Abstract—When a method throws an exception—its exception precondition—is a crucial element of the method’s documentation that clients should know to properly use it. Unfortunately, exceptional behavior is often poorly documented, and sensitive to changes in a project’s implementation details that can be onerous to keep synchronized with the documentation.

We present WIT, an automated technique that extracts the exception preconditions of Java methods. WIT uses static analysis to analyze the paths in a method’s implementation that lead to throwing an exception. WIT’s analysis is precise, in that it only reports exception preconditions that are correct and correspond to feasible exceptional behavior. It is also lightweight: it only needs the source code of the class (or classes) to be analyzed—without building or running the whole project. To this end, its design uses heuristics that give up some completeness (WIT cannot infer all exception preconditions) in exchange for precision and ease of applicability.

We ran WIT on 46 Java projects, where it discovered 11,875 exception preconditions in 10,234 methods, taking just 1 second per method on average. A manual analysis of a significant sample of these exception preconditions confirmed that WIT is 100% precise, and demonstrated that it can accurately and automatically document the exceptional behavior of Java methods.

I. INTRODUCTION

To correctly use a method, we must know its precondition, which specifies the valid inputs: those that the method’s implementation can handle correctly. In programming languages like Java, a method’s implementation may throw an exception to signal that a call violates its precondition. If it does so, knowing the method’s exceptional behavior is equivalent to knowing (the complement of) its precondition. Ideally, a method’s exceptional behavior should be described in the method’s documentation (for example, in its Javadoc comments) and thoroughly tested. In practice, it is known that a method’s documentation can be incomplete or inconsistent with its implementation [24, 40], and that only a fraction of a project’s test suite exercises exceptional behavior [21]. This ultimately limits the usability, in a broad sense, of insufficiently documented methods: without precisely knowing its precondition, programmers may have a hard time calling a method; test-case generation may generate invalid tests that violate the method’s precondition; program analysis may have to explicitly follow the implementation of every called method, which does not scale since it is not modular.

To alleviate these problems, we present WIT (What Is Thrown?): a technique to automatically infer the exception preconditions—the input conditions under which an exception is thrown—of Java methods. As we discuss in Sec. VII, extracting preconditions and other kinds of specification from implementations is a broadly studied problem in software engineering (and, more generally, computer science). Our WIT approach is novel because it offers a distinct combination of features. First, WIT is precise: since it is based on static analysis, it reports preconditions only when it can determine with certainty that they are correct. It is also lightweight, as it is applicable to the source code of individual classes of a large project without requiring to build the project (or even to have access to all project dependencies).

A key assumption underlying WIT’s design is that a significant fraction of a method’s exceptional executions are usually simpler, shorter, and easier to identify than the other, normal, executions. Therefore, WIT’s analysis (which we describe in detail in Sec. III) relies on several heuristics that drastically limit the depth and complexity of the program paths it explores—for example, it bounds the length of paths and number of calls that it can follow. Whenever a heuristics fails, WIT gives up analyzing a certain path for exceptional behavior. In general, this limits the number of exception preconditions that WIT can reliably discover. However, if our underlying assumption holds, WIT can still be useful and effective, as well as lightweight and scalable.

We implemented WIT in a tool with the same name, which performs a lightweight static analysis of Java classes using JavaParser for parsing and the Z3 SMT solver for checking which program paths are feasible. Sec. IV describes an experimental evaluation where we applied WIT to 46 Java projects—including several widely used libraries—to discover the exception preconditions of their public methods. WIT inferred 11,875 exception preconditions of 10,234 methods—running for 1 second on average on each of the analyzed methods. A manual analysis of a significant random sample of the inferred preconditions confirmed that WIT is precise; all manually checked preconditions were correct. It also revealed that it could retrieve 7–83% of all supported exception preconditions in project Apache Commons IO—achieving even higher recall on projects that use few currently unsupported Java features. Our empirical evaluation also indicates that WIT can be useful to programmers: 72% of the exception preconditions in the sample are not already documented; and 5 pull requests—
extending the public documentation of open-source projects with a selection of WIT-inferred preconditions—were accepted by the projects’ maintainers.

In summary, the paper makes the following contributions:

- **WIT**: a technique to automatically infer the exception preconditions of Java methods based on a novel combination of static analysis and heuristics that trade-off exhaustiveness for high precision.
- An implementation of WIT and an experimental evaluation targeting 46 open-source Java projects (including popular ones like Apache Commons Lang, and the h2database), which demonstrates WIT’s effectiveness, practical applicability to real-world projects, and usefulness.
- For reproducibility, WIT’s implementation and the detailed experimental outputs are available.

II. SHOWCASE EXAMPLES OF USING WIT

We briefly present examples of applying WIT to detect the exception preconditions of library functions in two Apache projects: Dubbo’s class Bytes and Apache Commons Lang’s class NumberUtils. The examples showcase WIT’s capabilities and practical usefulness: WIT could automatically extract exception preconditions in many methods of these two projects, including some that were not documented (Sec. II-A) or incorrectly documented (Sec. II-B). Sec. V-E reports further empirical evidence that WIT’s exception preconditions can be useful as a source of documentation.

To better gauge WIT’s capabilities, let us stress that the two Apache projects discussed in this section are widely used Java libraries; for instance, Dubbo’s GitHub repository has over 24 thousand forks and 36 thousand stars. As a result, they are exceptionally well documented and tested. The fact that WIT could find some of their few missing or inconsistent pieces of their documentation indicates that it has the potential to be practically useful and widely applicable.

A. Missing Documentation

Listing 1: Excerpts of the implementation of two methods in Apache Dubbo’s class Bytes.

```java
1 public static String bytes2base64(byte[] b, char[] code) {
2  \{ return bytes2base64(b, 0, b.length, code); \}
3 }
4 public static String bytes2base64(final byte[] bs, final int off, final int len, final char[] code) {
5  if (off < 0) throw new IndexOutOfBoundsException();
6  if (len < 0) throw new IndexOutOfBoundsException();
7  if (off + len > bs.length) throw new IndexOutOfBoundsException();
8  if (code.length < 64) throw new IllegalArgumentException();
9  //...
10 }
```

argument values, the first’s precondition is a special case of the second’s. Unfortunately, the documentation of these methods does not mention these preconditions: for example, the second method’s Javadoc comment vaguely describes `off` and `len` as simply “offset” and “length”, without clarifying that they should be non-negative values. This lack of documentation about valid inputs decreases the usability of the methods for users of the library.

