Improving Software Reliability and Productivity via Mining Program Source Code

Tao Xie, Mithun Acharya, Suresh Thummalapenta, Kunal Taneja
Department of Computer Science
North Carolina State University
Raleigh NC USA 27695
{xie, acharya}@csc.ncsu.edu, {sthumma, ktaneja}@ncsu.edu

Abstract

A software system interacts with third-party libraries through various APIs. Insufficient documentation and constant refactorings of third-party libraries make API library reuse difficult and error prone. Using these library APIs often needs to follow certain usage patterns. These patterns aid developers in addressing commonly faced programming problems such as what checks should precede or follow API calls, how to use a given set of APIs for a given task, or what API method sequence should be used to obtain one object from another. Ordering rules (specifications) also exist between APIs, and these rules govern the secure and robust operation of the system using these APIs. These patterns and rules may not be well documented by the API developers. Furthermore, usage patterns and specifications might change with library refactorings, requiring changes in the software that reuse the library. To address these issues, we develop novel techniques (and their supporting tools) based on mining source code, assisting developers in productively reusing third party libraries to build reliable and secure software.[1]

1 Introduction

The primary goal of software development is to deliver high-quality software efficiently and in the least amount of time whenever possible. To achieve the preceding goal, developers often want to reuse existing frameworks or libraries instead of developing similar code artifacts from scratch. The challenging aspect for developers in reusing the existing frameworks or libraries is to understand the usage patterns and ordering rules (specifications) among Application Programming Interfaces (APIs) exposed by those frameworks or libraries, because many of the existing frameworks or libraries are not well documented. Incorrect usage of APIs may lead to violated API specifications, leading to security and robustness defects in the software. Furthermore, usage patterns and specifications might change with library refactorings, requiring changes in the software that reuse the library.

To address these issues, we develop novel techniques (based on data mining) that automatically mine usage patterns and specifications, and detect refactorings from source code. Our techniques aid developers in productively reusing third party libraries to build reliable and secure software. We present three infrastructures based on mining source code to address the main issues faced by developers in reusing API libraries. The *tracing* infrastructure (Section 2) automatically mines API usage patterns and specifications from API client code in local source code repositories. The *searching* infrastructure (Section 3) expands the scope of mining to also include billions of lines of open-source API client code available on the web. The *refactoring-detection* infrastructure (Section 4) automatically detects refactorings in libraries by analyzing library API implementation code. We have implemented our techniques in a suite of tools. *Apiartor* [3], *ExitSafe* [2], and *IDeaMiner* [4] are based on the *tracing* infrastructure. *NEGWeb* [19] and *PARSEWeb* [18] are based on the *searching* infrastructure. Finally, *RefacLib* [17] is based on the *refactoring-detection* infrastructure. The high-level details of our infrastructures and the highlights of the evaluation results of our tools are provided in the subsequent sections.

2 Tracing Infrastructure

A software system interacts with third-party libraries through various APIs. Using these library APIs often needs to follow certain usage patterns (how to use a given set of APIs for a particular task?). Furthermore, ordering rules

[1]This work is supported in part by NSF grant CNS-0720641, CCF-0725190, and ARO grant W911NF-07-1-0431.

