Translating Code Comments to Procedure Specifications

Arianna Blasi
USI Università della Svizzera italiana
Switzerland

Alberto Goffi
USI Università della Svizzera italiana
Switzerland

Konstantin Kuznetsov
Saarland University / CISPA
Germany

Alessandra Gorla
IMDEA Software Institute
Spain

Michael D. Ernst
University of Washington
USA

Mauro Pezzè
USI Università della Svizzera italiana
Switzerland

Sergio Delgado Castellanos
IMDEA Software Institute
Spain

ABSTRACT

Procedure specifications are useful in many software development tasks. As one example, in automatic test case generation they can guide testing, act as test oracles able to reveal bugs, and identify illegal inputs. Whereas formal specifications are seldom available in practice, it is standard practice for developers to document their code with semi-structured comments. These comments express the procedure specification with a mix of predefined tags and natural language. This paper presents Jdoctor, an approach that combines pattern, lexical, and semantic matching to translate Javadoc comments into executable procedure specifications written as Java expressions. In an empirical evaluation, Jdoctor achieved precision of 92% and recall of 83% in translating Javadoc into procedure specifications. We also supplied the Jdoctor-derived specifications to an automated test case generation tool, Randoop. The specifications enabled Randoop to generate test cases that produce fewer false alarms and reveal more defects.

CCS CONCEPTS

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

KEYWORDS

Specification inference, natural language processing, software testing, automatic test case generation, test oracle generation

ACM Reference Format:
Arianna Blasi, Alberto Goffi, Konstantin Kuznetsov, Alessandra Gorla, Michael D. Ernst, Mauro Pezzè, and Sergio Delgado Castellanos. 2018. Translating Code Comments to Procedure Specifications. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3213846.3213872

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3213872

1 INTRODUCTION

A program specification expresses intended program behavior, and thereby it enables or automates many software engineering tasks. In software testing, it acts as an oracle to determine which inputs are legal and which outputs are correct [4, 17, 31, 37]. In debugging, it identifies faulty statements [29, 54]. In code synthesis, it is the goal that the synthesizer works toward [28, 47]. In refactoring, it ensures that transformations are consistent [23]. In formal verification, it discriminates correct and faulty implementations [53]. In runtime monitoring, it identifies anomalous behaviors [15]. Automating (parts of) these tasks requires a machine-readable format that tools can manipulate. A formal specification serves this purpose well. However, formal specifications are rarely available, because writing them is not a common software development practice.

By contrast, *informal* specifications are readily available as semi-structured and unstructured documents written in natural language. It is standard practice for programmers to specify preconditions, postconditions, and exceptional behaviors in procedure documentation. The Javadoc markup language and tool appeared in Java's first release in 1995, and Doxygen, a similar but cross-language tool, appeared 2 years later. Java IDEs automatically insert templates for Javadoc comments. Most importantly, programmers have become accustomed to writing these comments. As a result, significant amount of code contains an informal Javadoc specification. However, software engineering tools make limited use of these informal specifications.

This paper presents Jdoctor, a technique to *automatically construct executable procedure specifications from artifacts that programmers already create*, namely Javadoc code comments, without requiring programmers to change their development practice or to do extra work. An executable specification is one that can be executed, for example because it is written in a programming language rather than in some other logic; it also needs to be expressed procedurally, rather than (say) declaratively requiring the existence of some value without indicating how to compute it. The procedure specifications generated by Jdoctor can be used in multiple software engineering tasks, for instance to automate the generation of test cases, as shown in this paper.

1.1 Application: Test Case Generation

Automated test case generation can reduce both development costs and the societal cost of software failures [5, 27, 44]. To generate a test case automatically, a test generator must produce an *input* that causes the program under test to perform some actions, and an *oracle* that determines whether the program behaves correctly.

Creating accurate test oracles remains an open problem. This leads test generators to suffer both false alarms (tests failing when the program under test is correct) and missed alarms (tests passing when the program under test is buggy), as we now explain. As an example, suppose that a method call in an automatically-generated test throws an exception. This thrown exception does not necessarily mean that the tested method is buggy. There are the following possibilities:

(1) The thrown exception actually reveals an implementation defect. An example is triggering an assertion in the subject program or otherwise failing to complete the requested operation.
(2) The thrown exception is expected, desired behavior. An example is an `IllegalOperationException` when calling a mutator on an immutable object.
(3) The thrown exception is permitted, but not required, behavior. For example, a binary search routine has undefined behavior if its argument array is not sorted. It is allowed to throw any exception or to return a result that is not consistent with whether the searched-for element is in the array.

