What Is Thrown? Lightweight Precise Automatic Extraction of Exception Preconditions in Java Methods

Diego Marcilio
dvmarcilio.github.io

Carlo A. Furia
bugcounting.net

Software Institute – USI Università della Svizzera italiana – Lugano, Switzerland

***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], [46], 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—

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

```
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 }
```

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.[a]

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`[2] and `Commons Lang`.[3] 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[4] has over 24 thousand forks and 36 thousand stars. As a result, they are exceptionally well documented and tested [44], [24]. 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

Lst. 1 shows an excerpt of two overloaded implementations of method `bytes2base64`, which takes a byte array and represents it as a string in base 64. As we can see from the initial lines in `bytes2base64`'s second implementation, the two methods have fairly detailed preconditions; furthermore, since the first method calls the second with additional fixed 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. For instance, one of the exception preconditions found by WIT for Lst. 1's second method:

throws: `IndexOutOfBoundException`
when: `off >= 0 && len >= 0 && bs.length < len + off`
example: `[off=0, len=1, bs.length=0]`

corresponds to the path that reaches line 7 in Lst. 1. WIT also understands that the first method never throws this exception, but it can still throw others such as:

throws: `IllegalArgumentException`
when: `b.length >= 0 && code.length < 64`
example: `[b.length=0, code.length=0]`

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.

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

```
/** Returns the minimum value in an array.
* @param array an array, must not be null or empty
* @return      the minimum value in the array
* @throws      IllegalArgumentException if array is null
* @throws      IllegalArgumentException if array is empty */
public static int min(final int... array) {
{ validateArray(array); /* ... */ }
```

B. Inconsistent Documentation

Lst. 2 shows the complete Javadoc documentation and a brief excerpt of method `min` in the latest version of `Apache Commons Lang`'s class `NumberUtils`, which computes the minimum of an `array` of integers. Unlike the previous example, `min`'s documentation is detailed and clearly expresses the conditions under which an exception is thrown. Unfortunately, the documentation is partially incorrect: when `array` is null, `min` throws a `NullPointerException`, not an `IllegalArgumentException`, as precisely reported by WIT:

throws: `NullPointerException` *when:* `array == null`

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` switched[5] from calling `Validate.isTrue(a != null)` (which throws an `IllegalArgumentException` when the check

[a]A replication package is available.[1]

Listing 3: Excerpt of method `ArrayUtils.insert` in `Apache Commons`, and some of the methods it calls.

```
1 public static boolean[] insert(final int k, final boolean[] a
      , final boolean... v) {
2   if (a == null) { return null; }
3   if (isEmpty(v)) { return clone(a); }
4   if (k < 0 || k > a.length)
5       { throw new IndexOutOfBoundsException(); }
6   // ...
7 }
8
9 public static boolean isEmpty(boolean[] x)
10 { return getLength(x) == 0; }
11
12 public static int getLength(boolean[] y)
13 {  if (y == null) { return 0; } return y.length; }
```

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 overviews how WIT's analysis works. 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 JavaParser,[6] and constructs a control-flow graph (CFG) of the methods in the input classes using library `JGraphT`.[7] More precisely, we build a CFG for each method `m` individually; and annotate branches with each branch's Boolean condition.

Lst. 3 shows excerpts of 3 methods of class `ArrayUtils`[8] in `Apache Commons Lang`. Method `insert` puts some values `v` into an array `a` of Booleans at a given index `k`. The initial part of its implementation calls another method `isEmpty` of the same class to determine if `v` is empty; in turn, `isEmpty` calls method `getLength`. WIT builds CFGs for `insert`, `isEmpty`, and `getLength`, since they are all part of the input source code.

