

Inferring Concise Specifications of APIs

John L. Singleton

accesso Technology Group, plc
jls@cs.ucf.edu

Gary T. Leavens

University of Central Florida
leavens@cs.ucf.edu

Hridesh Rajan

Iowa State University
hridesh@iastate.edu

David R. Cok

GrammaTech, Inc
dcok@grammatech.com

Abstract—Modern software relies on libraries and uses them via application programming interfaces (APIs). Correct API usage as well as many software engineering tasks are enabled when APIs have formal specifications. In this work, we analyze the implementation of each method in an API to infer a formal postcondition. Conventional wisdom is that, if one has preconditions, then one can use the strongest postcondition predicate transformer (SP) to infer postconditions. However, SP yields postconditions that are exponentially large, which makes them difficult to use, either by humans or by tools. Our key idea is an algorithm that converts such exponentially large specifications into a form that is more concise and thus more usable. This is done by leveraging the structure of the specifications that result from the use of SP. 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 which 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 technique was able to reduce the length of SMT proofs needed for verifying implementations by 76.7% and reduced prover execution time by 26.7%.

Index Terms—specification inference, postconditions

I. INTRODUCTION

Frameworks and libraries are the basic building blocks of modern software, using them via their application programming interfaces (APIs), which are collections of classes and their methods. A specification for an API method is a contract [7], [36]. An API method’s *precondition* is a predicate that must hold when it is called; an API method’s *postcondition* is a predicate that the method ensures will hold when it completes. For instance, the `push(item)` method of `java.util.Stack` ensures that `item` is the top of the stack.

Knowledge of postconditions is very useful for automated software engineering tools such as formal verification of program correctness [3], [4], [52], test case generation [23], test oracle creation [39], detecting bugs [17], [32], [50], design by contract [7], [48], etc. Popular formal specification tools include ESC/Java [20], Bandera [9], Java Path Finder [2], JMLC [29], Kiasan [14], Code Contracts [1], etc., regularly use postconditions in place of a method call to gain scalability.

Unfortunately, pre- and postconditions specifications are not widely available, even for widely-used libraries. The main reason seems to be that the cost of writing such specifications is similar to the cost of writing the code itself [30]. To decrease the cost of writing specifications, several sets of techniques have been proposed to automatically derive specifications.

A first set of techniques analyzes call sites of an API method to collect a set of predicates at each of these call sites and then uses mining techniques, such as frequent items mining, to infer preconditions [40], [44]. Another body of work has focused on analyzing call sites to mine temporal patterns over API method calls, e.g. [26], [34], [37], [41], [47], [51]. However, these works do not infer postconditions.

A second set of techniques uses static analysis on the code of the API method to infer specifications; e.g. Cousot *et al.* uses abstract interpretation [10] to infer preconditions, and Buse *et al.* uses symbolic execution [8] to infer conditions leading to exceptions. However, Buse *et al.* do not infer conditions under which the API method terminates normally and Cousot *et al.* do not infer postconditions.

A third set of techniques uses *dynamic* approaches to mining specifications [3], [11], [12], [18], [22], [33], [35], [43], [49], [53]. Some of these works infer temporal patterns over API method calls [22], [33], [35], [53], object-usage specifications [43], and others strengthen existing specifications [49]. Although Daikon [18] can infer postconditions, its inference depends on the presence of an adequate test suite [28], [42].

This paper proposes a technique for inferring postconditions that combines forward symbolic execution [24] with predicate transformers [15]. Starting from a precondition (e.g., `true` or one inferred using prior techniques [40], [44]) our technique uses the body of the API method to produce a logical formula that can be converted into a specification, as shown in Figure 1.

To empirically evaluate our approach, we apply it to infer postconditions for seven popular Java libraries: JUnit4 (JU4), JSON-Java (JJA), Commons-CSV (CSV), Commons-CLI (CLI), Commons-Codec (COD), Commons-Email (EMA), and Commons-IO (CIO) totaling over 2300 methods. Our results show that our technique has a very high precision and recall. We were able to infer specifications for 75.7% methods, and all of the inferred specifications were verified to be correct using OpenJML’s Extended Static Checker (with Z3 [13] version 4.3.0). Our results show that our inferred specifications are both rich and concise. To evaluate richness, we study the presence of `assignable` and `purity` clauses (in addition to pre- and postconditions) in inferred specifications. The `purity` clause documents whether an API method will change memory locations, and the `assignable` clause documents which memory locations can be changed. We find that all the API methods with inferred specifications have either a `purity` clause or an `assignable` clause. To evaluate conciseness, we study the length of final specifications and find that 84.6% of inferred

```

//@ 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) A Java function `cmp`