In the absence of a specification, test generation tools can use heuristics to guess whether a given behavior is correct or incorrect. One heuristic is to consider as correct a method execution that terminates with a `NullPointerException` if one of the method's arguments was `null`, regardless of the specific requirements that may or may not consider this as an acceptable behavior [40]. Another heuristic is to use regression oracles, which deem as correct the behavior exposed with a previous version of the software under test, and as wrong any other behavior, even if the other behavior is also acceptable to the software designer [19].

These guesses lead to *false positives* or false alarms, in which the tool reports that a method has failed when in fact it behaved as specified or it was called with illegal inputs. The guesses also lead to *false negatives* or missed alarms, in which the tool fails to report a test that exposes an incorrect behavior. This can happen when the call should throw an exception but fails to, or when the call throws an exception but the tool heuristically disregards it in order to avoid false alarms.

We propose that test generators should rely on programmer-written informal specifications by exploiting the executable procedural specifications automatically generated with Jdoctor. The executable procedural specification can act as an oracle making automatically generated test cases more effective, and can avoid the generation of invalid test cases that derive from error-prone heuristics, as described in Section 6.

1.2 Contributions

The main contribution of the research work documented in this paper is Jdoctor, an approach to convert informal Javadoc code comments into executable procedure specifications. This approach is a novel combination of natural language parsing and pattern, lexical, and semantic matching techniques. A second contribution is an open-source implementation of the approach, which enables the replication of the experiments referred to in this paper and the execution of additional ones. A third contribution is multiple experimental evaluations of the approach that confirm its accuracy and usefulness reducing the number of false alarms reported by automatically generated tests.

We experimentally evaluated the accuracy of Jdoctor in translating Javadoc comments into procedure specifications. Jdoctor significantly outperforms other state-of-the-art approaches. Then, we showed that Jdoctor is not just accurate, but also useful in the domain of test generation. When integrated with a test generation tool (Randoop), Jdoctor's specifications reduce false positive alarms.

2 CODE COMMENT ANALYSIS

Our work draws inspiration and reuses ideas from code comment analysis. The closest work related to our technique of analyzing code comments is @tComment [50], ALICS [41], and Toradocu [24]. @tComment uses pattern-matching to determine three kinds of precondition properties related to nullness of parameters. It achieves high precision and recall for comments that match the patterns, but those patterns are quite narrow and do not generalize. ALICS generates procedure pre- and postconditions from code comments, using part-of-speech tagging and then pattern-matching against a small set of hard-coded nouns and jargon. Again, generalizability is not obvious at all, and would require manual extension for each new domain. Toradocu is our own initial work which resulted in Jdoctor. It uses natural language parsing, and matches identified nouns and verbs with arbitrary program expressions and operations using approximate lexicographic matching. Toradocu only works for exceptional conditions, which account for a fraction of Javadoc comments. Both @tComment and Toradocu had their extracted properties applied to the problem of test generation; ALICS has not been applied to any development task.

Jdoctor aims to combine and properly extend the best of these techniques. Pattern-matching cannot capture the expressivity and variety of natural language in Javadoc tags, not even if evaluated on just the programs in our case studies. Jdoctor complements pattern matching with natural language parsing, as do ALICS and Toradocu. However, unlike previous work, Jdoctor is not restricted to a small grammar of specific comments or specifications, but can express specifications in terms of abstractions defined in the program being analyzed. Unlike all previous work, Jdoctor adds a novel notion of semantic similarities. This handles comments that use terms that differ, despite being semantically related, from identifiers in code.

In a nutshell, Jdoctor produces specifications for all sorts of procedure behavior: preconditions (unlike Toradocu), normal postconditions (unlike @tComment and Toradocu), and exceptional postconditions (unlike @tComment), and is more general than previous work. Unlike @tComment, Jdoctor does not require the program to conform to specific behavior for illegal inputs (for null-related behaviors, in the case of @tComment). Unlike @tComment and ALICS, Jdoctor is not restricted to a small grammar of specific comments or specifications, but can express specifications in terms of abstractions defined in the program being analyzed. Jdoctor incorporates textual pattern-matching and natural language processing

(unlike @tComment), and introduces new techniques that are more sophisticated and effective than ALICS and Toradocu. Unlike ALICS, Jdoctor specifications are executable and involve data structures, like arrays and collections, and mathematical expressions.

