Mining Preconditions of APIs in Large-Scale Code Corpus

Hoan Anh Nguyen
hoan@iastate.edu

Robert Dyer
rdyer@iastate.edu

Tien N. Nguyen
tien@iastate.edu

Hridesh Rajan
hridesh@iastate.edu

Iowa State University
Ames, IA 50011, USA

ABSTRACT

Modern software relies on existing application programming interfaces (APIs) from libraries. Formal specifications for the APIs enable many software engineering tasks as well as help developers correctly use them. In this work, we mine large-scale repositories of existing open-source software to derive potential preconditions for API methods. Our key idea is that APIs' preconditions would appear frequently in an ultra-large code corpus with a large number of API usages, while project-specific conditions will occur less frequently. First, we find all client methods invoking APIs. We then compute a control dependence relation from each call site and mine the potential conditions used to reach those call sites. We use these guard conditions as a starting point to automatically infer the preconditions for each API. We analyzed almost 120 million lines of code from SourceForge and Apache projects to infer preconditions for the standard Java Development Kit (JDK) library. The results show that our technique can achieve high accuracy with recall from 75–80% and precision from 82–84%. We also found 5 preconditions missing from human written specifications. They were all confirmed by a specification expert. In a user study, participants found 82% of the mined preconditions as a good starting point for writing specifications. Using our mining result, we also built a benchmark of more than 4,000 precondition-related bugs.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs

Keywords

Specification Mining; JML; Preconditions; Big Code Mining

1. INTRODUCTION

Software in our modern world is developed using frameworks and libraries, which provide application programming interfaces (APIs) via classes and their methods. To be able to correctly use these APIs, programmers must conform to their specifications. For example, in the standard Java Development Kit (JDK), a call to next() in a LinkedList needs to be preceded by a call to hasNext() to ensure the list still has elements. For each API method, there are conditions that must hold whenever it is invoked. These are called the *preconditions* of the API. For example, in the JDK String class, the condition 'beginIndex <= endIndex' must hold when the method subString(beginIndex,endIndex) is called. These conditions, as part of the API's specification, have been shown to be useful for many automated software engineering tasks including the formal verification of program correctness [5, 7, 57], generation of test cases [22], building test oracles [41], bug detection [16, 31, 54], design by contract [9, 53], etc. Popular formal specification toolsets include ESC/Java [20], Bandera [11], Java Path Finder [2], JMLC [29], Kiasan [15], Code Contracts [1], etc.

Manually defining specifications for libraries is time-consuming. One must read the documentation of the APIs and even the source code and convert the conditions to the formats suitable for verification tools. To ease defining specifications, several approaches have been proposed to automatically derive the specifications. Generally, there are two types of approaches that complement each other: *program analysis-based* and *data mining-based* approaches.

Among program analysis approaches, dynamic approaches [5, 8, 17, 39, 54] could detect data and temporal invariants and recover program behaviors. However, they require a large number of test cases, and their results might be incomplete due to the incompleteness of the test suites. On the other hand, static analysis approaches do not require dynamic instrumentation but have high false-positive specifications [16, 28, 44, 53]. Importantly, those static techniques focus their analyses only on an *individual* project, which has the call sites for *only a small number of APIs*.

In contrast to program analysis-based approaches, other techniques in the mining software repositories (MSR) area have applied mainly data mining to derive API specifications from existing code repositories [21, 31, 34, 42, 51, 52, 58, 59]. The key difference of these mining approaches from the traditional program-analysis based approaches is that they consider *the usages of the APIs* at the call sites in the client programs of the APIs to derive the conditions *regarding only the usage orders or temporal orders* among the API calls. While some approaches detect such orders as pairs of method calls [52, 21, 55] (e.g., p must be called before q), other approaches mine the sequences of calls [59, 50] or even a graph or finite state diagram of method calls [42, 43, 51]. Other mining approaches focus on associations of API entities [31, 34]. Unfortunately, those mining approaches do not aim to recover pre- and post-conditions as part of specifications. Moreover, except a few methods [44], they mainly rely on mining techniques without in-depth analyzing the data and control properties in the mined code.

This paper introduces an approach that puts forth the idea of mining API specifications that combines both static analysis and source

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635924

code mining from a very large code corpus in open-source repositories to *derive the preconditions of APIs* in libraries and frameworks. We expect that the **APIs' preconditions would appear frequently in an ultra-large corpus of open-source repositories** that contain a very large number of the usages of those APIs, while **project-specific conditions will occur less frequently**. Importantly, we *combine the strength of both static analysis approaches (via control dependency analysis) and MSR approaches (via mining)* to make it scale to large corpus. Moreover, we can derive preconditions for a large number of APIs or entire library at the same time.

Specifically, we used a very large-scale data set from SourceForge consisting of 3,413 Java projects with 497,453 source files, 4,735,151 methods and 92,495,410 SLOCs, and from Apache consisting of 146 projects with 132,951 source files, 1,243,911 methods and 25,117,837 SLOCs. To analyze the APIs' client code in such large data set, we did not choose the dynamic analysis approach since it would require the generation of a very large number of test cases and a great deal of execution time. Instead, we develop a light-weight, intra-procedural, static analysis technique to collect all predicates for every API method in the data set. Our technique first builds the control dependence relation for each method. It then analyzes different paths and conditions that lead to each method to recover all primitive predicates for all API methods in the data set. After that, it will start mining on the preconditions by performing normalization, merging, filtering, and ranking on them.

In our empirical evaluation, we compared the mined preconditions with the real-world JML specifications for several JDK APIs that are created and maintained by the JML team [29]. The results show that our precondition mining technique can achieve high accuracy with recall from 75–80% and precision from 82–84% for the top-ranked results. We also found 5 new preconditions (for two JDK classes) that were not listed by the JML team. We reported to the team and got their confirmation on those preconditions. Moreover, we filed to the JML team the preconditions for 11 previously unspecified methods in 2 JDK classes, and they accepted all proposals. Importantly, our method is light-weight and scales to such large amount of code, allowing us to derive preconditions for entire JDK library. We also conducted a user study on human subjects who have experience with specifications on the usefulness and correctness of our mined preconditions. **82%** of the participants found that our result is a good starting point for writing specifications for APIs under study. In addition to supporting specification writing, we show the usefulness of our mined preconditions by using them to build a benchmark of more than 4,000 API call sites that might be buggy due to missing precondition checking. It is useful for tools to detect neglected conditions [10]. Our key contributions include

1. A novel approach that combines the strength from both code mining in a *ultra-large code corpus* and program analysis, to derive the preconditions of APIs in libraries and frameworks,

2. An empirical evaluation on a very large-scale data set to mine preconditions of JDK APIs.

Section 2 will explain an example that motivates our approach. Our key program analysis and mining technique is presented in Section 3. Section 4 is for our empirical evaluation. Related work is described in Section 5. Conclusions appear last.

2. MOTIVATING EXAMPLE

We first present an example of an API in Java Development Kit (JDK), and then discuss the observations motivating our approach.