B. Local Exception Paths

When analyzing a method `m`, WIT collects its *local exception paths* ("expaths" for short). These are all simple directed paths[b] on `m`'s CFG that end with a node that may throw an 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:

$$p\colon \mathbf{if}_2 \xrightarrow{\texttt{a!=}\mathbf{null}} \mathbf{if}_3 \xrightarrow{\texttt{!isEmpty(v)}} \mathbf{if}_4 \xrightarrow{\texttt{k<0 || k>a.length}} \mathbf{throw}_5$$

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, for each node $n_{\texttt{x}}$ in ℓ that calls some other 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 $n_{\texttt{x}}$. 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 p (Lst. 3's example) since `isEmpty` is part of the same analyzed class `ArrayUtils`. Inlining replaces p's edge $\mathbf{if}_3 \xrightarrow{\texttt{!isEmpty(v)}} \mathbf{if}_4$ with `getLength`'s only path: $\mathbf{if}_3 \xrightarrow{\texttt{!(getLength(v)==0)}} \mathbf{if}_4$. Since the implementation of `getLength` is available too, WIT recursively inlines its two paths, which finally gives two global expaths p_1, p_2 that inline `insert`'s local expath p's calls:

$$p_1\colon \mathbf{if}_2 \to \mathbf{if}_3 \to \mathbf{if}_{13} \xrightarrow{\texttt{v==}\mathbf{null}\texttt{, 0 != 0}} \mathbf{if}_4 \to \mathbf{throw}_5$$

$$p_2\colon \mathbf{if}_2 \to \mathbf{if}_3 \to \mathbf{if}_{13} \xrightarrow{\texttt{v!=}\mathbf{null}\texttt{,v.length != 0}} \mathbf{if}_4 \to \mathbf{throw}_5$$

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

Listing 4: Excerpt of the SMT encoding corresponding to global expath p_1 of method `insert` in Lst. 2.

```
# logic variables
k = Int('k')
a_null = 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 isEmpty
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)
c += [Or(k < 0, k > a_length)] # k < 0 || k > a.length
```

[b] A simple path is one where any one node appears at most once. We compute them using `JGraphT`'s `AllDirectedPaths` method.[9]

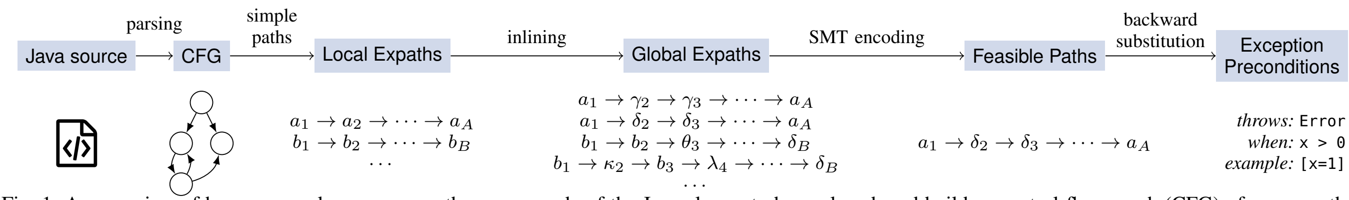

Fig. 1: An overview of how WIT works. WIT parses the source code of the Java classes to be analyzed, and builds a control-flow graph (CFG) of every method. It enumerates the simple paths in every method's CFG that may end with an exception (expaths). It then transforms these expaths local to a specific 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*.

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 ϕ corresponding to each global expath p: if ϕ is satisfiable, then the global expath p is feasible, and hence it corresponds to a possible exceptional behavior of method `m`.

WIT encode ϕ as a Python program using the Z3 SMT solver's `Z3Py` Python API.[c] Lst. 4 shows a simplified excerpt of the SMT program encoding the feasibility of `insert`'s global expath p_1. First, it declares logic variables of the appropriate types to encode program variables (e.g., `k`), their basic properties (e.g., `a_length`, which corresponds to the Java expression `a.length`), and the values passed via method calls (e.g., `getLength` is an integer variable storing `getLength()`'s output). Then, it builds a list `c` of constraints that capture the path constraints and the semantics of the statements along the path. For example, `a_length` must be nonnegative, since it corresponds to array `a`'s length (line 4); the properties of array `v` are copied to those of `x`, since `insert`'s argument `v` is the actual argument for `isEmpty`'s formal argument `x` (line 6); and path constraint `!isEmpty(v)` corresponds to the complement of Boolean variable `isEmpty` (line 11). In this case, Z3 easily finds that the constraints in `c` are unsatisfiable, since `Not(0 == 0)` is identically false. In contrast, the constraints corresponding to path p_2 are satisfiable, and thus Z3 outputs a satisfying assignment of all variables in that case.