3 MOTIVATING JAVADOC EXAMPLES

It is standard practice for Java developers to annotate code with informal specifications in the form of Javadoc comments. The Javadoc tool automatically generates API documentation in HTML format from such comments. Javadoc comments consist of free-form text, some of which is preceded by "tags": the `@param` tag for preconditions and the `@return` and `@throws` tags for regular postconditions and exceptional behaviors respectively.[1] We now present some examples of Javadoc comments taken from popular open-source Java code, together with the output that Jdoctor produces, to highlight the challenges of this work and the limitations of previous work.

3.1 Preconditions

`@param` tags characterize method parameters and state the preconditions that callers must respect. Consider the following comment, taken from the `BloomFilter` class of Google Guava. Jdoctor transforms this comment into the executable specification that is shown in the box below the Javadoc comment. The clauses are conjoined with the Java conditional operator "and" (`&&`) to form the complete procedure specification.

```
1  /**
2   * @param funnel the funnel of T's that the constructed {@code
3   *     BloomFilter<T>} will use
4   * @param expectedInsertions the number of expected insertions to the
5   *     constructed {@code BloomFilter<T>}; must be positive
6   * @param fpp the desired false positive probability (must be positive
7   *     and less than 1.0)
8   */
9  public static <T> BloomFilter<T> create(
10   Funnel<? super T> funnel,
11   int expectedInsertions,
12   double fpp) { ... }
```

expectedInsertions > 0
fpp > 0 && fpp < 1.0

Jdoctor correctly handles comments using math expressions (second and third `@param` comment) and compound conditions (third `@param` comment). Jdoctor also understands that the comment regarding the first parameter does not specify any precondition and thus does not produce any specification regarding parameter `funnel`.

3.2 Exceptional Postconditions

`@throws` and `@exception` tags represent postconditions of exceptional executions. Let us consider the following excerpt from the class `ClosureUtils` of the Apache Commons Collections library.

```
1  /**
2   * @throws NullPointerException if the closures array is null
3   * @throws NullPointerException if any closure in the array is null
4   */
5  public static <E> Closure<E> chainedClosure(
6    final Closure<? super E>... closures) { ··· }
```

[1] http://www.oracle.com/technetwork/articles/java/index-137868.html; `@exception` is equivalent to `@throws`.

closures == null ⟶ java.lang.NullPointerException
java.util.Arrays.stream(closures).anyMatch(e -> e == null) ⟶
java.lang.NullPointerException

The first exceptional postcondition is straightforward, and the state of the art can handle it. However, only Jdoctor understands properties related to elements in containers, as in the second `@throws` comment.

Jdoctor also handles more complex comments such as the following, which comes from the `CollectionUtils` class of the Apache Commons Collections library:

```
1  /**
2   * @throws NullPointerException if either collection or the comparator is null
3   */
4  public static <O> List<O> collate(
5    Iterable<? extends O> a,
6    Iterable<? extends O> b,
7    Comparator<? super O> c) { ··· }
```

(a == null || b == null || c == null) ⟶ java.lang.NullPointerException

Although this comment describes null conditions, the state of the art cannot produce a correct assertion. Jdoctor determines that "either collection" refers to parameters `a` and `b`, and "the comparator" refers to parameter `c`.

3.3 Normal Postconditions

`@return` tags represent postconditions of regular executions of methods and are the most varied and therefore the most challenging comments. Here are three examples that no other technique can handle.

The first example is from class `BagUtils` in the Apache Commons Collections library:

```
1  /** @return an empty Bag */
2  Bag emptyBag()
```

true ⟶ result.equals(Bag.EMPTY_BAG)

Jdoctor produces a postcondition using a constant declared in class `Bag`. Matching code comments to methods or other code elements in the code under test is not always this straightforward, as exemplified by the following comment for method `Graph.addEdge()` of the `JGraphT` library.

```
1  /** @return true if this graph did not already contain the specified edge */
2  boolean addEdge(V sourceVertex, V targetVertex, E e)
```

!this.containsEdge(sourceVertex, targetVertex) ⟶ result == true

Jdoctor infers that "this graph" refers to the graph instance itself, and that method `containsEdge` can check the postcondition. Jdoctor correctly passes the two vertexes as parameters of this method to form an edge.

