Automatic Documentation Inference for Exceptions

Raymond P.L. Buse and Westley R. Weimer
Department of Computer Science
University of Virginia
Charlottesville, VA, USA
{buse, weimer}@cs.virginia.edu

ABSTRACT

Exception handling is a powerful and widely-used programming language abstraction for constructing robust software systems. Unfortunately, it introduces an inter-procedural flow of control that can be difficult to reason about. Failure to do so correctly can lead to security vulnerabilities, breaches of API encapsulation, and any number of safety policy violations.

We present a fully automated tool that statically infers and characterizes exception-causing conditions in Java programs. Our tool is based on an inter-procedural, context-sensitive analysis. The output of this tool is well-suited for use as human-readable documentation of exceptional conditions.

We evaluate the output of our tool by comparing it to over 900 instances of existing exception documentation in almost two million lines of code. We find that the output of our tool is at least as good as existing documentation 85% of the time and is better 25% of the time.

Categories and Subject Descriptors

F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Specification techniques; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation

General Terms

Documentation, Human Factors

Keywords

Software Documentation, Documentation Inference, Exception Handling

1. INTRODUCTION

Modern exception handling allows an error detected in one part of a program to be handled elsewhere depending on the context. Most language-level exception schemes are based on the replacement model [16, 40]. A method may not have enough information to handle erroneous or otherwise "exceptional" conditions. In such cases the method "raises" or "throws" an exception to a parent method farther up the call stack where sufficient context exists to properly handle the event. This non-sequential control flow is simultaneously convenient and problematic [25, 29]. Uncaught exceptions and poor support for exception handling are reported as major obstacles for large-scale and mission-critical systems (e.g., [1, 3, 4, 34]). Some have argued that the best defense against this class of problem is the complete and correct documentation of exceptions [23]. The use of so-called "checked" exceptions, which force developers to declare the presence of uncaught exceptions, is a partial solution. However, in practice, many developers have found this to be burdensome requirement. Often exceptions are caught trivially (i.e., no action is taken to resolve the underlying error [39]) or the mechanism is purposely circumvented [30, 32].

Reasoning about programs that use exceptions is difficult for humans and also for automatic tools and analyses (e.g., [7, 8, 15, 17, 35, 36]). We propose to relieve part of that burden. We present an algorithm for inferring conditions that may cause a given method to raise an exception. Our automatic approach is based on symbolic execution and inter-procedural dataflow analysis. It alerts developers to the presence of "leaked" exceptions they may not be aware of, as well as to the causes of those exceptions. It also makes plain the concrete types of exceptions that have been masked by subsumption and subtyping; designing an exception handler often requires precise exception type information [23, 30]. Finally, but importantly, the tool can be used to automatically generate documentation for the benefit of the developers, maintainers, and users of a system.

Software maintenance, traditionally defined as any modification made on a system after its delivery, is a dominant activity in software engineering. Some reports place maintenance at 90% of the total cost of a typical software project [28, 33]. One of the main difficulties in software maintenance is a lack of up-to-date documentation [10]. As a result, some studies indicate that 40% to 60% of maintenance activity is spent simply studying existing software (e.g., [27] p. 475, [28] p. 35). Improving software documentation is thus of paramount importance, and in many cases our proposed algorithm does just that. In addition, our algorithm is fully automatic and quite efficient, making it reasonable to run it nightly and thus prevent drift between the maintained program and the documentation.

Many tools exist that allow for the automatic extraction of API-level documentation from source-code annotations. JAVADOC, a popular tool for Java, has been studied in the past (e.g., [21]) and is in common use among both commercial and open source Java projects. JAVADOC allows programmers to document the conditions under which methods throw exceptions, but in many existing projects this documentation is incomplete or inconsistent [14, 19, 37]. Users report that they prefer documentation that, among other properties, "provides explanations at an appropriate level of detail", is "complete" and is "correct" [26]. The documentation produced by our prototype tool integrates directly with JAVADOC and we evaluate it in terms of its completeness, correctness and use of potentially-inappropriate information.

The main contributions of this paper are:

- A study of the documentation of exceptions in several existing open source projects.
- An automatic algorithm for determining conditions sufficient to cause an exception to be thrown. Those conditions are used to generate human-readable documentation for explicitly-thrown and implicitly-propagated exceptions.
- Experimental evidence that our tool generates documentation that is at least as accurate as what was generated by humans in 85% of cases.

The structure of this paper is as follows. In Section 2 we present a motivating example related to exception documentation inference. We present our algorithm in Section 3. We describe a prototype tool based on that algorithm in Section 4. In Section 5 we empirically study existing programs and motivate the need for automatic exception documentation. Section 6 presents experimental results about the efficacy of our algorithm on off-the-shelf programs. We present future work in Section 7, and Section 8 concludes.