Sometimes WIT does not have sufficient information to determine with certainty whether a path is feasible. When a path includes a call to an opaque method (whose implementation is not available) that is not one of the basic JDK library methods that we provided a logic encoding for, WIT's feasibility check is underconstrained. In these cases, WIT still performs a feasibility check but reports any results as *maybe*, to warn that the output may not be correct.

In Lst. 3's example, suppose that `getLength`'s implementation wasn't available. Then, WIT would only know that `getLength` returns an integer; therefore it would classify path p as feasible but mark it as *maybe* since it is just an educated guess without correctness guarantees.

[c] 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.

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, method `void f(int x){int y=x+1; if(y > 0)throw;}` has a single feasible expath with path condition `y > 0`, which becomes `x+1 > 0` after backward substitution through the assignment to variable `y`. Since `x+1 > 0` only mentions argument `x`, it is a suitable exception precondition for method `m`.

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`, where calling `inc` increments the value of `x`. Here too, 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],[10] 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 `(!(x==null))&&(!(x==null))&&(0+1==1)&&(y<0||y>x.length)` into the much more readable `(y>x.length || y<0) && null!=x`, which doesn't repeat `x!=null` and omits the tautology `0+1==1`.

WIT's final output consists of: (a) the exception precondition,

(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 [41]" [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.)[d] 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).

[d]Even mature static analysis frameworks such as Spoon have only partial/-experimental support for features such as try/catch.[11]

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 [42], [44], [16] and the automatic generation of tests for some of these libraries [24] (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], [24]. 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[e] (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 $\langle m, p \rangle$, 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 $\langle m, p \rangle$ 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 e, 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 **`int`** `x` and `String y`, whose class includes fields **`int`**`[] a`, **`int`** `count`, and method **`boolean`** `active()`:

`void m(int x, int[] y)`	*argument*	*state*
null	`y == null`	`this.a != null`
value	`x == 1`	`this.count > 0`
query	`y.isEmpty()`	`!this.active()`

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 that were not already documented, and submitted them as 8 pull requests in 5 projects: `Accumulo`,[12] `Commons Lang`,[13,14,15] `Commons Math`,[16,17] `Commons Text`,[18] 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 "expres" the former and "maybes" the latter. The experimental evaluation focuses on the former kind: "exception precondition" (without qualifiers) means "expre".

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.[f] This manual analysis found that *all* expres were indeed correct, that is 100% precision.

[e] We exclude 6 innacurate cases.

[f] We pick all expres for 7 projects with less than 10 in total.