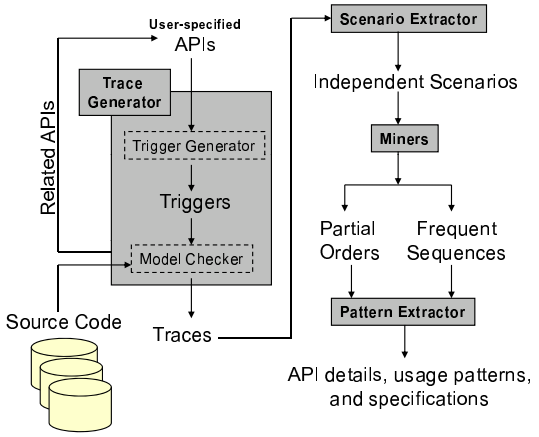

Figure 1. Tracing Infrastructure

(specifications) exist between APIs, and these rules govern the secure and robust operation of the system using these APIs. Unfortunately, API usage patterns and various API specifications are not well documented by the API-library developers. API patterns cut across procedural boundaries and an attempt to infer these patterns by manual inspection of source code (API client code) is often inefficient and inaccurate. Several problems exist even when the API specifications are known. API specifications (when known) can be formally written for third-party APIs and statically verified against a software system. But manually writing a large number of formal API specifications for static verification is often inaccurate or incomplete, apart from being cumbersome. Formal specifications are complicated and lengthy mainly due to the various API details (such as input/return type, error flags, and return values for APIs on success/failure) and language syntax considerations required for the specification to be accurate and complete. To address these issues, we present the *tracing* infrastructure that mines API details, patterns, and specifications by analyzing the source code (API client code). In this section, we present our tracing infrastructure and the three tools based on the infrastructure, namely, Apiartor (Section 2.1), ExitSafe (Section 2.2), and IDeaMiner (Section 2.3).

The high-level overview of the tracing infrastructure [2, 3] is shown in Figure 1. The tracing infrastructure has four main components: trace generator, scenario extractor, miners, and pattern extractor. The trace generator uses compile-time push-down model-checking (PDMC) [10] to generate inter-procedural static traces, which approximate run-time API behaviors. The PDMC process verifies a property specified in the form of Finite State Machine (FSM) over a given program. Using *Triggers* [3], a form of FSM, we adapt the PDMC process to output static traces in the program involving APIs of interest. A single static trace from the model checker might involve several API usage scenarios, being often interspersed. The scenario extractor separates different usage scenarios from a given trace, so that each scenario can be fed separately to the miners, our next component. The miner component employs various data mining techniques on these static traces to output frequent partial orders or frequent sequences (based on the employed data-mining technique) among APIs. The miner output is then processed by the pattern extractor to output API details, patterns, and specifications.

2.1 Apiartor

Apiartor employs the tracing infrastructure to mine usage patterns and specifications that involve multiple-API sequences from the static traces. Previous approaches [14, 22] mine frequent association rules, itemsets, or subsequences that capture API call patterns shared by API client code. However, these frequent API patterns cannot completely capture some useful orderings shared by APIs, especially when multiple APIs are involved across different procedures. Apiartor summarizes API usage patterns as partial orders [16]. Different API usage scenarios are extracted from the static traces by our scenario extraction algorithm and fed to a Frequent Closed Partial Order (FCPO) miner [16]. The miner summarizes different usage patterns as compact partial orders. The usage patterns can be used as a recommender, which shows how to use a set of APIs for a particular task. Our experience of applying the framework on 72 clients of the X11 library with 200K LOC in total has shown that the extracted API partial orders are useful in assisting effective API reuse and checking. We also compare Apiartor against an existing dynamic-trace miner [5]. Our results [3] highlight the advantages of our static-trace mining.

2.2 ExitSafe

Incorrect handling of errors incurred after API invocations (in short, API errors) can lead to security and robustness problems, two primary threats to software reliability. Correct handling of API errors can be specified as formal specifications, verifiable by static checkers, to ensure dependable computing. But API error specifications are often unavailable or imprecise, and cannot be inferred easily by source code inspection. Based on our tracing infrastructure, we develop a novel technique called ExitSafe, for statically mining API error specifications automatically from software package repositories, without requiring any user input. Similar to Apiartor, ExitSafe employs the tracing infrastructure to approximate run-time API error behaviors with static traces. Frequent sequence mining [20] is used on these static traces to mine specifications that define the correct handling of errors for relevant APIs used in the soft-

ware packages. The mined specifications are then used to uncover API error-handling bugs. We have implemented our technique in a tool called ExitSafe, and validated the effectiveness of ExitSafe on 82 widely used open-source software packages with approximately 300KLOC in total [2].

2.3 IDeaMiner

Manually writing formal specifications (when known) for static verification can be cumbersome. Based on the tracing infrastructure, we implement IDeaMiner [4], which infers API details such as return values on success/failure, error flags, and return value type from the static traces. IDeaMiner implements simple data-flow extensions to the PDMC process to infer API details. Based on these inferred API details and the language syntax (user-provided, as a one-time AST database for a given language), our Specifier tool [1] translates user-specified generic API rules to concrete formal specifications verifiable by static checkers. Users can specify generic rules at an abstract level that needs no knowledge of the source code, system, or API details. We apply IDeaMiner and Specifier on 10 Redhat-9.0 packages that use the well-known POSIX-API library to expose around 200 robustness problems [4].