Matching code comments to code elements may also require some notion of *semantic similarity*. Take the following example from the same `Graph` class.

```
1  /** @throws NullPointerException if vertex is not found in the graph */
2  Set edgesOf(Object vertex)
```

this.contains(vertex)==false ⟶ java.lang.NullPointerException

Jdoctor infers that "not found" is semantically related to the concept of "an element being contained in a container", thanks to Jdoctor's novel semantic similarity analysis. @tComment and

Toradocu lack any semantic similarity analysis, while ALICS only supports a limited list of manually-defined synonyms.

4 JDOCTOR

Jdoctor[2] translates Javadoc comments related to constructors and methods into executable Java expressions. Its key insight is the observation that nouns in a natural language comment tend to correspond with variables or expressions in the code, and verbs/predicates in a comment correspond with operations or methods. Jdoctor handles Javadoc preconditions (`@param`), normal postconditions (`@return`), and exceptional postconditions (`@throws` or `@exception`).

Jdoctor works in four steps:

(1) *Text normalization* (§4.1): Jdoctor preprocesses the text in the Javadoc `@param`, `@return`, `@throws`, and `@exception` block tags to prepare for the analysis of the natural language. This phase includes several text transformations to facilitate the next steps.
(2) *Proposition identification* (§4.2): Jdoctor uses a natural language parser to identify the propositions (subject–predicate pairs) of each clause in the comment.
(3) *Proposition translation* (§4.3): Jdoctor matches each identified proposition to a Java element, such as an expression or operation. This step is the core of Jdoctor, and relies on a combination of pattern, lexical, and semantic similarity matching.
(4) *Specification creation*: Jdoctor creates Java boolean expressions that encode the natural language Javadoc comment. It replaces each subject and predicate in the text with Java code and traverses the parse tree to create legal Java code, including method calls, operations, and boolean connectives (from grammatical conjunctions).

4.1 Text Normalization

Javadoc comments are rarely complete grammatical English sentences. For example, they often lack punctuation, have implicit subjects and/or verbs, and intermix mathematical notation with English. Current NLP parsers cannot always handle the style of Javadoc comments as written by programmers. Jdoctor makes the text parseable by preprocessing it into a grammatical English sentence before using NLP parsers. This phase properly extends previous work and allows to deal with many more Javadoc comments.

Punctuation: Jdoctor adds a terminating period when it is absent. It also removes spurious initial punctuation, which stems from programmers (incorrectly) using commas in Javadoc comments to separate parameter or exception names from their descriptions.

Implicit subject: Comments may refer to a subject that was previously mentioned. For instance, a typical `@param` comment is "Will never be null." Since Jdoctor parses sentences in isolation, each sentence needs an explicit subject. For `@param` comments Jdoctor adds the parameter name at the beginning of the comment text. Jdoctor also heuristically resolves pronouns such as "*it*", replacing them with the last-used noun in the comment.

Implicit verb: Some comments have implicit verbs, such as "`@param` num, a positive number". Jdoctor adds "is" or "are" depending on whether the first noun, which is assumed to be the subject, is singular or plural.

Incomplete sentences: Jdoctor transforms dependent clauses into main clauses when no main clause exists.

Vocabulary standardization: To accommodate later pattern-matching, Jdoctor standardizes text relating to nullness, if, and empty patterns. For example, Jdoctor standardizes "non-null" and "nonnull" to "not null".

Mathematical notation: Jdoctor transforms inequalities to placeholders that can be parsed as an adjective. For instance, Jdoctor transforms the clause `if {@code e} < 0` into the expression `e < 0`, and then into `e is LT0`.

4.2 Proposition Identification

Given an English sentence, Jdoctor identifies ⟨subject,predicate⟩ pairs, also called *propositions* [14], and conjunctions or disjunctions that connect propositions (if any). Extracting ⟨subject,predicate⟩ pairs from natural language sentences is referred to as Open Information Extraction (OIE) [1, 14, 18, 46].