PROJECT	KLOC	TIME	EXPRES #	M	P	MAYBES ?#	?P
accumulo	33	124	325	309	1.0	941	0.5
Activiti	103	152	374	338	1.0	154	0.4
asm	28	136	110	81	1.0	409	0.8
asterisk-java	30	5	16	14	1.0	9	0.4
AutomatedCar	4	0	0	0	–	2	–
Baragon	15	1	1	1	1.0	12	0.6
bigtop	6.5	0	1	1	1.0	4	1.0
byte-buddy	57	12	170	164	1.0	214	0.8
camel	972	1962	663	600	1.0	778	0
closure-compiler	287	66	46	45	1.0	62	0.6
commons-bcel	35	12	32	31	1.0	284	1.0
commons-configuration	20	10	73	60	1.0	55	0.8
commons-io	9.5	16	106	78	1.0	183	0.9
commons-lang	29	14	435	354	1.0	245	0.8
commons-math	61	96	627	357	1.0	627	0.8
commons-text	10	9	116	82	1.0	212	0.6
confucius	0.5	0	0	0	–	0	–
curator	26	5	18	13	1.0	40	0.6
dubbo	99	102	202	167	1.0	141	0.6
flink	568	888	1996	1634	1.0	3992	0.8
gae-java-mini-profiler	0.5	0	0	0	–	0	–
h2database	150	178	355	350	1.0	1104	0.6
httpcomponents-client	32	4	9	8	1.0	9	0.8
itext7	145	313	356	2325	1.0	1239	0.8
jackrabbit	260	66	465	446	1.0	481	0.8
jackrabbit-oak	26	111	175	175	1.0	460	0.8
jackson-databind	63	26	82	81	1.0	138	1.0
jfreechart	84	50	846	780	1.0	470	1.0
jmonkeyengine	19	279	388	354	1.0	832	0.2
joda-time	29	33	140	130	1.0	255	0.8
logging-log4j2	99	52	166	93	1.0	106	0.6
lucene-solr	685	721	1627	1406	1.0	2692	0.6
pdfbox	106	139	202	189	1.0	218	0.6
poi	260	448	387	352	1.0	2003	0.2
santuario-xml-security-java	35	8	78	73	1.0	59	0.8
shiro	27	10	70	63	1.0	101	0.4
spoon	75	38	213	207	1.0	187	0.2
spring-cloud-gcp	20	5	7	7	1.0	4	0.7
spring-data-commons	28	5	5	4	1.0	105	0.5
swingx	72	47	86	85	1.0	99	0.8
traccar	54	6	1	1	1.0	10	0.6
visualee	1.8	1	0	0	–	6	0
weiboclient4j	7.8	1	5	5	1.0	0	–
wicket	109	129	453	386	1.0	463	0.8
wildfly-elytron	80	33	190	175	1.0	227	0.6
xmlgraphics-fop	165	99	258	210	1.0	1357	0.4
overall	5,720	6,412	11,875	10,234	1.0	20,989	0.7

TABLE I: Exception preconditions inferred by WIT. For each analyzed PROJECTS: 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 these that are correct.

As we explained in Sec. III, WIT's maybes still have a chance of being correct exception preconditions, but they remain educated guesses in general. We randomly picked 206 maybes uniformly in each of the 43 projects that report someand manually checked them as we did for the expres. We found that 71% (128) of them are indeed correct; thus, WIT's precision remains high ($87\% = (128 + 390)/(206 + 390)$) even if we consider all maybes. As we further discuss in Sec. V-B, in most cases, WIT could not confirm the maybes as correct because they involve methods whose implementation is not available or unsupported Java features.

Listing 5: Excerpt from class `FileUtils` in project `Commons IO`.

```
static void copyToDir(File src, File destDir) {
    if (src == null) { throw new NullPointerException(); }
    if (src.isDirectory()) { copyDirToDir(src, destDir); }
    else if (src.isFile()) { copyFileToDir(src, destDir); }
    else { throw new IOException("Source does not exist"); }
}

static void copyDirToDir(File srcDir, File destDir) {
    if (srcDir == null) { throw new NullPointerException(); }
    if (srcDir.exists() && !srcDir.isDirectory())
    { throw new IllegalArgumentException(); }
    if (destDir == null) { throw new NullPointerException();}
    if (destDir.exists() && !destDir.isDirectory())
    { throw new IllegalArgumentException(); }
    // ...
}
```

Manually analyzing a significant sample of exception preconditions confirmed that WIT *is 100% precise.*

B. RQ2: Recall

Out of DSC [24]'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 `IOUtils`, 31 in `FileUtils`, 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,[g] 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, Lst. 5 shows an excerpt of two methods in `Commons IO`'s class `FileUtils`. WIT reports most of their exception preconditions as maybes. For example,

[g]Excluding unsupported annotation kinds is a common practice in the empirical evaluation of tools that infer annotations [46].

