Poster: An Algorithm and Tool to Infer Practical Postconditions

John L. Singleton
Department of Computer Science
University of Central Florida
Orlando, FL, USA
jls@cs.ucf.edu

Gary T. Leavens
Department of Computer Science
University of Central Florida
Orlando, USA
leavens@cs.ucf.edu

Hridesh Rajan
Department of Computer Science
Iowa State University
Ames, USA
hridesh@iastate.edu

David Cok
GrammaTech, Inc
USA
dcok@grammatech.com

ABSTRACT

Manually writing pre- and postconditions to document the behavior of a large library is a time-consuming task; what is needed is a way to automatically infer them. Conventional wisdom is that, if one has preconditions, then one can use the strongest postcondition predicate transformer (SP) to infer postconditions. However, we have performed a study using 2,300 methods in 7 popular Java libraries, and found that SP yields postconditions that are exponentially large, which makes them difficult to use, either by humans or by tools.

We solve this problem using a novel algorithm and tool for inferring method postconditions, using the SP, and transmuting the inferred postconditions to make them more concise.

We applied our technique to infer postconditions for over 2,300 methods in seven popular Java libraries. Our technique was able to infer specifications for 75.7% of these methods. Each of these inferred postconditions was verified using an Extended Static Checker. We also found that 84.6% of resulting specifications were less than 1/4 page (20 lines) in length. Our algorithm was able to reduce the length of SMT proofs needed for verifying implementations by 76.7% and reduced prover execution time by 26.7%.

KEYWORDS

Specification inference, JML, predicate transformers

ACM Reference Format:
John L. Singleton, Gary T. Leavens, Hridesh Rajan, and David Cok. 2018. Poster: An Algorithm and Tool to Infer Practical Postconditions. 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/10.1145/3183440.3194986

1 INTRODUCTION

There is a rich body of work on specification mining and inference [1–3]. A major category of these approaches analyze call sites of an API method to collect the set of predicates at each of these call sites and then use mining techniques such as frequent items mining to infer preconditions [4, 6]. Another body of work has focused on analyzing call sites to mine temporal patterns over API method calls, e.g. [5, 7]. These works have not focused on inferring postconditions.

The main contribution of this work is a postcondition inference technique to lower the cost of producing specifications. Our technique for method postcondition inference is based on the strongest postcondition predicate transformer: starting from a precondition (e.g., `true`) this uses the body of a method to produce a logical formula that can be converted into a specification as shown in Figure 1. A key aspect of our work is a novel algorithm for simplifying the often complex specifications that result from the strongest postcondition transformer (SP), which is discussed in Section 2. An illustration of our technique that shows the inferred specification for the method in Figure 1a is shown in Figure 1c.

2 TRANSMUTING SP WITH THE FAR ALGORITHM

The main problem with formulas computed by *sp* is that when the results are converted into specifications, they can become very large, deeply nested, and repetitive. This effect appears in the `IF` SP rule, which computes the disjunction of two different branch paths according to the equation $(\mathsf{sp}\ S_1(P \wedge B)) \vee (\mathsf{sp}\ S_2(P \wedge \neg B))$. where P is a precondition, B is the branch conditions, and S_1 and S_2 are the body of the `IF` and `ELSE` branches respectively.

At each branch point in the program the branch's precondition is fed back into *sp* as an argument, which causes the number of specification cases in the resulting specification to grow exponentially. In this section we describe the Flattening and Recombination (FAR) algorithm that produces an equivalent specification that is shorter and more practical than the input specification. We show the effect of our algorithm on a small example in Figure 1a, 1b and 1c.