2. MOTIVATING EXAMPLE

In this section we illustrate some of the challenges in documenting exceptions with a simple example drawn from `FreeCol`, an open-source game. The class in question is called `Unit` and is intended to be sub-classed many times.

```
 1  /**
 2   * Moves this unit to america.
 3   *
 4   * @exception IllegalStateException
 5   *            If the move is illegal.
 6   */
 7  public void moveToAmerica() {
 8    if (!(getLocation() instanceof Europe)) {
 9      throw new IllegalStateException("A unit"
10        + " can only be moved to america from"
11        + " europe.");
12    }
13    setState(TO_AMERICA);
14    // Clear the alreadyOnHighSea flag:
15    alreadyOnHighSea = false;
16  }
```

Our goal is to determine what exceptions `moveToAmerica()` can raise, and what conditions are sufficient to cause them to be raised. On the surface, this seems fairly simple: if the result of a call to `getLocation()` returns an object that is not of type `Europe` an `IllegalStateException` will be thrown on line `9`.

There is more to consider, however. Any method invocation, including `getLocation()` (line `8`) and `setState()` (line `13`), has the potential to throw an exception as well. We thus need to inspect those methods to see which exceptions they may throw. Additionally, because this class is likely to be extended and method invocation is handled by dynamic dispatch, the call to either method may resolve to an implementation for any subclass of `Unit`. Any of those methods may contain method invocations of their own. Finding all relevant exception sources can thus require significant tracing, and the more complex the code gets, the harder it becomes to track what must be true to reach those `throw` statements. Java addresses part of this problem by requiring "throws clause" annotations for checked exceptions and limiting what checked exceptions can be declared for subtypes. However, for unchecked exceptions in Java (or all exceptions in C#, etc.) the problem remains.

Software evolution and maintenance present additional major concerns. If a `throw` statement is added to any method reachable from `moveToAmerica`, the documentation of it, and potentially many other methods, including any method that calls `moveToAmerica`, would have to be amended. Notice also that the human provided JAVADOC documentation for this method is "`If the move is illegal`". This hides what constitutes an illegal move, which might be desirable if we expect that the implementation might change later. However, an automated tool could provide specific and useful documentation that would be easy to keep synchronized with an evolving code base. The algorithm we present in this paper generates "`IllegalStateException thrown when getLocation() is not a Europe`."

3. ALGORITHM DESCRIPTION

We now present an algorithm that generates documentation characterizing the runtime circumstances sufficient to cause a method to throw an exception. First, our algorithm locates exception-throwing instructions and tracks the flow of exceptions through the program. Second, the algorithm symbolically executes control flow paths that lead to these exceptions. This symbolic execution generates predicates describing feasible paths yielding a boolean formula over program variables. If the formula is satisfied at the time the method is invoked, then the exception can be raised.

The first phase of our algorithm is a refinement of JEX [6, 30]. For each method we generate a set of possible exception types that could be thrown by (and thus escape) that method. Our algorithm contains two improvements over previous work: a pre-processing of the call graph for increased speed, and a more precise treatment of throw statements to ensure soundness. This phase takes as input a program, its call graph, and the results of a receiver-class analysis on dynamic dispatches, and produces as output a mapping from methods to information about thrown exceptions.

The second phase of our algorithm starts with the exception information generated in the first phase and uses it to produce predicates that describe paths that throw exceptions. Those predicates become the documentation describing the conditions under which the exception is thrown. The second phase uses a form of symbolic execution to trace through method bodies and gather constraints about excep-

Input: Receiver-class information R.
Input: Program call graph C (built using R).
Input: Maximum desired propagation *depth*.
Output: Mapping M : methods $\rightarrow$ exception information.

```
 1: for all all methods $m \in C$ do
 2:     $M(m) \leftarrow \emptyset$
 3: end for
 4: $Worklist \leftarrow$ methods in $C$
 5: Topological sort $Worklist$
 6: while $Worklist$ is not empty do
 7:     for all $m \in c$ do
 8:         $Explicit \leftarrow \{(e, l, 0) \mid m$ has throw $e$ at $l\}$
 9:         $Propa \leftarrow \{(e, l, d{+}1) \mid m$ has a method call at $l \quad \wedge$
            $m' \in R(l) \quad \wedge \quad (e, l', d) \in M(m')\}$
10:         for all $(e, l, d) \in Explicit \cup Propa$ do
11:             if $l$ not enclosed by catch $e'$ with $e \leq e'$ then
12:                 if $d \leq depth$ then
13:                     $M(m) \leftarrow M(m) \cup \{(e, l, d)\}$
14:                 end if
15:             end if
16:         end for
17:         if $M$ changed then
18:             Add $m$ and methods calling $m$ to $Worklist$
19:         end if
20:     end for
21: end while
```

Figure 1: Algorithm for determining method exception information. If $M(m)$ is (e, l, d) then m can propagate (leak) exception e from location l, with e having already propagated through d other methods.

tions. While doing so, in some cases we are able to prune infeasible (i.e., over-constrained) paths. For each method and each exception type that method may raise, the output of phase two is a human-readable documentation instance representing conditions the may cause that method to raise an exception of that type.