Running WIT on class Bytes automatically finds the preconditions of these (as well as many other) methods, thus providing a useful form of rigorous documentation. One of the exception preconditions found by WIT for ```min```'s second method:

```java
1 /** Returns the minimum value in an array.
2 * @param array an array, must not be null or empty
3 * @return the minimum value in the array
4 * @throws IllegalArgumentException if array is empty
5 * @throws IndexOutOfBoundsException when: \[off >= 0 \land \{b.length=0, code.length=0\}\]
6 */
7 public static int min(final int... array) {
8  \{ ValidateArray(array); \}/ ...
9 }
```

In fact, WIT only reports exception preconditions that correspond to feasible paths. Each precondition comes with an example of argument values that make the precondition true. These are not directly usable as test inputs, since they describe the input’s properties without constructing them; but they are useful complements to the precondition expressions, and help users get a concrete idea of the exceptional behavior.

B. Inconsistent Documentation

Listing 2: Excerpt of the Javadoc comment and implementation of a method in Apache Commons Lang’s class NumberUtils.

```java
1 /** Returns the minimum value in an array.
2 * @param array an array, must not be null or empty
3 * @return the minimum value in the array
4 * @throws IllegalArgumentException if array is empty
5 * @throws IndexOutOfBoundsException when: \[off >= 0 \land \{b.length=0, code.length=0\}\]
6 */
7 public static int min(final int... array) {
8  \{ ValidateArray(array); \}/ ...
9 }
```

This inconsistency is due to a change in the implementation of ```ValidateArray```, which is called by ```min``` to validate its input and uses methods of class Validate to perform the validation. In version 3.12.0 of the library, ```validateArray``` switches from calling ```Validate.isTrue(a != null)``` (which throws an ```IllegalArgumentException``` when the check
Listing 3: Excerpt of method ArrayUtils.insert in Apache Commons, and some of the methods it calls.

```java
public static boolean[] insert(final int k, final boolean[] a, final boolean... v) {
    if (a == null) { return null; }
    if (isEmpty(v)) { return clone(a); }
    if (k < 0 || k > a.length) {
        throw new IndexOutOfBoundsException();
    }
    // ...
}
```

fails to calling Validate.notNull(a) (which throws a NullPointerException instead) to check that a is not null.

To help locate the source of any exceptional behavior, WIT also outputs the line where the exception is thrown, and often the triggering method call. In this example, it would clearly indicate that the exceptional behavior comes from a call to Validate.notNull. This information can help detect and debug such inconsistencies, which can be quite valuable to project developers and users (see Sec. V-E).

III. HOW WIT WORKS

Fig. 1 provides a high-level overview of WIT’s analysis workflow. WIT inputs the source code of some Java classes; it analyzes the methods of those classes to determine their exception preconditions, that is, the conditions on the methods’ input that lead to the methods throwing an exception. It then outputs the exception preconditions it could find, together with their matching exception class, as well as examples of inputs that satisfy the exception preconditions. WIT’s analysis only needs the source code of the immediate classes to be analyzed: it does not need a complete project’s source code, nor to compile or build the project.

A. Parsing and CFG

WIT parses the source code given as input using library JavaParser and constructs a control-flow graph (CFG) of the methods in the input classes; it then inlines the call to other methods. More precisely, we build a CFG for each method m individually; and annotate branches with each branch’s Boolean condition.

Listing 4: Excerpt of the SMT encoding corresponding to global expath p1 of method insert.

```plaintext
# logic variables
k = Int('k')
a = Bool('a!=null'); a.length = Int('a.length')
c = [a.length >= 0, v.length >= 0] # implicit
c == [Not(a==null)] # a != null
x.null, x.length = v.null, v.length # call insert
y.null, y.length = x.null, x.length # call getlength
c == [y==null] # y == null
getLength = 0 # return 0
isEmpty = (getLength == 0) # return getLength(x)==0
c == [Not(isEmpty)] # isEmpty(v)
k < 0 || k > a.length # k < 0 || k > a.length
```

```plaintext
exception—either explicitly with a throw or indirectly with a call (which may return exceptionally).

In Lst. 3’s example, one of insert’s local expaths p goes through the else branch on lines 2–3 and through the then branch on line 4 ending with the throw on line 5:
```
p: i10 => i11, isEmpty(v), i12, k<0 || k>a.length, throw3
```

C. Global Exception Paths

After collecting expaths local to each method, WIT converts them into global expaths by inlining calls to other methods.

Given a local expath L from some node n in that calls another method x, WIT checks whether x’s CFG is available (that is, whether x’s implementation was part of the input). If it is, WIT enumerates all simple paths that go through the CFG of x, and splices each of them into L at n. In other words, it transforms the local path L so that it follows inter-method calls. Since a method usually has multiple paths, one local expath may determine several global expaths after inlining.

WIT inlines calls recursively (with some limits that we discuss in Sec. III-F). If a called method’s CFG is not available, WIT doesn’t inline calls to it and marks them as “opaque”.

WIT inlines the call to isEmpty in local expath p1 (Lst. 3’s example) since isEmpty is part of the same analyzed class and is a simple path that calls some other method m. WIT inlines replaces p’s edge i10 with getlength’s only path: i11 -> i12 -> i13

Since the implementation of getlength is available too, WIT recursively inlines its two paths, which finally gives two global expaths p1, p2 that inline insert’s local expath p’s calls:

