# Automatic Generation of Oracles for Exceptional Behaviors

Alberto Goffi [1] alberto.goffi@usi.ch
Alessandra Gorla [2] alessandra.gorla@imdea.ch
Michael D. Ernst [3] mernst@cs.washington.edu
Mauro Pezzè [1] mauro.pezze@usi.ch

[1] USI Università della Svizzera italiana, Lugano, Switzerland
[2] IMDEA Software Institute, Madrid, Spain
[3] University of Washington, Seattle, WA, USA

## ABSTRACT

Test suites should test exceptional behavior to detect faults in error-handling code. However, manually-written test suites tend to neglect exceptional behavior. Automatically-generated test suites, on the other hand, lack test oracles that verify whether runtime exceptions are the expected behavior of the code under test.

This paper proposes a technique that automatically creates test oracles for exceptional behaviors from Javadoc comments. The technique uses a combination of natural language processing and run-time instrumentation. Our implementation, Toradocu, can be combined with a test input generation tool. Our experimental evaluation shows that Toradocu improves the fault-finding effectiveness of EvoSuite and Randoop test suites by 8% and 16% respectively, and reduces EvoSuite's false positives by 33%.

## CCS Concepts

**•Software and its engineering → Software testing and debugging;**

## Keywords

Testing, oracle problem, automatic test oracle, oracle generation

## 1. INTRODUCTION

This paper addresses the oracle problem: the creation of assertions that indicate the expected behavior of a software system. A test case consists of two parts: an input that triggers the behavior of the unit under test, and an oracle that indicates whether the behavior was correct. Oracles are an essential part of a test case, since they determine whether a test execution should pass or fail. Oracles encode the *intended* behavior of the software system, so they must be provided by a human or generated from human-provided information such as a formal specification. However, developers often fail to write oracles, and they even more rarely write formal specifications.

This work automatically generates oracles from information that developers do commonly write: natural language documentation. It is considered standard practice to write semi-structured natural-language documentation. For example, Java programmers write Javadoc comments to document their code. Our approach is to *convert such natural-language documentation into test oracles*, and use them in combination with a test input generator to automatically create complete test cases that expose defects.

Our technique uses natural language processing (NLP) techniques to parse the documentation into grammatical relations between words in a sentence. Next, it matches parsed subjects and predicates to source code elements, thus creating conditional expressions that describe the programmer-intended behavior. The technique then converts these conditional expressions into test oracles. We integrated our oracle-generation technique with existing test input generation tools, yielding a system that can automatically create complete tests from scratch, without any human involvement.

This paper focuses on exceptional behavior — situations in which a procedure should raise an exception. Exceptional behavior is particularly interesting because it is a significant cause of failures, and because it tends to be poorly covered by both manual and automatically-generated test suites.

Exceptional behavior is a frequent cause of field failures [47] because developers usually have the common case in mind when writing code, because recovering from failures is inherently challenging, and because developers often forget to account for failures that occur during exception processing.

Exceptional behavior is poorly covered in manually-written test suites, as confirmed by the experimental data reported in Section 4.1. The reasons are similar to those above, and also because it can be difficult to force software into the unusual situations that involve exceptions and exception processing.

Exceptional behavior is poorly covered in automatically-generated test suites for some of the same reasons, but primarily because, in the absence of a human-written specification, a test generation tool cannot determine whether a thrown exception is legal or illegal behavior. If a test throws an exception, then there are four main possibilities: (i) the exception might be specified behavior, given the inputs, (ii) the exception might reveal an error in the software under test, (iii) the exception might be legal but unspecified behavior caused by incorrect usage of the software, such as when the test supplies an invalid input, or (iv) the exception might be due to a system failure such as out-of-memory, timeout, or hardware failure.

If there is no specification of exceptional behavior, then when a tool generates a test that throws an exception, the tool has to use heuristics to classify the test into the four categories listed above. (Section 5.1 gives examples of such heuristics.) Use of heuristics makes the generated tests suffer false alarms and/or missed alarms, and a poor choice of heuristics can make test generation tools practically unusable [13, 14, 41]. Our goal is to *automati-*

*cally generate oracles from human-written documentation, such as Javadoc comments, which is a reliable and commonly-available source of information.*

We have implemented our oracle generation technique in a tool called Toradocu, and we used it in combination with the EvoSuite and Randoop test input generation tools to generate complete test cases that expose exception-related errors in Java programs.

The contributions of this paper include:

- An empirical study showing that developers tend to neglect exceptional behavior when they test their code. The results of this study motivate our work.
- A technique that combines natural-language processing and runtime instrumentation to create executable oracles that can be used during testing.
- A publicly-available implementation, Toradocu.[1]
- An experimental evaluation of the technique showing that:
  - Toradocu reduces the number of false positive reports from automatic test case generation tools.
  - Toradocu can reveal defects. Toradocu's oracles revealed 4 defects in Google Guava, and also defects in reimplementations of parts of Apache Commons.

The remainder of the paper is structured as follows. Section 2 presents a motivating example, Section 3 describes the main ingredients of our technique, Section 4 reports on our experimental evaluation, and Section 5 describes related work.

## 2. MOTIVATING EXAMPLE

Automatically-generated test cases tend to perform poorly with respect to exceptional behaviors of a system under test (SUT). This is because automatically-generated test oracles have to guess at the specification of the SUT — that is, they have to guess at the SUT's expected behavior. When a test input generator creates a test input whose execution throws an exception, the tool must guess whether the execution is a failure and the test case reveals a defect in the SUT.

As an example, consider the Java class FixedOrderComparator from the Apache Commons Collections library.[2] This comparator allows a developer to specify a comparison order among a set of objects. A FixedOrderComparator instance cannot be modified, i.e., it is *locked*, once it has performed a comparison. Method checkLocked is specified to raise an UnsupportedOperationException if the comparator is locked:

```
1 /**
2  * Checks to see whether the comparator is now locked
3  * against further changes.
4  *
5  * @throws UnsupportedOperationException if the
6  * comparator is locked
7  */
8 protected void checkLocked() {...}
```

State-of-the-art test case generators do not produce test cases that effectively test this code. Some test case generators, such as JCrasher [13] and Check 'n' Crash (CnC) [14], assume that a method should not raise any exception unless the method signature explicitly declares it. Such a test generator would produce a failing test case such as the following and wrongly indicate that it reveals a fault in the code under test:

```
1 void test() {
2   FixedOrderComparator c = new FixedOrderComparator(...);
3   ...
4   c.compare(...);
5   ...
6   c.checkLocked();
7 }
```

This would be a *false alarm*: method checkLocked correctly throws an exception because c is locked, given that it has already been used for a comparison.

Other test case generators, such as EvoSuite [19], either ignore test cases that throw exceptions, or assume any observed behavior is correct and create regression test suites that expect run-time exceptions. Such a test case generator would suffer *missed alarms* and possibly create misleading tests. This would happen if the SUT implementation throws an exception in violation of its specification, such as if the developer wrongly implemented method checkLocked to throw the exception under all circumstances.

Randoop [32] can be configured to behave in any of the above ways, but no heuristic or combination of them can reliably check that the behavior of checkLocked matches its specification, i.e., that the method should throw an exception only under specific circumstances.

On the other hand, the Javadoc comment of method checkLocked precisely describes its intended exceptional behavior. A human or tool that reads the Javadoc specification, and understands the semantics of natural-language sentences such as "if the comparator is locked", could produce the following test case that embeds line 6 within an oracle:

```
1  void test() {
2    FixedOrderComparator c = new FixedOrderComparator(...);
3    ...
4    c.compare(...);
5    ...
6    if (c.isLocked()) {
7      try {
8        c.checkLocked();
9        fail("Expected exception not thrown");
10     } catch (UnsupportedOperationException e) {
11       // Expected exception!
12     }
13   } else {
14     c.checkLocked();
15   }
16 }
```

The test oracle in this test case correctly checks that checkLocked throws UnsupportedOperationException only when the comparator is locked, reducing both false and missed alarms. Our Toradocu tool produces a test case that is semantically equivalent to this one. The Toradocu-generated test case is textually different because it is implemented using aspects (see Section 3.3).