```

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) Postcondition of `cmp` using the standard SP

```

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) Postcondition using our technique

Fig. 1: The postcondition of `cmp` function obtained using the standard definition of SP is verbose, exposes implementation details, and is tied to the structure and flow of `cmp`. Our technique infers concise postconditions that eliminates those limitations.

specifications are less than 1/4 page in length.

Our key contributions include:

- 1) A novel technique for mitigating the effect of state space explosion in the context of inferring specifications using strongest postcondition predicate transformers.
- 2) The implementation of a tool, **Strongarm**, that embodies our postcondition inference technique as an extension of the OpenJML [7], [29] program verification tool. Our tool will be part of a future OpenJML release that will make our techniques available to formal methods community.
- 3) A series of techniques and a novel, graph-based algorithm for producing concise specifications.
- 4) An experimental evaluation of Strongarm on seven popular Java libraries that demonstrates its effectiveness.
- 5) A benchmark of seven specified libraries.

Next, we will motivate our approach. Section III describes our approach, Section IV presents its evaluation, Section V compares it with related work, and Section VI concludes.

II. MOTIVATION

The standard technique for statically computing the postcondition for a piece of code was formalized by Dijkstra in the form of the *strongest postcondition predicate transformer* [15] (SP). SP is a functions that takes a predicate describing a starting state (i.e., a pre-state) and a statement, and produces predicate that describes the set of states that may result from executing that statement on any pre-state that satisfies the given predicate. Many researchers have used the opposite predicate transformer, the weakest precondition transformer, also introduced by Dijkstra, for building verification tools, including ESC/Java, Boogie [31], OpenJML, and many others.

However, SP itself has not been previously leveraged for successful tools that infer postconditions.

To understand reasons why SP has not been successfully used before, consider inferring the postcondition of the Java function in Figure 1a. Figure 1b shows the results of applying SP on that code, starting from the predicate `true`, and translating the resulting proposition to the JML notation. The JML keyword `normal_behavior` means that this specification is about the normal termination. As is usual in SP, conditions are accumulated as the tool traverses the control flow of the function. Since this function has three exits, this results in three alternative postconditions. The syntax `{| A also B |}` means that the implementation must obey both specifications *A* and *B*, or, equivalently, that a caller may conclude the postcondition of either *A* or *B*. Starting with the precondition of `true`, the first assignment leads to accumulation of the condition `c==a`. The if-condition leads to accumulation of the condition `c<b` in the true branch, and its negation `!(c<b)` in the false branch. The first (lexical-order) return statement is an exit point, and thus leads the postcondition `\result == -1`. Thus, at the first return statement we get four accumulated conditions. `c<b` (from the if condition), `true` (from the precondition), `\result == -1` (from the return), and `c==a` (from the assignment). The second if statement (within the body of else) leads to accumulation of the condition `c>b` in the true branch and its negation in the false branch. The second return statement thus leads to the postcondition `\result == 1`, along with `!(c<b)` (from the first if), `(c>b)` (from the second if), and `true` (from the precondition). The third return statement is similar.

As the resulting postconditions show, a straightforward use of the predicates produced by SP for specification inference

leads to verbose results that: (1) expose the internal details of the function such as internal variables (e.g., c) and assignments to such variables that are not part of the interface, (2) postconditions that are directly tied to the structure of the code and its control flow, and (3) significant redundancies that could increase the cost of reading and using such specifications.

Figure 1c shows the postconditions produced by our approach and previews its advantages. These postconditions are in terms of function arguments, reorganized to reduce redundancies, and eliminate internal details and tautologies.

III. INFERRING CONCISE POSTCONDITIONS

Figure 2 shows an overview of our approach for inferring postconditions that consists of the following steps.

- 1) The input is the code of the API method for which postconditions are needed. If preconditions are available, they can also be provided. Otherwise, our approach starts with the default precondition **true**.
- 2) Next, the code is symbolically executed to produce traces. The symbolic execution uses the strongest post-condition predicate transformer semantics (SP). These traces are converted to a raw specification. (See Sec. III-A below.)
- 3) Then, we flatten the raw specification into groups of clauses (cases). (See Sec. III-B)
- 4) Next, we compute the overlapping states found between groups of clauses. (See Sec. III-C.)
- 5) Finally, we recombine states and convert the results to the final specification. (See Sec. III-D.)

A. Producing Raw Specifications

We use forwards symbolic execution [24] to symbolically execute the annotated AST of each API method. The rules used by our symbolic execution engine are shown in Fig. 3.