An Example. Let us consider an example of a commonly-used API from the Java Development Kit (JDK): String.substring(int, int) in the java.lang package. The method takes as input two integer values: beginIndex, the index of the starting character (inclusive) and endIndex, the index of the the ending character (exclusive). The method returns a new string that is the substring of the original string, using the two indices. Examining this API, we could learn that there are three preconditions that must hold before it is called:

```
1 public boolean setPathFragmentation(int servletPathStart, int extraPathStart){
2   if (servletPathStart < 0 || extraPathStart < 0  ||
3       servletPathStart > completePath_.length()  ||
4       extraPathStart > completePath_.length() ||
5       servletPathStart > extraPathStart)
6       return false;
7   if (servletPathStart == completePath_.length()) {
8       ...
9       return true;
10  }
11  if (completePath_.charAt(servletPathStart) != '/')
12      return false;
13  if (extraPathStart == completePath_.length()) {
14      ...
15      return true;
16  }
17  if (completePath_.charAt(extraPathStart) != '/')
18      return false;
19  contextPath_ = completePath_.substring(0, servletPathStart);
20  servletPath_ = completePath_.substring(servletPathStart, extraPathStart);
21  ...
22  return true;
23 }
```

Figure 1: Client code of API String.substring(int,int) in project SeMoA at revision 1929. `http://goo.gl/u0HK16`

1. beginIndex ≥ 0,
2. endIndex $\leq$ this.length() and
3. beginIndex $\leq$ endIndex.

A precondition for an API to be used could involve the *receiver object* of the API and/or one or multiple of *its arguments*. Identifying the complete set of preconditions for an API is a difficult and time-consuming task. However, this particular API is extremely popular (one of the most frequently used APIs from the JDK) and there is another way to learn these preconditions, without having to even look at the documentation or source code for the method. Consider one example usage of this API as shown in Figure 1. The method Request.setPathFragmentation(...) in the SeMoA [46] project uses this API (lines 19 and 20). Examining the source, we can see several conditions that must be false in order for the control-flow to reach the API calls. For example, the if statement on line 2 must be false, meaning that both indices servletPathStart and extraPathStart must be non-negative, the indices must not be greater than the length of the string completePath_, and servletPathStart must not be greater than extraPathStart. These are the same conditions we saw in the documentation. This gives us our first observation:

OBSERVATION 1. *Preconditions can be inferred by looking at the conditions that must be satisfied before calling the APIs, i.e., the guard conditions of the API call sites.*

Let us consider line 19 of Figure 1. It contains another call to the API. The only difference is that at this call site, instead of a variable, constant value 0 is passed as the first argument. Thus, the conditions on an argument of an API can be derived from the properties of such value passed to the API. This gives the observation:

OBSERVATION 2. *The mining tool should take into account the properties of the arguments passed as the APIs' parameters.*

This client code however contains other conditions checked before the API call. Some of these conditions are specific to the logic of the client (lines 11 and 17). This gives our next observation:

```
1 private String getCommand(int pc, boolean allThisLine, boolean addSemi){
2   if (pc >= lineIndices.length)
3     return "";
4   if (allThisLine) {
5     ...
6     return ...;
7   }
8   int ichBegin = lineIndices[pc][0];
9   int ichEnd = lineIndices[pc][1];
10  ...
11  String s = "";
12  if (ichBegin < 0 || ichEnd <= ichBegin || ichEnd > script.length())
13    return "";
14  try {
15    s = script.substring(ichBegin, ichEnd);
16    ...
17  }
18}
```

Figure 2: Client code of API String.substring(int,int) in project Jmol at revision 18626. `http://goo.gl/Qa8NiS`

OBSERVATION 3. *Call sites might contain client-specific conditions, which could cause noise when inferring preconditions. Thus, an approach that mines preconditions from call sites should attempt to minimize such noise.*

This has been a challenge for the existing static program analysis-based approaches [44] when they examine the call sites of the APIs only within the code of the APIs' programs.

One way to minimize noise is to mine preconditions from *a large number of projects*. The valid preconditions should appear more frequently, while client-specific conditions should appear infrequently. Figure 2 shows another client, Jmol [27], that uses the same API (line 15) in the method ScriptEvaluator.getCommand(...).

The if statement on line 12 checks the three required preconditions. Note that, in this case, the checked condition is stronger than the required one: the beginning index ichBegin is *strictly less* than the ending index ichEnd. This gives our next observation:

OBSERVATION 4. *The relationship between conditions should be considered when mining preconditions.*

For example, a stronger condition should be counted as an instance of a weaker one. A mining tool must consider the relations among conditions to derive a precondition. Similar to the previous client code, this method also contains client-specific conditions (lines 2 and 4). Again, these conditions are project-specific and not actual preconditions for the API in question. However, these conditions do not appear in the first client code, and thus we start to see evidence that such noise would appear less frequently.

Motivation. This example motivates us to use an approach to mine the preconditions via the guard conditions of the call sites of the APIs under study in a very large number of projects in a large-scale corpus. That would help to minimize the *project-specific conditions* (as noises) because they will appear less frequently in the large corpus. The true preconditions would occur more frequently.

In this paper, we introduce such an approach that mines the preconditions of the APIs. In fact, after running our mining tool on a very large data set from SourceForge (consisting of 3,413 Java projects with 497,453 source files, 600,274 classes, 4,735,151 methods, and 92,495,410 SLOCs), we are able to derive the preconditions for the String.substring method in JDK. The columns in Table 1 show the preconditions with highest frequencies in the corpus that we mined for the receiver String object, and the arguments beginIndex and endIndex, respectively. As seen, the aforementioned true preconditions have among the highest frequencies. Project-specific conditions did not make the top of the list.

Table 1: Mined Preconditions for String.substring(int,int)

Receiver Object (rcv)	beginIndex	endIndex
rcv.length() > 0	rcv.length() > beginIndex	endIndex >= 0
rcv.length() >= endIndex	beginIndex <= endIndex	endIndex != -1
rcv.length() > beginIndex	beginIndex >= 0	rcv.length() >= endIndex

3. MINING WITH LARGE CODE CORPUS

Let us outline our approach for mining the preconditions for API methods. Figure 3 gives an overview, which can be summarized as:

1. The input is the set of all API methods under analysis and client projects to mine.
2. For each method in the corpus that calls an API, we build the control dependence relation between each method call and the predicates in the method (from the control-flow graph) and identify all preconditions of API calls. (Section 3.1)
3. Next, we normalize the preconditions to identify and combine the equivalent ones. (Section 3.2)
4. We then analyze the preconditions to infer additional ones which are not directly present in the client code. (Section 3.3)
5. Finally we filter out non-frequent preconditions (Section 3.4, and rank the remaining ones in our final result. (Section 3.5)

3.1 Control Dependence and Preconditions

In order to identify the preconditions of API calls, we need to identify all predicates that guard the evaluation of each method call in the program. This can be done by building the control dependence relation [18], based on the control-flow graph (CFG). In a CFG, each predicate node has exactly two outgoing edges labeled TRUE and FALSE representing the two corresponding branches.

DEFINITION 1. *A method call C is **control-dependent** on a predicate expression p if and only if on the corresponding CFG, all directed paths from p to C go out of p on the same edge - either TRUE or FALSE.*

This means that C is control-dependent on p if C is executed in only one branch of p. If C could be called in both branches of p, then C's execution does not depend on p. For example, in Figure 2, String.substring on line 15 is called only in the FALSE branch of the predicate on line 12, thus, it is control-dependent on that predicate. Our definition is stricter than the traditional definition by Ferrante *et al.* [18], which requires C always be called in one branch of p and not called in at least one path in the other branch. According to that, a method could be called in both TRUE and FALSE branches of the predicate on which it is control-dependent, thus the value of the predicate does not control the execution of the method call. This is the reason we give an adaptation in Definition 1.

DEFINITION 2. *An API method M is **control-dependent** on a predicate expression p in a client method if and only if all call sites of M in the client method are control-dependent on exactly one branch of p (TRUE or FALSE).*

When M is control-dependent on the FALSE branch of p, the predicate that guards M will be the negation of the predicate expression in p. We now define what we consider to be a *precondition* for calling a method.