3.1 Phase 1: Exception Information

The goal of this phase is to produce a mapping from methods to information about thrown exceptions. Figure 1 describes this algorithm formally. For each method, we call each separate exception that can escape that method and be propagated to its callers an *exception instance.* Each exception instance is an opportunity for documentation. An exception instance can be thrown directly from the method in question or thrown (but not caught) in a called method. For each exception instance, we trace the statement that raises it to each the methods can propagate it. We produce a set of exception instances for each method in the input program.

We require the program call graph C and a mapping R from method invocations to sets of concrete methods that could be invoked there at runtime. Our approach takes the form of a fixpoint worklist algorithm that considers and processes methods in turn. Since exceptions can be propagated through method invocations, we first process leaf methods that make no such invocations. We process a method only when we have precise information about all of methods it may invoke. This process terminates because the updates (line 13) are monotonic (i.e., we can learn additional exceptions that a method may raise, but we never subtract information) and the underlying lattice is of finite height (i.e., at worst a method can raise all possible exceptions mentioned in the program).

Processing a method consists of determining the set of exception instances that method can raise. We consider both explicit `throw` statements (line 8) and also method invocations (line 9) as possible sources of exceptions. We associate a new exception instance with the given method if a thrown exception is not caught by an enclosing `catch` clause (line 11) and thus may propagate to the caller. With respect to sources of exceptions that we consider, this analysis is conservative and may report exceptions that could never be thrown at run-time (e.g., `if (1 == 2) throw Exception();`).

We do *not* consider non-throw statements as possible sources of exceptions (e.g., division raising a divide-by-zero exception). While there may be some utility in a system that attempts to exhaustively track *all* exceptions, those exceptions that are implicitly-thrown are generally indicative of programming errors rather than design choices [39], and their inclusion in an API-level documentation might often be considered inappropriate [14]. Furthermore, estimating all implicitly thrown exceptions is likely to result in many false positives (as noted by [30], strictly speaking any statement can throw any exception).

This algorithm is a refinement of JEX [6, 30]. The primary differences are that we topologically sort the call graph and also that we maintain completeness by considering `throw` statements, instead of just those where the operand is a new object allocation. We also track additional information, such as the exception instance *depth.* We define depth to be the minimum number of methods an exception must propagate through before it reaches the method in question. For example, if an exception is explicitly thrown in the method body, its depth is 0. If `foo` calls `bar` and `bar` raises an exception, that exception has depth 1 in `foo`. In section Section 5 we will relate depth to the likelihood of human-written documentation.

3.2 Phase 2: Generating Documentation

In the first phase of the algorithm we derived *where* exceptions can be thrown. In this phase, we will use that information to discover, for each exception, *why* it might be thrown. Specifically, we use inter-procedural symbolic execution to discover predicates over program variables that would be sufficient to trigger a `throw` statement. Figure 2 describes this algorithm formally. The algorithm produces a mapping D; if $D(m, e) = doc$ then m can raise exception e when *doc* is true.

The algorithm is a fixed point computation using a worklist of methods. Each method is processed in turn (line 7) and a fixed point is reached when there is no change in the inferred documentation (line 20). Processing a method involves determining path predicates that describe conditions under which exceptions may be propagated from that method.

To process a method, we first enumerate all of the control flow paths (line 9) that can lead from the method head to the statements that we have previously determined can raise exceptions. We construct these control flow paths by starting at the exception-throwing statement and working backward

Input: Mapping M : method $\rightarrow$ exception information.
Input: Receiver-class information R.
Input: Program call graph C (built using R).
Output Documentation D : method $\times$ exception $\rightarrow$ predicate.

```
1: for all all methods $m \in C$ and all exceptions $e$ do
2:   $D(m,e) \leftarrow$ false
3: end for
4: $Worklist \leftarrow$ methods in $C$
5: Topological sort $Worklist$
6: while $Worklist$ is not empty do
7:   for all $m \in c$ do
8:     for all $(e,l,d) \in M(m)$ do
9:       for all loop-free paths $p$ from $start$ to $l$ do
10:        $Preds \leftarrow$ symbolically execute $p$
11:        if $l$ is a method call then
12:          for all methods $m'$ in $R(l)$ do
13:            $D(m,e) \leftarrow D(m,e) \vee (Preds \wedge D(m',e))$
14:          end for
15:        else
16:          $D(m,e) \leftarrow D(m,e) \vee Preds$
17:        end if
18:      end for
19:    end for
20:    if $D$ changed then
21:      Add $m$ and methods calling $m$ to $Worklist$
22:    end if
23:  end for
24: end while
```

Figure 2: Algorithm for inferring exception documentation that explains *why* an exception is thrown. If $D(m,e) = P$ then method m can raise exception e when P is true; the predicate P is the documentation.

through the control flow graph of the method until we reach the method entry point; we ignore forward edges during this traversal and thus obtain loop-free paths. When a statement has two predecessors, we duplicate the dataflow information and explore both: our analysis is thus path-sensitive and potentially exponential.