A challenge in this step was to avoid existential quantifiers in SP’s output. Existential quantifiers are problematic because they require the use of a constraint solver. However, the assignment rule in Equation 2 in Fig. 3 uses an existential quantifier. We avoid this problem by using a type of program representation called the “Optimal Passive Form” [5], [25], [31]. In Optimal Passive Form, which is a variant of single static assignment (SSA) form, in which every assignment to a variable results in a new variable. In addition to simplifying the control flow graph of a program, this form eliminates the need for such existential quantifiers since one does not need to search for previous assignments of variables when making new assignments. Next, we convert the propositions produced by SP into a raw specification. We define an abstract form of specifications independent of any concrete specification language, such as JML. The specifications themselves are to be thought of as method specifications. The only interesting properties we assume about specifications are that one can extract cases (such as those produced by *SP*) from a specification and that each such case has a set of preconditions (conceptually a conjunction) and a set of other clauses (also conjoined). Thus each specification is modeled as a non-empty

Algorithm 1: FAR

Input: S , a specification in SNF
 \sim , a sound equivalence relation
Output: A specification
begin
2 $V \leftarrow \text{cases}(S)$
3 $G \leftarrow \text{EmptyGraph}()$
 // merge the overlapping state spaces of S
5 **repeat**
6 $(G', W) \leftarrow \text{ToGraph}(V)$ // (Alg. 2)
7 $G'' \leftarrow \text{StronglyConnected}(G')$
8 $(G, V) \leftarrow \text{MergeSCC}(V, W, G'', G, \sim)$ // (Alg. 3)
 until $|G''| = 0$
 // connect each unmerged vertex to the root
11 $G.E = G.E \cup \{(G.\text{root}, v) \mid v \in V\}$
12 $G.V = G.V \cup \{V\}$
 // convert the residual graph back to a specification
14 **return** $\text{ToSpec}(G, G.\text{root})$ // (Alg. 4)
end

list of cases, separated by a special operator \vee . Within a case, clauses such as preconditions and postconditions are modeled as “atomic formula,” assertions whose structure is not further examined. This step produces specifications in a form that we call *specification normal form* (SNF). While specifications allows preconditions to distribute over cases, in SNF no such distribution of preconditions is allowed; thus in SNF each case is a non-empty list of cases, each separated by the operator \vee .

Definition III.1 (Specification Normal Form (SNF)). A specification is in *specification normal form* (SNF) if it follows the grammar in Figure 4.

To illustrate this phase of our technique consider the function in Figure 1a. The computation of $sp(\text{cmp}, \phi)$ produces the following formula:

$$\begin{aligned} &(a < b \wedge \phi \wedge \mathbf{result} = -1) \\ &\vee ((a > b \wedge \neg(a < b) \wedge \phi \wedge \mathbf{result} = 1) \\ &\vee (\neg(a < b) \wedge \neg(a > b) \wedge \phi \wedge \mathbf{result} = 0)) \end{aligned} \quad (6)$$

Formula (6) corresponds to the following specification.

$$\begin{aligned} &(\mathbf{pre} \ a < b \wedge \phi; \ \mathbf{rest} \ \mathbf{result} = -1) \vee \\ &(\mathbf{pre} \ a > b \wedge \neg(a < b) \wedge \phi; \ \mathbf{rest} \ \mathbf{result} = 1) \vee \\ &(\mathbf{pre} \ \neg(a < b) \wedge \neg(a > b) \wedge \phi; \ \mathbf{rest} \ \mathbf{result} = 0) \end{aligned} \quad (7)$$

B. Flattening Specification Cases and Computing Weights

Algorithm 1 describes the next three components of our technique. The input of this component is the raw specification produced by the predicate transformer semantics in SNF.

In the first step of the algorithm, these clauses are placed in a data structure suitable for a graph-based analysis; our implementation uses the JPaul program analysis library [46].

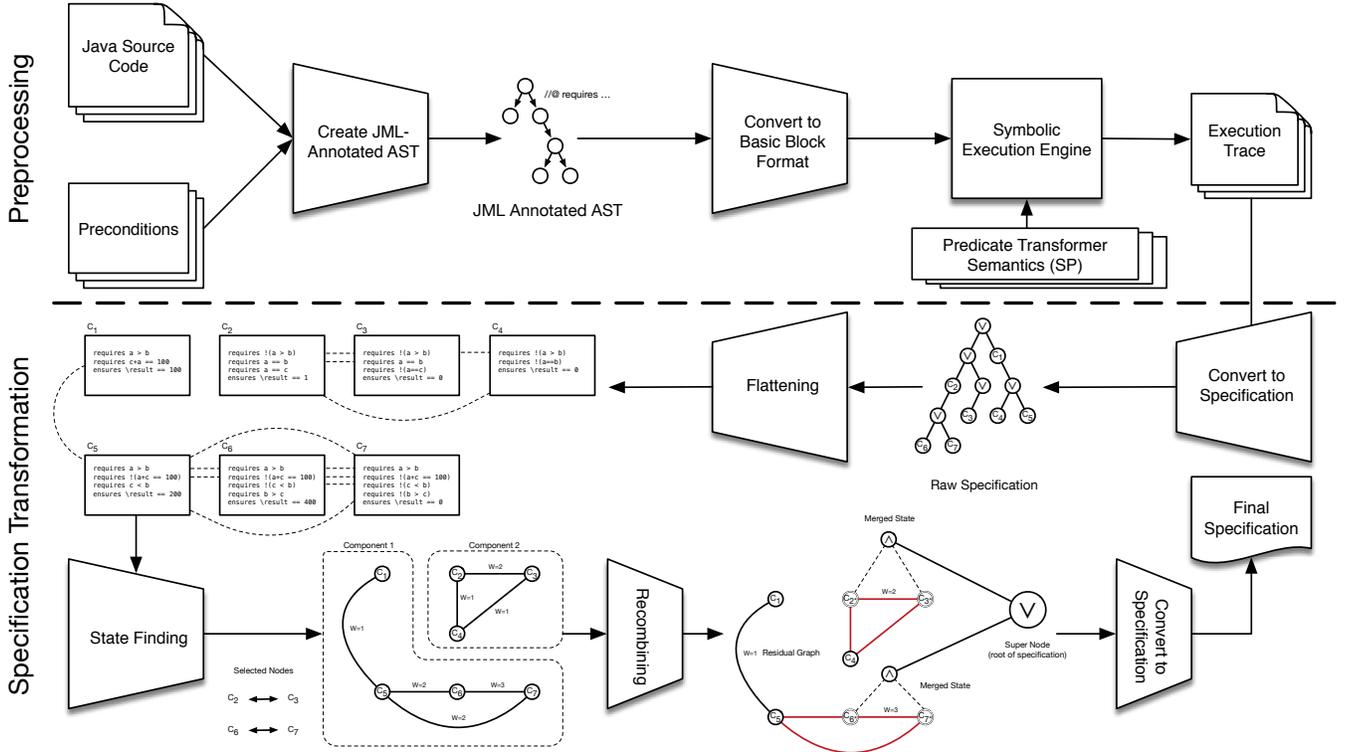


Fig. 2: Approach overview: The top-half produces raw specifications, and the lower half makes them concise.

- $$\begin{aligned}
 \text{sp SKIP } P &= P & (1) \\
 \text{sp } (V := E) P &= \exists v. (V = E[v/V]) \wedge P[v/V] & (2) \\
 \text{sp } (S_1; S_2) P &= \text{sp } S_2(\text{sp } S_1 P) & (3) \\
 \text{sp } (\text{IF } B \text{ THEN } S_1 \text{ ELSE } S_2) P &= (\text{sp } S_1(P \wedge B)) \vee \\
 & \quad (\text{sp } S_2(P \wedge \neg B)) & (4) \\
 \text{sp } (\text{WHILE } B \text{ DO } S) P &= (\text{sp } (\text{WHILE } B \text{ DO } S) \\
 & \quad (\text{sp } S(P \wedge B))) \vee (P \wedge \neg B) & (5)
 \end{aligned}$$

Fig. 3: Predicate transformer rules used by our approach.

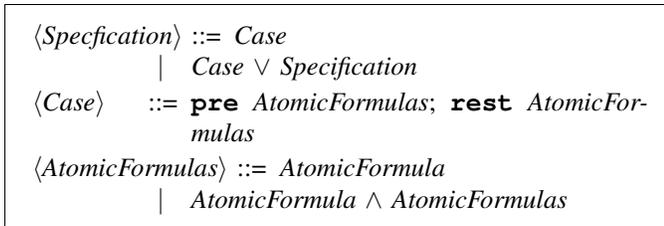


Fig. 4: The abstract syntax of specification normal form.

First we collect the cases of the specification, represented as *cases* in Algorithm 1, as a set. The semantics of case extraction is as follows.

$$\begin{aligned}
 \text{cases} : \text{Specification} &\rightarrow \mathcal{P}(\text{Case}) \\
 \text{cases}(S) &\stackrel{\text{def}}{=} \mathbf{let} \ c_1 \vee \dots \vee c_n = S \ \mathbf{in} \ \{c_1, \dots, c_n\}
 \end{aligned}$$

Algorithm 2: ToGraph

Input: V , a set of specification cases
 \sim , a sound equivalence relation

Output: A graph G and a table of weights W

begin

- 2 $G \leftarrow \text{EmptyGraph}()$
- 3 **for** $(l, r) \in V \times V$ such that $l \neq r$ **do**
- // add l and r to the list of vertices
- $G.V \leftarrow G.V \cup \{l\} \cup \{r\}$
- // compute the weight modulo \sim
- $W[l][r] \leftarrow |\{(x, y) \in \text{pre}(l) \times \text{pre}(r) \mid x \sim y\}|$
- // $l \rightarrow r$ if they share any clauses in common
- if** $W[l][r] > 0$ **then**
- $G.E \leftarrow G.E \cup \{(l, r)\}$
- end**
- 13 **end**
- 13 **return** (G, W)
- end**

With the cases of the specification collected, the next step is to flatten the specification. This is done by taking each case of the raw specification and detaching it from the tree. This forms a disjoint forest of all the specifications cases.

Once flattening is complete, Strongarm computes weights for each specification case using an equivalence relation on pairs of distinct cases as shown in Algorithm 2. On Line 7 of Algorithm 2 this equivalence relation is denoted by the symbol

Algorithm 3: MergeSCC

Input: V , a set of specification cases
 W , a table of weights between edge pairs of Scc
 SCC , the strongly connected component, a set of graphs of the form $\langle V, E \rangle$
 R , a residual graph of the form $\langle V, E \rangle$
 \sim , a sound equivalence relation

Output: A pair consisting of the R and remaining vertices V