DEFINITION 3. *A **precondition** of a method call is a single clause in the conjunctive normal form (CNF) of a predicate on which the method call is control-dependent.*

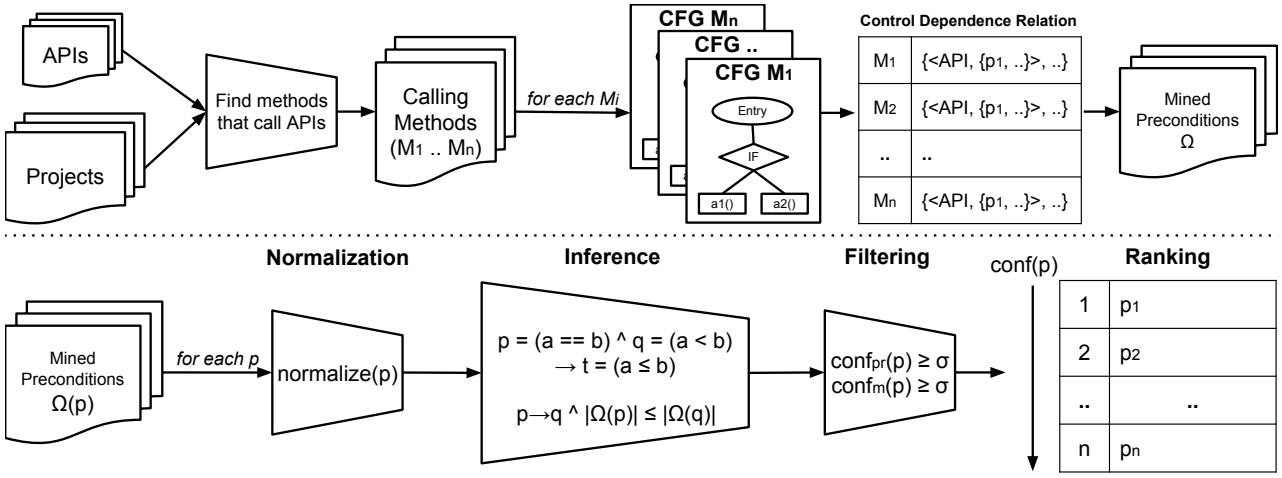


Figure 3: Approach Overview: We first find all methods that call each API and compute the control-flow graph for each. Then, we generate the control dependence graph to identify conditions leading to an API call. From that, we create an inverted index, then normalize each condition. We infer and merge conditions and then filter some out. The final list is ranked, giving us our result.

In Figure 2, the API call on line 15 is control-dependent on the FALSE branch of the if statement on line 12, so the predicate is negated and gives us: !(ichBegin < 0 || ichEnd ≤ ichBegin || ichEnd > script.length()). This predicate is represented in CNF as !(ichBegin < 0) && !(ichEnd ≤ ichBegin) && !(ichEnd > script.length()). Moving the negations inside, we have a set of three preconditions: ichBegin ≥ 0, ichEnd > ichBegin and ichEnd ≤ script.length().

For the goal of deriving general specifications, the context-specific names/expressions must be **abstracted** away from the individual method call sites. Since each call contains a receiver object and list of arguments, we are interested in the preconditions on each of these components. We use **rcv** and **argi** as the **symbolic names** for the receiver object and the i-th argument in the list of arguments, respectively. First, we match the expression of the receiver and that of each parameter of the method call against the expression of the precondition. Then, we try all possible substitutions of occurrences of the receiver and parameters with their corresponding symbolic names. If the condition contains a variable/field, its latest value will be used in the precondition. Its latest value is the expression in the right hand side of its most recent assignment (if any). In the above example, processing the three preconditions ichBegin ≥ 0, ichEnd > ichBegin and ichEnd ≤ script.length() of the method call script.substring(ichBegin, ichEnd) will result in the following abstracted preconditions arg0 ≥ 0, arg1 > arg0 and arg1 ≤ rcv.length(). A condition that does not involve any component of the call (i.e., having no symbolic names) will be discarded.

Finally, to follow Observation 2, for each expression e passed as argument argi to a method call, we create a precondition in three cases. First, if e is a constant of a primitive type, we create a precondition argi == c. Second, if e is an expression that can be recognized via its syntax as returning a non-null object, e.g., object instantiation or array initialization expression, we create a precondition argi != null. Third, if e involves any component of the call, i.e., having some symbolic names, we create a precondition argi == e', where e' is obtained from e by replacing identifiers with the corresponding symbolic names, e.g., arg1 == rcv.length(). These equality preconditions are used to support the inference of the non-strict inequality preconditions such as arg1 >= 0 or arg1 >= rcv.length().

Table 2 shows the resulting preconditions mined from our example API using this process. For each API, the preconditions are

Table 2: Extracting preconditions for String.substring(int, int) from the usages in Figures 1 and 2.

Figure 1, line 19	Figure 1, line 20	Figure 2, line 15
arg0 == 0 rcv.charAt(arg0) == '/' arg1 >= 0 arg1 != rcv.length() == '/' rcv.charAt(arg1)	arg0 >= 0 arg0 <= rcv.length() arg0 != rcv.length() rcv.charAt(arg0) == '/' arg0 <= arg1 arg1 >= 0 arg1 <= rcv.length() arg1 != rcv.length() rcv.charAt(arg1) == '/'	arg0 >= 0 arg1 > arg0 arg1 <= rcv.length()

stored in a map Ω, in which $\Omega(p)$ returns the set of calling methods containing precondition p before calling the API.

3.2 Precondition Normalization

Since we collect preconditions from call sites in different methods and projects, there are conditions that are equivalent but expressed in different forms. For example, the following: arg1 > arg0, arg0 < arg1, (arg0 - arg1) < 0, (arg1 - arg0) > 0 and arg0 - 1 < arg1 - 1 express the same conditions. Thus, we need to normalize the preconditions. The first step is to ensure every unary/binary expression is enclosed by exactly one pair of opening and closing parentheses. The next step is to order the operands in the binary operation(s) (such as >, <, etc.) of the preconditions so that they are comparable between call sites. Whenever two operands of a binary operation are re-ordered, the operator is reversed correspondingly.

For any comparison expression $E = E_l \trianglerighteq E_r$, where $\trianglerighteq$ is a comparison operator, we transform it into $E' = E'_l \trianglerighteq E'_r$, where E'_r contains only literals and E'_l contains all symbolic and other identifier names. If E'_r contains all numeric literals, it will be evaluated. The terms in E'_l are ordered in the ascending order of its symbolic names. For example, all 5 conditions above will be normalized into the same condition (arg0 - arg1) < 0. Finally, the map Ω is updated with the normalized preconditions for each API.

3.3 Precondition Inference

Inferring non-strict inequality preconditions. In the client code, a non-strict inequality precondition (a >= b or a <= b) might be split into strict inequality (a > b or a < b) and equality (a == b) condi-

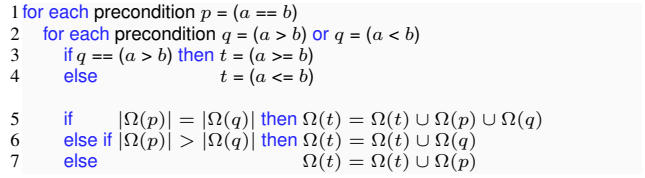


Figure 4: Inferring non-strict inequality preconditions.

tions, and checked at different call sites. Figure 4 shows our algorithm for inferring the non-strict inequality precondition. When the two preconditions p and q are used equally, all call sites for both of them are counted toward the inferred condition (line 5). Otherwise, only the call sites of the less-frequently used precondition are added (lines 6 and 7). This helps us avoid counting the occurrence frequencies of incorrect conditions toward the inferred one.