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.[h] 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 [44]'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[21] 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`[22] 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%).

> WIT *inferred 7–83% of the exception preconditions in* `Commons IO`*. Its recall varies considerably depending on the analyzed project's characteristics.*

C. RQ3: Features

Sec. V-B's 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.

[h] Indeed, both methods may throw a `SecurityException`,[20] which makes them unsuitable, strictly speaking, to express a precondition in the most general case.

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.[23] 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], [25]: a method's precondition should only be weakened [23], but no Boolean expression is weaker than **`true`**.

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[24] 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 `ConcurrentModification` exception,[25] or after collecting constant values from other classes.[26] In all, WIT's output is often concise and to the point—and thus readable and useful.

> *The exception preconditions inferred by* WIT *are usually succinct and mainly involve checks of method arguments.*

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. I, 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 to methods without any Javadoc, and 153 to methods with some Javadoc that does not describe that exceptional behavior. In contrast, 21% (110) of WIT's exception preconditions are properly documented; and 6% (31) of them are only partially documented (usually with a `@throws Exception` tag that does not specify the conditions under which an `Exception` is thrown). The remaining 1% corresponds to 6 exception preconditions whose documentation is incorrect. We also found that 38% (196) of the 518 preconditions occur in nested calls (when an exception is propagated from a method call); only 23% (45) of them are documented.

> *In a manually analyzed sample, 72% of WIT's exception preconditions were **not** documented.*

While there may be situations where documenting every source-code method is not needed or recommended, properly documenting *public* methods of APIs (remember that all of WIT's exception preconditions refer to public methods) is an accepted best practice [46], [24]. To determine whether WIT's inferred preconditions can be a valuable source of API documentation, we collected 90 exception preconditions extracted by WIT in 5 Apache projects and submitted them as 8 pull requests (as described in Sec. IV-B). At the time of writing, maintainers accepted (without modifications) 5 pull requests containing 71 preconditions—54 (76%) of them occurring in nested calls. Two pull requests to project `Commons Math` have not been reviewed yet; one to project `Commons Lang` is on hold because the project maintainers realized that the 10 methods whose exceptional behavior we document are inconsistent in using `IllegalArgumentException` vs. `NullPointerException`, and they prefer to fix this inconsistency before updating the documentation.

It is significant that the projects that accepted these pull requests are known for their extensive and thorough documentation practices [44], [24]. The fact that WIT could automatically detect several exception preconditions that were missing from their documentation, and promptly added following our pull requests,[i] indicates that WIT's output can be quite useful. We expect that WIT's precise output can have an even bigger impact on scarcely documented projects.

> *WIT's precise exception preconditions can be useful to improve also large and mature projects: maintainers from 4 Apache projects accepted 79% of a sample of WIT preconditions submitted as pull requests.*

VI. Threats to Validity

The main threat to the *internal validity* of our assessment of WIT's *precision* (Sec. V-A) comes from the fact that it is based on manual inspection of Java code and documentation. Like all manual analyses, we cannot guarantee that no mistakes were made. Nevertheless, various evidence corroborates the claim that WIT's precision is high. First, WIT's precision follows from its design; therefore, the manual analysis was primarily a validation of WIT's *implementation*, checking that no unexpected source of incorrectness occurred in practice. Second, we inspected not only the source code but also any official documentation, tests, as well as the datasets of related studies of Java exceptions [21], [24]. Third, the authors extensively discussed together the few non-obvious cases, and were as conservative as possible in the assessment. We followed similar precautions to mitigate threats to our assessment of WIT's *recall* (Sec. V-B), where we relied on [24]'s manual analysis as ground truth.

Our selection of 46 Java projects includes several very popular Java open source libraries, which were used in recent related work; this helps reduce threats to *external validity*. It remains that the exceptional behavior of libraries may be different than that of other kinds of projects. Since library APIs tend to perform more input validity checks [31], it is possible that WIT would report fewer exception preconditions simply because fewer are present in other kinds of software.