Each full control flow path is then symbolically executed (line 10, as in, e.g., [9]). We track conditional statements (e.g., `if` and `while`) and evaluate their guarding predicates using the current symbolic values for variables. We collect the resulting predicates; conjuncted together they form a constraint called a *path predicate* that describes conditions under which that path can be taken [2, 5, 20, 31]. An off-the-shelf path predicate analysis could also be used; we interleave path predicate generation with fixed-point method processing for efficiency.

If the exception-throwing statement is a dynamic dispatch method invocation, we process the dataflow information for each concrete method that may be invoked (line 11). We use an off-the-shelf conservative receiver class analysis [22] to obtain this information (line 12). A method invocation is thus handled by assigning the actual arguments to the formal parameters and including the body of the callee. We thus model execution paths that extend all of the way to the original `throw` statement. If the callee method m' raises an exception e under conditions $D(m',e)$ and the caller m can reach that point with path predicate *Preds* then the propagated exception occurs on condition $D(m',e) \wedge Preds$ (line 13).

To guarantee termination we only enumerate and explore paths that do not contain loops involving backwards branches. That is, we only consider paths in which loop bodies are evaluated at most once. Because we do not filter out paths with infeasible path predicates, employing this common heuristic [13] will *not* cause us to miss exception-throwing statements. Instead, it will cause us to generate potentially-inaccurate predicates, and thus potentially-inaccurate documentation, in some cases. We make this trade off favoring speed over precision because exception-throwing conditions typically do not depend on particular values of loop induction variables (see Section 6). We are interested in documenting exceptions in terms of API-level variables and not in terms of local loop induction variables. Exception-throwing conditions often depend on whether a loop or conditional is taken at all, but rarely depend on more detailed iteration conditions.

After a fixed point is reached, the path predicate becomes our documentation for the exception instance. If one exception type can be thrown via multiple different paths, those path predicates are combined with a logical disjunction to form a single path predicate. The result is, for each method and for each exception raised by that method, a boolean expression over program variables which, if satisfied at runtime, is sufficient to cause the exception to be raised.

3.3 Documentation Post-Processing

We post-process and pretty-print the resulting path predicates to produce the final human-readable documentation output of our tool. We employ a small number of recursive pattern-matching translations to phrase common Java idioms more succinctly. For example:

- `true` becomes `always`
- `false` $\vee$ `x` becomes `x`
- `x != null` becomes "`x` is not null"
- `x instanceof T` becomes "`x` is a `T`"
- `x.hasNext()` becomes "`x` is nonempty"
- `x.iterator().next()` becomes "`x`.{some element}"

We also apply some basic boolean simplifications (e.g., replacing `x && not(x)` with "false"). The most complicated transformation we use replaces `(x && y) || (x && z)` with `x && (y || z)`. An off-the-shelf term rewriting engine or algebraic simplification tool could easily be employed; in our experiments the generated exception documentation did not involve terms that could be simplified arithmetically.

4. PROTOTYPE TOOL AND EXPERIMENTAL SETUP

To evaluate the effectiveness of our algorithm, we elected to target Java programs, anticipating that we could use existing JAVADOC documentation as a basis for comparison. We chose several popular systems, detailed in Figure 3, for our experiments.

Our algorithm requires a call graph that maps invocation sites to possible concrete target methods. The problem

Name	Version	Domain	kLOC	methods	throws	documented
Azureus	2.5	Internet File Sharing	470	14678	655	35.42%
DrJava	20070828	Development	131	2215	121	19.01%
FindBugs	1.2.1	Program Analysis	142	2965	245	7.76%
FreeCol	0.7.2	Game	103	3606	330	75.15%
hsqldb	1.8.0	Database	154	2383	315	26.03%
jEdit	4.2	Text Editor	138	1110	53	15.09%
jFreeChart	1.0.6	Data Presentation	181	41	7	42.86%
Risk	1.0.9	Game	34	366	15	40.00%
tvBrowser	2.5.3	TV guide	155	3306	21	57.14%
Weka	3.5.6	Machine Learning	436	127	9	88.89%
Total or Mean			1944	30797	1771	40.73%

Figure 3: The set of Benchmark programs used in this study. The "methods" column gives the total number of methods reachable from `main()`. The "throws" column gives the number of `throw` statements contained in those methods. The "documented" column gives the percentage of those exceptions that are documented with a non-empty JAVADOC comment.