Jdoctor first performs *partial POS tagging* [41]. It marks parameter names as nouns and inequality placeholders such as `LT0` as adjectives. Jdoctor completes the POS tagging process by means of the Stanford Parser[3] [34], which produces a semantic graph (an enriched parse tree) representing the input sentence. The nodes of the semantic graph correspond to the words of the sentence, and the edges correspond to grammatical relations between words.[4]

Jdoctor identifies the words that comprise the *subject* and the ones that comprise the *predicate*, based on the sentence structure and the grammatical roles encoded in the graph. More precisely, given the single node (i.e., word) marked as subject in the semantic graph, Jdoctor identifies the complete subject phrase by visiting the subgraph with the subject node as root node and collecting all the words involved in a relation of type compound, adverbial modifier, adjectival modifier, determiner, and nominal modifier. This is substantially different from ALICS [41], which only uses the ready-to-use POS tagging provided by the Stanford Parser, thus missing information from all the other words. Jdoctor identifies the predicates by collecting words with the following grammatical relations: auxiliary, copula, conjunct, direct object, open clausal complement, and adjectival, negation, numeric modifiers.

Jdoctor extracts one proposition from simple sentences and multiple propositions from multi-clause sentences. By processing dedicated edges for grammatical conjunctions like "and" and "or" in the semantic graph, Jdoctor correctly supports multi-clause sentences. While traversing the graph, Jdoctor identifies propositions interconnected with the proper boolean conjunctions or disjunctions that reflect the grammatical conjunctions in the input sentence.

4.3 Proposition Translation

Jdoctor translates each proposition by consecutively applying complementary heuristics of pattern matching (Section 4.3.1) and lexical

[2] Jdoctor is open-source and publicly available: https://github.com/albertogoffi/toradocu/releases/tag/v3.0

[3] http://nlp.stanford.edu/software/lex-parser.html
[4] http://universaldependencies.org/u/dep/all.html

Algorithm 1 Comment Translation

```
1:  /** Translate comment text into Java expressions. Given the English text
    and the list of propositions, the function matches each part. */
2:  function TRANSLATE(set of propositions)
3:      if return-comment then
4:          IDENTIFY-GUARD-AND-PROPERTIES(set of propositions)
5:          MATCH-PROPOSITION(proposition-in-guard)
6:          MATCH-PROPOSITION(proposition-in-trueProperty)
7:          MATCH-PROPOSITION(proposition-in-falseProperty)
8:      else
9:          for all proposition ∈ text do
10:             MATCH-PROPOSITION(proposition)
11:         end for
12:     end if
13: end function

14: /** Given a proposition (i.e. a pair of subject and predicate), find code el-
    ements to match subject and predicate. */
15: function MATCH-PROPOSITION(proposition)
16:     subjCandidateList = GET-SUBJECT-CANDIDATES(subject)
17:     matchedSubject = LEXICAL-MATCH(subject, subjCandidateList)
18:     if no match for subject then
19:         return
20:     end if
21:     predCandsList = GET-PREDICATE-CANDIDATES(predicate)
22:     matchedPredicate = PATTERN-MATCHING(predicate, predCandsList)
23:     if no match for predicate then
24:         matchedPredicate = LEXICAL-MATCH(predicate, predCandsList)
25:     end if
26:     if no match for predicate then
27:         matchedPredicate = SEMANTIC-MATCH(predicate, predCandsList)
28:     end if
29: end function
```

matching (Section 4.3.2) with semantic similarity analysis (Section 4.3.3). Jdoctor uses the first of these approaches that succeeds.

This section describes the translation of a single proposition. Jdoctor handles multiple propositions by merging the translations of the component propositions according to the grammatical conjunctions ("or", "and") in the input comment.

Algorithm 1 shows how Jdoctor processes the (normalized) text of Javadoc comments. The text may contain multiple propositions. The algorithm translates each proposition independently. Then, these translations (which are Java expressions and operations) are recombined to create the full executable specification. The recombination is done specially for `@return` comments. Jdoctor first identifies the *guard*, the *true property* and the *false property*. For instance, the return comment for `ArrayStack.search()` in Apache Commons Collections is "the 1-based depth into the stack of the object, or -1 if not found". Jdoctor identifies "if not found" as the guard, "the 1-based depth into the stack of the object" as the true property, that is, the property that holds when the guard is true, and "-1" as the false property, that is, the property that holds when the guard is evaluated to false.