WIT's implementation has a number of limitations; some reflect deliberate trade-offs, while others could simply be removed by extending its implementation. In its current state, WIT has demonstrated to produce useful output and to be precise and scalable.

VII. Related Work

Assertion Inference. 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 be not applicable to all features of a realistic programming language [6], [8], [20], [7], [33]. Daikon [13] 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 [9], [17], [40], [10], [37], [26]. WIT is fundamentally based on static analysis, which can be very precise but incomplete [18]; 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 [3], [35], [27], [45], [38], [38]. 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 [28], [15].

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 [25], [29], [39], [30], [36], [43], [34]: the

[i]One maintainer from `Accumulo` remarked that ours "are nice fixes to the javadoc, thanks for finding them."

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 DScribe's approach.

VIII. Discussion and Conclusions

We presented WIT: a technique to extract exception preconditions of Java methods that focuses on precision and is lightweight.

The output of WIT's analysis can be useful to extend, complement, and revise the documentation of public methods' exceptional behavior. Accurately documenting exceptions is crucial for developers [46], but writing documentation is onerous [24], [25]; as a result, APIs often lack documentation [31], especially for exceptions [4]. WIT's high *precision* ensures that its output can generally be trusted without requiring manual validation, and hence it can directly help the job of developers writing documentation (or tests).

WIT's 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

[1] Angello Astorga, Siwakorn Srisakaokul, Xusheng Xiao, and Tao Xie. PreInfer: Automatic inference of preconditions via symbolic analysis. In *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2018, Luxembourg City, Luxembourg, June 25-28, 2018*, pages 678–689. IEEE Computer Society, 2018. doi:10.1109/DSN.2018.00074.

[2] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli. Satisfiability modulo theories. In A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors, *Handbook of Satisfiability*. IOS Press, 2009.

[3] Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. Translating code comments to procedure specifications. ISSTA 2018, page 242–253, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213872.

[4] Raymond P. L. Buse and Westley Weimer. Automatic documentation inference for exceptions. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 273–282. ACM, 2008. doi:10.1145/1390630.1390664.

[5] Satish Chandra, Stephen J. Fink, and Manu Sridharan. Snugglebug: a powerful approach to weakest preconditions. In Michael Hind and Amer Diwan, editors, *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009*, pages 363–374. ACM, 2009. doi:10.1145/1542476.1542517.

[6] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th Annual ACM Symposium on Principles of Programming Languages (POPL'78)*, pages 238–252, 1977.

[7] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In Roberto Giacobazzi, Josh Berdine, and Isabella Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013, Rome, Italy, January 20-22, 2013. Proceedings*, volume 7737 of *Lecture Notes in Computer Science*, pages 128–148. Springer, 2013. doi:10.1007/978-3-642-35873-9_10.

[8] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[9] Christoph Csallner and Yannis Smaragdakis. Dynamically discovering likely interface invariants. In *ICSE*, pages 861–864, 2006.

[10] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, pages 281–290, 2008.

[11] Wayne W. Daniel. *Biostatistics: A Foundation for Analysis in the Health Sciences*. Wiley, 7 edition, 1999.

[12] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. doi:10.1007/978-3-540-78800-3_24.

[13] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[14] Andrew Forward and Timothy Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM Symposium on Document Engineering, McLean, Virginia, USA, November 8-9, 2002*, pages 26–33. ACM, 2002. doi:10.1145/585058.585065.

[15] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*, pages 200–20010, 2018.

[16] M. Kechagia, S. Mechtaev, F. Sarro, and M. Harman. Evaluating automatic program repair capabilities to repair api misuses. *IEEE Transactions on Software Engineering*, (01):1–1, mar 5555. doi:10.1109/TSE.2021.3067156.

[17] Nadya Kuzmina, John Paul, Ruben Gamboa, and James Caldwell. Extending dynamic constraint detection with disjunctive constraints. In *WODA*, pages 57–63, 2008.