Name	Paths	Instances	Mean depth	RCA	EFA	DocInf	Total
Azureus	93,052	17,296	3.30	703.2s	2.32s	504.56s	1,210s
DrJava	1,288	317	1.66	606.0s	0.22s	46.33s	653s
FindBugs	9,280	1,160	1.99	431.0s	0.27s	62.00s	493s
FreeCol	79,620	6,635	4.18	454.5s	2.49s	135.80s	593s
hsqldb	30,569	2,779	2.98	153.6s	0.84s	323.77s	478s
jEdit	6,216	588	4.80	430.1s	0.16s	54.40s	485s
jFreeChart	17	9	1.11	368.7s	0.01s	0.02s	369s
Risk	74	33	0.91	374.0s	0.09s	10.90s	385s
tvBrowser	1,557	283	5.37	490.6s	1.30s	43.30s	535s
Weka	18	14	0.57	458.0s	0.08s	55.34s	513s
Total or Mean	221,691	29,114	2.69	4,470s	7.79s	1,236s	5,714s

Figure 4: Measurements from running our tool on each benchmark, generating documentation for every exception. "Paths" is the number of control flow paths enumerated and symbolically executed. "Instances" is the number of documentations generated.. "Mean Depth" is the mean distance between the method where the documentation is generated, and the original throw statement. "RCA" is runtime for receiver class analysis (we did not implement this) and includes the time for loading and parsing the program files. "EFA" is the runtime for Exception Flow Analysis (Phase 1). "DocInf" is runtime for documentation generation (Phase 2) and includes all post processing. All experiments were conducted on a 2GHz dual core system.

of constructing an accurate and precise call graph for dynamic languages typically involves receiver class analysis or alias analysis. Many off-the-shelf solutions exist. In fact, this problem has become one of the most heavily-researched topics in program analysis [18]. We employed Spark [22] to produce call graphs. Spark is reasonably fast (terminating in less than one hour for all of our benchmarks) and is integrated into Soot [12], the Java bytecode analysis framework we used to parse input programs. We used the Eclipse jdt to parse source code and extract Javadocs.

5. EXISTING EXCEPTION DOCUMENTATION

We claim that complete documentation of exceptions is important to software developers because incompleteness can lead to problems with security, reliability, and encapsulation, as well as to difficulties in software maintenance. In this section we will provide evidence to demonstrate that the conditions under which real world programs can raise exceptions are often not completely documented.

To evaluate documentation completeness we first explore the rate at which exceptions of each type are documented. We would expect "complete" documentation to either include every instance where a particular exception type can be raised, or almost no instances. That is, we hypothesize that if `SecurityException` is deemed worth documenting some of the time, it should be worth documenting all of the time. This is similar to the assumption used by Engler et al. [11] to find bugs from inconsistencies in systems code. This notion of consistency allows for the possibility that the developers may consciously decide not to document certain kinds of exceptions, but views partial documentation of an exception type as a mistake.

Experimentally, we found many examples of inconsistent documentation in our benchmarks. Figure 5 presents some of the most frequently documented, yet sometimes neglected, exception types. For each benchmark chosen, the two most commonly documented exception types are listed. For each exception type we list the static percentage of methods documenting that exception as a percentage of methods that can throw that exception (as determined by our algorithm, see Section 3.1). We do not consider "stub" annotations, which indicate an exception type without additional comment (e.g., just `@throws Exception`), to be actual documentations since they do not convey any information regarding the triggering of the exception, and may have been automatically inserted by a development environment. We conjecture it is unlikely that documentation for a particular exception type is needed or appropriate in over 90% of methods that raise it, but not needed or desired in the other 10%. It is interesting to note that some commonly-documented exceptions are associated with standard library exception types (e.g., `NullPointer`) while others are program-specific (e.g., `DataflowAnalysis`).

We also hypothesize that some exception instances may not be documented because of the difficulty of (or lack of programmer interest in, e.g., [28] p. 45 and [38]) manually modeling exception propagation. Figure 6 indicates that the propagation depth of an exception, in practice, has a strong inverse correlation with the probability that it will be documented. In particular, it is quite rare for documentation to appear for any exception not explicitly raised in the current method (i.e., for any exception instance at depth > 0). For propagation depths larger than three, documentation becomes almost non-existent.

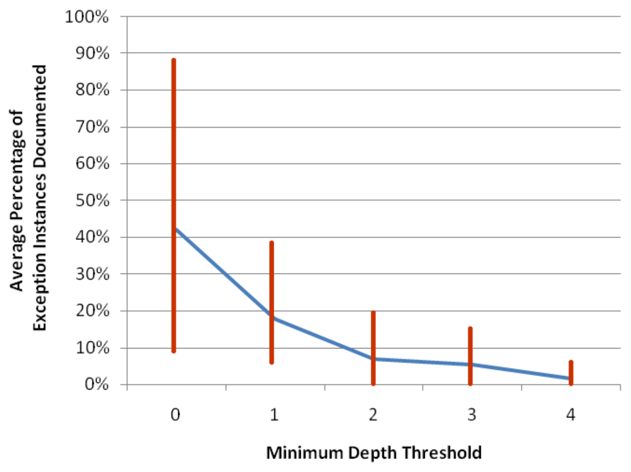

Figure 6: Exception documentation as a function of exception propagation depth. The line represents the average over all exception instances within all of our benchmarks. The vertical bars indicate the maximum and minimum observed value. All of our benchmarks are included. An exception instance explicitly thrown within the method in question has depth zero; higher depths indicate the number of method invocations through which the exception must be propagated.