Merging strong and weak conditions. Among the preconditions, some imply others (Observation 4). If a stronger condition holds, the weaker condition holds too. This means that all call sites of the stronger condition could be merged to (counted toward) those of the weaker. However, merging can lead to inferring wrong preconditions if the weaker one is in buggy code or specific to a particular client (Observation 3). We avoid this noise by using the assumption that the more frequently a precondition is checked, the more likely it is correct. Thus, if the stronger condition is less-frequently checked than the weaker one, its call sites will be merged to those of the weaker and it will be removed from the set of preconditions.

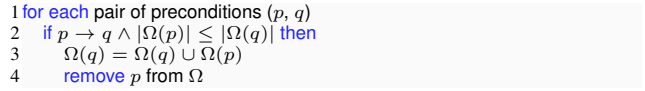


Figure 5: Merging preconditions with implication.

The procedure is shown in Figure 5. Note that this merging will remove all equality and/or strict inequality preconditions composing the non-strict ones. For example, if two conditions p: (arg == 0) and q: (arg > 0) infer the condition t: (arg >= 0) and p is stronger and less-frequently checked than t, as at line 7 in Figure 4, its call sites containing p will be added to those of t. Then, p is removed.

Dealing with dynamic dispatch. Since the data types cannot be precisely resolved at static time, some actual API calls could be missed in our static analysis, thus, all their preconditions at those call sites could be missed too. For example, method obj.add() which is resolved at static time as List.add() because obj is declared as List could actually be ArrayList.add() at runtime. We address this with a conservative solution that whenever a set of preconditions is extracted for a call of API m, that set is also considered as the preconditions of all APIs that override or implement m in the library. The rationale behind this is the assumption of behavioral subtyping in which preconditions cannot be strengthened in a subtype [32]. Thus, this heuristic will enrich the set of extracted preconditions for a sub-type with those from the super-type, which are the same or stronger than the actual ones. Those preconditions could be merged to the actual ones and increase the confidence of the actual ones.

3.4 Precondition Filtering

Since we mine preconditions from many projects/methods in a large-scale code corpus, there are conditions which are context-specific or might even be incorrect. These conditions are not useful for building the API specifications and should be filtered out. First, we remove all conditions which are checked only once in the whole code corpus. Then, for each API, we remove all conditions which have low confidence in being checked before calling the API.

The confidence of a precondition for an API is measured as the ratio between the number of code locations checking the condition before calling the API over the total number of locations calling the API. We compute two values for confidence corresponding to two types of locations: one over client projects ($conf_{pr}$) and another over client methods ($conf_m$):

$$conf_{pr}(p) = |\Psi(p)| / \left|\bigcup_q \Psi(q)\right|$$
$$conf_m(p) = |\Omega(p)| / \left|\bigcup_q \Omega(q)\right|$$

where $\Psi(p)$ is the set of projects with condition p before the API call. For each API, we keep only the preconditions that have both confidence values higher than or equal to a certain threshold σ. We use $\sigma = 0.5$ in our experiment.

3.5 Precondition Ranking

For each API, we rank the preconditions based on their total confidence, which is computed as $conf(p) = conf_{pr}(p) \times conf_m(p)$. Using only $conf_m(p)$ might favor the conditions used a lot but only in a small number of projects. In contrast, using only $conf_{pr}(p)$ might favor the conditions which are accidentally repeated in many projects but not used frequently. Thus, our approach combines both confidence values to rank the preconditions. Different from the traditional ranking scheme that puts all items in one list, our approach uses different ranked lists for the receiver object, the arguments of an API, and any combinations of them. Only the top-1 precondition in each ranked list is kept in the final result.

4. EVALUATION

In this section, we aim to answer two research questions:
RQ1. *How accurate are the preconditions mined by our approach?* The answer to this question would tell whether our approach works in identifying the preconditions from usages in a large code corpus.
RQ2. *How useful are the mined preconditions as a starting point in writing API specifications?*

4.1 Data Collection

We collected a large code corpus from two sources: SourceForge.net (SF) [48] and Apache Software Foundation (ASF) [6]. SF is a free source code hosting service for managing open source software projects. ASF is an American non-profit corporation who manages the development of Apache open source projects.

For SF, we downloaded project metadata in JSON format from its website and collected information about all projects that are self-classified to be written in Java. To get higher quality code for mining the preconditions, we filtered out the projects that might be experimental or toy programs based on the number of revisions in the history. We only kept projects with at least 100 revisions. We downloaded the last snapshots of each project. We eliminated from the snapshot of a project the duplicated code from different branches/versions of the project. For ASF, we checked the list of all Apache projects [6] and downloaded the source code of the latest stable releases of all projects written in Java.

Table 3 shows the statistics on our datasets. SF has 3,413 projects satisfying the above criteria and ASF has 146 projects. They both have hundreds of thousand of source files. The total amount of code is almost 120 million lines of code (SLOCs) where SF contributes about four times more than ASF. The projects are written by thousands of developers and cover a variety of domains and topics.

Table 3: Collected projects and API usages.

	SourceForge.net	Apache
Projects	3,413	146
Total source files	497,453	132,951
Total classes	600,274	173,120
Total methods	4,735,151	1,243,911
Total SLOCs	92,495,410	25,117,837
Total JDK public classes	1,275	1,275
Total JDK public methods	11,049	11,049
Total used JDK classes	806 (63%)	918 (72%)
Total used JDK methods	7,592 (63%)	6,109 (55%)
Total method calls	22,308,251	5,544,437
Total JDK method calls	5,588,487	1,271,210

```
1 /*@  public normal_behavior
2  @   requires 0 <= beginIndex
3  @            && beginIndex <= endIndex
4  @            && (endIndex <= length());
5  @   ensures \result != null && \result.stringSeq.equals(this.stringSeq.
       subsequence(beginIndex,endIndex));
6  @ also
7  @  public exceptional_behavior
8  @   requires 0 > beginIndex
9  @            || beginIndex > endIndex
10 @            || (endIndex > length());
11 @   signals_only StringIndexOutOfBoundsException;
12 @*/
13 public String substring(int beginIndex, int endIndex);
```

Figure 6: JML Specification for String.substring(int, int)

In this experiment, we focus on the APIs in the JDK library. Analyzing all APIs from the java packages, we found that there are 1,275 public classes and 11,049 public methods in the library. We also observed that many APIs have not been used at all in the studied projects. Only 63% and 72% of the accessible JDK classes have been used in SF and Apache, respectively. The corresponding numbers of JDK methods used are 63% and 55%, respectively. In both SF and ASF, about one-fourth of the number of all method calls are the calls to JDK methods. This number shows that those open-source projects are heavily based on the JDK library.

4.2 Ground-truth: Java Modeling Language (JML) Preconditions

In order to evaluate the accuracy of our mined preconditions, we used a ground-truth of known-correct preconditions. The Java Modeling Language (JML) is a language for specifying the behavior of Java classes and methods. Specifications are defined using a custom syntax inside of special comments that start with '@'. Figure 6 shows part of the specification in JML for the substring(int, int) method discussed in Section 2. The specification defines both normal behavior (lines 1–5) and exceptional behavior (lines 7–10), and signals certain Exception when certain preconditions hold (line 11).

The normal behavior for this method requires three conditions to hold prior to calling the method. These conditions are declared using requires statements and boolean expressions (lines 2–4). The specification also ensures that after finishing normal execution two conditions hold. These are declared using ensures statements and boolean expressions (line 5). A precondition is 1) a clause in the conjunctive normal form of the boolean expression following a requires keyword in a *normal behavior*, or 2) a clause in the conjunctive normal form of the *negation* of the boolean expression following a requires or signals keyword in an *exceptional behavior*. If a specification has multiple normal and/or exceptional behaviors, we combine them by taking the union set of the preconditions. For example, if preconditions i > 0 and i == 0 appear in two normal behaviors, they will be combined into a precondition i >= 0. The preconditions are then abstracted using the symbolic names.