```
begin
2  for  $G \in SCC$  such that  $|G.V| > 1$  do
3     $maxW \leftarrow$  the max weight of  $G$  via  $W$ 
4     $(L, R) \leftarrow$  a pair  $\in G.E$  with weight  $maxW$ 
      // compute intersection modulo  $\sim$ 
6     $c \leftarrow \{l \mid (l, r) \in pre(L) \times pre(R), l \sim r\}$ 
7     $rcmn \leftarrow \{r \mid (l, r) \in pre(L) \times pre(R), l \sim r\}$ 
      // remove the common vertexes
9     $L' \leftarrow$  pre ( $pre(L) \setminus c$ ); rest  $rest(L)$ 
10    $R' \leftarrow$  pre ( $pre(R) \setminus rcmn$ ); rest  $rest(R)$ 
      // make a conjunction node between  $L$  and  $R$ 
12    $R.E \leftarrow R.E \cup (\{c\} \times \{L', R'\}) \cup \{(R.root, c)\}$ 
13    $R.V \leftarrow R.V \cup \{L'\} \cup \{R'\}$ 
      // remove merged nodes
15    $V \leftarrow V \setminus (\{L\} \cup \{R\})$ 
    end
17  return  $(R, V)$ 
end
```

Algorithm 4: ToSpec

Input: R , a residual graph of the form $\langle V, E \rangle$
Root, the root of the graph

Output: A specification

