Towards Combining Usage Mining and Implementation Analysis to Infer API Preconditions

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

Tien N. Nguyen
The University of Texas at Dallas, USA
tien.n.nguyen@utdallas.edu

Hridesh Rajan
Iowa State University, USA
hridesh@iastate.edu

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

ABSTRACT

The preconditions of an API method are constraints on the states of its receiver object and arguments intended by the library designer(s) to correctly invoke it in the client code. There have been two main kinds of approaches for automatically inferring API preconditions. The first kind of approaches mines the frequently checked conditions guarding the API usages in the client code and generalize them into preconditions. The second kind of approaches analyzes the implementation of the API to compute preconditions.

In this paper, we report an observation that the usage-based approach often produces preconditions stronger than those intended while the implementation-based produces weaker ones. Our finding calls for a new direction of integrating those kinds of precondition inference approaches and refinement solutions to reduce the differences between sets of inferred preconditions.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**;

KEYWORDS

API Preconditions, Specification Mining, Program Analysis

ACM Reference Format:
Hoan Anh Nguyen, Tien N. Nguyen, Hridesh Rajan, and Robert Dyer. 2018. Towards Combining Usage Mining and Implementation Analysis to Infer API Preconditions. In *Proceedings of the 1st ACM SIGSOFT International Workshop on Automated Specification Inference (WASPI '18), November 9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3278177.3278182

1 INTRODUCTION

Libraries enable pragmatic software reuse by allowing developers to access the library's functionality via Application Programming Interfaces (APIs). To correctly use an API method, developers must ensure the receiver and arguments conform to certain conditions that are *intended by the library/framework's designers*. Such conditions are referred to as the *preconditions* of an API. Violating a precondition will lead to undesired behavior and possibly an exception. For example, `begin <= end` is a precondition of `String.substring(begin,end)`.

WASPI '18, November 9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-6057-9/18/11…$15.00
https://doi.org/10.1145/3278177.3278182

Preconditions, and specifications in general, tell API users about intended behaviors of APIs and make verification possible. Useful, comprehensible, and efficiently checkable formal specifications can help contain the cost of developing high assurance, reliable, and secure software systems [2, 5]. Ideally, designers would specify all the preconditions for all of their API methods in the library/framework. However, in practice, due to the large number of APIs, not all libraries/frameworks are delivered with the specified preconditions.

To reduce the manual effort required to define the preconditions for software libraries, several approaches have been introduced to automatically derive the preconditions for the API methods. Broadly speaking, existing precondition mining approaches can be classified into two kinds: mining software repositories (MSR) based approaches and program-analysis based approaches.

The first kind relies on *(MSR) techniques* applied to the *code using the APIs* (called *client code*). Specifically, they analyze *the usages of APIs* at call sites in the client code to derive the preconditions [3, 6]. The idea behind these approaches is that the preconditions of an API method could be mined from the *guard conditions* of the calls to that API in the client code. Let us call them *usage-based approaches*.

The second kind of approaches is program analysis-based. Specifically, they rely on analyzing the *implementation of APIs*. They analyze the guard conditions along the exception paths to detect preconditions. Most of those approaches rely on static analysis [1, 6]. Let us call them *implementation-based approaches*.

Despite their successes, no existing research has studied the nature and the relation between the two kinds of approaches. In this paper, we report two observations. First, ***the usage-based approach often produces preconditions stronger than those intended by the APIs' designers***. That is, the set of valid values for the arguments of an API according to the mined rules from the client code is *smaller* than the set of valid values allowed by the true precondition of the API. On the other hand, ***the implementation-based approach often produces weaker ones***. That is, violating these preconditions will lead to exception behaviors, however, satisfying them does not guarantee no exception behaviors. The former often produces the *upper bounds* and the latter often produces the *lower bounds* of the preconditions.

Our finding calls for a direction of integrating precondition inference approaches and refinement solutions to reduce the difference between upper and lower bounds of the inferred preconditions.

2 MOTIVATION FOR THE COMBINATION

We will first use several examples to demonstrate the limitations of the usage- and implementation-based inference techniques.

Example 1. Too weak preconditions when inferring from implementation. Fig. 1a shows the implementation of method

```
1 public E get(int index) {
2   rangeCheck(index);
3   return elementData(index); }
4 private void rangeCheck(int index) {
5   if (index >= size) throw new IndexOutOfBoundsException(...); }
6 E elementData(int index) { return (E) elementData[index]; }
```

a) java.util.ArrayList

```
1 public E removeFirst() { return remove(header.next); }
2 private E remove(Entry<E> e) {
3   if (e == header) throw new NoSuchElementException();
4   ... }
```

b) java.util.LinkedList

Figure 1: Excerpts of implementations.