[18] Claire Le Goues and Westley Weimer. Specification mining with few false positives. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 292–306. Springer, 2009.

[19] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. doi:10.1145/197320.197383.

[20] Francesco Logozzo. Automatic inference of class invariants. In *VMCAI*, volume 2937 of *LNCS*, pages 211–222. Springer, 2004.

[21] Diego Marcilio and Carlo A. Furia. How Java programmers test exceptional behavior. In *18th IEEE/ACM International Conference on Mining Software Repositories, MSR 2021, Madrid, Spain, May 17-19, 2021*, pages 207–218. IEEE, 2021. doi:10.1109/MSR52588.2021.00033.

[22] Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondřej Čertík, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Štěpán Roučka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. Sympy: symbolic computing in python. *PeerJ Computer Science*, 3:e103, January 2017. doi:10.7717/peerj-cs.103.

[23] Bertrand Meyer. *Object-oriented software construction*. Prentice Hall, 2nd edition, 1997.

[24] Mathieu Nassif, Alexa Hernandez, Ashvitha Sridharan, and Martin P. Robillard. Generating unit tests for documentation. *IEEE Transactions on Software Engineering*, pages 1–1, 2021. doi:10.1109/TSE.2021.3087087.

[25] Hoan Anh Nguyen, Robert Dyer, Tien N. Nguyen, and Hridesh Rajan. Mining preconditions of apis in large-scale code corpus. FSE 2014, page 166–177, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2635868.2635924.

[26] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. DIG: A dynamic invariant generator for polynomial and array invariants. *ACM Trans. Softw. Eng. Methodol.*, 23(4):30:1–30:30, 2014. doi:10.1145/2556782.

[27] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. Inferring method specifications from natural language api descriptions. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 815–825, 2012. doi:10.1109/ICSE.2012.6227137.

[28] Hung Phan, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. Statistical learning for inference between implementations and documentation. In *39th IEEE/ACM International Conference on Software Engineering: New Ideas and Emerging Technologies Results Track, ICSE-NIER 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 27–30. IEEE Computer Society, 2017. doi:10.1109/ICSE-NIER.2017.9.

[29] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Path-sensitive inference of function precedence protocols. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, page 240–250, USA, 2007. IEEE Computer Society. doi:10.1109/ICSE.2007.63.

[30] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. Static specification inference using predicate mining. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 123–134, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1250734.1250749.

[31] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. Automated api property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013. doi:10.1109/TSE.2012.63.

[32] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. 12(2):191–221, April 2003. doi:10.1145/941566.941569.

[33] Mohamed Nassim Seghir and Peter Schrammel. Necessary and sufficient preconditions via eager abstraction. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 236–254. Springer, 2014. doi:10.1007/978-3-319-12736-1_13.

[34] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, page 174–184, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1273463.1273487.

[35] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 260–269, 2012. doi:10.1109/ICST.2012.106.

[36] Suresh Thummalapenta and Tao Xie. Alattin: Mining alternative patterns for detecting neglected conditions. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 283–294, 2009. doi:10.1109/ASE.2009.72.

[37] Nikolai Tillmann, Feng Chen, and Wolfram Schulte. Discovering likely method specifications. In *ICFEM*, volume 4260 of *LNCS*, pages 717–736. Springer, 2006.

[38] Chong Wang, Xin Peng, Mingwei Liu, Zhenchang Xing, Xuefang Bai, Bing Xie, and Tuo Wang. A learning-based approach for automatic construction of domain glossary from source code and documentation. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 97–108, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3338906.3338963.

[39] Andrzej Wasylkowski and Andreas Zeller. Mining temporal specifications from object usage. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, page 295–306, USA, 2009. IEEE Computer Society. doi:10.1109/ASE.2009.30.

[40] Yi Wei, Carlo A. Furia, Nikolay Kazmin, and Bertrand Meyer. Inferring better contracts. In Richard N. Taylor, Harald Gall, and Nenad Medvidović, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*, pages 191–200. ACM, May 2011.