We claim that the lack of exception documentation in practice follows from the difficulty in developing it and not from a lack of need for it. Both of these observations serve as evidence that the presence of Javadocs for exceptions is inconsistent and incomplete.

6. EVALUATION

The purpose of our algorithm is the automatic construction of useful documentation for exceptions. In this section we compare the documentation produced by our prototype implementation with pre-existing documentation already present in programs.

Having ran our tool on each of the benchmarks in Figure 3 and generated documentation for each exception, we now restrict our attention to only those instances for which a Javadoc comment exists. For our benchmarks there were 951 human-documented exception instances. We paired each existing piece of documentation with the documentation suggested by our tool and then manually reviewed each documentation pair. We categorized each according whether the tool-generated documentation was *better*, *worse*, or about the *same* as the human-created version present in the code.

We consider the tool-generated documentation to be better when it is more precise. For example:

```
     Worse:  if inappropriate
(Us) Better: parameter:params not a KeyParameter
```

We also consider our documentation to be better when it contains more complete information or otherwise includes

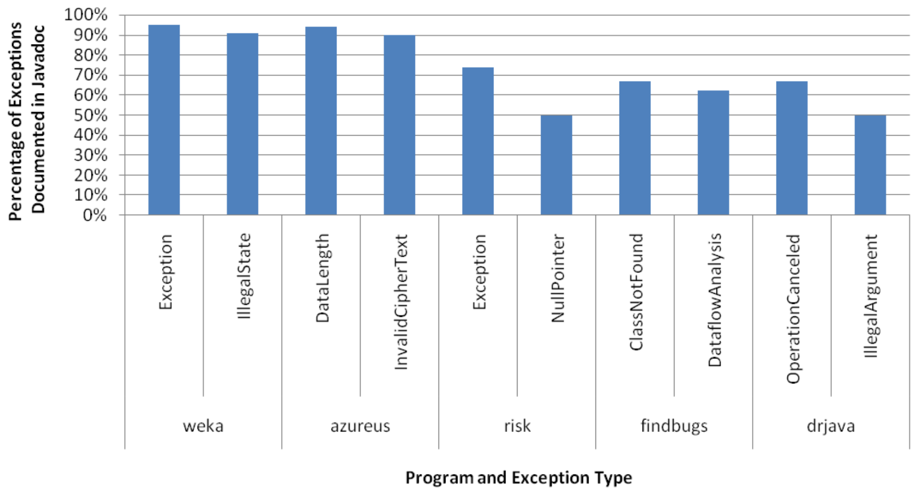

Figure 5: Frequency of exception documentation. Each column represents methods that document the given exception expressed as a percentage of methods that throw the exception. For each benchmark, the two most frequently documented exceptions where the rate is below 100% are listed.

cases that were forgotten in the human-written documentation. For example:

```
     Worse:  id == null
(Us) Better: id is null or id.equals("")
```

Often, both pieces of documentation conveyed the condition. For example:

```
     Same: has an insufficient amount of gold.
(Us) Same: getPriceForBuilding() >
                                   getOwner().getGold()
```

In all other cases, we considered the tool-generated documentation to be worse. This typically happened when human created documentation contained special insight into the relationship between program variables, or expressed high-level information about the program state that could not be readily inferred from a set of predicates without additional knowledge. For example:

```
     Better: the queue is empty
(Us) Worse:  private variable m_Head is null
```

All ties or difficult decisions were resolved in favor of the existing documentation. The total time taken to evaluate all 951 documentation pairs was 3.5 hours; the determination was usually quite direct.

Figure 7 shows relative quality of the documentation produced by our tool for the 951 exception instances in our benchmarks. The ten columns on the left give breakdowns for individual benchmarks. The three rightmost columns give overall performance; we describe them in more detail below.

We also measured whether the generated conditions could be expressed strictly in terms of non-private API-level variables. This property is generally beyond our control, although techniques involving Craig interpolants can be used to express predicates in terms of certain variables instead of others [24]. Predicates over private or local variables might not be useful because their meanings might not be clear at the API-level, and because users of the system cannot directly affect them. Furthermore, documentation involving private variables has the potential to expose implementation details that were intended to be hidden.

In our experiments 29% of the documentation instances involved private or local variables; 71% involved parameters and public variables only. Surprisingly, however, documentation instances involving private variables were nearly as good (83% vs 88% same as or better than existing) as those that were constructed strictly in terms of public variables. This follows largely from the descriptiveness of private variable names in our benchmarks. Consider this indicative example from FindBugs:

```
     Same: the class couldn't be found
(Us) Same: {private classesThatCantBeFound}.
                         contains(parameter:className)
```

Even though `classesThatCantBeFound` is a private field, and we may not even know what type of object it is, this generated documentation is still as good as the human-written one. Not all private variables were descriptive, as this example from Weka shows:

```
     Better: Matrix must be square
(Us) Worse:  {private m} == {private n}
```

Without knowing that `m` and `n` are matrix dimensions, our documentation is not useful. In general, we need not always reject generated documentation that contains a private reference. The decision to accept or reject such documentations depends on several factors: the extent to which encapsulation is a concern, the relative readability of private vs. public data, and on the intended purpose of the documentation (e.g., internal or external use).