Table 4: Specifications for JDK classes from JML Website

Package	Number of classes			
	Full Spec	Some Spec	No Spec	Total
java.io	3	0	7	10
java.lang	14	3	1	18
java.net	2	0	0	2
java.sql	0	0	5	5
java.util	23	2	0	25
java.util.regex	0	0	2	2
All	42	5	15	62

Table 5: JML preconditions of JDK methods in classes with full specifications

	Number of Preconditions							
	0	1	2	3	4	5	6	7
Methods	78	465	144	62	36	4	4	4

Number of methods: 797. Number of preconditions: 1155.

The authors and maintainers of JML have written specifications for several popular Java packages from the JDK and published them on their website [26]. We downloaded and analyzed these specifications. As shown in Table 4, there are specification files for 62 classes from 6 JDK packages. After analyzing, we learned that, in 15 class files, there are no specifications for any method (column No Spec) and in 5 other files, there are specifications for some methods but not all (column Some Spec). We read the remaining 42 files, which contain specifications for all methods (column Full Spec), and extracted all preconditions for all of their methods.

Table 5 summarizes the number of extracted preconditions of the methods in those 42 classes. We group the methods based on the numbers of extracted predicates in the preconditions. In total, there are 1,155 preconditions for 797 methods in which 78 of them have no preconditions, 465 of them have one precondition, and so on. As seen, most of them have from 0 to 3 preconditions. A much smaller percentage of methods has more than 3 preconditions.

4.3 RQ1: Accuracy

4.3.1 Result

We ran our tool on the two datasets, and compared the mined preconditions with those in the JML ground-truth. We used two metrics: precision and recall. *Precision* is measured as the ratio between the number of correctly-mined preconditions and the total number of mined preconditions. *Recall* is measured as the ratio between the number of correctly-mined preconditions and the total number of preconditions. A mined condition is considered correct if it is exactly matched with one precondition of the same method in the ground-truth using syntactic checking. If a mined condition is not in the ground-truth, we manually verified it. If it is a not-yet defined one, or semantically equivalent with a precondition (e.g., !rcv.isEmpty() and rcv.size() > 0) or implied by the preconditions of that method in the ground-truth (e.g., b > 0 is implied by a > 0 and a < b), it is counted as correct.

Table 6: Mining accuracy over preconditions

	Mined	Precision	Recall	Time
SourceForge	1,098	84%	79%	17h35m
Apache	1,065	82%	75%	34m
Both	1,127	83%	80%	18h03m

Table 7: Mining accuracy over methods

Dataset	Fully-covered				Total.Inc.
	Total	Perfect	1 Extra	>1 Extra	
SF	613 (77%)	492 (62%)	118 (15%)	3 (0.38%)	60 (8%)
Apache	593 (74%)	457 (57%)	126 (16%)	10 (1.25%)	78 (10%)
Both	628 (79%)	489 (61%)	131 (16%)	8 (1.00%)	47 (6%)

Table 6 shows the accuracy for all mined preconditions. In both datasets, the tool achieved high accuracy with recall from 75–80% and precision from 82–84%. The accuracy for two sources is comparable. The accuracy for SourceForge is a bit higher than that for Apache. When both datasets are combined, precision lies between those for two datasets. However, recall is slightly improved since a few more API methods, which were not seen in either dataset, have been included in the result for the combined dataset.

Table 7 shows more detailed numbers on the mining accuracy for all the API methods. As seen, with the SourceForge dataset, our tool can *cover all* of the preconditions for 613 out of 797 (77%) JDK methods in the ground-truth. That is, in 77% of given methods under investigation, specification writers would just have to verify and remove some incorrect ones. Among those 613, we can derive *perfectly* the preconditions for 492 methods. That is, in 62% of methods, specification writers would use the set of preconditions as is. There are 118 (15%) and 3 (0.38%) methods having 1 and more than 1 extra (incorrect) preconditions, respectively. Our tool cannot produce any correct preconditions for only 8% of the methods. The numbers are comparable for Apache and the combined dataset.

Thanks to our light-weight analysis, the running time for Apache, which has more than 25M SLOCs and 1.2M JDK API calls, is just 34 minutes. The time for SourceForge and for both is much longer mainly due to accessing the local SVN repositories.

4.3.2 Analysis

Incorrect Cases. We first analyzed the *incorrect cases*. Since the JML specifications were manually built by the JML team, it is possible that some preconditions are still missing from the current version of their specifications. Thus, for the mined preconditions that are not in the ground-truth from the JML team, we manually verified them to see if they are truly incorrect cases. We found 5 correctly mined preconditions that were missing in the ground-truth (Table 8). We sent them to the main author of JML. He kindly confirmed all five cases. This is evidence that our tool could help specification writers reduce their effort and mistakes.

Table 9 shows the summary of the incorrectly-mined preconditions, which are classified into 3 types. For majority of the incorrect cases, the mined preconditions are **stronger** than the actual ones. The reason is that our tool cannot distinguish between the precondition as part of an API usage and the one as part of the API specification. For example, the API java.util.List.add(Object) accepts a null argument. However, in many usages of that API in the client code, developers often perform null checking for the argument before calling it. Thus, our tool reported the incorrect condition: arg0 != null. Another example is the API File.mkdir(), which does not require any preconditions in its specification. If the operation fails for some reason, it will return null. However, to avoid unnecessary operations to the file system and control the reason of the failure, developers often check file existence with !exists() before calling mkdir(). Another example is the method valueOf(Object obj). Our tool detects the null checking on the argument arg0 != null from several client projects, but it is not part of its specification. These examples show an interesting gap between the actual API usages from client code and the intended usages from the API designers. This suggests a further investigation for API designers on how to adjust to support developers better in the APIs' client code.

Table 8: Newly found preconditions in JML specifications

Class	Method	Precondition
String	getChars(int,int,char[],int)	arg3 >= 0
StringBuffer	append(char[])	arg0 != null
BitSet	flip(int, int)	arg0 <= arg1
	set(int, int)	arg0 <= arg1
	set(int, int, boolean)	arg0 <= arg1

Table 9: Different types of incorrectly-mined preconditions

Dataset	Total	Stronger	Irrelevant	Analysis.Err.
SourceForge	173	118	53	2
Apache	187	121	65	1
Both	195	129	66	0

In the second type of incorrect cases, the conditions along the path to an API call are **irrelevant** to the preconditions of the API. For example, it is frequent that developers check if both arguments are positive before calling Math.min(). Those checks might make sense in term of the logic of the program, however, they are not relevant as the preconditions.

For the third type, a few incorrect cases are caused by the **imprecision in our light-weight static program analysis**. An example is incorrectly-mined precondition arg0 <= 0 of StringBuffer.ensureCapacity(int). In the code, the call to this API belongs to the branch satisfying arg0 <= 0, however, the sign of arg0 is reversed before the call. Our analysis did not keep track of the value change in the code leading to the call, thus, extracted incorrect condition. To track value changes, we can use dynamic symbolic execution.

Missing Cases. To better understand the missing cases, we examined all the preconditions which are in the ground-truth but were not mined by our tool. We classified the missing cases into four categories as shown in Table 10. Each cell of the table shows the ratio between the number of missing cases in the corresponding category over the total number of preconditions in the ground-truth.