```
p1: i10 => i11 => i12 => v==null, 0 != 0 => i13 => throw3
p2: i10 => i11 => i12 => v!=null, v.length != 0 => i13 => throw3
```

D. Path Feasibility

WIT builds global expaths only based on syntactic information in the CFGs; therefore, some paths may be infeasible (not executable). To determine whether a global expath is feasible, WIT encodes it as an SMT (Satisfiability Modulo Theory) formula [2], and uses the Z3 SMT solver [12] to determine whether the expath’s induced constraints are feasible.

To this end, it first transforms the path into SSA (static single assignment) form, where complex statements are broken down into simpler steps, and fresh variables store the
intermediate values of every expression. We designed a logic encoding of Java’s fundamental types (int, boolean, byte, arrays, strings) with their most common operations (including arithmetic, equality, length, contains, isEmpty), as well as of a few widely used JDK library methods (such as Array.getLength). WIT uses this encoding to build an SMT formula \( \phi \) corresponding to each global expath \( p \): if \( \phi \) is satisfiable, then the global expath \( p \) is feasible, and hence it corresponds to a possible exceptional behavior of method \( m \).

WIT encodes \( \phi \) as a Python program using the Z3 SMT solver’s Z3Py Python API. It parses the source code of the Java classes to be analyzed, and builds a control-flow graph (CFG) of every method. It enumerates the simple paths in every method’s CFG that may end with an exception (expaths). It then transforms these expaths local to a specific method into global expaths by inlining method calls; this may transform a single local expath into multiple global expaths. To determine which expaths are feasible, WIT encodes their constraints as an SMT problem and uses the Z3 SMT solver to check if they are satisfiable. It finally transforms all feasible paths into exception preconditions.

E. Exception Preconditions

A feasible path \( p \) identifies a range of inputs of the analyzed method \( m \) that trigger an exception. In order to characterize those input as an exception precondition, WIT encodes \( p \)’s constraints as a formula that only refers to \( m \)’s arguments, as well as to any members that are accessible at \( m \)’s entry (such as the target object this, if \( m \) is an instance method). To this end, it works backward from the last node of exception path \( p \); it collects all path constraints along \( p \), while replacing any reference to local variables with their definition. For example, a precondition includes calls to methods (as opposed to just their return values) when: (i) \( x \) is not null; (ii) \( x \)’s type is a string; (iii) \( x \)’s length is not available, any expressions including them may not make sense in a precondition. In all these cases, WIT still reports the exception expression obtained by backward substitution, but marks it as a maybe to indicate that it may not be correct. Another, more subtle case occurs when the exception precondition includes calls to methods (as opposed to just variable lookups). If these methods are not pure (that is, they do not change the program state), the precondition may be not well-formed. For instance, a precondition \( x\_inc() == 0 \) may not be correct. WIT is conservative and marks as maybe any exception precondition that involves calls to methods that are not known to be pure.

Sometimes WIT cannot build an exception precondition expression that only mentions arguments and other visible members. A common case is when a path includes opaque calls: since the semantics or implementation of these calls is not available, any expressions including them may not make sense in a precondition. In all these cases, WIT still reports the exception expression obtained by backward substitution, but marks it as a maybe to indicate that it may not be correct. Another, more subtle case occurs when the exception precondition includes calls to methods (as opposed to just variable lookups). If these methods are not pure (that is, they do not change the program state), the precondition may be not well-formed. For instance, a precondition \( x\_inc() == 0 \) may not be correct. WIT is conservative and marks as maybe any exception precondition that involves calls to methods that are not known to be pure.

Before outputting any exception preconditions to the user, WIT simplifies them to remove any redundancies and display them in a form that is easier to read. To this end, it uses SymPy [22] a Python library for symbolic mathematics. Java’s syntax is sufficiently similar to C’s that we can also enable SymPy’s pretty printing of expressions using C syntax, and then additionally tweak it to amend the remaining differences with Java. While conceptually simple, the simplification step is crucial to have readable exception preconditions. For example, SymPy simplifies the ugly expression \((1\_x==null)\&(\_x==null))\&(\_x+1==1)\&(y>0)\&(y\_x.length)\) into the much more readable \((y=x\_length)\) without losing any information.

WIT’s Z3 ad hoc encoding also handles aliasing by explicitly keeping track of possible aliases along each checked path. Thanks to the other heuristics that limit path length [Sec. III-F], this approach is feasible in practice.
(b) whether it is a maybe, (c) the thrown exception type, (d) and an example of inputs that satisfy the precondition (given by Z3’s successful satisfiability check). For debugging, wit can also optionally report the complete throw statement (including any exception message or other arguments used to instantiate the exception object), and the line in the analyzed method m where the exception is thrown or propagated.

F. Heuristics and Limitations

Let us now zoom in into a few details of how wit’s implementation works. To put these details into the right perspective, let us recall wit’s design goals: it should be precise and lightweight; it’s acceptable if achieving these qualities loses some generality—as long as a sizable fraction of exception preconditions can be precisely determined.

Implicit exceptions. wit only tracks exceptions that are explicitly raised by a throw statement; it does not consider low-level errors—such as division by zero, out-of-bound array access, and buffer overflow—that are signaled by exceptions raised by the JVM. This restriction is customary, in techniques that infer exceptional behavior, since implicitly thrown exceptions are “generally indicative of programming errors rather than design choices” [31] [4], and usually do not belong in API-level documentation [14]. Extending wit to also track implicit exceptions would not be technically difficult, but would produce a vast number of boilerplate exception preconditions that are not specific to a method’s behavior.