## 3. TORADOCU

This section explains how Toradocu converts Javadoc comments into test oracles. Toradocu works in three phases that correspond to the three components illustrated in Figure 1:

1. The *Javadoc extractor* (Section 3.1) identifies all the Javadoc comments that are related to exceptional behaviors. The Javadoc extractor outputs a list of the exceptions that each method of the class under test is specified to throw, together with a natural-language description of the conditions under which each exception is intended to be thrown. For instance, referring to the example of Section 2, the Javadoc extractor outputs that method

[1] https://github.com/albertogoffi/toradocu
[2] http://commons.apache.org/proper/commons-collections

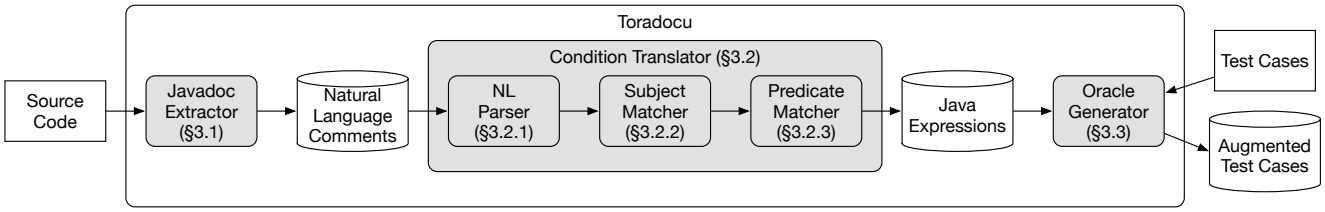


**Figure 1: Architecture of Toradocu. The three components of Toradocu are represented by gray rounded rectangles. The test cases may be human-written or automatically-generated.**

checkLocked should throw an UnsupportedOperationException under the condition "the comparator is locked".

2. The *condition translator* (Section 3.2) translates each natural-language condition into Java boolean expressions. In our example, the condition translator converts "the comparator is locked" into target.isLocked()==true. The variable target in the produced boolean expression is just a placeholder representing the receiver object of the method call.
3. The *oracle generator* (Section 3.3) produces test oracles in the form of assertions and embeds them in the provided test cases. The oracle generator also performs viewpoint adaptation, replacing the target placeholder with the proper variable name that the test case instantiates. For the example presented in Section 2, it replaces target with c.

## 3.1 Javadoc Extractor

Javadoc comments are semi-structured natural-language text blocks that specify a code element (a Java method, field, class, or package). Javadoc comments use tags (keywords starting with @) to structure the documentation. For example, the @param tag marks the specification of a formal parameter, @return marks the specification of the value returned by a method, and @throws and @exception mark a specification of exceptional behavior. An exceptional behavior specification is an exception type *e* plus some natural-language sentences describing when *e* should be thrown by the method.

Toradocu's Javadoc extractor identifies all the comments related to exceptional behaviors for each method of a class under test (CUT). Given the example of Section 2, the Javadoc extractor produces the following information:

**method:** checkLocked()
**exception:** UnsupportedOperationException
**condition:** "*if the comparator is locked*"

We implemented this component as a Javadoc doclet. As a result, it also handles Javadoc comments that are inherited from a superclass and do not directly appear in the source code of the class under test. Our current prototype identifies descriptions tagged with @throws or @exception, and will miss untagged descriptions of exceptional behavior.

## 3.2 Condition Translator

The condition translator uses a mix of natural language processing techniques and pattern-matching to translate Javadoc conditions into Java boolean expressions. For example, the condition translator converts the condition "*if the comparator is locked*" to c.isLocked()==true, converts "*if expectedKeys or expectedValuesPerKey is negative*" to expectedKeys<0 || expectedValuesPerKey<0, and converts "*if both iterator and the element are null*" to iterator==null && element==null. Algorithm 1 details the behavior of the condition translator. The algorithm takes as input a condition as extracted by the Javadoc extractor, and it outputs a list of Java boolean expressions. It works in three phases that are described in the following sections.

### 3.2.1 Natural Language Parsing

The condition translator starts transforming a given Javadoc comment text by invoking function PARSE-INTO-PROPOSITION-GRAPH at line line 4 of Algorithm 1. This function, defined at line 17, relies on the Stanford Parser library [25, 26] to process the natural language Javadoc comment transforming it into a forest of binary trees where each tree represents a single sentence of the original natural language comment. The leaves of each binary tree represent propositions (i.e., pairs of subject and predicate), and the internal nodes connecting such leaves represent either *and* or *or* conjunctions. As an example, the Javadoc comment *"either the iterator or the element are null"* would become a tree where the root node represents the conjunction *"or"*, the left leaf is the proposition with subject *"iterator"*, and the right leaf is the proposition with subject *"element"*. Both subjects would be bound to the same predicate *"are null"*.

Given the Javadoc comment text, the condition translator first uses the Stanford Parser to identify separate sentences (line 19), and to produce a *semantic graph* for each of the sentences (line 20). The semantic graph consists of a parse tree plus additional information, such as grammatical relations.[3]

Next, the condition translator identifies subjects (line 21) and related predicates (line 23). In the NLP community, this step is called *open information extraction* [1, 16]. A subject is a noun phrase that the sentence is about, and the predicate is the remainder of the sentence, which says something about the subject. Given the sentence *"either the iterator or the element are null"*, the identified subjects are *"iterator"* and *"element"*, while the predicate is *"are null"* for both of them. In that predicate, the object of the copular verb *"are"* is *"null"*. The function IDENTIFY-SUBJECTS (line 21) returns all the subjects of a semantic graph. In the case of a compound noun, the function IDENTIFY-SUBJECTS returns the head noun plus the noun compound modifier.

For each identified subject, the function IDENTIFY-PREDICATE (line 23) searches the semantic graph for matches to patterns that commonly appear in Javadoc documentation. Our prototype supports sentences structured as in Table 1. Notice that Toradocu identifies as the predicate only some words of the sentence. Furthermore, the predicate includes a negation modifier when present. This pattern-matching approach is not complete, in the sense that it will not extract meaning from the predicate in every possible English sentence. However, it is effective, and it was adequate in our experiments. Future work could devise a more comprehensive and general approach for interpreting the predicate.

Line 24 shows what the function PARSE-INTO-PROPOSITION-GRAPH produces: a forest of binary trees where each tree represents a single sentence of the original natural language comment.