3 EVALUATION

We designed a series of experiments designed to examine the effectiveness of Strongarm and FAR at inferring practical specifications. We selected a cross section of popular Java libraries: JUnit4 (JU4), JSON-Java (JJA), Commons-CSV (CSV), Commons-CLI (CLI),

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

ACM ISBN 978-1-4503-5663-3/18/05.
https://doi.org/10.1145/3183440.3194986

```
//@ requires true;
public int cmp(int a, int b){
    int c = a;
    if (c < b) {
        return -1;
    } else {
        if (c > b) {
            return 1;
        }
        return 0;
    }
 }
```

(a)

```
public normal_behavior
  requires true;
  {|
    requires (c < b);
    ensures true;
    ensures \result == -1;
    ensures c == a;
  also
  {|
    requires !(c < b);
    requires (c > b);
    ensures true;
    ensures \result == 1;
    ensures c == a;
  also
    requires !(c < b);
    requires !(c > b);
    ensures true;
    ensures \result == 0;
    ensures c == a;
 |}
|}
```

(b)

```
public normal_behavior
  requires !(a < b);
  {|
    requires (a > b);
    ensures \result == 1;
  also
    requires !(a > b);
    ensures \result == 0;
  |}
  also
    requires (a < b);
    ensures \result == -1;
```

(c)

Figure 1: (a) A simple comparison function. Using the standard definition of strongest postconditions, one obtains the postcondition shown in (b) for `cmp`. This postcondition contains a lot of repetition, tautologies, and is nearly twice the size of the original procedure. (c) The specification for `cmp` after being processed by our tool. Note that not only is this final specification shorter, it is lexically different than (b) and takes advantage of the fact that `!(a < b)` is true in multiple branches.

Commons-Codec (COD), Commons-Email (EMA), and Commons-IO (CIO). Code metrics pertaining to these libraries are summarized in Table 1.

Table 1: Code metrics for APIs used in evaluation.

API Name	SLOC	Methods	Files	Version
JUnit4	10,018	1,230	193	4.13
JSON-Java	3,201	200	18	20160212
Commons-CSV	1,501	158	10	1.4
Commons-CLI	2,666	194	22	1.3.1
Commons-Codec	6,607	509	60	1.10
Commons-Email	2,734	192	22	1.4
Commons-IO	9,836	955	115	2.5
Total	**36,563**	**2,331**	**440**	-

Discussion. Our evaluation determined that our technique was able to infer specifications for 75.7% of these methods. We also found that 84.6% of resulting specifications were less than 1/4 page (20 lines) in length. Our algorithm was able to reduce the length of SMT proofs needed for verifying implementations by 76.7% and reduced prover execution time by 26.7%. These results indicate the effectiveness of our technique at producing smaller and possibly easier to understand specifications. We plan to further investigate the impact of our technique on usability of inferred specifications in future work.

4 ACKNOWLEDGMENTS

The work of the authors was supported in part NSF grants 1518789 and CNS1228695.

REFERENCES

[1] Gordon Fraser and Andreas Zeller. 2011. Generating Parameterized Unit Tests. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, New York, NY, USA, 364–374. https://doi.org/10.1145/2001420.2001464

[2] Mark Gabel and Zhendong Su. 2012. Testing Mined Specifications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE '12)*. ACM, New York, NY, USA, Article 4, 11 pages. https://doi.org/10.1145/2393596.2393598

[3] Anh Cuong Nguyen and Siau-Cheng Khoo. 2011. Extracting Significant Specifications from Mining Through Mutation Testing. In *Proceedings of the 13th International Conference on Formal Methods and Software Engineering (ICFEM'11)*. Springer-Verlag, Berlin, Heidelberg, 472–488. http://dl.acm.org/citation.cfm?id=2075089.2075130

[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 2014)*. ACM, New York, NY, USA, 166–177. https://doi.org/10.1145/2635868.2635924

[5] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Graph-based Mining of Multiple Object Usage Patterns. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 383–392. https://doi.org/10.1145/1595696.1595767

[6] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Static Specification Inference Using Predicate Mining. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. ACM, 123–134. https://doi.org/10.1145/1250734.1250749

[7] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting Object Usage Anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering (ESEC-FSE '07)*. ACM, 35–44. https://doi.org/10.1145/1287624.1287632