Java features. wit’s CFG construction currently does not fully support some Java features: for-each loops, switch statements, and try/catch blocks; and does not analyze the exceptional behavior of constructors. When these features are used, the CFG may omit some paths that exist in the actual program. (Supporting these features is possible in principle, but would substantially complicate the CFG construction.)

The SMT encoding used for path feasibility (Sec. III-D) is limited to a core subset of Java features and standard library methods. wit won’t report exception preconditions that involve unsupported features (or will report them as maybe, that is without correctness guarantee).

Path length. In large methods, even some local expaths can be too complex, which bogs down the whole analysis process. Therefore, wit only enumerates paths of up to \( N = 50 \) nodes, which have a much higher likelihood of being manageable.

Inlining limits. Inlining can easily lead to a combinatorial explosion in the expaths; therefore, a number of heuristics limit inlining. First, a path can be inlined only if it is up to \( N = 50 \) nodes—the same limit as for local expaths. Second, wit stops inlining a call in a path after it has reached a limit of \( I = 100 \) inlined paths—that is, it has branched out the call into \( I \) different ways. It can still inline other calls in the same path, but this limit avoids recursive inlinings that blow up. Third, wit enumerates the inlinings of a call in random order; in cases where the limit \( I \) is reached, this increases the chance of collecting a more varied set of inlined paths instead of getting stuck in some particularly complex ones (if the limit \( I \) is not reached, the enumeration order is immaterial).

Timeouts. Z3’s satisfiability checks (to determine if a path is feasible) may occasionally run for a long time. wit limits each call to Z3 to a 15-second timeout; when the timeout expires, Z3 is terminated and the path is assumed to be infeasible. There is also an overall timeout of 10 minutes per analyzed class. If wit’s analysis still runs after the timeout, to remain lightweight, wit skips to the next class.

Extensions. The parameters regulating these heuristics can be easily changed if one needs to analyze code with peculiar characteristics, when a large running time is not a problem.

IV. Experimental Evaluation

This section describes the empirical evaluation of wit, which targets the following research questions.

RQ1 (precision): How many of the exception preconditions detected by wit are correct?

RQ2 (recall): How many exception preconditions can wit detect?

RQ3 (features): What are the most common features of the exception preconditions detected by wit?

RQ4 (efficiency): Is wit scalable and lightweight?

RQ5 (usefulness): Are wit’s exception preconditions useful to complement programmer-written documentation?

A. Experimental Subjects

In our evaluation, we ran wit on 46 open-source Java projects surveyed by recent papers investigating the (mis)use of Java library APIs [22], [44], [16] and the automatic generation of tests for some of these libraries [23] (see Tab. I). Several of these projects are large, widely-used, mature Java projects in various domains (base libraries, GUI programming, security, databases)—especially the 26 projects from the Apache Software Foundation, which recent empirical research has shown to be extensively documented and thoroughly tested [44], [23]. On the other hand, a few projects taken from [16] are smaller, less used, or both. For instance, projects gae-java-mini-profiler, visualee, and AutomatedCar are no longer maintained. This minority of projects makes the selection more diverse, so that we will be able to evaluate wit’s capabilities in different scenarios.

B. Experimental Setup

We ran wit on the source code of all projects, after excluding directories that usually contain tests (e.g., src/test/) or other auxiliary code. All experiments ran on a Windows 10 Intel i9 laptop with 32GB of RAM. By default, wit only infers the exception preconditions of public methods; if a public method calls a non-public one, wit will also analyze the latter but report only public exception preconditions.

To answer RQ1 (precision), we performed a manual analysis of a sample of all exception preconditions reported by wit to determine if they correctly reflect the exceptional behavior of the implementation. One author tried to map each inferred
exception precondition to the source code of the analyzed method. In nearly all cases, the check was quick and its outcome clear. The few exception preconditions whose correctness was not obvious were analyzed by the other authors as well, and the final decision was reached by consensus. We were conservative in checking correctness: we only classified an exception precondition as correct if the evidence was clear and easy to assess.

To answer RQ2 (recall), we used Nassif et al. [24]'s dataset—henceforth, DSC—as ground truth. DSC includes 842 manually-collected exception preconditions [5] (expressed in structured natural language, e.g., “if offset is negative”) for all public methods in Apache Commons IO’s base package collected from all origins (package code, libraries, tests, documentation, …). We counted the exception preconditions inferred by WIT that are semantically equivalent to some in DSC. Matching DSC’s natural-language preconditions to WIT’s was generally straightforward, as we didn’t have to deal with subtle semantic ambiguities: since WIT is very precise, it only reports correct exception preconditions.

Using DSC as ground truth assesses WIT’s recall in a somewhat restricted context: (i) DSC targets exclusively the Commons IO project, whose extensive usage of I/O operations complicates (any) static analysis; (ii) DSC describes all sorts of exceptional behavior, including the “not typically documented” runtime exceptions [24]. To assess WIT’s recall on a more varied collection of projects, we also considered Zhong et al. [44]'s dataset—henceforth, DPA—which includes 503 so-called “parameter rules” of public methods in 9 projects (a subset of our 46 projects described in Sec. IV-A). A parameter rule is a pair \((m, p)\), where \(m\) is a fully-qualified method name and \(p\) is one of \(m\)’s arguments; it denotes that calling \(m\) with some values of \(p\) may throw an exception. Importantly, parameter rules do not express the values of \(p\) that determine an exception, and hence they are much less expressive than preconditions; however, they are still useful to determine “how much” exceptional behavior WIT captures. We counted the exception preconditions inferred by WIT that match DPA: a precondition \(c\) matches a parameter rule \((m, p)\) if \(c\) is an exception precondition of method \(m\) that depends on the value of \(p\). This is a much weaker correspondence than for DSC, but it’s all the information we can extract from DPA.