[3] http://nlp.stanford.edu/software/dependencies_manual.pdf

**Algorithm 1** Condition Translator

```
1: Translates the English condition extracted by the Javadoc extractor into
   a list of Java boolean expressions (one per sentence).
2: INPUT: t (English text describing a condition), m (method commented
   by t)
3: function TRANSLATE-CONDITION(t, m)
4:     propositionGraph := PARSE-INTO-PROPOSITION-GRAPH(t)
5:     for each proposition p = ⟨subj, pred⟩ in propositionGraph do
6:         jExpr := ''
7:         jSubjs := MATCH-SUBJECT(subj, m, m.getClass())
8:         for each subject jSubj in jSubjs do
9:             jPred := MATCH-PREDICATE(pred, jSubj)
10:            jExpr := CREATE-JAVA-EXPR(jSubj, jPred, jExpr, pred)
11:        end for
12:        p.jExpression := jExpr
13:    end for
14:    return IN-ORDER-TRAVERSE(propositionGraph)
15: end function
16: Translates English text into logical formulas, as a forest of binary trees
    (one per sentence). Internal nodes represent "and" and "or".
17: function PARSE-INTO-PROPOSITION-GRAPH(t)
18:    graph := {}
19:    for each sentence s in t do
20:        semGraph := GET-SEMANTIC-GRAPH(s)   // Stanford Parser
21:        subjectList := IDENTIFY-SUBJECTS(semGraph)
22:        for each subject subj in subjectList do
23:            proposition := ⟨subj, IDENTIFY-PREDICATE(subj, semGraph)⟩
24:            graph := graph.add(proposition)
25:        end for
26:    end for
27:    return graph
28: end function
29: Tries to match a string subj, representing a subject, to the most likely
    Java element(s).
30: function MATCH-SUBJECT(subj, m, cut)
31:    jElements := GET-JAVA-ELEMENTS-FOR-SUBJECT(m, cut)
32:    return GET-MOST-SIMILAR-ELEMENTS(subj, jElements)
33: end function
34: Returns Java elements that a subject can potentially match: m's parame-
    ters, cut, and cut's public fields and public nullary non-void methods.
35: function GET-JAVA-ELEMENTS-FOR-SUBJECT(m, cut)
36: Given a set of elements, forms a set of strings that contains the name
    of each element, the name of its type, and the name of each supertype.
    Finds the strings that have minimum Levenshtein edit distance from s.
    Returns each element associated with those strings. Returns multiple
    elements if there is a tie. Returns the empty set if the minimum edit
    distance is greater than 8.
37: function GET-MOST-SIMILAR-ELEMENTS(s, elements)
38: Tries to match a string pred, representing a predicate, to the most likely
    Java element(s).
39: function MATCH-PREDICATE(pred, jSubj)
40:    match := PATTERN-MATCHING(pred)   // see Section 3.2.3
41:    if match ≠ empty then
42:        return match
43:    else
44:        jElements := GET-JAVA-ELEMENTS-FOR-PREDICATE(jSubj)
45:        return GET-MOST-SIMILAR-ELEMENTS(pred, jElements)[0]
46:    end if
47: end function
48: Returns Java elements that a predicate can potentially match: the public
    fields and public nullary non-void methods of jSubj.
49: function GET-JAVA-ELEMENTS-FOR-PREDICATE(jSubj)
50: Returns a new Java expression that is the combination of the existing
    Java expression jExpr and a new expression composed of subject jSubj
    and predicate jPred. This function handles cases where a single subject is
    actually referring to multiple Java elements. Also, this function decides
    whether a condition should be evaluated to either true or false. See
    Section 3.2.4.
51: function CREATE-JAVA-EXPR(jSubj, jPred, jExpr, pred)
```

**Table 1: List of sentence structures from which Toradocu can correctly extract the predicate.**

| Sentence Form | Predicate comprises |
|---|---|
| Copula | Negation modifier, copular verb, complement |
| Active Form | Verb, complement |
| Passive Form | Negation modifier, verb, passive auxiliary |

### 3.2.2 Matching a Subject to a Java Element

Having parsed English text into propositions, where each proposition consists of a subject and a predicate, Toradocu now tries to match each subject and predicate to a Java element in the code, as explained in this section and the next one.

The function MATCH-SUBJECT, called on line 7, tries to match the subject to *identifiers* and *types* that appear in the code (lines 30–33). In particular, a subject can be matched with (i) a formal parameter of the documented method, (ii) a method of the CUT, or (iii) the target object.

For case (ii), the algorithm considers matches with CUT's nullary non-void methods — that is, those that take no arguments and that return a value. Matching the subject with such methods handles cases in which the subject refers to a non-public field of the CUT for which a getter method exists; the test oracle can access the getter method. For instance, a sentence such as *"the capacity of the container"* refers to capacity, which is a private field of the CUT. This field cannot be accessed directly, but only by means of the getter method getCapacity().

Javadoc comments often refer to parameter types, as in the case of *"the collection is empty"*, which refers to a parameter of type java.lang.Collection. For this reason the algorithm considers type names, in addition to identifiers. It considers all supertypes: transitive superclasses and implemented interfaces.

The example introduced in Section 2 is an example of case (iii) and supertype names, since in the sentence "if the comparator is locked" the subject (i.e., "the comparator") refers to the instance of the CUT itself (i.e., FixedOrderComparator), which implements the Comparator interface.

Among all the possible matching candidates, the algorithm picks the Java element with the smallest Levenshtein edit distance (number of character insertions, deletions, and replacements). It returns no candidate if the edit distance is greater than 8; we determined this value experimentally.

### 3.2.3 Matching a Predicate to a Java Element

Given a Java element that matches the subject of the sentence, the condition translator has to identify the Java element that the predicate refers to. Toradocu's main strategy is the same lexical matching approach it takes for subjects (Section 3.2.2). Toradocu takes advantage of the English text that the programmer has embedded in identifier names, and looks for similarities with the English text in the specification.

This approach works when the predicate can be represented by a method call with a camelCase name. However, it does not work when the needed Java expression does not contain English text, as for arithmetic and null comparisons. Therefore, the function MATCH-PREDICATE (lines 39–47) implements two strategies:

- *Textual pattern matching* (lines 40–42): The function PATTERN-MATCHING looks for an exact match of the predicate with a set of predefined patterns that covers common cases (listed in Table 2). For instance, the pattern "*is positive*", which can be applied to numeric types (byte, short, int, long, float, and double), produces the Java predicate *subjectElement*>0. When the subject

**Table 2: List of predicate patterns that Toradocu directly translates.**

| Predicate | Translates to |
|---|---|
| is/are positive | >0 |
| is/are negative | <0 |
| is/are true | ==true |
| is/are false | ==false |
| is/are null | ==null |
| is/are < 1 | <1 |
| is/are <= 0 | <=0 |

element's type is a reference type (a non-primitive type), the only relevant pattern is a check whether an object is null (i.e., "is null" → == null). If pattern matching does not produce any result, MATCH-PREDICATE moves on to lexical matching.

- *Lexical matching* (lines 44–45): MATCH-PREDICATE looks for matching candidates among the public fields and methods of the subject instance. Thus, if the subject refers to the class itself, MATCH-PREDICATE looks at methods declared and inherited by the CUT. If the subject refers to a non-primitive variable (i.e., parameter or field), then the predicate matcher considers the methods in the subject's declared type. For example, the "is locked" predicate would match the isLocked() method that FixedOrderComparator class declares. MATCH-PREDICATE limits its search to methods that return boolean values and take no parameters. Similarly to the logic of the function MATCH-SUBJECT, the matcher selects the method that has the lowest Levenshtein distance to the predicate that is less than 8.

### *3.2.4 Creating the Java Condition*

The last step of Algorithm 1 is to produce the final Java "condition template" (line 10) that the oracle will instantiate.

This includes the decision whether to produce a condition that should be evaluated to either true or false. If the predicate contains the substring "not" or "n't", as in the sentence *"the comparator is not locked"*, the predicate matcher produces a negative condition such as (comparatorInstance.isLocked() == false).

CREATE-JAVA-EXPR also concatenates multiple Java expressions into a single one. Consider the comment *"either array is null"* where subject *"either array"* refers to multiple Java elements (e.g., the formal parameters of the commented method). The function CREATE-JAVA-EXPR concatenates the expressions with a suitable conjunction. If the sentence starts with *"either"*, it uses the conjunction is "||" otherwise it uses "&&".

## 3.3 Oracle Generator

At this point, a Javadoc @throws comment has been translated to a boolean expression by the condition translator. More concretely, the output of the condition translator is a triple ⟨method m, expected exception type e, boolean Java condition c⟩ for a given method m that is supposed to throw exception e when condition c holds.

The oracle generator converts the output of the condition translator into a test oracle and injects the oracle into any test case that invokes the method m. The oracle generator modifies the test case such that, right before each invocation of m, it checks whether c holds or not. If c holds, the oracle expects an exception e as the result of the invocation of m, and makes the test case fail if this does not happen.

Toradocu employs aspect oriented programming and AspectJ[4] to embed oracles in existing test suites.