Finally, note that in this section we have provided experimental evidence of the utility of our tool primarily with respect to low-depth (i.e., < 4) exceptions: those for which we have a significant baseline for comparison. While higher-depth exceptions will, in general, be represented by more

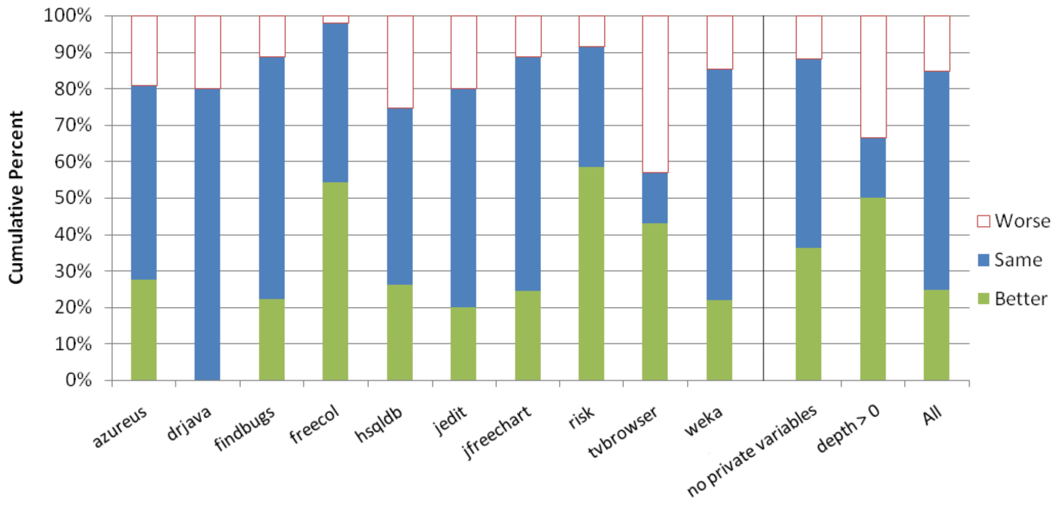

Figure 7: The relative quality of the documentation produced by our tool for 951 exception instances in 1.9MLOC that were already human-documented.

complex predicates, it is not clear at what point they become too complex to be useful to human readers as documentation. In any case, we can see in Figure 4 that our inference algorithm does in fact scale to large programs, where nearly 100,000 control flow paths are enumerated and depths can sometimes average greater than 5. In practice, the determination of which exception types and depths should be documented is likely to depend on program-specific concerns (e.g., the meaning of the exception) as well as usage (e.g., whether the documentation is to be employed as a debugging aide or an API-level documentation supplement, etc.).

7. FUTURE WORK

We found documentation produced by our tool to be surprisingly readable after only a few simple transformations. However, it would be interesting to enhance readability by simulating human prose or re-arranging predicates. Even though most of the generated documentation did not involve complex logic, we suspect that a system capable of symbolic algebra simplification would be useful in certain application domains.

Documentations generated by our tool only involve predicates over program variables. However, there is other information available in the source code that might provide additional insight into the exceptional condition. For example, we might incorporate the argument (typically a string) provided to the exception constructor into the associated documentation.

In some obvious cases, our implementation prunes infeasible paths. Integrating a theorem prover would allow us to rule out more infeasible paths and thus reduce false positives.

Finally, during the course of our experiments, we noticed that certain exception types are frequently documented in a standardized or templated manner by developers. For example, for nearly every `ClassNotFoundException` the corresponding comment is a variation on "thrown if the class couldn't be found." In such cases, it may be appropriate to "copy" or "instantiate" existing documentation structure rather than inferring it.

8. CONCLUSION

In this paper we have described an algorithm for automatic exception documentation in 2 phases. Phase 1 determines which exceptions may be thrown or propagated by which methods; this algorithm is a slight refinement of previous work. The analysis is conservative; it will not miss exceptions but may report false positives. Phase 2, the primary contribution of this work, then characterizes the conditions, or path predicates, under which exceptions may be thrown. This analysis is also conservative; it may generate poor predicates for exceptions that depend on loops or where the call graph is imprecise. We convert these predicates into human-readable documentation. We are able to generate documentation involving only public and API-level variables in 71% of 951 instances associated with 1.9M lines of code.

The documentation instances we generated are at least as accurate as what was created by humans in 85% of the instances, and are strictly better in 25% of them.

Our study of existing documentation suggests that many exception instances remain undocumented in practice. This is especially true when exceptions are propagated through methods. Our algorithm is completely automatic and handles both propagated and local exceptions. It is efficient enough to be used nightly, taking 95 minutes for two million lines of code, thus reducing drift between an implementation and its documentation.