To better characterize the exception preconditions that WIT could not infer, we performed an additional manual analysis of: (a) 679 of DSC’s exception preconditions among those that WIT did not infer, (b) 118 throw statements among those that WIT could not capture in each of the other projects, and (c) 185 exception preconditions reported by WIT as “maybe” (that is, which may be incorrect). These 982 cases help assess what it would take to improve WIT’s recall.

To answer RQ3 (features), during the manual analysis of precision we also classified the basic features of each exception precondition \(r\) of a method \(m\). We determine whether \(r\) corresponds to an exception that is thrown directly by \(m\) or propagated by \(m\) (and thrown by a called method). We count the number of Boolean connectives \(||\) and \&\& in \(r\), which gives an idea of \(r\)’s complexity. Then, we determine if each subexpression \(e\) of \(r\) constraints \(m\)’s arguments, or \(m\)’s object state; and we classify \(r\)’s check according to whether it is: (a) a null check (whether a value is null), (b) a value check (whether a value is in a certain set of values), (c) a query check (whether a function call returns certain values). For example, here are expressions of each kind for a method \(m\) with arguments \(\text{int} x\) and \(\text{String} y\), whose class includes fields \(\text{int}[]\) a, \(\text{int} count\), and method \(\text{boolean} active()\):

<table>
<thead>
<tr>
<th>void m((int x, int[] y)</th>
<th>argument</th>
<th>state</th>
</tr>
</thead>
<tbody>
<tr>
<td>null</td>
<td>y == null</td>
<td>this.a != null</td>
</tr>
<tr>
<td>value</td>
<td>x == 1</td>
<td>this.count &gt; 0</td>
</tr>
<tr>
<td>query</td>
<td>y.isEmpty()</td>
<td>!this.active()</td>
</tr>
</tbody>
</table>

To answer RQ5 (usefulness), we first inspected the source code documentation (Javadoc and comments) of all methods with exception preconditions analyzed to answer RQ1, looking for mentions of the thrown exception types and of the conditions under which they are thrown. We also selected 90 inferred exception preconditions among those were not already documented, and submitted them as 8 pull requests in 5 projects: Accumulo [22], Commons Lang [13, 14, 15], Commons Math [16, 17], Commons Text [11] and Commons IO [19]. We selected these 5 projects as they are very active and routinely spend effort in maintaining a good-quality documentation. Each pull request combines the exception preconditions of methods in the same class or package, and expresses WIT’s exception preconditions using Javadoc @throws tags.

V. EXPERIMENTAL RESULTS

As described in Sec. III-E, WIT produces two kinds of exception preconditions. The main output are those whose feasibility was fully checked (Sec. III-D); others are marked as maybe and can still be correct but have no guarantee. In this section, we call “express” the former and “maybes” the latter. The experimental evaluation focuses on the former kind: “exception precondition” (without qualifiers) means “expr”.

A. RQ1: Precision

Overall, WIT reported 11,875 expres and 20,989 maybes in 17,688 methods (10,234 and 8,391 respectively)—out of a total of 388,000 analyzed public methods from 57,000 classes in 46 projects. As shown in Tab. I, WIT detected some preconditions in 44 of these projects.

We manually analyzed a sample of 390 expres to determine if they are indeed correct. This sample size is sufficient to estimate precision with up to 5% error and 95% probability with the most conservative (i.e., 50%) a priori assumption [11]: thus, it gives our estimate good confidence without requiring an exhaustive manual analysis [46, 24]. We applied stratified sampling to pick the 390 expres: we randomly sampled 10 instances in each of the 44 projects where WIT detected some expres [1]. This manual analysis found that all expres were indeed correct, that is 100% precision.

We exclude 6 inaccurate cases.

We pick all expres for 7 projects with less than 10 in total.
TABLE I: Exception preconditions inferred by WIT. For each analyzed PROJECT: the size of the analyzed source code in thousands of lines (KLOC); WIT’s total running TIME in minutes; the number # of inferred exception preconditions (EXPRES), the number M of methods with some inferred exception preconditions, the resulting precision P, the number #? of MAYBE exception preconditions, and the percentage ?P of those that are correct.

<table>
<thead>
<tr>
<th>PROJECT</th>
<th>KLOC</th>
<th>TIME</th>
<th>EXPRES</th>
<th>M</th>
<th>P</th>
<th>#?</th>
<th>?P</th>
</tr>
</thead>
<tbody>
<tr>
<td>accumulo</td>
<td>33</td>
<td>124</td>
<td>325</td>
<td>309</td>
<td>1.0</td>
<td>941</td>
<td>0.5</td>
</tr>
<tr>
<td>Activiti</td>
<td>103</td>
<td>152</td>
<td>374</td>
<td>338</td>
<td>1.0</td>
<td>154</td>
<td>0.4</td>
</tr>
<tr>
<td>asm</td>
<td>28</td>
<td>136</td>
<td>110</td>
<td>81</td>
<td>1.0</td>
<td>400</td>
<td>0.2</td>
</tr>
<tr>
<td>asterisk-java</td>
<td>30</td>
<td>5</td>
<td>16</td>
<td>14</td>
<td>1.0</td>
<td>9</td>
<td>0.4</td>
</tr>
<tr>
<td>AutomatedCar</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>Baragon</td>
<td>15</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>12</td>
<td>2</td>
<td>1.0</td>
</tr>
<tr>
<td>bigtop</td>
<td>6.5</td>
<td>0</td>
<td>1</td>
<td>1.0</td>
<td>4</td>
<td>1</td>
<td>1.0</td>
</tr>
<tr>
<td>byte-buddy</td>
<td>57</td>
<td>12</td>
<td>170</td>
<td>164</td>
<td>1.0</td>
<td>214</td>
<td>0.8</td>
</tr>
<tr>
<td>camel</td>
<td>972</td>
<td>1962</td>
<td>663</td>
<td>600</td>
<td>1.0</td>
<td>778</td>
<td>0.5</td>
</tr>
<tr>
<td>closure-compiler</td>
<td>287</td>
<td>66</td>
<td>46</td>
<td>45</td>
<td>1.0</td>
<td>62</td>
<td>0.6</td>
</tr>
<tr>
<td>commons-bcel</td>
<td>35</td>
<td>12</td>
<td>32</td>
<td>31</td>
<td>1.0</td>
<td>284</td>
<td>1.0</td>
</tr>
<tr>
<td>commons-configuration</td>
<td>20</td>
<td>10</td>
<td>73</td>
<td>60</td>
<td>1.0</td>
<td>55</td>
<td>0.8</td>
</tr>
<tr>
<td>commons-io</td>
<td>9.5</td>
<td>16</td>
<td>106</td>
<td>78</td>
<td>1.0</td>
<td>183</td>
<td>0.9</td>
</tr>
<tr>
<td>commons-lang</td>
<td>29</td>
<td>14</td>
<td>435</td>
<td>354</td>
<td>1.0</td>
<td>245</td>
