Poster: Using Consensus to Automatically Infer Post-conditions

Jingyi Su
Bowling Green State University
Bowling Green, Ohio
jsu@bgsu.edu

Mohd Arafat
Bowling Green State University
Bowling Green, Ohio
marafat@bgsu.edu

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

ABSTRACT

Formal behavioral specifications help ensure the correctness of programs. Writing such specifications by hand however is time-consuming and requires substantial expertise. Previous studies have shown how to use a notion of consensus to automatically infer pre-conditions for APIs by using a large set of projects. In this work, we propose a similar idea of consensus to automatically infer post-conditions for popular APIs. We propose two new algorithms for mining potential post-conditions from API client code. The first algorithm looks for guarded post-conditions that test the value returned from the API and throws an exception. The second algorithm looks for values flowing from the API to another API with already known preconditions, which recommends them as post-conditions of the first API.

CCS CONCEPTS

• **Theory of computation → Logic and verification**;

KEYWORDS

specification inference, consensus, post-condition

ACM Reference Format:
Jingyi Su, Mohd Arafat, and Robert Dyer. 2018. Poster: Using Consensus to Automatically Infer Post-conditions. In *ICSE '18 Companion: 40th International Conference on Software Engineering Companion, May 27-June 3, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3183440.3195096

1 INTRODUCTION

Behavioral interface specifications in the form of pre- and post-conditions for methods, invariants for classes, etc [2, 3] help ensure the correctness of programs. However writing such specifications manually is a huge burden on programs, incurring large cost in both money and time and requiring substantial expertise [4, 5].

Much previous work has been done on automating the inference of such specifications [4–6]. Prior work focused on leveraging the collective intelligence found in Big Code to automatically infer pre-conditions for library functions [4, 5]. This allows inferring the specification of an API whose code may not be available. Having specifications for commonly used APIs can further aid tools and humans in developing specifications for client code.

ICSE '18 Companion, May 27-June 3, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3195096

In this work, we extend the notion of consensus-based mining of specifications to automatically infer post-conditions of API methods. We extend the approach of Nguyen et al. [4] and replace the pre-condition mining logic with several new post-condition inference algorithms. In the future we hope to fully implement and evaluate the approach's effectiveness using Boa [1].

2 APPROACH

Our approach builds off the approach proposed by Nguyen et al. [4]. An overview of the approach is shown in Figure 2. In this figure, the majority of the pre-condition inference approach used by the prior work is re-used while the pre-condition specific components are replaced with post-condition inference algorithms (box in the top-right of the figure). The function $\Omega(p)$ gives the set of all mined postconditions for a particular project. In this work, we present two possible algorithms for post-condition inference.

The general steps of the overall inference algorithm include: finding calls to a set of target APIs, inferring post-conditions for each target API by examining code at the client call sites for the API, normalizing the inferred conditions ($x > 0$ and $0 < x$ are the same), using logical inference to find more or strengthen the conditions, then filtering and ranking them to help remove noise and then propose the results to the user. Next we describe the two new post-condition inference algorithms.

Algorithm 1 The first algorithm attempts to find the pattern shown in Figure 1, where a target API is called and the return value is stored in a variable. If the variable is not modified and we see a guarded throws clause, where the guard uses the variable in the condition, then we assume the negation of the condition is a possible post-condition of the API. This is a form of defensive programming.

```
1 o = API();
2 ... // code that does not modify o
3 if (o > 5) // "o <= 5" is a potential post-condition
4   throw Exception();
```

Figure 1: Code example with a possible post-condition guard

Algorithm 2 If a value returned from an API flows into a different API, assuming it was unaltered on the way, then we may be able to gain knowledge from that pattern. If we have a large set of pre-conditions and know the second API's pre-conditions we can use that to infer the post-condition of the first API. An example of such code is shown in Figure 3.

An overview of the inference algorithm is shown in Figure 4. Here we have used some helper functions: `rhs(e)` means the right-hand side of an assignment expression, `lhs(e)` means the left-hand side of an assignment expression, `vars(e)` means the variables in