The *first category* (column 'No-call') consists of the preconditions of the API methods that have their JML specifications in the ground-truth, but have *never been called* in the client code in our datasets. For SourceForge, there are 46 such methods with 45 preconditions. For Apache, the corresponding numbers are 49 and 58. For those methods and preconditions, which contribute about 4% and 5% of the total numbers of preconditions, respectively, our tool can not mine the preconditions.

The *second category* (column Private) contains the preconditions involving the APIs' private and internal fields or methods, which are inaccessible from client code. Examples of this category are

1. Precondition !changed of Observable.notifyObservers(): changed is a private field of the Observable class to represent the

Table 10: Four Types of Missing Preconditions

	No-call	Private	No occur	Low freq.
SF	4%	4%	9%	3%
Apache	5%	5%	12%	3%
Both	2%	5%	10%	4%

internal state of the object. The method notifyObservers is called only if the object's state was changed.

2. Precondition parseable(s) of Integer.parseInt(String s): this condition requires the string argument of parseInt to be parseable.

3. Precondition capacityIncrement >= 0 of Stack.push(Object): The stack can only be pushed if its internal capacity is larger than 0.

The first two categories are due to the inherent limitation of mining approaches on client code, however, their percentages are small.

The *last two categories* contain the preconditions which could occur in the client code but are not in our result due to the limitations of our static analysis that cannot detect the occurrences of the conditions (No occur.) or due to the cut-off thresholds (Low freq.).

4.3.3 Accuracy by data size

When computing accuracy, we also analyzed the impact of the size of dataset in our algorithm. We ran our tool on various data sizes. From each full dataset, SourceForge and Apache, we created the datasets of size B by randomly selecting the projects of the full dataset into bins having the same number of B projects. Using each bin as input, we ran our tool on it and recorded the accuracy (*precision* and *recall*) for that bin. Then, we computed the average accuracy over all bins, and used that accuracy for that size. In this experiment, we chose $B = 2^i$, meaning that we kept increasing the data sizes by a power of 2 until reaching the full dataset. To consider both precision and recall, we used Fscore. Fscore is the harmonic mean of precision and recall, which is computed as $Fscore = 2 \times Precision \times Recall/(Precision + Recall)$.

Figure 7 shows the result. The values on the lines at $B = 1$ shows the accuracy for the dataset containing individual projects and those at $B = Full$ shows the accuracy for the full dataset as input. As the data size increases, precision decreases and recall increases. The gain in recall is much higher than the loss in precision making their harmonic mean Fscore increases significantly: 7% to 82% for SourceForge and 21% to 79% for Apache.

4.3.4 Accuracy Sensitivity Analysis

In this experiment, we studied the impact of different components in our method on the accuracy. In Figure 8, the baseline (group Base) is the solution that extracts the preconditions by looking at only the guard conditions (e.g., the ones in if statement(s)) on the path leading to the API calls. This baseline does not consider the properties of the passed arguments, normalization and merging, nor deal with dynamic dispatch. Then, we successively add other components one by one to the baseline solution to see changes in accuracy. The second solution (Arg) adds the preconditions that are obtained from the properties of the passed arguments, e.g., arg0 == 0, arg1 != null. The third one includes normalization of preconditions and the fourth one includes merging. The last one covers all components in our approach by adding the subtyping information in which the preconditions of a method are also collected from the call sites of its overridden methods to deal with dynamic dispatch.

As more components are added, recall increases significantly from 60 to 79% in SourceForge and from 55 to 75% in Apache, while precision is maintained. Among the components, adding properties of arguments passed to APIs improves the recall 6% in SourceForge and 8% in Apache. The respective improvements from adding merging conditions are 7% and 5%. Adding subtyping contributes 4% and 6%. Normalization contributes 2% for both.

4.4 RQ2: Usefulness

We also studied how useful our automatically mined preconditions are for writing specifications via two experiments.

4.4.1 Suggesting preconditions in specifications

Our first experiment looks at the mined preconditions for API methods that currently do not have a JML specification provided. We run our tool to automatically mine preconditions for the APIs and then manually transformed them into JML syntax. We then sent these JML-styled specifications to one of the original authors of JML. If he agreed these specifications are correct, it lends evidence that our approach is useful as a tool for suggesting preconditions when writing the initial specification for APIs.

Our results are summarized in Table 11. In total, we prepared specifications for 11 API methods from 2 JDK classes which previously had no JML specifications. Our tool generated a total of 29 mined preconditions (column M). For our approach, one author transformed the automatically generated preconditions into JML specifications. A second author, who has extensive experience with JML's syntax including designing and implementing the JML research compiler JAJML, then performed a manual validation of the results and removed 4 preconditions (column Rm) which are incorrect for the corresponding APIs. Five preconditions are deemed close (column Fix), but require modifications of the comparison operator from strictly greater than ($>$) to greater than or equal to ($>=$). The remaining 20 were accepted exactly as the tool mined them. After this step, the specifications containing 25 preconditions (including the 5 modified) were sent to the JML team member.

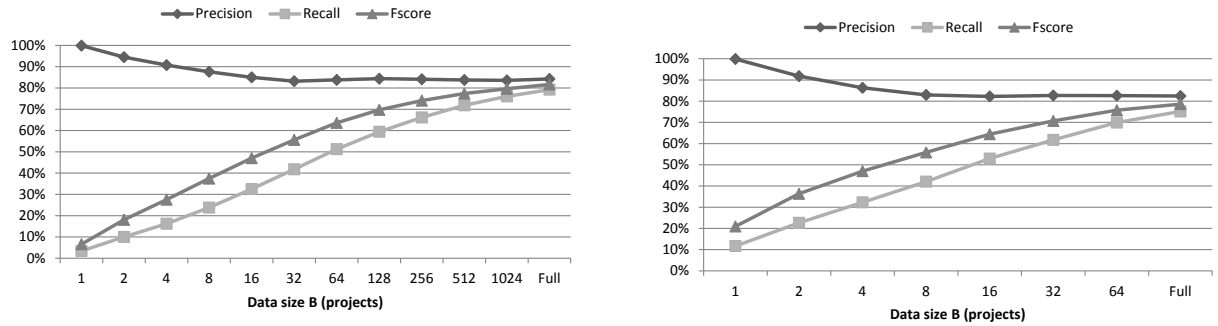
Table 11: Suggesting preconditions

Class	Method	M	Rm	Fix	Accept
StringBuffer	delete(int,int)	4	1	2	Y
	replace(int,int,String)	3	1	0	Y*
	setLength(int)	2	1	0	Y
	subSequence(int,int)	4	1	1	Y
	substring(int,int)	3	0	1	Y
LinkedList	add(int,Object)	2	0	0	Y
	addAll(int,Collection)	3	0	1	Y
	get(int)	2	0	0	Y
	listIterator(int)	2	0	0	Y
	remove(int)	2	0	0	Y
	set(int,Object)	2	0	0	Y
		29	4	5	

As seen, the JML team member agreed on 10 out of 11 methods's specifications, such that the set of suggested preconditions is complete and precise (**Y** in column Accept). For only one method StringBuffer.replace (**Y*** in column Accept), the preconditions are correct however two other ones are missing.