<td>0.7</td>
</tr>
<tr>
<td>commons-math</td>
<td>61</td>
<td>9</td>
<td>627</td>
<td>357</td>
<td>1.0</td>
<td>627</td>
<td>0.8</td>
</tr>
<tr>
<td>commons-text</td>
<td>10</td>
<td>9</td>
<td>116</td>
<td>82</td>
<td>1.0</td>
<td>212</td>
<td>0.6</td>
</tr>
<tr>
<td>confucius</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>curator</td>
<td>26</td>
<td>5</td>
<td>18</td>
<td>13</td>
<td>1.0</td>
<td>40</td>
<td>0.6</td>
</tr>
<tr>
<td>dubbo</td>
<td>99</td>
<td>102</td>
<td>202</td>
<td>167</td>
<td>1.0</td>
<td>141</td>
<td>0.6</td>
</tr>
<tr>
<td>flink</td>
<td>568</td>
<td>688</td>
<td>1996</td>
<td>1634</td>
<td>1.0</td>
<td>3992</td>
<td>0.7</td>
</tr>
<tr>
<td>gae-java-mini-profiler</td>
<td>0.5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>h2database</td>
<td>150</td>
<td>178</td>
<td>355</td>
<td>350</td>
<td>1.0</td>
<td>1104</td>
<td>0.6</td>
</tr>
<tr>
<td>hhttpcomponents-client</td>
<td>32</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>1.0</td>
<td>9</td>
<td>0.8</td>
</tr>
<tr>
<td>itext7</td>
<td>145</td>
<td>313</td>
<td>356</td>
<td>2325</td>
<td>1.0</td>
<td>1239</td>
<td>0.8</td>
</tr>
<tr>
<td>jackrabbit</td>
<td>260</td>
<td>66</td>
<td>465</td>
<td>446</td>
<td>1.0</td>
<td>481</td>
<td>0.9</td>
</tr>
<tr>
<td>jackrabbit-oak</td>
<td>26</td>
<td>111</td>
<td>175</td>
<td>175</td>
<td>1.0</td>
<td>460</td>
<td>0.8</td>
</tr>
<tr>
<td>jackson-databind</td>
<td>63</td>
<td>26</td>
<td>82</td>
<td>81</td>
<td>1.0</td>
<td>138</td>
<td>1.0</td>
</tr>
<tr>
<td>jfreechart</td>
<td>84</td>
<td>50</td>
<td>846</td>
<td>780</td>
<td>1.0</td>
<td>470</td>
<td>1.0</td>
</tr>
<tr>
<td>jmonkeyengine</td>
<td>19</td>
<td>279</td>
<td>384</td>
<td>354</td>
<td>1.0</td>
<td>832</td>
<td>0.9</td>
</tr>
<tr>
<td>joda-time</td>
<td>29</td>
<td>33</td>
<td>140</td>
<td>130</td>
<td>1.0</td>
<td>255</td>
<td>0.8</td>
</tr>
<tr>
<td>logging-log4j2</td>
<td>99</td>
<td>52</td>
<td>166</td>
<td>93</td>
<td>1.0</td>
<td>106</td>
<td>0.6</td>
</tr>
<tr>
<td>lucene-solr</td>
<td>685</td>
<td>721</td>
<td>1627</td>
<td>1406</td>
<td>1.0</td>
<td>2692</td>
<td>0.6</td>
</tr>
<tr>
<td>pdbbox</td>
<td>106</td>
<td>139</td>
<td>202</td>
<td>189</td>
<td>1.0</td>
<td>218</td>
<td>0.6</td>
</tr>
<tr>
<td>poi</td>
<td>260</td>
<td>448</td>
<td>387</td>
<td>352</td>
<td>1.0</td>
<td>2003</td>
<td>0.2</td>
</tr>
<tr>
<td>santuario-xml-security-jaava</td>
<td>35</td>
<td>8</td>
<td>78</td>
<td>73</td>
<td>1.0</td>
<td>59</td>
<td>0.8</td>
</tr>
<tr>
<td>shiro</td>
<td>27</td>
<td>10</td>
<td>70</td>
<td>63</td>
<td>1.0</td>
<td>101</td>
<td>0.4</td>
</tr>
<tr>
<td>spoon</td>
<td>75</td>
<td>38</td>
<td>213</td>
<td>207</td>
<td>1.0</td>
<td>187</td>
<td>0.2</td>
</tr>
<tr>
<td>spring-cloud-gcp</td>
<td>20</td>
<td>5</td>
<td>7</td>
<td>7</td>
<td>1.0</td>
<td>4.7</td>
<td>0.7</td>
</tr>
<tr>
<td>spring-data-commons</td>
<td>28</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td>1.0</td>
<td>105</td>
<td>0.5</td>
</tr>
<tr>
<td>swingx</td>
<td>72</td>
<td>47</td>
<td>86</td>
<td>85</td>
<td>1.0</td>
<td>99</td>
<td>0.8</td>
</tr>
<tr>
<td>tracecar</td>
<td>54</td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>1.0</td>
<td>10.6</td>
<td></td>
</tr>
<tr>
<td>visualae</td>
<td>1.8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>6</td>
<td>0.0</td>
</tr>
<tr>
<td>weibullocolatej4</td>
<td>7.8</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1.0</td>
<td>0</td>
<td>0.0</td>
</tr>
<tr>
<td>wicket</td>
<td>109</td>
<td>129</td>
<td>453</td>
<td>386</td>
<td>1.0</td>
<td>463</td>
<td>0.8</td>
</tr>
<tr>
<td>wildly-elytron</td>
<td>80</td>
<td>33</td>
<td>190</td>
<td>175</td>
<td>1.0</td>
<td>227</td>
<td>0.6</td>
</tr>
<tr>
<td>xmlgraphics-fop</td>
<td>165</td>
<td>99</td>
<td>258</td>
<td>210</td>
<td>1.0</td>
<td>1357</td>
<td>0.4</td>
</tr>
<tr>
<td>overall</td>
<td>5,720</td>
<td>6,412</td>
<td>11,875</td>
<td>10,234</td>
<td>1.0</td>
<td>20,989</td>
<td>0.7</td>
</tr>
</tbody>
</table>

B. RQ2: Recall

Out of DSc’s 842 manually identified exception preconditions, WIT detected 61 expres in 6 classes of Commons IO (1 in FilenameUtils, 1 in LineIterator, 8 in FileCleaningTracker, 17 in IOTests, 31 in FileUtilizes, and 3 in HexDump), that is a recall of 7% (61/842). However, 678 out of DSc’s 842 exception preconditions are of kinds unsupported by WIT (see Sec. III-F). After excluding unsupported kinds (WIT’s recall is 37% (61/(842 − 678)).

Analysis of missed expres. To better understand WIT’s recall, we manually analyzed 781 (842 − 61) Commons IO exception preconditions from DSc that WIT didn’t report as expres. We can classify these missed preconditions in 3 groups.

1) Unsupported features: As mentioned, the largest group of missed preconditions (449 or 66% of the analyzed sample) involve Java language features that WIT does not support.

2) Implicit exceptions: Another group of missed preconditions (138 or 20% of the analyzed sample) correspond to implicit exceptions that are thrown by the Java runtime (e.g., when a null pointer is dereferenced), which we deliberately ignore (as discussed in Sec. III-F).

3) Maybes: exception preconditions in the third group (91 or 14% of the analyzed sample) refer to opaque methods (Sec. III-C) whose implementation is not available to WIT. When opaque methods are involved, WIT simply doesn’t have the information needed to conclude that these features determine a feasible precondition. Nevertheless, it doesn’t completely give up; it often still reports as “maybe” a “plausible” exception precondition that may be correct. Indeed, 75 of the 91 manually analyzed exception preconditions that WIT detected as maybes were correct. WIT’s recall on Commons IO becomes 83% ((75 + 61)/(842 − 678)) if we also count all (correct) maybes (with 87% precision, see Sec. V-A).

As a concrete example, Listing 5 shows an excerpt of two methods in Commons IO’s class FilenameUtils. WIT reports most of their exception preconditions as maybes. For example,

<table>
<thead>
<tr>
<th>Listing 5: Excerpt from class FilenameUtils in project Commons IO.</th>
</tr>
</thead>
<tbody>
<tr>
<td>static void copyDirToDir(File src, File destDir) {</td>
</tr>
<tr>
<td>if (src == null) { throw new NullPointerException(); }</td>
</tr>
<tr>
<td>if (src.isDirectory()) { copyDirToDir(src, destDir); }</td>
</tr>
<tr>
<td>else if (src.isFile()) { copyFileToDir(src, destDir); }</td>
</tr>
<tr>
<td>else { throw new IOException(&quot;Source does not exist&quot;); }</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

Manually analyzing a significant sample of exception preconditions confirmed that WIT is 100% precise.