```
begin
2   $B \leftarrow \emptyset$ 
3  for each  $v$  adjacent to Root do
4    if  $v$  is a leaf then
5       $B \leftarrow B \cup v$ 
6    else
7       $B \leftarrow B \cup (v \wedge ToSpec(R, v))$ 
8    end
9  end
10 return  $\bigvee B$ 
end
```

' \sim '. For simplicity, our prototype uses lexical identity for the relation \sim . In our experimental analysis (Section IV) we find that this choice works out well.¹ The weights assigned are the number of preconditions that are \sim between the two cases; for example, if cases l and r have 3 preconditions that are related

¹The \sim relation used need not be lexical identity; we speculate that more interesting results could be achieved by considering other relations, such as logical equivalence, but that is beyond the scope of this paper.

by \sim , then the weight assigned to the edge from l to r is 3. In Fig. 2, an edge of weight n is shown as n dashed edges.

C. State Finding

Returning to Algorithm 1, the next step is to compute the connected components of the resulting graph. In Figure 2, this process produces Component 1 and Component 2.

D. Recombining and Conversion to Contract

The next step of Algorithm 1 selects, from each connected component, the pair of vertices with the largest weight edges connecting them (i.e., the most similarity), by calling MergeSCC (Algorithm 3). For example, in Figure 2 this produces the choice (C_6, C_7) for Component 1 and (C_2, C_3) for Component 2.

Algorithm 3 works on each connected component as follows. First it selects from the connected component, the maximum weight (Line 3), and a pair of vertices connected by an edge with that weight (Line 4). The selected pair of vertices are combined into a single node by extracting the common preconditions between the two vertices (Line 6) and returning a modified residual graph with an edge connecting the common preconditions (c) to both vertices without those common specifications (L' and R' , Lines 9 to 13). For example, in Fig. 2 C_2 and C_3 are combined into a single node, as are C_6 and C_7 . Next, we examine the connected component to which the pair belongs. If any vertex is adjacent to any of the elements of the pair, we remove that edge from the graph (Line 15).

Upon returning to Algorithm 1, Lines 11 to 12 make a root node for the specification. When converted to a specification, this root node will be a disjunction (\vee) of all the newly-created conjunction nodes (the merged states in Fig. 2). In the example in Figure 2, this leaves us with vertices C_1 and C_5 . This subgraph is then fed back into the FAR algorithm until there are no remaining vertices. Completely disjoint vertices (those with no preconditions in common with any of the other vertices) form a disjoint forest in the final iteration of the algorithm. Once this outcome is reached, each vertex in the forest is connected unmodified to the root. This resulting tree is then converted back into a specification.

E. Soundness of Our Approach

We have proved the soundness of our approach in a detailed report. Briefly, soundness for our approach is defined with respect to satisfaction by programs (method bodies). In essence a program C satisfies a specification S if for every pre-state s that satisfies some case c 's precondition, the semantics of C is such that the post-state s' that C produces (when run on s) satisfies c 's postcondition. The FAR algorithm uses an equivalence relation \sim to compute the intersection between two sets of preconditions. However, not just any equivalence relation on atomic formulas will do; a *sound* relation must preserve the meaning of atomic formulas. Lexical equivalence is sound. Our report shows that if \sim is sound, then the semantics of specifications is preserved by Algorithm 1.

F. Implementation Details

After an execution trace of an API method is obtained, we need to transform the trace and the underlying predicate AST into a raw specification. Our implementation includes several transformation tasks such as translation from internal variable representations to externalized (source code level) representations, removal of tautologies, determination of purity, and inference of frame axioms. For space considerations we omit a detailed description of these techniques.

IV. TECHNICAL EVALUATION

We have implemented our techniques in a tool, **Strongarm**, that is an extension of the OpenJML [7], [29] program verification tool. Our tool will be part of a future OpenJML release. In this section we evaluate the performance of our technique on the task of inferring specifications. In the next subsection we explain our experimental setup and provide specific details about the source code we conducted our experiments on. Following this, our evaluation looks at the performance of our technique from the following five perspectives:

- 1) **Effectiveness of Inference** How many of the candidate methods we were able to infer?
- 2) **Efficiency of Reduction** By how much were the specifications reduced?
- 3) **Complexity of Inferred Specifications** Overly complex and overly simplistic specifications are not practical. What are the characteristics of the inferred specifications?
- 4) **Performance of Inference Procedure** How well (with respect to time) did our technique perform at the task of inferring specifications?
- 5) **Performance of Inferred Specifications** How effective is our technique at reducing specification nesting, decreasing specification length, decreasing prover execution time, and decreasing proof length?

Finally, in Section IV-H we conclude with a human evaluation of the inferred specifications produced by our technique.

A. Evaluation Methodology

We designed a series of experiments designed to examine the effectiveness of our technique at inferring practical specifications. We selected a cross section of popular Java libraries: JUnit4 (JU4), JSON-Java (JJA), Commons-CSV (CSV), Commons-CLI (CLI), Commons-Codec (COD), Commons-Email (EMA), and Commons-IO (CIO). The static characteristics of these libraries are summarized in Table I.

A method for inferring preconditions in large corpora was recently investigated in the work of Nguyen et al. [40]. However, rather than specifying the preconditions for the methods in our study, we assume a vacuously true precondition, namely `true`. Prior to construction of the final specification, the default precondition is removed and not tabulated in the final specification analysis. We do this so as to not simultaneously test the results of our work in parallel with the technique of Nguyen. As noted in Section IV-B, we do not specifically attempt to infer invariants for loops and instead rely on user-written loop

TABLE I: 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	-

invariants. For the study presented in this paper loops were not annotated with such invariants. Additionally, our technique’s inference technique assumes any field referenced in the body of a method should be visible in the inferred specification. However, in JML this is considered an error by default. For this reason, all private fields referenced in specifications have been given the special annotation `spec_public` which allows private fields to appear in specifications. This promotion of private fields to `spec_public` is performed automatically by our technique during inference and reflected in the inferred specifications; future work includes using JML features such as model fields to avoid declaring all fields to be `spec_public`. Additionally, during our evaluation we discovered there were several methods we were unable to validate due to bugs in OpenJML; these methods were explicitly skipped in our evaluation.

Experiments were performed on the Stokes HPC cluster² at the University of Central Florida. Each job node was configured with 6 Intel Xeon 64-bit processors, 42GB of RAM, and used Oracle JDK 1.8.0.131. Our technique produces extensive telemetry data as well as the inferred specifications. The results presented below are based on mining this telemetry data.

1) *Verification of Inferred Specifications*: Once a specification is inferred, we must have a standard way of knowing if the specification itself is correct. In our experiments we validated the inferred specifications in three different ways. First, in the creation of our technique, we built a comprehensive test suite consisting of approximately 100 hand-written and hand-verified test cases. Our tool passes this test suite. Second, once inferred specifications are produced, we use OpenJML’s to type check the produced specifications. All of the specifications produced by our technique type check. Lastly, we use OpenJML’s Extended Static Checker (with Z3 [13] version 4.3.0). 70% of the inferred specifications were verified by OpenJML. In practice our technique would check these specifications before submitting them to a user and therefore would not produce an invalid specification. The cases we were unable to verify were caused by tool error or implementation issues in our technique. Verification in this fashion also checks for unsatisfiable clauses, which should not be present in practical specifications. In doing this, as noted

²<https://arcc.ist.ucf.edu/index.php/resources/stokes/about-stokes>

in Section IV-B, we do not consider the exceptional behavior of the method in question.