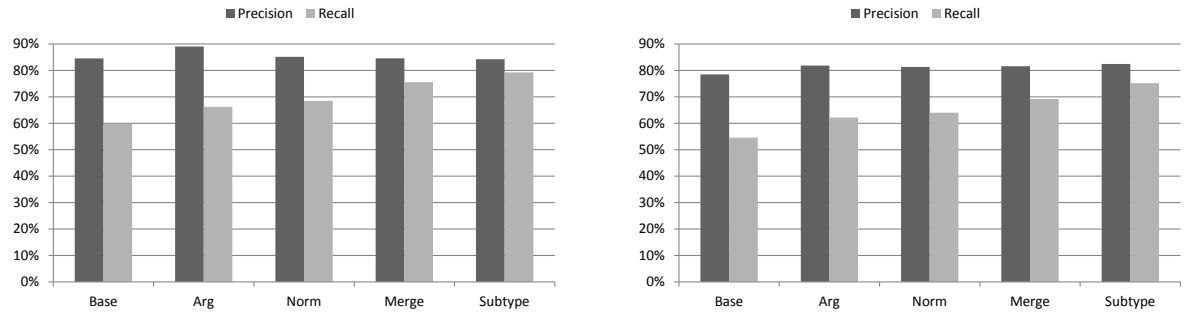
4.4.2 Web-based survey

In the second experiment, we created a web-based survey and asked human subjects who have experience with using JDK library and/or formal specification languages such as JML to evaluate the resulting preconditions. We had a total of **15** respondents. Participants were asked to rate their experience with Java, JML, reading specifications, and writing specifications. Two thirds self-indicated



(a) SourceForge (b) Apache

Figure 7: Mining accuracy as the size of dataset varies



(a) SourceForge (b) Apache

Figure 8: Mining accuracy as technical components are successively added

having more than 6 months experience writing specifications and many with experience in JML specifically.

Participants were shown an example method (e.g., the substring method from Section 2) along with the set of proposed preconditions we mined for that method. We then pre-selected the correct answers (based on the JML ground-truth) for each condition and explained why it was "correct", "a good starting point", or "incorrect". "Correct" means that this precondition can be used as-is in the specification. "Good starting point" means that it might need small modifications to be used in a specification, such as changing a comparison operator from strict to non-strict. "Incorrect" means that the condition is irrelevant in building the specification.

Next, users were shown 5 methods one at a time and the mined preconditions for them. They were asked to rate each individual precondition as mentioned. We also asked them to give an overall, more subjective, rating for the entire method on whether our mined preconditions are useful. After 5 methods, they were given an opportunity to write general feedback. They also had an opportunity to continue rating more preconditions for other methods. On average each participant graded 20 preconditions.

When randomly choosing methods for a user, we enforced that the first two were APIs that existed in the ground-truth and the last three were APIs that did not. Using the responses from the first two, we were able to grade the users on their expertise by calculating the answers that matched the ground-truth out of the total number of ground-truth answers. For this study, we only keep responses from users who scored 100% on this grading. In total, there were **9** users grading **75** methods with **104** preconditions.

The following table shows the correctness of the preconditions as rated by participants. Excluding the 'Not Sure' responses, the participants rated **63**% as Correct. What the results in Section 4.3 could not show however was the amount of almost correct preconditions, which the participants rated as almost **19**%.

Correct	Good Starting Point	Incorrect	Not Sure
64	19	18	3
63%	19%	18%	–

Overall, participants found that **82**% of the mined preconditions are useful as the starting point for writing the specification. The following table shows the responses for rating the tool's usefulness:

Agree+	Agree	No Opinion	Disagree	Disagree+
23	33	6	9	4
33%	48%	–	13%	6%

Again, excluding the 'No Opinion' responses, the participants rated the tool as useful for **81**% of the methods shown!

4.4.3 A benchmark of precondition-related bugs

In this section, we show an application of our mined preconditions in building a benchmark of bugs caused by missing precondition checking. An example of this type of bug is that a developer does not check the condition beginIndex $\leq$ endIndex before calling String.substring(int, int) when the logic of the program does not ensure it. This type of benchmark is very useful for bug detection tools that look for neglected condition checking such as Chang *et al.*'s tool [10] and Alattin[50]. It was reported that neglected conditions are an important but difficult-to-find class of defects [10].

To build the benchmark, we processed all **1,966,563** revisions with changed Java files for all 3,413 Java projects in SourceForge

dataset. For a project P, we first identified the fixing revisions by the popular method [60] that uses the heuristic of searching in commit logs for the phrases indicating fixing activities. For each fixing revision r_i, we used our prior origin analysis tool to compare it with the previous one r_{i-1}. We detected the mapped methods and API calls between two revisions. For each pair of mapped API calls in a method, we computed two sets of guard conditions. We compared each set with the mined preconditions of the API to find the set of preconditions that are implied by a guard condition. If there exists such a precondition in r_i but not in r_{i-1}, we add the API call sites and (r_{i-1}, r_i) to our benchmark. In total, there are **369,532** fixing revisions. Among them, **3,130** (**0.85%**) in **931** projects are detected as related to missing preconditions. The total number of call sites related to those fixes is **4,399**. To check its quality, we manually checked a sample of 100 call sites in the benchmark, and found that 80 of them are related to preconditions. We will manually check all and make our benchmark available. We found that null-pointer and index-out-of-bounds exceptions are the two most common sub-types in those bugs. Our result confirms this type of bug and calls for detection tools. This shows the usefulness of our mined preconditions in building the benchmark. Our mined preconditions can also be used in such detection tools.

Threats to Validity. The two chosen datasets might not be representative. The criteria of 100 revisions might not have filtered out all experimental and toy projects. We conducted experiments only on JDK. The ground-truth was built by us. Thus, human errors could occur. The two chosen classes in the usefulness study might not be representative. Our human study suffers from selection bias, as not all participants have the same level of expertise on formal specifications. There is possible construct bias as we chose the APIs in JDK. We did not compare our tool to a related one in [44]. Similar to ours, their tool is also based on both mining and program analysis. However, their tool is for C code and re-implementing it for Java code is infeasible due to their algorithm's complexity as well as the differences between two languages. Moreover, their approach operates on a single project while we rely on large number of projects. Thus, the two approaches require inputs with different nature. Other mining-based approaches do not work for preconditions, while other static and dynamic analysis methods for specification inference do not have a mining component (Section 5).

5. RELATED WORK

The condition mining work that is closest to our approach is from Ramanathan, Grama, and Jagannathan (RGJ) [44]. Similar to ours, the RGJ approach tightly integrates *program analysis* with *data mining techniques*. They proposed a static inference mechanism to identify the preconditions that must hold when a method is called. They first analyze the call sites of the method in its containing program and then use a path-sensitive inter-procedural static analysis to collect the predicates at each program point. To compute preconditions, RGJ collects a predicate set along each distinct path to each call-site. The intersection of predicate sets is then constructed at the join points where distinct paths merge. Predicates computed within a procedure are memorized and used to compute preconditions that capture inter-procedural control- and data-flow information. RGJ then runs frequent itemset mining on data-flow predicate sets, and sub-sequence mining for control-flow conditions to derive preconditions. They reported a precision level of 77.13%.

Our approach has several key differences. First, it operates on a *very large-scale corpus* of client programs of the libraries that contain the call sites of APIs. In contrast, RGJ is designed to perform its inter-procedural analysis on only an *individual client program* containing the APIs' call sites. Thus, RGJ can be used to improve our analysis technique when running on each project. Second, their mining algorithm works on the data-flow predicate sets in an individual program, while our mining technique operates on the comparable preconditions across an ultra-large number of projects. In contrast, they find conditions using sophisticated data- and control-flow analyses on a single program. Their mining algorithm does not consider the predicates across projects.