When translating propositions into Java expressions (line 15), Jdoctor attempts to match each subject and predicate to code elements. Intuitively, nouns correspond to objects, which are represented in source code as expressions, and verbs correspond to actions or predicates, which are represented in source code as operations.

Jdoctor starts by analyzing the subject of the proposition (line 17) and tries to match the subject to a code element, which may be a parameter of the method, a field of the class, or the class itself. Jdoctor retrieves the *identifiers* and *types* of all code elements in scope as candidates and looks for the best match. For instance, when processing the comment "if the comparator is null" presented in Section 3, Jdoctor matches the subject "comparator" to the parameter of the method whose type is `Comparator`. Jdoctor implements this task with lexical matching, which we explain in Section 4.3.2.

If Jdoctor finds a matching expression for the subject, it proceeds looking for a corresponding matching predicate (such as "is null" in the example). More in detail, Jdoctor retrieves the code identifiers of all public methods and fields within the scope of the subject as possible candidates. For example, after matching the subject comparator to type `Comparator`, Jdoctor retrieves the whole list of public methods and fields of class `Comparator` as possible candidates. Once it has identified the possible candidates, Jdoctor incrementally exploits a set of heuristics to infer the correct matching of the predicate to the code element among the possible candidates: (i) It checks whether the predicate matches a set of predefined translations (line 22), (ii) it looks for lexically similar matches (line 24), (iii) it searches for matches according to semantic similarity (line 27) .

4.3.1 Pattern Matching. Jdoctor uses *pattern matching* to map common phrases such as "is positive", "is negative", and "is null" to the Java expression fragments `>0`, `<0`, and `==null`, respectively. In line with previous work [41, 51, 55], Jdoctor employs a set of extensible patterns that covers properties for primitive types, Strings, and nullness checks. Pattern matching can efficiently translate common patterns, but it cannot handle domain-specific concepts or jargon specific to the code, like the concepts "vertex" and "graph" in "if vertex is not found in the graph".

4.3.2 Lexical Matching. Jdoctor tries to match a subject or a predicate to the corresponding code element based on the intuition that words in Javadoc comments are lexically similar to code elements. Jdoctor (i) tokenizes code candidates into separate terms according to camel-case convention, (ii) computes the Levenshtein distance between each term and each word in the subject/predicate, and (iii) selects the candidate with the smallest Levenshtein distance, as long as it does not exceed a threshold (with a very small default threshold (i.e. two) to avoid wrong matches as much as possible). For example, when processing the comment "if the comparator is null" presented in Section 3, Jdoctor matches the subject "comparator" to the parameter of the method whose type is `Comparator` with distance 0.[5]

Jdoctor uses lexical matching similarly to Toradocu [24]. This technique outperforms simple pattern matching, like the one implemented by @tComment, and straightforward natural language processing, like the one implemented in ALICS. Its limitations are due to the assumption that nouns in Javadoc comments should be similar to identifiers in the code, which may not always hold in practice.

4.3.3 Semantic Matching. To illustrate the limits of pattern and lexical matching, consider the predicate "is not found in the graph". The desired translation would be `!graph.containsVertex(vertex)`. Pattern-matching would work only if specific patterns were available to handle this case. Lexical matching fails because the code

[5] Jdoctor's algorithm is case-insensitive.

element `containsVertex` is not lexically close to the terms "is not found in the graph" that occur in the comment. The Jdoctor semantic matching approach builds on the observation that syntactically different terms can have a close semantics. For example, method `containsVertex` in the code and the concept "is not found in the graph" in the comment are lexically different, although their *semantics* is related. Jdoctor deals with matches that both pattern and lexical matching miss thanks to *semantic similarity* between words. Jdoctor uses *word embedding*, which has been proved to be a powerful approach to represent semantic word relations. It embeds words in a high-dimensional vector space such that distances between words are closely related to the semantic similarities, regardless of the syntactic differences. Jdoctor uses GloVe,[6] a two-layer neural network model for this purpose.

