Consensus-Based Mining of API Preconditions in Big Code *

Hoan Anh Nguyen
Iowa State University, USA
hoan@iastate.edu

Robert Dyer
Bowling Green State University, USA
rdyer@bgsu.edu

Tien N. Nguyen Hridesh Rajan
Iowa State University, USA
{tien,hridesh}@iastate.edu

Abstract

Formal specifications for APIs help developers correctly use them and enable checker tools automatically verify their uses. However, formal specifications are not always available with released APIs. In this work, we demonstrate an approach for mining API preconditions from a large-scale corpus of open-source software. It considers conditions guarding API calls in client code as potential preconditions of the corresponding APIs. Then it uses consensus among a large number of API usages to keep the ones appearing in the majority. Finally, the mined preconditions are ranked based on their frequencies and reported to users.

Categories and Subject Descriptors D.2.1 [*Software Engineering*]: Requirements/Specifications.

Keywords API Preconditions, Software Mining, JML, Big Code.

1. Introduction

Modern software is usually developed using frameworks and libraries via invoking and extending their application programming interface (API) classes and methods. Developers must respect API specifications in order to use them correctly in their code. For an API method, one part of its specification is the set of conditions that must hold before it is invoked. These conditions are called preconditions of the API. For example, in the standard Java Development Kit (JDK) library, API method substring(begin, end) in package java.lang requires three preconditions: (1) begin $\geq$ 0, (2) end $\leq$ this.length() and (3) begin $\leq$ end.

Misunderstanding API preconditions could lead to bugs in software. For example, when migrating code from C# to Java, developers in project MSSCodeFactory[1] assumed that, in the above substring API, the second argument is the length of the substring, which is the case in the corresponding API in C#. Thus, in many calls to this API in their code, they passed value 1 to this second argument without checking if it satisfied the preconditions regarding the length of original string and the begin index in the first argument or not. Those calls caused IndexOutofBounds exceptions and were later fixed in revision 2463.

We have conducted a large-scale study [3] on the bug fixing histories of 3,413 Java projects on SourceForge and found more than 4 thousand potential bugs related to checking preconditions. Manual checking on 100 randomly-sampled ones confirmed that 80% of samples are actual precondition violation bugs.

Ideally, the preconditions should be manually specified by the API designer(s). One could also read the documentation of the APIs and even the source code to derive preconditions and convert them to the suitable formats. However, this manual process is time-consuming, tedious and error-prone leading to the practice that not many APIs are released with formal specifications or even informal specifications (such as Javadoc).

This work introduces an approach that puts forth the idea of mining API specifications that combines both static analysis and source code mining from a very large code corpus in open-source repositories to *derive the preconditions of APIs* in libraries and frameworks. Our approach is based on the observation that developers commonly check preconditions of methods before calling them. This style of programming makes the software more resilient to the unexpected inputs, thus, avoids unexpected program behaviors and bugs. We expect that the **APIs' preconditions would appear frequently in a large corpus of open-source projects** that contain a very large number of the usages of those APIs, while **project-specific conditions will occur less frequently**. We *combine the strength of both static analysis approaches (via control dependency analysis) and mining software repositories (MSR) approaches (via mining)* to make it scale to large corpus. Importantly, we can derive preconditions for a large number of APIs or entire library at the same time.

2. Consensus-Based Mining Approach

Figure 1 shows our approach overview. It takes as input a set A of API methods under analysis and client projects P in the code corpus and mines the preconditions for all APIs in the following steps.

First, we **scan** for all methods that call at least one API in A. We parse all projects in P and resolve types for all expressions to find API calls. Since most projects in P are not compilable due to missing dependencies, we use several heuristics for partial parsing to resolve types as much as possible. Then, for each (calling) method, we **build** its **control flow graph** (CFG) and use CFG to **analyze** the **control dependence relation** for all API calls in the method. For each called API, we extract $\Omega(p)$ containing all API call sites having p as the guard condition.

Since the preconditions are collected from multiple call sites and projects written by different developers and in different styles, the same precondition could be expressed in different forms, e.g., $a>b$, $b<a$ and $(a-b)>0$. Thus, after extracting, we **normalize** them so that they are comparable between call sites. We also analyze the preconditions to **infer** additional ones which are not directly present in the client code. For example, a non-strict inequality ($a\geq b$ or $a\leq b$) might not be checked directly but in the forms of

* The full paper has been published in FSE '14 [3].