Our work is also related to **static approaches** for mining specifications. Those static approaches *rely more on data mining*, while using *more light-weight static analyses* than our approach and RGJ. Gruska *et al.* [23] introduce the idea of wisdom of the crowds similar to our approach on 6,000 Linux projects (about 200MLOCs). However, their technique mines only temporal properties in term of pairs of method calls. They used 16 million mined temporal properties to check the anomalies in a new project. Our prior work, GrouMiner [42] performs frequent subgraph mining to find API programming patterns. JADET [52], Dynamine [34], Williams and Hollingsworth [55], CodeWeb [40] mine pairs of calls as patterns. MAPO [59, 3] expresses API patterns in term of partial orders of API calls. Tikanga [51] mines temporal specification in term of Computation Tree Logic formulas. Shoham *et al.* [47] use inter-procedural analysis to mine API specification in term of FSAs.

Other static approaches to mine API specifications and then leverage them to detect bugs [16, 28, 30, 31, 33]. FindBugs [25] looks for specified bug patterns. Tools suggest code examples related to specific APIs and types [38, 45, 49, 56]. *All above static approaches do not recover APIs' preconditions*.

There are several **dynamic approaches** in mining specifications [5, 12, 13, 17, 21, 33, 35, 43, 53, 58]. Daikon [17] automatically detects invariants in a program via running test cases. Wei *et al.* [53] infer complex post-conditions from simple programmer-written contracts in the code. Weimer *et al.* [54] mine method pairs from exception control paths and identify temporal safety rules. In brief, *our approach can complement well to dynamic approaches*.

There are other approaches that require annotations on partial specifications on desired invariants, and then verify program properties and detect violations [4, 19, 24]. Our approach is *automatic*.

Our work is also related to research to derive the behavior model of a program or software component for verification [14, 36, 37]. These approaches aim to recover the formal model for a program with pre/post-conditions of the states' transitions. *In contrast, our approach focuses at a more fine-grained level of individual APIs*.

6. CONCLUSIONS

In this paper, we propose a novel approach to mine the preconditions of API methods using a large code corpus. Our key idea is that the true API preconditions appear frequently in their usages from a large code corpus with large number of API usages, while project-specific conditions occur less frequently. We mined the preconditions for JDK methods on almost 120 million SLOCs on SourceForge and Apache projects. Comparing to the human-written preconditions in JML, our approach achieves high accuracy with recall from 75–80% and precision from 82–84% for the top-ranked results. In our user study, participants found 82% of the mined preconditions as a good starting point for writing specifications.

7. ACKNOWLEDGMENTS

Hoan Anh Nguyen and Tien N. Nguyen are funded in part by NSF grants CCF-10-18600, CNS-12-23828, CCF-1320578, CCF-1349153, and CCF-1320578. Hridesh Rajan and Robert Dyer are funded in part by NSF grants CCF-11-17937 and CCF-08-46059.

8. REFERENCES

[1] Code Contracts at Rise4Fun. http://rise4fun.com/CodeContracts.

[2] Java Path Finder (JPF). http://babelfish.arc.nasa.gov/trac/jpf.

[3] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34. ACM, 2007.

[4] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of interface specifications for java classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 98–109. ACM, 2005.

[5] G. Ammons, R. Bodík, and J. 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.

[6] Apache Software Foundation. http://apache.org.

[7] T. Ball and S. 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.

[8] I. Beschastnikh, Y. Brun, S. Schneider, M. Sloan, and M. D. Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 267–277. ACM, 2011.

[9] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Int. J. Softw. Tools Technol. Transf.*, 7(3):212–232, June 2005.

[10] R.-Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Trans. Softw. Eng.*, 34(5):579–596, 2008.

[11] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. 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.

[12] P. Cousot, R. Cousot, and F. 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.

[13] V. Dallmeier, C. Lindig, and A. 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.

[14] G. de Caso, V. Braberman, D. Garbervetsky, and S. Uchitel. Automated abstractions for contract validation. *IEEE Trans. Softw. Eng.*, 38(1):141–162, Jan. 2012.

[15] X. Deng, Robby, and J. 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.

[16] 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.

[17] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE'99, pages 213–224. ACM, 1999.

[18] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[19] J. Fischer, R. Jhala, and R. Majumdar. Joining dataflow with predicates. In *Proceedings of the 13th Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 227–236. ACM, 2005.

[20] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. 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. ACM, 2002.

[21] M. Gabel and Z. 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.

[22] P. Godefroid, N. Klarlund, and K. 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.

[23] N. Gruska, A. Wasylkowski, and A. 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.

[24] T. A. Henzinger, R. Jhala, and R. Majumdar. Permissive interfaces. In *Proceedings of the 13th Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 31–40. ACM, 2005.

[25] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.

[26] JML. Examples page. http://www.eecs.ucf.edu/~leavens/JML/examples.shtml, 2013.

[27] Jmol. http://sourceforge.net/projects/jmol/.

[28] T. Kremenek, P. Twohey, G. Back, A. Ng, and D. Engler. From uncertainty to belief: inferring the specification within. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 161–176. USENIX Association, 2006.

[29] G. T. Leavens. The Java Modeling Language (JML). http://www.eecs.ucf.edu/~leavens/JML.

[30] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE Trans. Softw. Eng.*, 32(3):176–192, 2006.

[31] Z. Li and Y. 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.

[32] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, Nov. 1994.

[33] C. Liu, E. Ye, and D. J. Richardson. Software library usage pattern extraction using a software model checker. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, ASE '06, pages 301–304. IEEE Computer Society, 2006.

[34] B. Livshits and T. Zimmermann. Dynamine: finding common error patterns by mining software revision histories. *SIGSOFT Softw. Eng. Notes*, 30(5):296–305, 2005.

[35] D. Lo and S. 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] D. Lo, L. Mariani, and M. Pezzè. Automatic steering of behavioral model inference. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 345–354. ACM, 2009.

[37] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 501–510. ACM, 2008.

[38] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proceedings of the 2005 conference on Programming language design and implementation*, PLDI '05, pages 48–61. ACM, 2005.

[39] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 2008 19th International Symposium on Software Reliability Engineering*, ISSRE '08, pages 117–126. IEEE Computer Society, 2008.

[40] A. 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.

[41] C. D. Nguyen, A. Marchetto, and P. 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.

[42] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392. ACM, 2009.

[43] M. Pradel and T. 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] 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.

[45] N. Sahavechaphan and K. Claypool. Xsnippet: Mining for sample code. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 413–430. ACM, 2006.

[46] SeMoA - Secure Mobile Agents. `http://sourceforge.net/projects/semoa/`.

[47] S. Shoham, E. Yahav, S. Fink, and M. Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 174–184. ACM, 2007.

[48] SourceForge.net. `http://sourceforge.net/`.

[49] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 204–213. ACM, 2007.

[50] S. Thummalapenta and T. Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 283–294. IEEE Computer Society, 2009.

[51] A. Wasylkowski and A. Zeller. Mining temporal specifications from object usage. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 295–306. IEEE Computer Society, 2009.

[52] A. Wasylkowski, A. Zeller, and C. Lindig. Detecting object usage anomalies. In *Proceedings of the Symposium on Foundations of Software Engineering*, ESEC-FSE '07, pages 35–44. ACM, 2007.

[53] Y. Wei, C. A. Furia, N. Kazmin, and B. Meyer. Inferring better contracts. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 191–200. ACM, 2011.

[54] W. Weimer and G. 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.

[55] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve bug finding techniques. *IEEE Trans. Softw. Eng.*, 31(6):466–480, 2005.

[56] T. Xie and J. Pei. Mapo: Mining api usages from open source repositories. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 54–57. ACM, 2006.

[57] Y. Xie and A. 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.

[58] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. 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.

[59] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, pages 318–343. Springer-Verlag, 2009.

[60] T. Zimmermann and P. Weißgerber. Preprocessing cvs data for fine-grained analysis. In *Proceedings of the First International Workshop on Mining Software Repositories*, pages 2–6, May 2004.