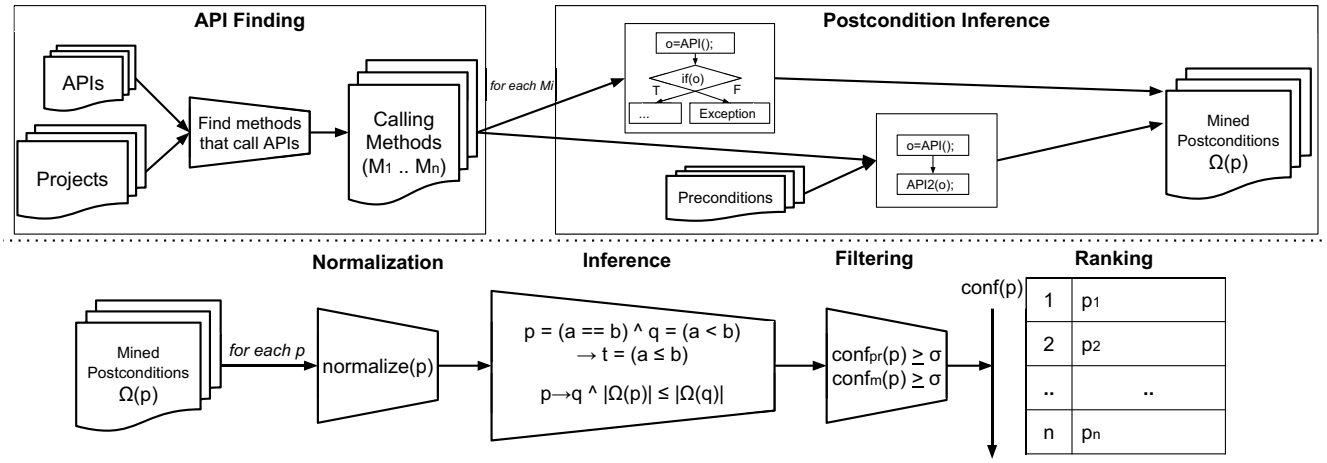

Figure 2: Overview of consensus based post-condition inference (based on [4, Fig.3]) - new components in top-right box

```
o = API();
... // code that does not modify o
anotherAPI(o);
```

Figure 3: Code example requiring pre-conditions

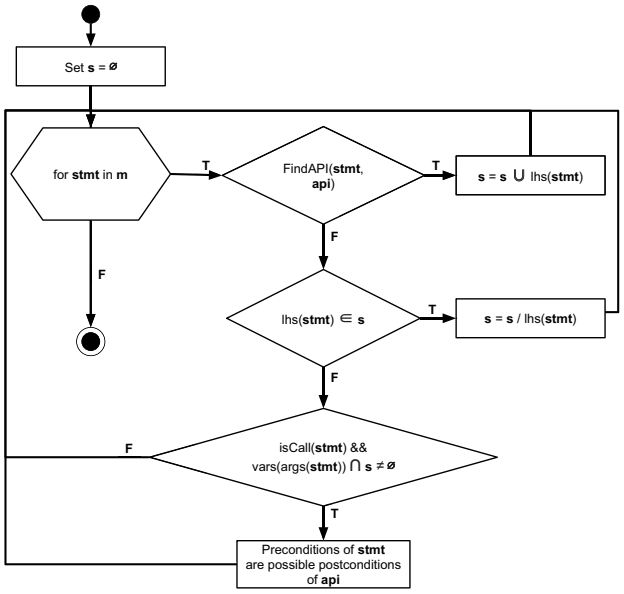

Figure 4: Use pre-conditions to infer post-conditions

the expression, and `args(e)` means the arguments to a method call. `FindAPI(stmt, api)` is a function for checking if the statement is a method call calling a target API.

The algorithm examines each statement in a method, looking for calls to a target API. If it finds such a call, it keeps track of the variable name used to store the result. If it sees an assignment to any variable being tracked, it removes that variable from the set. Finally, if it sees a method call using one of the tracked variables, it will suggest the pre-conditions of the second API as possible post-conditions of the target API.

This algorithm examines a single method sequentially, which may lead to imprecision. In the future we hope to strengthen this algorithm by using program analysis techniques such as generating control-flow graphs and data-flow analysis to properly track the use-def relationship of the variables.

3 CONCLUSION

Automatically inferring API specifications aids developers wishing to formally specify their code. In this paper, we presented several algorithms for inferring post-conditions of API methods using a notion of consensus. In the future we hope to implement and evaluate the approach using the Boa infrastructure and dataset [1].

Acknowledgements This work was supported by the National Science Foundation under grants 1512947 and 1518776.

REFERENCES

[1] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N. Nguyen. 2013. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 422–431.

[2] John V Guttag, James J Horning, and Jeannette M Wing. 1985. The Larch family of specification languages. *IEEE Software* 2, 5 (1985).

[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. 2006. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes* 31, 3 (May 2006), 1–38.

[4] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. 2014. Mining Preconditions of APIs in Large-scale Code Corpus. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'14)*. 166–177.

[5] Hridesh Rajan, Tien N. Nguyen, Gary T. Leavens, and Robert Dyer. 2015. 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)*. 579–582.

[6] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. 2011. Inferring Better Contracts. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. 191–200.