ArrayList.get(index). This API returns the element at the specified position index of the list. It throws IndexOutOfBoundsException when accessing the array on line 6 if the specified position index is negative, or greater than or equal to the size of the list. In its implementation, in method rangeCheck (lines 4–5), index is explicitly checked against the size of the list, size, because the length of the internal array could be greater than the actual size. However, the developer did not check the precondition index >= 0 and let this unchecked exception be thrown at runtime. This practice could be an optimization for performance purposes. However, this would make an automated technique based on implementation *miss the precondition* index >= 0 and the inferred precondition *weaker than expected*.

On the other hand, due to the practice of defensive programming, developers of client code of this API often write explicit checks of index >= 0 and index < size() before calling it to avoid IndexOutOfBoundsException. In this example, the preconditions mined from usages match exactly with the expected ones.

Example 2. Inaccessible preconditions when inferring from implementation. Fig. 1b shows the implementation of method LinkedList.removeFirst() whose purpose is to remove and return the first element from the list. It throws NoSuchElementException if invoked on an empty list, thus, its precondition is !isEmpty(). In the client code, developers usually check this condition before calling it, so this precondition could be mined from the API usages. However, it could not be inferred from the API's implementation because it is never checked there. Instead, an equivalent condition header.next == header is checked (line 3). The condition header.next == header is not useful for the API users since it is inaccessible from outside of the library implementation. In this example, *the precondition mined from implementation is inaccessible*, thus, not useful for API users, and *exposes implementation details* which is not desirable [4].

Example 3. Too strong preconditions when mining from usages. String.substring(begin) returns a substring in the string that starts from the character at the index begin and extends to the end of the string. The expected preconditions are begin >= 0 and begin < length(). In the usages of this API, developers usually check the precondition begin < length() before calling it. However, instead of begin >= 0, developers often check the stronger condition of begin > 0 because when begin is 0 the API returns the original string, which is not very useful. This example illustrates a phenomena that when using APIs, *developers could check conditions which are stronger than the actual preconditions of the APIs*.

2.1 Observations

The above examples show that both implementation-based and usage-based approaches face the problem of incompleteness. The preconditions inferred from the former are likely to be weaker than the expected ones. There would exist values of the receiver object and/or argument(s) satisfying the inferred preconditions, but still leading to exception. In the example with ArrayList.get(index), an implementation-based solution can only guarantee that an exception will be thrown if index > size() but does not know that there are values of index in the range $(-\infty, 0)$ that still lead to an exception.

In contrast, the preconditions mined from the usage-based approach are likely to be stronger than expected. There would exist values of the receiver object and/or argument(s) violating the mined preconditions, but not causing exception. In the third example with String.substring(begin), the preconditions mined from a usage-based solution imply that passing a value of begin in the interval (0, length()) would not throw exception, but cannot tell whether the value of begin == 0 would or not.

Importantly, the examples suggest that the boundary of the expected preconditions could be in the gap between the results inferred from the two approaches. For example, this gap is the interval index < 0 in the first example with ArrayList.get(index), isEmpty() in the second example with LinkedList.removeFirst(), and begin == 0 in the third example with String.substring(begin).

This observation suggests that we could perform an additional step to validate the gap. For example, we can validate if the program would throw an exception or not if passing a negative value of index to ArrayList.get(index) or calling removeFirst() on a LinkedList in the state satisfying isEmpty() or passing begin == 0 when calling String.substring(begin). If an exception is thrown, like in the first two cases, we would conclude that the gap is not part of the preconditions. Otherwise, like in the last case, the gap would be included as part of the preconditions.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1518897,1518776,1512947. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles*, SOSP'01, pages 57–72. ACM, 2001.

[2] A. Hall. Seven myths of formal methods. *IEEE Softw.*, 7(5):11–19, Sept. 1990.

[3] H. A. Nguyen, R. Dyer, T. N. Nguyen, and H. Rajan. Mining preconditions of apis in large-scale code corpus. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 166–177, New York, NY, USA, 2014. ACM.

[4] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, Dec. 1972.

[5] H. Rajan, T. N. Nguyen, G. T. Leavens, and R. Dyer. Inferring behavioral specifications from large-scale repositories by leveraging collective intelligence. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 579–582, Piscataway, NJ, USA, 2015. IEEE Press.

[6] M. K. Ramanathan, A. Grama, and S. Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 123–134. ACM, 2007.