Given the triple ⟨method m, exception e, condition c⟩, Toradocu automatically generates a custom *aspect* for method m, and later uses this aspect to instrument the bytecode of a provided test suite. As an example, the custom aspect for method FixedOrderComparator.checkLocked() appears below. AspectJ will weave it into the test case and it will execute when the test case does.

```
1  @Around("call(protected void checkLocked())")
2  public Object advice(ProceedingJoinPoint jp) {
3    Object target = jp.getTarget();
4    Object[] args = jp.getArgs();
5    List⟨Class⟩ expectedExcepts = getExpectedExcepts(target, args);
6    if (! expectedExcepts.isEmpty()) {
7      try {
8        jp.proceed(args); // checkLocked() is invoked here
9        fail("Expected exception not thrown");
10     } catch (Throwable e) {
11       if (! expectedExcepts.contains(e.getClass())) {
12         fail("Unexpected exception thrown");
13       } else {
14         return null; // success
15       }
16     }
17   }
18   return jp.proceed(args); // checkLocked() is invoked here
19 }

21 // Returns a list of exceptions that method checkLocked is expected to
22 // throw, based on run-time checks of conditions from its Javadoc.
23 private List⟨Class⟩ getExpectedExcepts(Object target, Object[] args) {
24   List⟨Class⟩ expectedExcepts = new ArrayList⟨Class⟩();
25   if (target.isLocked()) { // condition is checked here
26     expectedExcepts.add(
27       Class.forName("java.lang.UnsupportedOperationException"));
28   }
29   return expectedExcepts;
30 }
```

The Toradocu-generated custom aspect checks whether method checkLocked() behaves correctly according to the described exceptional behavior. At run time, the aspect first builds a list of expectedExceptions (lines 5, 23–30) by performing a run-time check of each boolean Java condition that was output by the condition translator, adding exceptions to the list whose condition currently holds (lines 25–27). In our example there is only 1 @throws clause and at most 1 element in the list. The list of expected exceptions may contain more than 1 element if multiple conditions hold at run time that lead to different exceptions. As an example, consider method foo(Object x, int y) with Javadoc "*@throws NullPointerException if x is null*" and "*@throws IllegalArgumentException if y is negative*". The invocation foo(null, -10) is permitted to raise either of the two exceptions, and the aspect accepts either one as correct behavior.

If Toradocu was not able to derive any information about the exceptional behavior of the method, then getExpectedExcepts returns an empty list. If none of the extracted conditions is currently true, the list of expected exceptions is empty (line 6). In such cases, the method checkLocked() is invoked just as in the original test case (line 18).

When the list of expected exceptions is not empty (line 6), as in our example, the aspect invokes the method at line 8, catching any exception that its execution would generate. If the method correctly raised the expected exception, the aspect masks the exception by returning null (line 14), because a non-void aspect requires a return value. The test case fails if

- no exception is thrown (line 9), or
- the exception type differs from the list of expected exceptions (line 12).

If a method that is not expected to throw any exception throws an exception during its execution, and thus executes line 18, the test case fails, flagging a *likely* problem due to: (i) a missing @throws

[4] http://www.eclipse.org/aspectj

tag in the documentation documenting the thrown exception, or (ii) an error in the implementation that led to the exception. It is also possible that the thrown exception is a permitted behavior that is intentionally not documented.

Using aspects, Toradocu automatically injects test oracles every time method checkLocked() is invoked in a test case. Thus, given a test case such as this one from Section 2:

```
1 void test() {
2   FixedOrderComparator c = new FixedOrderComparator(...);
3   c.compare(...);
4   c.checkLocked();
5 }
```

Toradocu uses AspectJ to weave the aspect into the test case, yielding this final test case:

```
1  void test() {
2    FixedOrderComparator c = new FixedOrderComparator(...);
3    c.compare(...);
4
5    List<Class> expectedExcepts = getExpectedExcepts(c, new Object[0]);
6    if (! expectedExcepts.isEmpty()) {
7      try {
8        c.checkLocked();
9        fail("Expected exception not thrown");
10     } catch (Throwable e) {
11       if (! expectedExcepts.contains(e.getClass())) {
12         fail("Unexpected exception thrown");
13       }
14     }
15   } else {
16     c.checkLocked();
17   }
18 }
19
20 // Returns a list of exceptions that method checkLocked is expected to
21 // throw, based on run-time checks of conditions from its Javadoc.
22 private List<Class> getExpectedExcepts(Object target, Object[] args) {
23   List<Class> expectedExcepts = new ArrayList<Class>;
24   if (target.isLocked()) { // condition is checked here
25     expectedExcepts.add(
26       Class.forName("java.lang.UnsupportedOperationException"));
27   }
28   return expectedExcepts;
29 }
```

Toradocu does not explicitly output the source code of the resulting test cases. Rather, it outputs the source code of the aspects and uses AspectJ to instrument the bytecode of given test cases to add the oracles.

# 4. EXPERIMENTAL EVALUATION

The key contribution of Toradocu is the ability to automatically generate test oracles from Javadoc comments. Combined with test input generation tools such as EvoSuite and Randoop, Toradocu can automatically generate test cases with oracles that reveal faults in exceptional behaviors. Our experiments evaluate the effectiveness of our test oracles when integrated with automatically-generated test inputs. Thus, our experiments show the effectiveness of Toradocu combined with test generators as a uniquely powerful tool to automatically generate complete and accurate test suites for exceptional behavior.

Our evaluation addresses the following research questions:

**RQ1** Do developers test exceptional behavior less than normal behavior?
This research question assesses whether developers tend to overlook exceptional behaviors when testing their software. A positive answer to this question would make Toradocu relevant in practice. Section 4.1 reports the results.

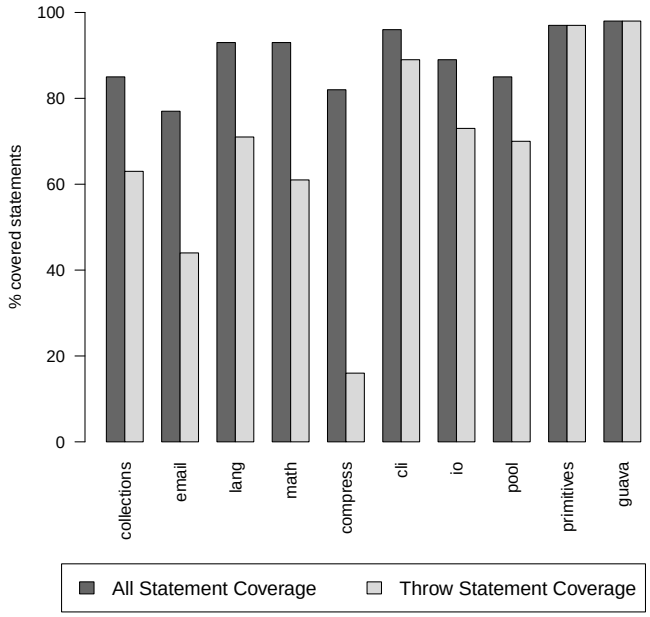


**Figure 2: All statements coverage vs. exceptional statements coverage achieved by the test suites written by the Apache Commons and Google Guava developers.**

**RQ2** To what extent does Toradocu reduce the number of false positives that test input generation tools produce?
Test input generation tools such as EvoSuite and Randoop do not produce test oracles, or they use simple heuristics to classify exceptional behaviors. Depending on the heuristic, they suffer false positives (reporting as a failure an intended behavior) and/or false negatives (considering correct a failing execution). Toradocu reduces the number of such errors. Section 4.2 reports the design and results of this experiment.

**RQ3** Does Toradocu reveal faults in the implementation of exceptional behavior?
Test input generation plus Toradocu has the potential to reveal bugs where developers do not correctly implement the expected exceptional behavior. Section 4.3 describes our experiment in which this approach found defects in student assignment submissions, and in which it found previously-unknown defects in Google Guava, which the developers have acknowledged and fixed.