2) *Threats to Validity*: As mentioned in Section IV-A1, we use the OpenJML tool to check the results of our inferred specifications. A positive result from OpenJML certifies that the program code satisfies the given specification. This is an especially strong guarantee; since checking is done statically, this certifies that *for all runs* (and potential input values) the specifications are valid. However, this depends on the soundness of the tool. This is a limitation in the following ways. First, OpenJML itself might have bugs and therefore might certify programs that are not correct. Second, the theory behind OpenJML itself might not be sound, i.e., it might admit incorrect programs (programs that do not satisfy their specification) under certain circumstances; however, it is believed that this second problem is limited to JML features with semantics that are not quite settled yet (such as details concerning invariants). Our study is not impacted by known problems in the semantics of JML. Lastly, in choosing JML as our target specification language, our approach to specification inference may not generalize to other specification languages as well as it has with JML. However, there are many Hoare-style specification languages that use pre- and postcondition specifications, such as Eiffel, and retargeting our technique to these specification languages should be a straightforward exercise for future work.

B. Limitations

In designing our technique, we intentionally made some trade-offs to simplify its implementation. The two major limitations of our technique are seen in our handling of exceptions and in our handling of loop constructs. First, although JML allows for descriptions of exceptional method behavior, in our technique we do not attempt to infer this behavior (although information about the exceptional behavior of codes is present in our AST; it is simply elided). Instead this task is relegated to future work [16]. Second, as a simplifying assumption, in our implementation we assume that in the presence of loops that our technique will always have access to expert-written loop invariants. From these loop invariants we apply the standard Hoare loop rules to facilitate inferring postconditions.

C. Effectiveness of Inference

For each of the libraries we categorized the status of an inference attempt in one of four different categories. We give an explanation of the categories in this section as well as provide some analysis of our findings; see Figure 5.

The status *Inferred* indicates that our technique successfully inferred a specification for a method. We define success as containing at least one postcondition in the form of an `ensures` or `assignable` clause (in the case a frame axiom is necessary). Other clauses may be (and often are) present. For all experiments our technique succeeded in producing a specification more than 74.0% of the time (overall), and in its worst performance produced a specification 48.1% (in Commons-CSV). Our technique’s best success in producing

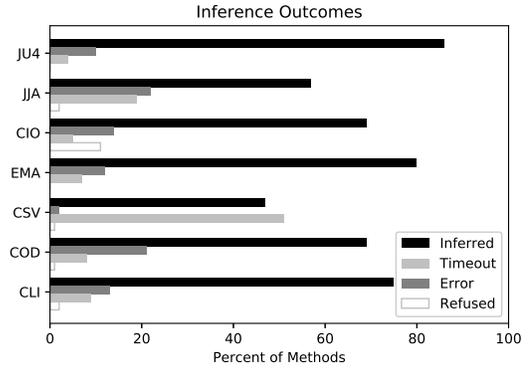


Fig. 5: A summary of inference outcomes for our technique on our 7 test libraries.

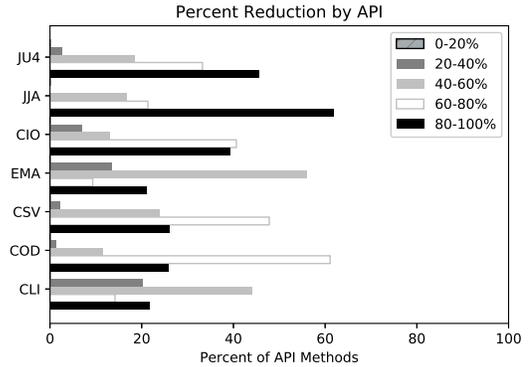


Fig. 6: A summary of specification length reduction for our technique on our 7 test libraries.

specifications was on JUnit4 (92.5%); this was largely due to JUnit4 having significantly smaller control flow graphs compared to other libraries. In our analysis we found that the CFGs for JUnit4 were less than 400 nodes, except for one case of 594 nodes.

A status of *Timeout* indicates that the inference process was aborted before inference could complete. For this paper we used a timeout of 300 seconds (5 minutes). We determined this timeout through repeated experimentation with different timeouts ranging up to 20 minutes. In our initial tests we determined that inference attempts that did not complete within 5 minutes were not able to complete in 20 minutes either. While higher settings for timeout timeouts might reduce the number of specifications that fail to be inferred, the combined timeout was only about 9%. When a timeout occurs the intermediate results are discarded and not included in our remaining analyses. In the results reported in Figure 5 we can see that the two worst performing subjects in terms of timeout are Commons-CSV (51.3%) and JSON-Java (23.0%). Unlike the other test candidates, both Commons-CSV and JSON-Java are both parsers that contain deeply nested code. In all other libraries timeout performance was excellent and perhaps these results suggest that inferring specifications for parsers, since they are inherently very recursive, is more difficult than

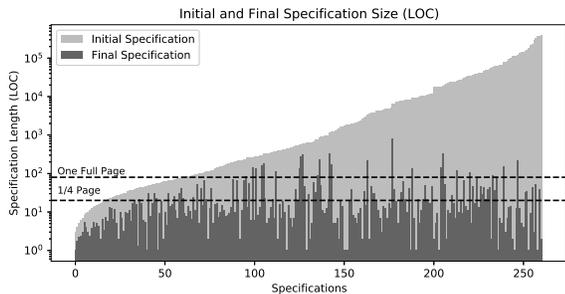


Fig. 7: The initial and final specification length for all inferred specifications (all test libraries combined, binned to 262 equal width bins). Light lines indicate length before applying our technique; dark lines show our technique’s effect on length. In this paper we consider 1/4 page to be equal to 20 lines.

inferring specifications for general purpose code.

A status of *Refused* means that our technique did not attempt to infer a specification, because the control flow graph (CFG) size was larger than a preset limit (a CFG size of 500 nodes). Similar to the timeout parameter, larger CFGs typically take much longer to infer. In our evaluation a size of 500 proved to be a reasonable choice, since the number of refused methods represented only 3.9% overall. This number is partially inflated by the unusually high number of refused methods in Commons-IO. This is explained by higher CFG complexity relative to the other libraries in the test suite. This additional complexity comes from the way the verification conditions for exceptional code are generated. Although we are not inferring the specifications for the exceptional specifications cases, the exceptional information exists in the CFG that our technique analyzes to infer the normal specification cases. Since Commons-IO deals with input/output related functions it has an unusually high number of exceptions. This greatly inflates the size of the CFG, which made fewer of its methods usable for our study. This effect could be mitigated if the exceptional nodes were removed from the CFG prior to inference but such a change would make it then impossible to later infer the exceptional behavior of methods.

A status of *Error* means that our technique encountered an internal error during inference. We manually investigated these errors and found them to be generated from current limitations in OpenJML itself. For example, certain features, such as enumerated types, are not currently supported in OpenJML (although they are valid in JML itself). Our implementation is based on OpenJML 0.8.12; we expect to be able to reduce the amount of internal error our tool encounters as errors are corrected in OpenJML.

D. Efficiency of Reduction

As discussed in Sections I and III, one of the problems impacting specification inference by symbolic execution (and therefore predicate transformers) is the length of the resulting inferred specification. In this section we will evaluate how

effective our technique was at reducing the length of specifications inferred by symbolic execution.

In Figure 7, we show the length of the initial and final inferred specifications in terms of lines of specification. Additionally, since short specification length can be a desirable quality from a software engineering and readability perspective, we also provide two reference lines: one at one full page (80 lines) and one at a quarter page (20 lines). In our analysis, 95.0% of specifications fell below one full page length and 84.6% fell below one quarter page length. This suggests that our techniques for reducing specification size were effective in reducing the size of most of the inferred specifications.

The results in Figure 7 give an overview of all inferred specifications. However, our technique’s effectiveness at reducing the final specification length of individual code bases varied significantly. In Figure 6, we can see a breakdown of our technique’s effectiveness at reducing the final size of inferred specifications. Of the most interest are 80-100% category (the *most* reduction), and the 20-40% reduction category (the *least* reduction). Although we created a category for it, none of the code bases contained specifications that were reduced in the 0-20% category. In our experiment, we found that the 80-100% category contained 41.6% of all methods. In Figure 6 we see that the worst performer (least reduction) was Commons-Email. We did further analysis of this phenomenon where we looked at the length of the methods being inferred, the control flow depth, and the resulting reduction classification. We did not manage to find a relationship between these variables, which may suggest that these contracts failed to reduce more significantly because they described essential (non-reducible) logical states rather than non-practical artifacts as described in Section III.

E. Complexity of Inferred Specifications