the precondition leading to the throw on line 5 requires !src.isDirectory() and !src.isFile()—two calls to methods of Java system class File. WIT doesn’t know whether these two methods have any side effects, whether their return values are somehow related, and whether calling them may throw an exception. However, it still reports the correct condition as maybe; the user can take it as a suggestion that still requires manual validation but is grounded in the analysis of the methods’ control flow. Even in cases where Z3’s feasibility check cannot be done, WIT’s simplification step can still prune out logically inconsistent conditions and avoid reporting them even as maybes. In Lst. 5, the exception thrown by copyDirToDir at line 11 is infeasible from copyToDir, since it requires both src.isDirectory() (line 3) and !src.isDirectory() (line 10): thus WIT doesn’t report anything.

**Dataset DPA.** Using [24]’s DPA dataset of 503 parameter rules as reference suggests that WIT’s recall varies considerably depending on the characteristics of the analyzed project. Overall, WIT inferred 104 matching expres and 61 matching maybes, corresponding to a recall of 21% (expres only) and 33% (expres+maybes). If we exclude the parameter rules involving features unsupported by WIT, the recall becomes 49% (expres only) and 77% (expres+maybes). WIT struggles the most on projects like asm, which extensively uses features and coding patterns that WIT currently doesn’t adequately support: as a result, WIT’s recall is fairly low (considering all parameter rules, 4% with expres only and 15% with expres+maybes; considering only supported ones, 17%/67%). In contrast, more “traditional” Java projects like JFreeChart extensively follow programming practices such as validating a method’s input, which are a better match to WIT’s current capabilities: as a result, WIT’s recall is quite high (considering all parameter rules, 50% with expres only and 61% with expres+maybes; considering only supported ones, 92%/96%).

### C. RQ3: Features

WIT inferred 7–83% of the exception preconditions in Commons 10. Its recall varies considerably depending on the analyzed project’s characteristics.

#### Sec. V-B

The comparison of WIT’s preconditions with those in DSC [24]’s extensive collection confirmed what also reported by other empirical studies [3, 46]: exception preconditions are often concise and structurally simple. This was also reflected in our manual sample of 390 expres inferred by WIT. In terms of size, 69% of them are simple expressions without Boolean connectives &|/; and only 10% include more than one connective. In terms of control-flow complexity, 68% of WIT’s expres involve exceptions that are thrown directly by the analyzed method (as opposed to propagated from a call). These measures of complexity are very similar for maybes, which indicates again that it’s not their intrinsic complexity but the lack of information (for example: purity) that prevents WIT from confirming them as correct.

Over 97% of all expres constrain a method’s arguments (72% constraint only the arguments), whereas about 32% predicate over object state. Value checks are most frequent (60% of expres), followed by null checks (50% of expres); and 97% of expres have either or both. Query checks are considerably less frequent (18% of expres include one). If we look at maybes, they tend to include query checks more frequently (35%), which is to be expected since a method call can be soundly used in a precondition only when it is provably pure (Sec. III-E).

Up to 12% of the expres in the sample are the simplest possible Boolean expression: true. All of the 6 expres of spring-cloud-gcp are of this kind. These usually correspond to methods that unconditionally throw an UnsupportedOperation exception to signal that they are effectively not available; see project lucene-solr’s class ResultSetImpl for an example [21]. In Java, this is a common idiom to provide “placeholders,” which will be replaced by actual implementations through overriding in subclasses. While this is a common programming pattern that leverages polymorphism, it nominally breaks behavioral substitutability [19]. Some of the exception preconditions that we manually inspected revealed interesting and non-trivial features. WIT could infer expres embedded in complex expressions, such as in the case of an empty string that triggers an exception in the “else” part of a ternary expression. It also followed method calls collecting complex conditions and presenting them in a readable, simplified form. For example, for a ConcurrentHashMap exception or after collecting constant values from other classes. In all, WIT’s output is often concise and to the point—and thus readable and useful.

#### D. RQ4: Efficiency

Thanks to the heuristics it employs (Sec. III-F) and to the nature of exception preconditions (which tend to be simpler compared to general program behavior), WIT’s analysis is quite lightweight and scalable. As shown in Tab. 1 its running times are generally short: it processed the entire Apache Commons Lang in just 14 minutes—4.2 seconds on average for each of the project’s 200 top-level classes. It also scales well to very large projects: it analyzed the 9 780 classes of Apache Camel (the largest project in our collection) in 33 hours—just 12 seconds per class on average. Key to this performance is WIT’s capability of analyzing each class in isolation, without requiring any compilation or build of the whole project.

**WIT’s analysis is lightweight: on average, it takes 7 seconds per class; 32 seconds per exception precondition.**

#### E. RQ5: Usefulness

Out of all 518 expres and maybes that WIT correctly inferred, 72% (372) are not documented; precisely, 283 of them belong
We expect that pull requests, from their documentation, and promptly added following our NullPointerException impact on scarcely documented projects.

Automatically inferring preconditions and other specification elements from implementations is a long-standing problem in computer science, which has been tackled with a variety of different approaches. Historically, the first approaches used static analysis and thus were typically sound (the inferred specification is guaranteed to be correct, that is 100% precision) but incomplete (not all specifications can be inferred, that is low recall), and may not be applicable to all features of a realistic programming language. Daikon was the first, widely successful approach that used dynamic analysis, which offers a different trade-off: it is unsound (the "inferred" specifications are only "likely" to be correct) but it is applicable to any program that can be executed. Daikon approach’s practicality also yielded a lot of follow-up work. wit is fundamentally based on static analysis, which can be very precise but incomplete; its heuristics further make it lightweight, and hence applicable to real-world Java projects.