The time costs are low, no annotations are required, and the potential benefits are large. We believe this work is a solid step toward making automatic documentation generation for exceptions a reality.

9. REFERENCES

[1] G. Alonso, C. Hagen, D. Agrawal, A. E. Abbadi, and C. Mohan. Enhancing the fault tolerance of workflow management systems. *IEEE Concurrency*, 8(3):74–81, July 2000.

[2] T. Ball and J. R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[3] M. Bruntink, A. van Deursen, and T. Tourwé. Discovering faults in idiom-based exception handling. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 242–251, 2006.

[4] T. Cargill. Exception handling: a false sense of security. *C++ Report*, 6(9), 1994.

[5] L. Carter, B. Simon, B. Calder, L. Carter, and J. Ferrante. Path analysis and renaming for predicated instruction scheduling. *International Journal of Parallel Programming*, 28(6):563–588, 2000.

[6] B.-M. Chang, J.-W. Jo, K. Yi, and K.-M. Choe. Interprocedural exception analysis for java. In *SAC '01: Proceedings of the 2001 ACM symposium on Applied computing*, pages 620–625, 2001.

[7] R. Chatterjee, B. G. Ryder, and W. Landi. Complexity of points-to analysis of java in the presence of exceptions. *IEEE Trans. Software Eng.*, 27(6):481–512, 2001.

[8] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar. Efficient and precise modeling of exceptions for the analysis of java programs. In *Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31, 1999.

[9] M. Das, S. Lerner, and M. Seigle. ESP: path-sensitive program verification in polynomial time. *SIGPLAN Notices*, 37(5):57–68, 2002.

[10] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *International Conference on Design of Communication*, pages 68–75, 2005.

[11] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *Symposium on Operating Systems Principles*, pages 57–72, 2001.

[12] R. V.-R. et. al. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[13] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *Programming Language Design and Implementation (PLDI)*, pages 234–245, 2002.

[14] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 26–33, 2002.

[15] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 230–239, 20-26 May 2007.

[16] J. B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[17] M. Gupta, J.-D. Choi, and M. Hind. Optimizing java programs in the presence of exceptions. In *European Conference on Object-Oriented Programming*, pages 422–446, London, UK, 2000. Springer-Verlag.

[18] M. Hind. Pointer analysis: haven't we solved this problem yet? In *Workshop on Program Analysis for Software Tools and Engineering*, pages 54–61, 2001.

[19] S. Huang and S. Tilley. Towards a documentation maturity model. In *International Conference on Documentation*, pages 93–99, 2003.

[20] R. Jhala and R. Majumdar. Path slicing. In *Programming Language Design and Implementation (PLDI)*, pages 38–47, 2005.

[21] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *International Conference on Computer Documentation*, pages 147–153, 1999.

[22] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In G. Hedin, editor, *Compiler Construction, 12th International Conference*, volume 2622 of *LNCS*, pages 153–169, Warsaw, Poland, April 2003. Springer.

[23] D. Malayeri and J. Aldrich. Practical exception specifications. In *Advanced Topics in Exception Handling Techniques*, pages 200–220, 2006.

[24] K. L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–12, 2005.

[25] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In *European Conference on Object-Oriented Programming*, pages 85–103, 1997.

[26] D. G. Novick and K. Ward. What users say they want in documentation. In *Conference on Design of Communication*, pages 84–91, 2006.

[27] S. L. Pfleeger. *Software Engineering: Theory and Practice*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[28] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.

[29] M. P. Robillard and G. C. Murphy. Regaining control of exception handling. Technical Report TR-99-14, Dept. of Computer Science, University of British Columbia, 1, 1999.

[30] M. P. Robillard and G. C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.

[31] T. Robschink and G. Snelting. Efficient path conditions in dependence graphs. In *International Conference on Software Engineering (ICSE)*, pages 478–488, 2002.

[32] B. G. Ryder, D. Smith, U. Kremer, M. Gordon, and N. Shah. A static study of java exceptions using jesp. In *International Conference on Compiler Construction*, pages 67–81, London, UK, 2000. Springer-Verlag.

[33] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[34] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith. Dealing with disaster: Surviving misbehaved kernel extensions. In *Symposium on Operating Systems Design and Implementation*, pages 213–227, Seattle, Washington, 1996.

[35] S. Sinha and M. J. Harrold. Criteria for testing exception-handling constructs in java programs. In *ICSM*, pages 265–, 1999.

[36] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. *icse*, 0:336–345, 2004.

[37] B. Thomas and S. Tilley. Documentation for software engineers: what is needed to aid system understanding? In *International Conference on Computer Documentation*, pages 235–236, 2001.

[38] S. Tilley and H. Müller. Info: a simple document annotation facility. In *International Conference on Systems Documentation*, pages 30–36, 1991.

[39] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Conference on Object-oriented programming, systems, languages, and applications*, pages 419–431, 2004.

[40] S. Yemini and D. Berry. A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, 7(2), Apr. 1985.