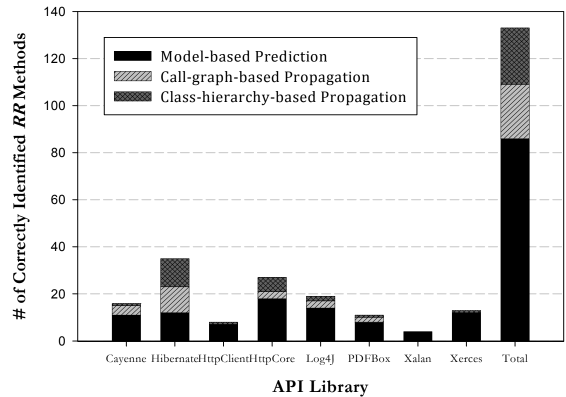


Figure 14. Distributions of the correctly identified *RR* methods.

[1] This model was trained with precision of 93.8%, recall of 85.4%, and F-Score of 92.8%.

HibernateAdapter.java Revision 625

```
public void update(Item item) {
    ……
    If (item.isDeleted()){
        Session session = newSession();
        SyncInfo syncInfo = syncDAO.get(item.getSyncId());
        closeSession();
        ……
    }
    ……
}

private void closeSession() {
    if(currentSession != null){
        currentSession.close();
        currentSession = null;
    }
}
private Session newSession() {
    currentSession = this.sessionFactory.openSession();
    return currentSession;
}
```

HibernateAdapter.java Revision 1525

```
public void update(Item item) {
    ……
    If (item.isDeleted()){
        Session session = newSession();
        SyncInfo syncInfo = null;
        try{
            syncInfo = syncDAO.get(item.getSyncId());
        } finally{
            closeSession();
        }
    ……
    }
    ……
}
    private void closeSession() {
        if(currentSession != null){
            currentSession.close();
            currentSession = null;
        }
    }
    Private Session newSession() {
        closeSession();
        currentSession = this.sessionFactory.openSession();
        return currentSession;
    }
```

Figure 15. A confirmed defect in Mesh4j.

### D. RQ4: Benefits of Propagations

To discover the benefits of the propagations, we investigated the distribution of the correctly identified *RR* methods considering the way that they are identified, shown in Figure 14. Of all the correctly identified *RR* methods, 17.3% are identified via call-graph-based propagation and 18.0% via class-hierarchy-based propagation. The results show that the propagations enable *RRFinder* to identify more *RR* methods.

### E. RQ5: Usefulness of the Mined Specifications

Finally, to evaluate the usefulness of the mined specifications, we used these specifications to detect defects in open source projects. Figure 15 shows a confirmed defect concerning the specification (*SessionFactory.openSession()*, *Session.close()*). The left-hand-side code snippet shows the found defect: when the method invocation *synsDAO.get()* throws exceptions, the *session* created in *newSession()* will be left unclosed. The right-hand-side code snippet shows how the defect is fixed. More examples of the detected defects are not shown due to the space limit.

## VII. Conclusion

We have proposed an approach, *RRFinder*, to automatically mining *RR* specifications for API libraries. *RRFinder* first identifies *RR* API methods, and then searches for the corresponding *RA* API methods. To identify *RR* API methods, *RRFinder* iteratively produces rich and precise results by interleaving the step of model-based prediction with the steps of call-graph-based and class-hierarchy-based propagations. Evaluation results on eight open source libraries show that our approach performed effectively and the mined specifications are useful in detecting resource leak defects.

## References


[1] E. Torlak, S. Chandra. Effective interprocedural resource leak detection. In *Proceedings of ICSE*, 535-544, 2010.

[2] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-me: a static analysis for automatic individual object reclamation. In *Proceedings of PLDI*, 364–375, 2006.

[3] D. L. Heine and M. S. Lam. A practical flow-sensitive and context-sensitive C and C++ memory leak detector. In *Proceedings of PLDI*, 168–181 , 2003.

[4] Sun's guide to Persistent Connections: http://tinyurl.com/6b5jc7.

[5] Proposal of Automatic Resource Management: http://mail.openjdk.java.net/pipermail/coin-dev/2009-February/000011.html.

[6] A. Salcianu and M. Rinard. Purity and side effect analysis for Java programs. In *Proceedings of VMCAI*, 199-215, 2005.

[7] WordNet, available at *http://wordnet.princeton.edu/*.

[8] B. Eckel. Thinking in Java. 3rd Edition. Prentice-Hall, 2002.

[9] Java7, available at *http://jdk7.java.net/*.

[10] I. H. Witten, E. Frank, and M. A. Hall. Data Mining: Practical Machine Learning Tools and Techniques. 3rd Edition. Morgan Kaufmann. 2011.

[11] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of SOSP*, 57-72, 2001.

[12] W. Weimer and G. Necula. Mining temporal specifications for error detection. In *Proceedings of TACAS*, 461–476, 2005.

[13] S. Thummalapenta and T. Xie. Mining exception-handling rules as sequence association rules. In *Proceedings of ICSE*, 496-506, 2009.

[14] J. Whaley, M. Martin, and M. Lam. Automatic extraction of object-oriented component interfaces. In *Proceedings of ISSTA*, 218–228, 2002.

[15] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments?*/. In *Proceedings of SOSP*, 145–158, 2007.

[16] H. Zhong, L. Zhang, T. Xie, H. Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of ASE*, 307-318, 2009.

[17] A. Igarashi, N. Kobayashi. Resource usage analysis. In *Proceedings of POPL*, 331-342, 2002.

[18] M. Bartoletti, P. Degano, G. Ferrari, and R. Zunino. Local policies for resource usage analysis. *TOPLAS*, 31(6), 1-43, 2009.

[19] I. Bate, G. Bernat, and P. Puschner. Java virtual-machine support for portable worst-case executiontime analysis. In *Proceedings of ISORC*, 83-90, 2002.

[20] R. Wilhelm. Timing analysis and timing predictability. In *Proceedings of FMCO*, 317-323, 2004.

[21] E. Albert, P. Arenas, S. Genaim, G. Puebla, D. Zanardini. Resource usage analysis and its application to resource certification. In *Proceedings of FOSAD*, 258-288, 2009.

[22] J. Navas, M. Mendez-Lojo, and M. V. Hermenegildo. User-definable resource usage bounds analysis for Java bytecode. ENTCS, 253(5), 65-82, 2009.