More recently, approaches based on natural language processing (NLP) have gained traction. A clear advantage of NLP is that it can analyze artifacts other than program code (e.g., comments and other documentation); on the other hand, machine learning is usually based on statistical models, and hence it cannot guarantee correctness and may be subject to overfitting.

Like the “classic” work on static assertion inference, wit extracts preconditions by directly analyzing the behavior of a method’s implementation. An alternative, complementary approach is extracting assertions indirectly by analyzing the clients of a method the
patterns used by many clients of the same API are likely to indicate suitable ways of using that API’s methods [31].

**Exception Preconditions.** Buse and Weimer’s work [4]—which is a refinement of Jex [32]—shares several high-level similarities with wit: it specifically targets the documentation of exceptional behavior, uses static analysis, and can often improve or complement human-written documentation. Nevertheless, ours and their approach differ in several important characteristics: (a) their approach works on instrumented bytecode, which requires a full compilation of a project to be analyzed (wit only needs the source code of the class to be analyzed); (b) they do not exhaustively check path satisfiability or that only pure method expressions are used in expressions, and hence they may report exception preconditions that are not valid; (c) their evaluation is solely based on a qualitative comparison with human-written documentation, whereas wit’s evaluation quantitatively estimates precision and recall.

SnuggleBug [5] is a technique to infer weakest preconditions that characterize the reachability of a goal state from an entry location. Like wit, SnuggleBug is sound and scales to real-world Java projects (even though it works on bytecode and hence requires full project compilation). SnuggleBug’s analysis is more general than wit’s, as it is not limited to exception preconditions, and handles calls (including recursion) by synthesizing over-approximated procedure summaries instead of inlining. This approach achieves a different trade-off than wit, which more aggressively gives up on long paths or complex, unsupported language features. SnuggleBug’s evaluation demonstrates one of its main usage scenarios: validating implicit exception warnings.

PreInfer [1] infers preconditions of C# programs using symbolic execution (through the Pex white-box test-case generator) by summarizing a set of failing tests’ paths. Compared to wit, PreInfer explores a different part of the assertion inference design space: where wit aims to infer simple preconditions with high precision and scalability, PreInfer focuses on complex preconditions that involve disjunctive and quantified formulas over arrays. These differences in aim are also reflected by the different experimental evaluations: we applied wit to 388 000 methods in 57 000 classes over 46 projects of diverse characteristics, where it inferred 11 875 preconditions; PreInfer’s evaluation targets 1 143 methods in 147 classes over 4 projects mainly consisting of algorithm and data structure implementations, where it inferred 178 preconditions. Since it relies on Pex, PreInfer’s inferred predicates are only “likely perfect because Pex may not explore all execution paths” [1].

A direct, quantitative comparison with these approaches [4], [5], [1] is not possible, since their implementations or experimental artifacts are not publicly available.

**Exceptional Behavior Documentation.** Other recent work uses static analysis to extract API specification with a focus on extending and completing programmer-written documentation. PaRu [44] is an automated technique that analyzes source code and Javadoc documentation to link method parameters to exceptional behavior. PaRu’s goal is to “identify as many parameter rules as possible [...] it does not comprehend or interpret any rule” [44]; hence, unlike wit, PaRu does not infer preconditions but just a mapping between parameters and the throw statements that depend on them.

Drone [46] compares the exceptional behavior of source code to that described in Javadoc in order to find inconsistencies. Similarly to wit, Drone analyzes a program’s control flow statically and uses constraint solving (i.e., Z3)—but to find inconsistencies rather than to analyze feasibility. wit and Drone also differ in some of the Java features they support; for example, Drone keeps track of try/catch blocks (wit misses some paths) but does not follow calls inside conditionals (wit fully supports them). The several differences between wit’s and Drone’s capabilities reflect their different goals (and, correspondingly, the different research questions of their respective evaluations): Drone aims at finding inconsistencies in whole projects, whereas wit infers preconditions with high precision and nimbly on individual classes. As a result, Drone is run on projects with some existing documentation to improve and extend it: the tool “takes API code and document directives as inputs, and outputs repair recommendations for directive defects” [46] [§3]; wit can run on projects without documentation and reliably find exception preconditions (Sec. V-E showed that 72% of the manually analyzed exception preconditions found by wit are completely undocumented).

DScribe [24] generates unit tests and documentation from manually written templates, which helps keep them consistent. An extensive manual analysis of the exceptional behavior of Apache Commons IO—which we used as ground truth in Sec. V-B’s experiments—found that 85% of exception-throwing methods are not documented, not tested, or both, which motivated their template-based approach. wit’s output could be used to write the templates, thus improving the automation in DSB. The other key feature (that it’s lightweight) would be beneficial in different scenarios. For research in mining software repositories, not requiring complete project builds enables scaling analyses to a very large number (e.g., several thousands) of projects—whereas building all of them would be infeasible. Using wit as a component of a recommender system that runs in real-time is another scenario where speed/scalability would be of the essence.
Acknowledgements. Work partially supported by SNF grant 200021-182060 (Hi-Fi).

REFERENCES