## 4.1 RQ1: Exceptional Behavior Coverage

We selected 10 popular, well-tested open source libraries (9 Apache Commons projects and Google Guava) to evaluate whether Java developers pay equal attention to exceptional behavior and normal behavior when writing tests.

We ran the developer-written test suites for each project, and Figure 2 plots two coverage metrics:

- *all statement coverage*: The standard quality metric for test suites, this is the number of statements executed at least once by the test suite, divided by the total number of statements. It is a ratio between 0 and 1, inclusive. Higher numbers are better.
- *throw statement coverage*: This ratio is the number of throw statements executed at least once by the test suite, divided by the total number of throw statements. This metric represents the amount of exceptional behavior that test suites cover.

**Table 3: Toradocu's effectiveness in avoiding false positives for selected classes in Google Guava. Specification size is reported as the number of @throws tags in the method and the number of those tags that Toradocu was able to convert into an oracle (number of correct and *c*omplete oracles, number of correct but *i*ncomplete oracles, and oracles that Toradocu was *u*nable to translate; Toradocu never produced an incorrect oracle). The EvoSuite-generated test suites are classified as passing, or failing due to an exception being thrown. Test failures are classified as true positives (defects in the code or the documentation) or false positives (Toradocu weaknesses or illegal inputs generated by EvoSuite).**

| | Spec size | | | | EvoSuite | | | EvoSuite + Toradocu | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Toradocu | | | | Fail | | | Fail | | |
| | | | | | | True | False Pos. | | True | False Pos. | |
| Subject class | @throws | C | I | U | Pass | Pos. | Input | Pass | Pos. | TW | Input |
| ArrayListMultimap | 1 | 1 | 0 | 0 | 3 | 0 | 6 | 4 | 0 | 0 | 5 |
| AtomicDoubleArray | 1 | 1 | 0 | 0 | 20 | 0 | 10 | 21 | 0 | 0 | 9 |
| ConcurrentHashMultiset | 12 | 8 | 1 | 3 | 32 | 0 | 15 | 40 | 0 | 1 | 6 |
| Doubles | 4 | 3 | 1 | 0 | 42 | 0 | 12 | 47 | 0 | 2 | 5 |
| Floats | 4 | 3 | 1 | 0 | 43 | 0 | 18 | 47 | 0 | 2 | 12 |
| MoreObjects | 1 | 1 | 0 | 0 | 17 | 0 | 4 | 18 | 0 | 0 | 3 |
| Shorts | 6 | 3 | 1 | 2 | 22 | 0 | 22 | 30 | 0 | 3 | 11 |
| Strings | 1 | 1 | 0 | 0 | 6 | 0 | 9 | 9 | 0 | 0 | 6 |
| Verify | 4 | 4 | 0 | 0 | 4 | 4 | 1 | 5 | 4 | 0 | 0 |
| Total | 34 | 25 | 4 | 5 | 189 | 4 | 97 | 221 | 4 | 8 | 57 |

The test suites achieve high code coverage: always higher than 75%, and usually higher than 90%. Developers do not pay equal attention to testing exceptional behavior: the coverage of throw statements is never higher than of the overall coverage, and is usually significantly lower, in two cases below 50%.

Exceptional behavior is just as legal and expected as normal behavior, and it should be tested just as well. The results of our empirical evaluation show that our Toradocu technique is needed to augment existing, high-quality manually written test suites too.

## 4.2 RQ2: Reducing False Positives in Test Executions

Test input generation tools such as EvoSuite and Randoop rely on heuristic oracles. For instance, one common heuristic is to consider any runtime exception as a failure without regard for the specification of the code under test. This heuristic results in many false alarms. Toradocu reduces the amount of false alarms in automatically-generated test suites by augmenting automatically-generated test cases with test oracles for exceptional behaviors. We evaluated how well it does so.

### *4.2.1 Methodology*

We selected 9 subject classes from the packages base, collections, and primitives of the Google Guava project (Table 3). We chose these packages because of our familiarity with them, which reduces the effort to manually inspect the results.

We used EvoSuite to generate a test suite for each class under test. We configured EvoSuite to avoid false negatives (missed alarms), by producing test suites without regression oracles. (EvoSuite can generate regression oracles with test suites. In this case, it treats every unchecked exception[5] as a failure and every other exception as expected behavior. These choices lead to both false positives and false negatives; we configured EvoSuite to avoid false negatives or missed alarms, by not treating any exception as expected behavior, as a careful test engineer might do.) We directed EvoSuite to maximize all its search goals with a search budget of 60 seconds.

We then used Toradocu on each class to automatically generate oracles describing the exceptional behavior of each method.

We ran each test suite with and without Toradocu's oracles embedded, and we manually analyzed the failing test cases.

[5]In Java, an "unchecked exception" is one that a client is not required to catch and handle: RuntimeException, Error, and their subclasses.

### *4.2.2 Results*

Table 3 reports the results of our empirical evaluation. Toradocu generates aspects only for those methods that contain or inherit Javadoc comments using the @throws or @exception tags. When a method can throw different exceptions, Toradocu generates one single aspect checking multiple exceptional behaviors of the method.

Out of 290 EvoSuite-generated test executions, 101, or 35%, fail because the test inputs trigger exceptional behaviors that throw exceptions. For classes ArrayListMultimap, Strings, and Verify, EvoSuite generated more failing tests than passing tests. This is because these classes have a large amount of exceptional behaviors and EvoSuite strives to avoid redundant tests. When Toradocu's oracles are embedded in the same test suites, the number of false positives is reduced from 97 to 65, which is an absolute decrease of 11% or a relative decrease of 33%.

We manually examined every test failure. There are three possibilities:

**True positive that reveals a defect in the CUT.** Three test failures are true positives due to missing @throws comments about unchecked exceptions that the code throws. Based on careful reading of the code and documentation, including comparing these methods with other similar methods, we judged that the developers had probably intended that these exceptions should be documented with a @throws comment.

Two examples are that method removeExactly throws IllegalArgumentException if its occurrences parameter is negative, and that method verify throws VerifyException if its argument expression is false.

We reported the 3 new documentation errors to the Guava developers.[6] The Guava developers acknowledged the problems and accepted our fixes to the documentation. The fact that all of these bug reports and fixes were accepted indicates that our categorization of tests was accurate.

One further test, in class Verify, is also a true positive due to an incorrect specification. The test failed Toradocu's oracle due to a null pointer exception thrown by the test code verifyNotNull(null, "dstk", null). Toradocu's oracle expected the invocation to throw a VerifyException instead, because the documentation indicates that if the first parameter is null, the method throws a VerifyException:

[6]https://github.com/google/guava/pull/2099 and https://github.com/google/guava/pull/2106

```
139 /**
152  * @throws VerifyException if reference is null
153  */
155 public static <T> T verifyNotNull(
156     @Nullable T reference,
157     @Nullable String errorMessageTemplate,
158     @Nullable Object... errorMessageArgs) {
```

The erroneous NullPointerException is due to the fact that errorMessageArgs is null, which the specification permits via the @Nullable annotation on that formal parameter.

Although we were not previously aware of this defect in Guava, it had been independently reported.[7] This independent report shows that documentation errors do cause problems and confusion. The Google developers confirmed the documentation bug. We speculate that they have not yet fixed it because of limitations of the FindBugs @Nullable annotation that they are using. Their confirmation of the bug indicated that the intended specification was that errorMessageArgs should be non-null but that its elements may be null. Use of a more expressive tool for specifying and checking nullness [17, 34] would resolve the problem.

**False positive due to a limitation of Toradocu.** The Javadoc comment correctly documents the expected exceptional behavior, but Toradocu could not parse the sentence correctly to generate a Java expression and an aspect. Therefore, the (correct) thrown exception was surfaced as a test failure.