Criticisms of prior work on specification inference have included that the inferred specifications were either too short and not descriptive enough or template based and not expressive enough to capture the meaning of programs not falling within the parameters of the templates. We examine complexity in three different ways. First, in Figure 7 we give metrics on the variation in the length of specifications produced by our technique. To quantify the distribution of types of clauses our technique was able to infer, in Table II, we report on the variation of clauses present in the specifications inferred by our technique for each of our target codebases. In Table II we can see that inferred specifications were comprised of mostly `requires` clauses, followed by roughly 4 times less `ensures` clauses as well as smaller numbers of `pure` and `assignable` clauses. Our technique is able to handle quantifiers such as `forall`, however we did not examine its distribution in the inferred specifications. In Figure 8, we detail the specification case nesting present in the inferred specifications. This is discussed further in Section IV-G.

TABLE II: Summary of JML clauses in inferred specifications.

API	Methods	requires	ensures	assignable	pure
JU4	1,230	845	631	153	588
JJA	200	209	183	24	40
CSV	158	70	91	34	23
CLI	194	1,301	422	118	27
COD	509	1,533	569	105	90
EMA	192	7,139	622	191	47
CIO	955	1,483	681	270	641
Total	2,331	12,580	3,199	895	1,456
Ratio	-	5.39	1.37	0.38	0.62

F. Performance of Inference

To better understand the runtime performance characteristics of our technique we collected telemetry data about its performance. In our analysis, we found that the data was tightly clustered between 10^1 and 10^2 ms. Note that inference time in our experiment was limited to 5 minutes (3×10^5 ms). Many of the observed data points fell within this region, suggesting a linear fit with the exception of some very large control flow graphs which fell in the region bordering the timeout. These data points were identified as belonging mostly to Commons-IO and due largely to the number of exceptional flows present in the resulting control flow graphs (see Section IV-C for more details). As discussed in Section IV-C, using the timeout threshold of 5 minutes with the current performance characteristics allowed us to infer more than 75% of the methods we considered for this study with an average inference time of just 2 seconds per method.

G. Performance of our technique

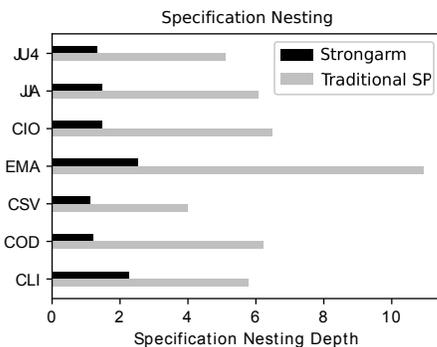


Fig. 8: The effect of our technique on the nesting of inferred specifications

To better understand the performance of our technique with respect to more standard techniques such as removing tautologies, we conducted further evaluation to study our technique. We conducted our study by taking the same 7 libraries used throughout this section and examining the effect of running all of the standard analysis types (Section III-F) with the exception of our technique in one run and all of the analysis types *with* our technique added back into the analysis pipeline.

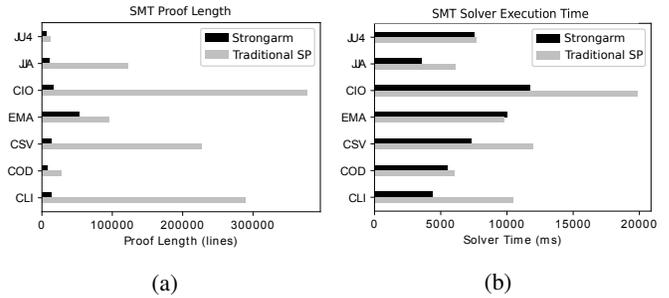


Fig. 9: The effect our technique on (a) the length of the SMT conditions generated from specifications (b) the time needed to check the inferred specifications.

To understand the characteristics of specifications before and after our technique is applied to them we observed 4 different metrics. The first metric we observed was **specification nesting**. Specification nesting is defined as the lexical depth of a specification. In Figure 8, we present the performance of our technique in reducing nesting of specifications. In this case of all libraries, our technique achieved a reduction in nesting, producing an overall average reduction of 73.8%.

In our examination of the **percent reduction** aspects of our technique, we found our technique’s effect on specifications is not necessarily to produce large reductions in the length (as measured in lines) of specifications; our technique’s contribution to the length (in lines) is small compared to the contribution of the other steps (our technique contributes an additional $\approx 10\%$ reduction overall).

Ultimately, for a specification to be most useful, it should be used to verify the implementation of the code it specifies. In Figure 9a, we examine the impact of our technique on the **proof length**, i.e., the length of the SMT conditions generated that are needed to verify the correctness of a specification in relation to its implementation. In Figure 9a we can see that our technique has a large impact on the size of the generated SMT proofs for the code samples we studied. Overall, *our technique reduces the size of the generated SMT conditions by 76.7%*.

In the previous paragraph we saw that our technique is effective for reducing the size of the proofs required to verify implementations of inferred specifications. To determine the impact of the reduced length on the time to run the SMT solver on these proofs, we studied the **solver times** with and without our technique. In Figure 9b we show the time taken to prove the specifications inferred with and without our technique. As can be seen in all cases, our technique reduces the time taken to verify the SMT conditions. Across all libraries, our technique reduces the prover execution time by 26.7%.

H. A Study of the Inferred Specifications

To determine if the specifications produced by our technique were useful to human readers, we conducted a small study to examine their usefulness. In general, we believe that practical specifications are more useful for human readers than those

that are not, since they tend to be shorter and more to the point. However, we defined additional criteria against which we designed our study.

Namely, we selected the following criteria and say that a specification is *practical* if they:

- do not contain redundant formulas,
- do not contain unsatisfiable formulas,
- do not contain tautological formulas,
- specify frame axioms [6]),
- specify when a method is pure (has no side-effects), and
- only use names that are visible to a method's clients.

Methodology. Using the specifications inferred by our technique, we conducted a survey of people familiar with the Java Modeling Language (JML) [29]. The survey used 12 pairs of method specifications inferred by our technique; to highlight the aspects of our definition of practical specifications, one element of each pair had one aspect, e.g., removal or unsatisfiable formulas, disabled. The survey was completed by 25 people, 18 of which said that they “definitely” had experience reading JML and 7 of which had some experience with JML. All correctly answered a simple question about JML that tested their understanding of a method specification with two specification cases. (A method *specification case* in JML is a pre- and postcondition specification, with optional frame axioms, that must be satisfied whenever the case's precondition is true when the method is called.) The questions in the survey were all simplifications of pairs of output given by our tool, one of which was not processed to remove a single impractical feature (according to the above definition).

Results. An unsatisfiable precondition, such as `!true`, causes the specification case it appears in to be useless. 87.5% of the survey respondents preferred a specification without unsatisfiable preconditions, including preconditions that required non-null fields to be null. Another example of unsatisfiable specifications comes when a specification case has two mutually-contradictory clauses; in this case all respondents preferred a specification without such unsatisfiable combinations of clauses (75% of them strongly so).

According to the survey, 95% of the respondents preferred a specification without the tautological postcondition `true == true`. Also, 62.5% of the respondents preferred a specification without tautologies such as `requires true` and more subtle tautologies involving non-null declarations. Another 87.5% preferred a specification without subtle redundancies such as `"No resource defined" != null`. In another question, 80% of the respondents preferred a specification without redundant clauses requiring non-null fields to be not null. In another question, 93.3% of the respondents preferred a specification without duplicated specification cases.

Regarding frame axioms, the survey contrasted a specification without the frame `assignable this.name` with a nearly identical specification with the assignable clause. Although the specification without the frame axiom is shorter, 86.7% of the respondents preferred the specification with the frame axiom (53% strongly so).

In JML a pure method is one without any side-effects, and is specified by the `pure` keyword. The `pure` keyword also functions as a strong frame axiom. 66.7% of the respondents preferred a specification to one just like it but without the keyword `pure`.

In sum, the majority of the survey respondents preferred specifications that satisfy our notion of a “practical” specification (the kind our technique produces), even if the differences were rather subtle.

V. RELATED WORK

In addition to the work discussed in Section I, there are several other systems and papers relevant to this paper. The closest related work to Strongarm is the Houdini system by Flanagan and Leino [19]. Houdini targets ESC/Java with the goal of statically inferring specifications for Java. Similarly, in our work we target OpenJML, a successor to ESC/Java. However, rather than symbolically computing specifications as Strongarm does, Houdini instead applies templates, i.e., commonly-found specification patterns; to search for a specification, Houdini tries all of these patterns and checks the candidate specifications using ESC/Java. In contrast to Strongarm, Houdini also attempts to infer object invariants using the same mechanism, which Strongarm does not yet attempt.