[1] http://msscodefactory.sourceforge.net/

SPLASH Companion'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814189.2816271

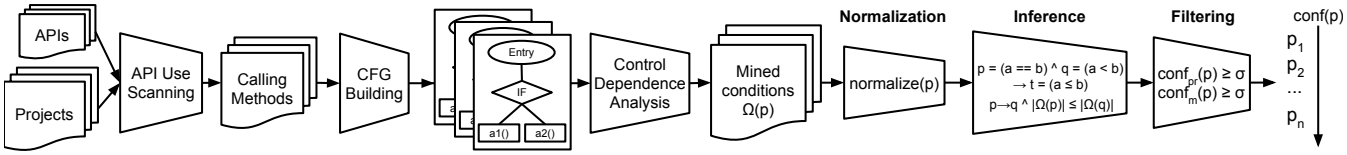

Figure 1. Approach Overview: Consensus-based Mining of Preconditions.

strict inequality ($a>b$ or $a<b$) and equality ($a==b$) conditions at different call sites. If there are sufficient occurrences of those two preconditions, we create the corresponding non-strict inequality precondition with the set of call sites derived from their two sets.

Finally we **filter** out non-frequent preconditions and **rank** the remaining ones in our final result. Filtering and ranking is based on the confidence of a precondition p for an API a which is measured as the ratio between the number of code locations checking p before calling a over the total number of locations calling a.

3. Precondition Mining Tool

We implemented our approach as a plug-in to Eclipse IDE. A user can activate the plug-in by showing its view in Eclipse. Currently, the tool provides four functions via four corresponding buttons in its view: selecting client code corpus, selecting library to be analyzed, showing statistics on the use of the library in the code corpus and starting precondition mining.

A user can **select a code corpus** by entering the path to a folder containing all projects and specifying if the projects' folders contain snapshots or repositories of the projects. Currently, we support Subversion (SVN) and Git repositories.

For **selecting a library**, a user can choose a project or jar file already loaded in the current workspace, or an external jar file. When a user selects to **show the statistics**, the tool will parse the library to get all API classes and methods and execute the scanning step in our approach to find all uses of these APIs in the code corpus. The result will be shown in the content of the tool's view. For example, JDK 6 has more than 11 thousand public methods, 63% of which are called at least once in our SourceForge projects and 25% of method calls in SourceForge are to JDK APIs.

When a user **starts mining**, he/she will be able to customize different parameters in our approach. He/she can choose the value for k of the top-k mined preconditions, the percentage of the dataset to be used or options for including/excluding different components in our approach. The larger k or the percentage is, the more preconditions are mined. There are two options for the output format: XML and Java project. In both cases, the output will be a folder with the same file structure as the library. For the second format, the output is a Java project with the same package/file/class/method structure with the library where methods do not have body and are annotated with mined preconditions in JML syntax. The user can also request to show preconditions of a selected method using context menu.

4. Demonstration Overview

This demonstration shows how to use our tool to mine API preconditions from our datasets of thousands of open-source projects.

- background and introduction to our tool,
- instructions on how to install our tool,
- showing an example use of our tool to mine a popular JDK API,
- explaining the format of the mined preconditions,
- demonstrating what to do with the mining results, and
- pointers on where to find help.

5. Presenter Biographies

Hoan Anh Nguyen is a post-doctoral researcher at Iowa State University. His expertise is in software evolution and mining software repositories. Robert Dyer has helped design and implement several programming languages. Currently his research is focused on mining software repositories with the Boa project. Tien N. Nguyen is a faculty of the Electrical and Computer Engineering and Computer Science departments at Iowa State University. His research focus is on software evolution, software mining and analysis. Hridesh Rajan is a computer science faculty at Iowa State University, where he works on the Boa infrastructure, and the Panini language.

6. Related Work

Our approach is closely related to the precondition mining work from Ramanathan, Grama, and Jagannathan (RGJ) [5] which also integrates *static program analysis* with *data mining*. The key difference is that our approach operates on a *large-scale corpus* of client programs that contain API calls and mine preconditions from predicates across projects. In contrast, RGJ is designed to perform within an *individual client program* containing the APIs' call sites.

There are several approaches for mining specifications using *dynamic analysis*. Daikon [1] automatically detects invariants in a program via running test cases. Wei *et al.* [7] infer complex post-conditions from simple programmer-written contracts in C# code.

Our work is also related to *static approaches* for mining temporal specifications among program elements [2, 4, 6, 8]. The mined specifications from those approaches express the orders among method calls (and control structures) rather than preconditions.

Acknowledgments

This work was supported in part by the US NSF under grants CCF-15-18897, CCF-15-18776, CNS-15-13263, CNS-15-12947, CCF-14-23370, CCF-13-49153, CCF-13-20578, TWC-12-23828, CCF-11-17937, CCF-10-17334, and CCF-10-18600.

References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. ICSE '99.

[2] N. Gruska, A. Wasylkowski, and A. Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. ISSTA '10.

[3] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of apis in large-scale code corpus. FSE '14.

[4] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. ESEC/FSE '09.

[5] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. PLDI '07.

[6] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. ESEC-FSE '07.

[7] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. ICSE'11.

[8] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending API usage patterns. ECOOP '09.