This caused 8 test failures in Toradocu test suites. In class ConcurrentHashMultiset, the method toArray(T[]) throws an exception when "*the runtime type of the specified array is not a supertype of the runtime type of every element in this collection*". Toradocu failed to create the right assertion for this method. Similarly, in classes Floats and Doubles it failed to correctly generate a correct *and complete* aspect from 4 Javadoc comments that look like "*if collection or any of its elements is null*". Toradocu correctly processed the information that when the collection is null the method should throw an exception. However, it failed to generate a complete oracle because it did not check that the method throws an exception when an element of the collection is null.

**False positive due to an unexpected input generated by EvoSuite.** The remaining 57 failing test cases are exceptions that the code under test is allowed, but not required, to throw, given the non-typical inputs that EvoSuite generated.

The Javadoc documentation does not mention any expected exception because the specification is intentionally incomplete. A common reason for this is that the method has precondition contracts that forbid clients from passing a particular parameter value. In such a case, the documentation should *not* specify what exception is thrown if the precondition is violated. For example, a binary search routine requires that its array input is sorted, but does not promise any particular behavior if the array input is not sorted. As an example, EvoSuite generated a test containing the method invocation Doubles.tryParse(null), which threw NullPointerException. As another example, an EvoSuite-generated test passed a very large value to the ArrayListMultimap static constructor create(int, int), then invoked the method createCollection(), which threw OutOfMemoryException.

We used our best judgment in determining whether the specification was intentionally incomplete. Our results are conservative, since this category might contain some cases where the developers truly intended to write a complete specification.

This experiment shows that Toradocu can reduce the number of false positive failures in a test suite, thus making the suite more fruitful for people to examine. Furthermore, Toradocu identified 4 defects in the Google Guava project, 3 of which were previously unknown and have now been fixed in response to our bug reports.

While these initial results are extremely encouraging, we caution that they have been performed on only 9 classes taken from one project. We cannot know whether the results will generalize.

[7] https://github.com/google/guava/issues/1701

**Table 4: Subjects of the evaluation of revealing implementation errors. Column *Methods* includes both methods and constructors.**

| Class | Methods | Implementation snapshots |
|---|---|---|
| collections.map.FilterIterator | 10 | 401 |
| collections.comparators.FixedOrderComparator | 9 | 391 |
| math.genetics.ListPopulation | 13 | 269 |
| collections.map.PredicatedMap | 7 | 283 |

## 4.3 RQ3: Revealing Implementation Errors

Our last experiment evaluated whether Toradocu effectively reveals buggy code: defects in the implementation of exceptional behavior. This experiment assumes a *complete* and *correct* natural-language Javadoc specification. In such a scenario, Toradocu should produce failing test cases when developers do not correctly implement the expected exceptional behavior.

### *4.3.1 Methodology*

Rojas et al. [37] studied how developers use the EvoSuite test input generation tool when developing new software. They asked 41 developers to implement 4 different Java classes (Table 4) starting from their Javadoc documentation. The 4 classes were selected from the Apache Commons Collections and Apache Commons Math libraries. Rojas et al. recorded a snapshot of the implementation whenever the developers chose to run EvoSuite to test their implementation.

For each subject class, we used Toradocu to generate the aspects describing its exceptional behavior. We then used EvoSuite and Randoop to generate test suites for each implementation snapshot of each class.[8]

For Randoop, we used its default heuristic for oracles. Randoop outputs two distinct test suites: a suite of passing tests (the Randoop manual calls these the "regression tests") and a suite of failing tests (the Randoop manual calls these the "error-revealing tests"). Since Randoop uses heuristics to classify tests as passing or failing, the regression tests may contain false negatives (missed alarms), and the error-revealing tests may contain false positive alarms. For EvoSuite, we used the default heuristic: any thrown exception is expected behavior. This avoids false positive alarms but is prone to false negatives (missed alarms).

We ran each test suite with and without Toradocu oracles, and we evaluated the fault-finding effectiveness of the test suites. We measured the fault-finding effectiveness in terms of number of test cases that fail, thus revealing an exception-related error in the implementation.

Similarly to the previous evaluation, we manually analyzed the test executions. Given the large amount of data, we sampled 20 executions for each kind of test outcome (passing, failing because of a runtime exception, and failing because of a failing assertion).

[8] We included Randoop because a recent study found that "debugging was significantly more efficient when Randoop test cases are used [compared to manual test cases]", but "debugging was equally effective with manual and EvoSuite test cases" [9].

**Table 5: Toradocu effectiveness with developers' implementations of a Javadoc specification. Specification size is as in Table 3. Tests are classified as passing, failing due to an exception being thrown (FE), or failing due to a test assertion (FA). Randoop generates two test suites: one containing only passing tests (RandoopR) and one containing only failing tests (RandoopE). The bottom row shows false negative rates for the "Pass" column, and false positive rates for the "FE" and "FA" columns; in both cases, smaller numbers are better. The bottom row was determined by manual inspection of 20 randomly-chosen tests from each cell of the table.**

| | Spec size | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Toradocu | | | EvoSuite | | | EvoSuite+Toradocu | | | RandoopR | | | RandoopR+Toradocu | | | RandoopE | | | RandoopE+Toradocu | | |
| Subject Class | @throws | C | I | U | Pass | FE | FA | Pass | FE | FA | Pass | FE | FA | Pass | FE | FA | Pass | FE | FA | Pass | FE | FA |
| FilterIterator | 4 | 2 | 0 | 2 | 2748 | 3 | 0 | 2325 | 3 | 423 | 17440 | 0 | 0 | 8927 | 0 | 8513 | 0 | 12453 | 0 | 1423 | 7998 | 3032 |
| FixedOrderComparator | 9 | 5 | 0 | 4 | 4045 | 0 | 0 | 3980 | 0 | 65 | 12910 | 0 | 0 | 12670 | 0 | 240 | 0 | 23670 | 0 | 26 | 23382 | 262 |
| ListPopulation | 10 | 2 | 0 | 8 | 3536 | 2 | 0 | 3421 | 2 | 115 | 11614 | 28 | 0 | 11543 | 21 | 78 | 0 | 12693 | 0 | 97 | 12434 | 162 |
| PredicatedMap | 4 | 2 | 0 | 2 | 1968 | 4 | 0 | 1548 | 4 | 420 | 6186 | 0 | 0 | 6076 | 0 | 110 | 0 | 24257 | 0 | 136 | 24084 | 37 |
| **False pos./neg. rate** | | | | | **45%** | **78%** | **n/a** | **40%** | **78%** | **0%** | **35%** | **43%** | **n/a** | **20%** | **24%** | **0%** | **n/a** | **90%** | **n/a** | **0%** | **90%** | **0%** |

### *4.3.2 Evaluation Results*

Table 5 presents the results of our evaluation. All of the aspects generated by Toradocu were correct and complete.

The results show that automatic test case generator tools are rather weak in revealing exception-related implementation errors, due to a lack of proper test oracles. For instance, EvoSuite generates 9 failing test cases, of which only 2 are true positives (2 NullPointerException due to implementation errors for ListPopulation). The other 7 are due to malformed test input (5 ClassCastException and 2 InitializationError).

When we augment the EvoSuite test suites with Toradocu's oracles, the number of test cases failing for a run-time exception remains unchanged, while the number of test cases failing due to an assertion dramatically increases (from 0 to 1023). This is the direct effect of the test oracles we injected in the test suites. We manually inspected 20 randomly sampled tests leading to failures and we confirmed that all of them are true positives: they fail because of a bug in the implementation or because the implementation is entirely missing.

We confirmed the same results with the Randoop test suites (from 0 to 8941 failed assertions and from 0 to 3493, all true positives among the inspected ones).

Overall, Toradocu makes the test suites much more useful to developers. EvoSuite alone generated only 9 failing tests, of which 78% were false positives. EvoSuite+Toradocu generated 1031 failing tests, of which less than 1% were false positives. Randoop's regression test suite contained 28 failing tests, of which 43% were false positives. Randoop+Toradocu eliminated 7 false positives (a reduction from 43% to 24%) and created 8941 true positive tests that revealed errors in the implementations. Randoop's error-revealing test suite contained 90% false positives, but Randoop+Toradocu correctly identified 1682 of them as false positives and identified 3493 of them as true positives, permitting a developer to focus on true positive test failures and to create richer, correct regression test suites.