Flanagan and Saxe's work on compacting verification conditions [21], unlike Strongarm, only deals with two types of duplication: multiple assignment and propagation of accumulated formulas. Strongarm can reduce these problems globally in the specification whereas the approach of Flanagan and Saxe is restricted to the branch level. Furthermore, Strongarm uses a pluggable equivalence relation (\sim , see Section III-B), which generalizes their work.

In contrast with Strongarm (and Houdini), Daikon [18] is a runtime approach. To produce specifications, Daikon requires a test suite that calls the code that is being targeted for inference. Even when such a test suite is available, dynamic inference can fail, since code coverage and branch coverage are typically not sufficient to produce specifications [27], [28], as many test suites focus on corner cases and are thus not suitable for specification inference [42].

Similarly, in their work on discovering relational specifications, Smith et al. discover specifications by looking at program outputs [45]. Our approach is fully static does not require running the code.

In their work on API usage error detection, Murali et al. investigate detecting API usage errors using Bayesian inference [38]. Our approach does not use machine learning to generate its specifications and therefore does not require training examples to infer specifications.

Nguyen et al. [40] infer method preconditions, by examining call sites. Our approach is different in that Strongarm computes postconditions based on each method's code, not preconditions based on calling code.

VI. CONCLUSION AND FUTURE WORK

Our approach to inferring practical postconditions takes advantage of structure the logical formulas that result from

the SP predicate transformer. We evaluated our technique on 7 popular Java libraries and found that 95.0% of the inferred specifications were less than 1 page long and 84.6% were less 1/4 of a page. Such concise specifications have the potential aid many areas of software engineering. Our future work will evaluate these implications.

REFERENCES

- [1] Code Contracts at Rise4Fun. <http://rise4fun.com/CodeContracts>.
- [2] Java Path Finder (JPF). <http://babelfish.arc.nasa.gov/trac/jpf>.
- [3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16. ACM, 2002.
- [4] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, SPIN '01, pages 103–122. Springer-Verlag, 2001.
- [5] Mike Barnett and K. Rustan M. Leino. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '05, pages 82–87. New York, NY, USA, 2005. ACM.
- [6] Alex Borgida, John Mylopoulos, and Raymond Reiter. “…and nothing else changes”: The frame problem in procedure specifications. In *Proceedings of the 15th International Conference on Software Engineering*, ICSE '93, pages 303–314. Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.
- [7] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, 2005.
- [8] Raymond P.L. Buse and Westley R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 273–282. New York, NY, USA, 2008. ACM.
- [9] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE '00, pages 439–448. ACM, 2000.
- [10] Patrick Cousot, Radhia Cousot, Manuel Fahndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *in Proceedings of the 14th Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'13)*. Springer Verlag, January 2013.
- [11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'11, pages 150–168. Springer-Verlag, 2011.
- [12] Valentin Dallmeier, Christian Lindig, and Andreas Zeller. Lightweight defect localization for java. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550. Springer-Verlag, 2005.
- [13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
- [14] Xianghua Deng, Robby, and John Hatcliff. Kiasan: A verification and test-case generation framework for java based on symbolic execution. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, ISOLA '06, pages 137–. IEEE Computer Society, 2006.
- [15] Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.
- [16] Isil Dillig, Thomas Dillig, Boyang Li, and Ken McMillan. Inductive Invariant Generation via Abductive Inference. *SIGPLAN Not.*, 48(10):443–456, 2013.
- [17] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin 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.
- [18] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon System for Dynamic Detection of Likely Invariants. *Sci. Comput. Program.*, 69(1-3):35–45, December 2007.
- [19] Cormac Flanagan and K. Rustan M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, FME '01, pages 500–517. London, UK, UK, 2001. Springer-Verlag.
- [20] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended Static Checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 234–245. New York, NY, USA, 2002. ACM.
- [21] Cormac Flanagan and James B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 193–205. New York, NY, USA, 2001. ACM.
- [22] Mark Gabel and Zhendong Su. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '08/FSE-16, pages 339–349. ACM, 2008.
- [23] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 213–223. ACM, 2005.
- [24] Mike Gordon and Hélène Collavizza. Forward with Hoare. In A. W. Roscoe, Cliff B. Jones, and Kenneth R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 101–121. Springer London, 2010.
- [25] Radu Grigore, Julien Charles, Fintan Fairmichael, and Joseph Kiniry. Strongest Postcondition of Unstructured Programs. In *Proceedings of the 11th International Workshop on Formal Techniques for Java-like Programs*, FTJLP '09, pages 6:1–6:7. New York, NY, USA, 2009. ACM.
- [26] Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 119–130. ACM, 2010.
- [27] Neelam Gupta and Zachary V. Heidepriem. A new structural coverage criterion for dynamic detection of program invariants. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, ASE'03, pages 49–58. Piscataway, NJ, USA, 2003. IEEE Press.
- [28] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving Test Suites via Operational Abstraction. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 60–71. Washington, DC, USA, 2003. IEEE Computer Society.
- [29] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, May 2006.
- [30] Gary T. Leavens and Curtis Clifton. Lessons from the JML project. In Bertrand Meyer and Jim Woodcock, editors, *Verified Software: Theories, Tools, Experiments, Zurich, Switzerland*, volume 4171 of *Lecture Notes in Computer Science*, pages 134–143. Springer-Verlag, 2008.
- [31] K. Rustan M. Leino. This is Boogie 2. Manuscript KRML 178, 2008. Available at <http://research.microsoft.com/~leino/papers.html>.
- [32] Zhenmin Li and Yuanyuan Zhou. Pr-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 13th Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 306–315. ACM, 2005.
- [33] Hui Liu, Zhiyi Ma, Lu Zhang, and Weizhong Shao. Detecting duplications in sequence diagrams based on suffix trees. In *APSEC '06: Proceedings of the XIII Asia Pacific Software Engineering Conference*, pages 269–276. IEEE CS, 2006.
- [34] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305. New York, NY, USA, 2005. ACM.
- [35] David Lo and Shahar Maoz. Mining hierarchical scenario-based specifications. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 359–370. IEEE Computer Society, 2009.

- [36] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, 1988.
- [37] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd International Conference on Software Engineering*, ICSE'00, pages 167–176. ACM, 2000.
- [38] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 151–162, New York, NY, USA, 2017. ACM.
- [39] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Automated oracles: An empirical study on cost and effectiveness. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE 2013, pages 136–146. ACM, 2013.
- [40] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh 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.
- [41] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H. Pham, Jafar M. Al-Kofahi, and Tien N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009.
- [42] Jeremy W. Nimmer and Michael D. Ernst. Invariant Inference for Static Checking: An Empirical Evaluation. *SIGSOFT Softw. Eng. Notes*, 27(6):11–20, November 2002.
- [43] Michael Pradel and Thomas R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 371–382. IEEE Computer Society, 2009.
- [44] Murali Krishna Ramanathan, Ananth Grama, and Suresh 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.
- [45] Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. Discovering relational specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 616–626, New York, NY, USA, 2017. ACM.
- [46] Alexandru D. Sălcianu. jpaul – Java Program Analysis Utilities Library. Available from <http://jpaul.sourceforge.net>, 2005.
- [47] Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. Detecting object usage anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, 2007.
- [48] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM, 2011.
- [49] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM, 2011.
- [50] Westley Weimer and George C. Necula. Mining temporal specifications for error detection. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 461–476. Springer-Verlag, 2005.
- [51] Chadd C. Williams and Jeffrey K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.
- [52] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '05, pages 351–363. ACM, 2005.
- [53] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. Perracotta: Mining temporal api rules from imperfect traces. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 282–291. ACM, 2006.