[41] Westley Weimer and George C. Necula. Finding and preventing run-time error handling mistakes. In John M. Vlissides and Douglas C. Schmidt, editors, *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2004, October 24-28, 2004, Vancouver, BC, Canada*, pages 419–431. ACM, 2004. doi:10.1145/1028976.1029011.

[42] Ming Wen, Yepang Liu, Rongxin Wu, Xuan Xie, Shing-Chi Cheung, and Zhendong Su. Exposing library api misuses via mutation analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 866–877, 2019. doi:10.1109/ICSE.2019.00093.

[43] Cheng Zhang, Juyuan Yang, Yi Zhang, Jing Fan, Xin Zhang, Jianjun Zhao, and Peizhao Ou. Automatic parameter recommendation for practical api usage. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 826–836, 2012. doi:10.1109/ICSE.2012.6227136.

[44] Hao Zhong, Na Meng, Zexuan Li, and Li Jia. An empirical study on api parameter rules. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 899–911, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3377811.3380922.

[45] Hao Zhong and Zhendong Su. Detecting api documentation errors. 48(10):803–816, October 2013. doi:10.1145/2544173.2509523.

[46] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. Automatic detection and repair recommendation of directive defects in java api documentation. *IEEE Transactions on Software Engineering*, 46(9):1004–1023, 2020. doi:10.1109/TSE.2018.2872971.

URL References

1. WIT replication package: https://doi.org/10.6084/m9.figshare.19086992
2. Apache Dubbo https://dubbo.apache.org/en/
3. Apache Commons Lang https://commons.apache.org/proper/commons-lang/
4. Apache Dubbo on GitHub https://github.com/apache/dubbo
5. https://github.com/apache/commons-lang/commit/ba607f525b842661d40195d0d4778528e2384e70
6. JavaParser: https://github.com/javaparser/javaparser
7. JGraphT: https://jgrapht.org/
8. https://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/ArrayUtils.html
9. https://jgrapht.org/javadoc-1.4.0/org/jgrapht/alg/shortestpath/AllDirectedPaths.html
10. SymPy https://www.sympy.org/en/index.html
11. https://github.com/INRIA/spoon/tree/master/spoon-control-flow
12. https://github.com/apache/accumulo/pull/2594
13. https://github.com/apache/commons-lang/pull/869
14. https://github.com/apache/commons-lang/pull/870
15. https://github.com/apache/commons-lang/pull/871
16. https://github.com/apache/commons-math/pull/206
17. https://github.com/apache/commons-math/pull/207
18. https://github.com/apache/commons-text/pull/311
19. https://github.com/apache/commons-io/pull/339
20. https://github.com/openjdk/jdk/blob/f77a1a156f3da9068d012d9227c7ee0fee58f571/src/java.base/share/classes/java/io/File.java#L889
21. https://asm.ow2.io/asm4-guide.pdf#page=62
22. https://github.com/jfree/jfreechart
23. https://github.com/apache/lucene-solr/blob/7ada4032180b516548fc0263f42da6a7a917f92b/solr/solrj/src/java/org/apache/solr/client/solrj/io/sql/ResultSetImpl.java#L631
24. https://github.com/apache/logging-log4j2/blob/59f6848b70eebbaa3aa0e14f7186b9b5e1942b5a/log4j-layout-template-json/src/main/java/org/apache/logging/log4j/layout/template/json/util/TruncatingBufferedWriter.java#L160
25. https://github.com/apache/logging-log4j2/blob/59f6848b70eebbaa3aa0e14f7186b9b5e1942b5a/log4j-perf/src/main/java/org/apache/logging/log4j/perf/nogc/OpenHashStringMap.java#L476
26. https://github.com/apache/jackrabbit/blob/35d5732bc1418718f49553a81e42ac4146619dcf/jackrabbit-spi-commons/src/main/java/org/apache/jackrabbit/spi/commons/name/PathFactoryImpl.java#L217