3 Searching Infrastructure

Open source projects available on the web include many valuable usage scenarios of the APIs of interest; these usage scenarios can help understand how to reuse those APIs and can give hints on API properties that must be satisfied while reusing those APIs. To gather such usage scenarios, we develop a searching infrastructure that leverages a code search engine such as Google code search [11] to gather a large number of code examples that reuse the APIs of interest. We analyze gathered code examples statically to extract the patterns of reuse and properties that should be satisfied for reusing those APIs.

While enjoying the benefits provided by a code search engine in terms of expanding the analysis scope to billions of lines of code available on the web, our infrastructure faces one new challenge: the code samples returned by a code search engine are often partial and not compilable, because a code search engine retrieves individual source files with usages of the given query API, instead of entire projects. Therefore, we develop several heuristics to tackle this challenge. We explain a heuristic used in our infrastructure through the code sample shown below:

```
public QueueSession test()   { ...
       return connect.createQueueSession(false,int);}
```

In this code sample, the method-invocation `createQueueSession` is a part of the return statement. The receiver object type of method-invocation

```
01: VerificationResult vr0, vr1;
02: Verifier verf = VerifierFactory.getVerifier(cName);
03: if(verf != null) {
04:   vr0 = verf.doPass1();
05:   if(vr0 != VerificationResult.VR_OK)
06:       return;
07:   vr1 = verf.doPass2(); ...
```

Figure 2. A code example of the BCEL library.

`createQueueSession` can be inferred by looking up the declaration of the `connect` variable. But as our infrastructure deals with a code sample that is partial, it is difficult to get the return type of the method-invocation without being able to access the corresponding method declaration in the downloaded file. However, the return type can still be inferred from the return type of the enclosing method declaration. As the enclosing method declaration has return type `QueueSession`, we can infer that the return type of method-invocation `createQueueSession` is `QueueSession` or one of its subtypes.

We next describe the tools developed based on our searching infrastructure.

3.1 NEGWeb

Neglected conditions, also referred as missing paths, are known to be an important class of software defects. In particular, 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. To detect neglected conditions, we develop a novel tool, called NEGWeb, that is based on our searching infrastructure. NEGWeb mines programming rules that must be satisfied for using an API from the related code examples gathered through a code search engine and applies mined rules to detect neglected conditions around that API in a given input application. As NEGWeb is developed based on our searching infrastructure, unlike the existing approaches such as Chang et al. [7], which focuses on mining one project code base, NEGWeb mines a much larger scope of code bases. In addition, NEGWeb's approach is based on simple statistical analysis and is more scalable than the approach by Chang et al., which is based on frequent sub-graph mining that suffers from scalability issues.

We explain our NEGWeb approach using the code sample related to the BCEL library shown in Figure 2. Initially, NEGWeb parses the code sample and builds a control flow graph. NEGWeb uses *dominance* and *data dependency*, and extracts preceding and succeeding conditions around the nodes that include any of the methods such as `doPass1` and `doPass2`. A rule candidate extracted from the example code sample is "`direct-const-check` on `return after doPass1` with `VerificationResult.VR_OK`", which describes that an equality check with constant `VerificationResult.VR_OK` should be performed on the

return object of the `doPass1` method. NEGWeb mines extracted rule candidates to compute frequent programming rules and applies mined rules to detect violations in a given input application.

We evaluated NEGWeb to detect violations in local code bases or open source code bases. NEGWeb confirms three real defects in Java code reported in the literature and also finds three previously unknown defects in a large-scale open source project called Columba (91,508 lines of Java code) that reuses 541 API classes and 2225 API methods. We also report a high percentage of real rules among the top 25 reported rules mined for five popular open source applications.