This evaluation shows that Toradocu improves the effectiveness of a test input generation tool. It reduces the number of false negative test results, thus revealing more true faults. It reduces the number of false positives, thus saving developers' time in inspecting test results.

# 5. RELATED WORK

Our work uses NLP to automatically extract conditional expressions regarding exceptional behavior from Javadoc comments. We see our work as particularly beneficial for test input generation techniques, since they typically lack a means to determine on the success (or failure) of a test execution.

Section 5.1 describes how test input generation tools treat exceptional behaviors. Section 5.2 surveys the related techniques that automatically generate test oracles from sources of information other than natural language documents. Section 5.3 concludes by presenting other techniques that infer specifications, which can sometimes be used as test oracles, from natural-language documents.

## 5.1 Treatment of Exceptions by Test Generation Tools

The state of the art in automated test generation uses heuristics to create oracles for exceptional behavior.

JCrasher [13] and CnC [14] use a heuristic for classifying whether a test that throws an exception should be reported as a possible bug. There are four cases. (1) Throws of java.lang.Error are always ignored, because they indicate a drastic problem. (2) Throws of declared exceptions are always ignored, because they might be part of the contract. (3) Throws of most undeclared exceptions are bugs only if they are thrown by the method under test; throws by transitively-called methods are ignored. (4) Throws of undeclared exceptions IllegalArgumentException, IllegalStateException, and NullPointerException are bugs if they are thrown by some public method that is (transitively) called by the method under test. They are ignored otherwise, including if they are thrown by the method under test itself or by a non-public method that is transitively called. These heuristics suffer both false alarms and missed alarms. JCrasher and CnC also use a heuristic to group exceptions before presenting them to the user — if the backtrace is the same, the tool just shows one of them.

Randoop [31, 32, 36] also uses heuristics to classify each test as passing, failing, or invalid. Passing tests are output as regression tests, failing tests are output as indicating a likely bug in the software under test, and invalid tests are ignored and never shown to the user. When a test throws a: (1) checked exception except as noted below, the test is treated as passing, (2) unchecked exception except as noted below, the test is treated as passing, (3) NullPointerException and no null values were used in the test, the test is treated as failing, (4) NullPointerException and some null value was used in the test, the test is treated as invalid, (5) out-of-memory exception, the test is treated as invalid, or (6) different exceptions on different test executions, the test is treated as invalid due to nondeterminism. (7) Furthermore, a test that violates a contract stated in the JDK documentation (such as reflexivity, symmetry, and transitivity of equals()) is treated as failing. All of the above behaviors can be customized by the user. Randoop's default settings aim to avoid false positive test failures, so as not to annoy the user by reporting too many possible errors in the program, when an exception may have been caused by incorrect usage of the software under test by the randomly-generated test. However, this does not eliminate all false positives, and it means that Randoop suffers false negatives — it fails to report some real errors that its test inputs have revealed.

The faults identified by JTest [35] "are usually ... illegal arguments or special object states" [51], where the latter probably means a violated precondition, such as calling pop() on an empty stack. JTest suppresses warnings about any exception documented with the @exception comment tag; that tag does not indicate conditions, just the exception name.

EvoSuite [19, 21, 46] is intended to produce regression tests, not to detect errors. Thus, by default EvoSuite assumes any exception thrown is the correct, intended behavior. This behavior can be overridden by the user.

Other test generators do not use any heuristic to treat exceptional behaviors. Unlike EvoSuite, they report every undeclared exception to the user, thus including many false alarms [5, 23, 40].

All the aforementioned tools and corresponding techniques are orthogonal to the contributions presented in this paper. Integrating Toradocu with any of these tools would improve them.

## 5.2 Automatic Generation of Test Oracles

Baresi et al. [6] and Barr et al. [7] wrote surveys on automatic test oracle generation techniques.

Many such techniques can automatically generate test oracles starting from formal specifications, which can be in the form of algebraic specifications [2, 22], assertions [3, 11, 28, 38, 45], Z specifications [27, 29], context-free grammars [15], JML annotations [12], and finite state machines [20], among others. However, formal specifications are seldom available, and this limits the applicability of these techniques in practice. By contrast, Toradocu relies on Javadoc comments, a form of specification that is frequently available in Java projects.

Metamorphic testing generates oracles by comparing two operations that are supposed to have the same behavior [8, 10, 18, 24, 30]. The effectiveness of these oracles is limited to cases where such symmetric properties exist.

Toradocu is currently limited to deal with exceptional behavior, but can potentially be extended to other cases. Moreover, Toradocu is likely to complement techniques that generate test oracles from other sources of information.

## 5.3 Using NLP to Extract Specifications

Toradocu uses NLP techniques and heuristics to extract test oracles from natural language comments. Other works use similar techniques for similar purposes.

A number of works use NLP to extract specifications from natural language documents. Wu et al. parse Web services documentation to infer dependency constraints on service parameters [49]. Xiao et al. extract security policies [50]. Wong et al. extract input constraints from program documents to provide a better guidance during symbolic execution [48]. Zhong et al.'s Doc2Spec analyzes Javadoc comments to infer automata describing the resource-related behavior of Java programs [54]. Toradocu uses similar techniques to the ones that these works employ, but for a different purpose.

Pandita et al. proposed an approach to automatically extract function contracts from natural language documents [33]. Their work extracts information that could also be useful to generate test oracles, but they did not use it for this purpose.

Natural language processing techniques have also been used to automatically detect inconsistencies between code and corresponding comments. For example, iComment [42] and aComment [43] derive rules about specific properties from code comments using both NLP and heuristics. Then they statically check the extracted rules to check whether there is any inconsistency between comments and code. DocRef combines NLP with island parsing to highlight inconsistencies between code and API documentation, thus revealing likely bugs in the documentation [53]. Rubio-González and Liblit exploit static analysis to collect all the return error codes of system calls in Linux. Then they compare this information with the documentation of the Linux manual to find undocumented error codes. Their analysis revealed over 1700 inconsistencies looking only at file-related system calls [39]. Although these techniques have similar goals as Toradocu, they do not produce test oracles.

Bacherler et al. [4] designed a technique to automatically generate test cases out of natural language documents. Their technique works under the assumption that the documents use a restricted, formalized English vocabulary. Toradocu is more flexible since it does not have this assumption.

Zhang et al. exploit NLP techniques to generate test inputs and a small set of related assertions from a set of test names provided by the developer [52]. Since Toradocu generates test oracles for arbitrary test inputs, the two techniques address complementary goals.

tComment by Tan et al. is the work that is most closely related to ours [44]. Similar to Toradocu, they aim to generate test cases with oracles derived from the Javadoc comments of the class under test. tComment, however, resolves only exact pattern matchings between Java elements in the code and names in the documentation, and handles only null value checking.

# 6. CONCLUSIONS

This paper presented Toradocu, a technique to automatically generate test oracles for exceptional behavior. It works from Javadoc comments, which are informal documentation that programmers are already accustomed to writing. Toradocu uses natural language processing techniques to identify subjects and predicates in comments, and then maps them to concepts in the source code, thus creating conditional expressions that describe the programmer-intended behavior. In combination with an automatic test input generation tool, Toradocu creates test suites with oracles that are able to detect errors in error-handling code in Java programs.

We performed an empirical evaluation that shows that developers under-test exceptional behavior, motivating the need for a technique that can automatically generate test cases for exceptional cases, even when manually-written test suites already exist.

Our evaluation shows that Toradocu reduces the number of false alarms that automatically-generated test cases typically report, and at the same time it is effective at detecting real faults in implementations. Toradocu identified 4 defects in the Google Guava documentation, 3 of them previously unknown.

Toradocu is just a first prototype of what we envision as a powerful and general tool to generate test oracles. In the future we plan to improve the abilities of the NLP component to resolve more complex conditions. Moreover, we also plan to look at other parts of the Javadoc comments besides @throws tags.

# 7. ACKNOWLEDGMENTS

This work was supported in part by the Swiss National Science Foundation with the project ReSpec (grant n. 146607), by the European Union FP7-PEOPLE-COFUND project AMAROUT II (grant n. 291803), by the Spanish Ministry of Economy project DEDETIS, and by the Madrid Regional Government project N-Greens Software (grant n. S2013/ICE-2731). This material is based on research sponsored by DARPA under agreement numbers FA8750-12-2-0107, FA8750-15-C-0010, and FA8750-16-2-0032. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

# 8. REFERENCES


[1] G. Angeli, M. J. J. Premkumar, and C. D. Manning. Leveraging linguistic structure for open domain information extraction. In *ACL 2015, Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics*, pages 344–354, 2015.

[2] S. Antoy and D. Hamlet. Automatically checking an implementation against its formal specification. *IEEE Transactions on Software Engineering*, 26(1):55–69, 2000.

[3] W. Araujo, L. C. Briand, and Y. Labiche. Enabling the runtime assertion checking of concurrent contracts for the Java modeling language. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 786–795, 2011.

[4] C. Bacherler, B. Moszkowski, C. Facchi, and A. Huebner. Automated test code generation based on formalized natural language business rules. In *ICSEA'12, Proceedings of the 7th International Conference on Software Engineering Advances*, pages 165–171, 2012.

[5] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: An evolutionary test approach for Java. In *ICST'10, Proceedings of the 3rd International Conference on Software Testing, Verification and Validation*, pages 185–194, 2010.

[6] L. Baresi and M. Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Department of Computer and Information Science, 2001.

[7] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, May 2015.

[8] A. Carzaniga, A. Goffi, A. Gorla, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE'14, Proceedings of the 36th International Conference on Software Engineering*, pages 931–942, 2014.

[9] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. Do automatically generated test cases make debugging easier? An experimental assessment of debugging effectiveness and efficiency. *ACM Transactions on Programming Languages and Systems*, 25(1):5:1–5:38, December 2015.

[10] T. Y. Chen, F.-C. Kuo, T. H. Tse, and Z. Q. Zhou. Metamorphic testing and beyond. In *STEP'03, Proceedings of the 11th International Workshop on Software Technology and Engineering Practice*, pages 94–100, 2003.

[11] Y. Cheon. Abstraction in assertion-based test oracles. In *QSIC'07, Proceedings of the 7th International Conference on Quality Software*, pages 410–414, 2007.

[12] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002 — Object-Oriented Programming, 16th European Conference*, pages 231–255, 2002.

[13] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, September 2004.

[14] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.

[15] J. D. Day and J. D. Gannon. A test oracle based on formal specifications. In *SOFTAIR'85, Proceedings of the 2nd Conference on Software Development Tools, Techniques, and Alternatives*, pages 126–130, 1985.

[16] L. Del Corro and R. Gemulla. Clausie: Clause-based open information extraction. In *WWW 2013, Proceedings of the 22nd International World Wide Web Conference*, pages 355–366, 2013.

[17] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, Waikiki, Hawaii, USA, May 25–27, 2011.

[18] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology*, 3(2):101–130, 1994.

[19] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, March–April 2012.

[20] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[21] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *ISSRE'13, Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 360–369, 2013.

[22] J. Gannon, P. McMullin, and R. Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

[23] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 13–15, 2005.

[24] A. Gotlieb. Exploiting symmetries to test programs. In *ISSRE'03, Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 365–375, 2003.

[25] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

[26] M. Marneffe, B. Maccartney, and C. Manning. Generating typed dependency parses from phrase structure parses. In *LREC'06, Proceedings of the workshop on Cross-Framework and Cross-Domain Parser Evaluation*, pages 449–454, 2006.

[27] J. Mcdonald. Translating Object-Z specifications to passive test oracles. In *ICFEM'98, Proceedings of the 1998 International Conference on Formal Engineering Methods*, pages 165–174, 1998.

[28] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1st edition, 1988.

[29] E. Mikk. Compilation of Z specifications into C for automatic test result evaluation. In *ZUM'95, Proceedings of the 9th International Conference of Z Users*, pages 167–180, 1995.

[30] C. Murphy, G. Kaiser, I. Vo, and M. Chu. Quality assurance of software applications using the in vivo testing approach. In *ICST'09, Proceedings of the 2nd International Conference on Software Testing, Verification and Validation*, pages 111–120, 2009.

[31] C. Pacheco and M. D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

[32] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *ICSE'07,*

*Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[33] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language API descriptions. In *ICSE'12, Proceedings of the 34th International Conference on Software Engineering*, pages 815–825, Zurich, Switzerland, 2012.

[34] M. M. Papi, M. Ali, T. L. Correa Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[35] Parasoft Corporation. *Jtest version 4.5*. http://www.parasoft.com/.

[36] Randoop Developers. Randoop manual. https://randoop.github.io/randoop/manual/, January 2016. Version 2.1.1.

[37] J. M. Rojas, G. Fraser, and A. Arcuri. Automated unit test generation during software development: A controlled experiment and think-aloud observations. In *ISSTA 2015, Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 338–349, 2015.

[38] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[39] C. Rubio-González and B. Liblit. Expect the unexpected: error code mismatches between documentation and the real world. In *PASTE'10, Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 73–80, 2010.

[40] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, September 7–9, 2005.

[41] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 201–211, Lincoln, NE, USA, November 11–13, 2015.

[42] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /*iComment: Bugs or bad comments?*/. In *SOSP 2007, Proceedings of the 21st ACM Symposium on Operating Systems Principles*, pages 145–158, Stevenson, WA, USA, October 14–17, 2007.

[43] L. Tan, Y. Zhou, and Y. Padioleau. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs. In *ICSE'11, Proceedings of the 33rd International Conference on Software Engineering*, pages 11–20, 2011.

[44] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Fifth International Conference on Software Testing, Verification and Validation (ICST)*, pages 260–269, Montreal, Canada, April 18–20, 2012.

[45] R. N. Taylor. An integrated verification and testing environment. *Software: Practice and Experience*, 13(8):697–713, 1983.

[46] M. Vivanti, A. Mis, A. Gorla, and G. Fraser. Search-based data-flow test generation. In *ISSRE'13, Proceedings of the IEEE International Symposium on Software Reliability Engineering*, pages 370–379. IEEE, 2013.

[47] W. Weimer and G. C. Necula. Finding and preventing run-time error handling mistakes. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 419–431, Vancouver, BC, Canada, 2004.

[48] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan. Dase: Document-assisted symbolic execution for improving automated software testing. In *ICSE'15, Proceedings of the 37th International Conference on Software Engineering*, pages 620–631, Florence, Italy, 2015.

[49] Q. Wu, L. Wu, G. Liang, Q. Wang, T. Xie, and H. Mei. Inferring dependency constraints on parameters for web services. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 1421–1432, Rio de Janeiro, Brazil, 2013.

[50] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *FSE 2012, Proceedings of the ACM SIGSOFT 20th Symposium on the Foundations of Software Engineering*, pages 12:1–12:11, Cary, North Carolina, 2012.

[51] T. Xie and D. Notkin. Tool-assisted unit test selection based on operational violations. In *ASE 2003: Proceedings of the 18th Annual International Conference on Automated Software Engineering*, pages 40–48, Montreal, Canada, October 8–10, 2003.

[52] B. Zhang, E. Hill, and J. Clause. Automatically generating test templates from test names. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*, pages 506–511, Lincoln, NE, USA, November 11–13, 2015.

[53] H. Zhong and Z. Su. Detecting API documentation errors. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2013)*, pages 803–816, Indianapolis, Indiana, USA, 2013.

[54] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *ASE 2009: Proceedings of the 24th Annual International Conference on Automated Software Engineering*, pages 307–318, Washington, DC, USA, 2009.