3.2 PARSEWeb

A common problem faced by developers while reusing existing frameworks or libraries is that the developers often know what type of object that they need, but do not know how to get that object with a specific method sequence. PARSEWeb, a tool developed based on our searching infrastructure, attempts to address the preceding issue by accepting a query of the form "Source → Destination" and suggests frequent method-invocation sequences that accept the *Source* object type as input and produce the *Destination* object type as output. We explain our PARSEWeb tool through an example query "`IEditorPart` → `ICompilationUnit`" related to the Eclipse IDE programming. PARSEWeb uses our searching infrastructure to gather related code examples with the usages of `IEditorPart` and `ICompilationUnit`. PARSEWeb analyzes gathered code examples statically and constructs a directed acyclic graph. PARSEWeb identifies nodes that contain the given Source and Destination object types and extracts a method-invocation sequence by calculating the shortest path between those Source and Destination nodes. A method-invocation sequence suggested by PARSEWeb is shown below:

```
IEditorPart iep = ...
IEditorInput editorInp = iep.getEditorInput();
IWorkingCopyManager wcm = JavaUI.getWorkingCopyManager();
ICompilationUnit icu = wcm.getWorkingCopy(editorInp);
```

Along with several heuristics described in the searching infrastructure, PARSEWeb includes other heuristics for clustering similar sequences and for ranking the clustered sequences. PARSEWeb uses an additional heuristic called query splitting that helps address the problem where code samples for the given query are split among different source files. In our evaluation, we show that PARSEWeb suggests solutions for real programming problems posted in developer forums of existing libraries and PARSEWeb performs better than existing related tools Prospector [15] and Strathcona [13]. Furthermore, as PARSEWeb is developed based on our searching infrastructure, unlike related tools, PARSEWeb is not limited to the queries of any specific set of libraries or frameworks.

4 Refactoring-Detection Infrastructure

Refactoring is a disciplined technique for improving the internal structure of a program while preserving its observable behavior. The problem with refactorings is that they can change an API and require software that uses the old API to be updated to use the new API. Conventionally, such updating is done manually, which is error-prone, tedious, and disruptive to the development process. Thus, such updating makes maintaining software expensive. This problem is exacerbated when refactorings change the APIs of reusable software components (e.g., libraries and frameworks). Previous study [9] of five popular components shows that refactorings cause more than 80% of API changes that were not backwards-compatible. We next describe the tool we developed based on our refactoring-detection infrastructure.

4.1 RefacLib

We have developed a novel technique and its supporting tool, `RefacLib`, to automatically infer refactorings that happened between two versions of a library. One of the key challenges is the size of real-life libraries (hundreds of KLOC). To reduce the search space, previous approaches [12, 21] assume that older program entities in one version are removed and replaced with refactored entities in the subsequent version. Thus, all these approaches start by analyzing only pairs of program elements that disappear from the old version and program elements that appear in the newer version. While this assumption is true for software that is built and used in-house, and does not need to be backwards-compatible, reusable software libraries follow a long *deprecate-replace-remove* cycle.

Moreover, most of the existing approaches [8, 21] rely on similarity of internal references among program entities to detect refactorings. However, some of the program entities might not have an internal reference from within a library. To overcome the lack of internal references in the case of libraries, we developed a new suite of analyses. `RefacLib` uses a fast and innovative syntactic analysis based on Shingles-encoding [6], a technique used in Information Retrieval to detect similarity in large bodies of text. The syntactic analysis produces pairs of program elements (across the two versions) that have similar bodies (e.g., methods with similar bodies). These pairs are fed into a suite of heuristic-based analyses that do not depend on references, and thus are able to analyze libraries.

Our approach consists of three phases. The *syntactic analysis* phase takes as input two versions of the com-

ponent, v1 and v2, and produces pairs of syntactically-matching entities. The *classification* phase classifies these pairs as candidates for various kinds of refactorings based on some syntactic checks. We currently support seven refactorings: `ChangeMethodSignature`, `RenameClass`, `PushDownMethod`, `RenamePackage`, `RenameMethod`, `PullUpMethod`, and `MoveMethod` (these were among the most frequently performed refactorings found in a previous study [9]). Finally, in the *heuristic-based analysis* phase, the algorithm computes a composite score for each pair based on various heuristics. We define a set of heuristics for each type of refactoring. For each candidate pair, the algorithm assigns a composite score that is a weighted sum of scores for all heuristics defined for that refactoring type. `RefacLib` reports as refactorings only the pairs with a score above a threshold. Our syntactic analysis phase returns a set of pairs of entities that are similar textually. These pairs of similar entities are classified by the *classification* phase as candidates of one of the seven refactoring types that we support. The classification is done using some syntactic checks.

These pairs of similar entities are suspected candidates of refactorings, but may contain many pairs that are not actual refactorings and need to be filtered out. For candidates of each refactoring type, our new heuristic-based analysis selects pairs that are real refactorings. In particular, the analysis gathers facts from the source code and Javadoc comments, and computes similarity measures to assign an overall score that reflects the likelihood of a candidate to be a refactoring. The facts and similarity measures vary for different types of refactorings. The process of classification and heuristic-based analysis iterates visiting the set of all pairs, and taking into account already detected refactorings. The process continues until a fixed point is reached. This process ensures that `RefacLib` detects pairs of entities that underwent multiple refactorings.

We compared the accuracy of `RefacLib` with that of a state-of-the-art tool `RefactoringCrawler` [8] for three frameworks, and two libraries. Our experiments show that `RefacLib` performs better than `RefactoringCrawler` when detecting refactorings in libraries. Moreover, the accuracy of `RefacLib` is comparable to that of `RefactoringCrawler` while detecting refactorings for the three framework subjects. Furthermore, the subjects that we chose were all real-world open-source components with up to 352K lines of code, and thus `RefacLib` is scalable to real-world software components.

References

[1] M. Acharya, T. Sharma, J. Xu, and T. Xie. Effective generation of interface robustness properties for static analysis. In *Proc. ASE 2006*, pages 293–296.

[2] M. Acharya and T. Xie. Static detection of API error-handling bugs via mining source code. Technical Report TR-2007-35, North Carolina State University Department of Computer Science, Raleigh, NC, October 2007.

[3] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proc. ESEC/FSE 2007*, pages 25–34.

[4] M. Acharya, T. Xie, and J. Xu. Mining interface specifications for generating checkable robustness properties. In *Proc. ISSRE 2006*, pages 311–320.

[5] G. Ammons, R. Bodik, and J. Larus. Mining specifications. In *Proc. POPL 2002*, pages 4–16.

[6] Broder. On the resemblance and containment of documents. In *Proc. SEQUENCES 1998*, pages 21–29.

[7] R.-Y. Chang, A. Podgurski, and J. Yang. Finding what's not there: a new approach to revealing neglected conditions in software. In *Proc. ISSTA 2007*, pages 163–173.

[8] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automatic detection of refactorings in evolving components. In *Proc. ECOOP 2006*, pages 404–428.

[9] D. Dig and R. Johnson. How do APIs evolve? a story of refactoring. *J. Softw. Maint. Evol.*, 18(2):83–107, 2006.

[10] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking push down systems. In *Proc. CAV 2000*, pages 232–247.

[11] Google Code Search Engine, 2006. `http://www.google.com/codesearch`.

[12] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE TSE*, 31(2):166–181, 2005.

[13] R. Holmes and G. Murphy. Using structural context to recommend source code examples. In *Proc. ICSE 2005*, pages 117–125.

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

[15] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. PLDI 2005*, pages 48–61.

[16] J. Pei, H. Wang, J. Liu, K. Wang, J. Wang, and P. Yu. Discovering frequent closed partial orders from strings. *IEEE TKDE*, 18(11):1467–1481, 2006.

[17] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *Proc. ASE 2007*, pages 377–380.

[18] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. ASE 2007*, pages 204–213.

[19] S. Thummalapenta and T. Xie. NEGWeb: Static defect detection via searching billions of lines of open source code. Technical Report TR-2007-24, North Carolina State University Department of Computer Science, Raleigh, NC, August 2007.

[20] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proc. ICDE 2004*, pages 79–90.

[21] P. Weissgerber and S. Diehl. Identifying refactorings from source-code changes. In *Proc. ASE 2006*, pages 231–240.

[22] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. Perracotta: Mining temporal API rules from imperfect traces. In *Proc. ICSE 2006*, pages 282–291.