Jdoctor removes a customized list of stopwords from the predicate and applies lemmatization before computing semantic similarity with GloVe. Lemmatization transforms terms to their root form, for instance, "playing" and "played" become "play", to simplify the semantic matching by reducing the amount of terms in the domain. By default, Jdoctor uses a list of stopwords that includes articles and words that belong to the Java language, such as "for", "do" and "null". The Glove model as is, however, can only capture the semantic similarity of single terms. Thus, it would report the terms "vertex" and "graph" as semantically related. However, most of the times *predicates* and *code identifiers* are composed of multiple words. For example, in JGraphT, the comment excerpt "vertex is found" should match method `containsVertex`. To compare multiple words at once, Jdoctor uses the *Word Mover's Distance (WMD)* algorithm [30].

WMD measures the semantic distance between two text snippets as the cumulative distance that all words in the comment ([`vertex`, `is`, `found`] in this case) have to be exactly as the words in the code element identifier ([`contain`, `vertex`] in this case). Similarly to what Jdoctor does for lexical matching, it selects the candidate that has the closest semantic distance up to a given threshold.

Despite offering different matching strategies, Jdoctor resorts only to lexical similarity for subject matching. This approach forces Jdoctor to match subjects to code elements with a very high precision (though it may miss some matches). This conservative decision is vital for the performance of Jdoctor, since subject matching gives the scope to later match the predicate. A wider — and possibly wrong — scope would direct the search for predicate matching towards completely wrong paths.

Jdoctor produces a single Java boolean condition as translation of `@param` comments, and a pair ⟨expected exception type, Java boolean condition⟩ as translation of `@throws` comments. Translations of `@return` comments are not a single boolean Java condition; instead, a single translation is composed of three Java boolean conditions corresponding to guard, true-, and false-property.

5 EVALUATION: TRANSLATION ACCURACY

We evaluated the translation accuracy of Jdoctor by answering the following research questions.

RQ1 What is the effectiveness (precision and recall) of Jdoctor in translating Javadoc comments to procedure specifications?

[6] https://nlp.stanford.edu/projects/glove/

Table 1: Subject programs and ground-truth translations. Column "Doc'd Classes" reports the total number of classes with documentation, out of which we selected "Analyzed Classes". "Analyzed Methods" reports the methods with Javadoc tags that the authors of this paper could express in executable form.

Subjects	Classes Doc'd	Classes Analyzed	Methods Analyzed	Normal Pre.	Normal Post.	Excep. Postcond.
Commons Collections 4.1	196	20	179	146	26	166
Commons Math 3.6.1	519	52	198	57	23	198
GraphStream 1.3	67	7	14	3	8	1
Guava 19	116	19	66	30	12	48
JGraphT 0.9.2	47	10	23	0	6	28
Plume-lib 1.1	26	10	83	7	43	27
Total	971	118	563	243	118	468

https://commons.apache.org/collections, https://commons.apache.org/math, http://graphstream-project.org, http://github.com/google/guava, http://jgrapht.org, http://mernst.github.io/plume-lib

RQ2 How does Jdoctor's effectiveness compare with state-of-the-art approaches, namely @tComment and Toradocu?

We measure effectiveness in terms of precision and recall, the standard metrics for an information retrieval task such as translating Javadoc comments to procedure specifications.

Precision measures correctness as the proportion of the output that is correct, with respect to missing and wrong outputs. The output is correct (C) when Jdoctor produces a specification that matches the expected specification. The output is missing (M) when Jdoctor does not produce any specification. The output is wrong when Jdoctor either produces a specification when no specification is expected (W1), or a specification that does not match the expected one (W2). Precision is defined as the ratio between the number of correct outputs and the total number of outputs:

$$precision = \frac{|C|}{|C| + |W1| + |W2|}$$

Recall measures completeness as the proportion of desired outputs that the tool produced, and it is defined as the ratio between the number of correct outputs and the total number of desired outputs:

$$recall = \frac{|C|}{|C| + |W2| + |M|}$$

Our measurements conservatively consider a partially correct translation to be wrong. For example, if the comment is "`@throws` Exception if x is negative or y is null", then the translation "`x < 0`" is deemed as wrong.

5.1 Experimental Setup

For the evaluation we selected 6 well-maintained open-source Java systems (see Table 1). For each system we (i) discarded classes with no or limited documentation, that is, with less than 5 Javadoc comments, and (ii) ignored comments documenting methods inherited from `java.lang.Object`, getters and setters (methods whose name starts with "get" or "set"). Column "Doc'd Classes" in Table 1 reports the number of classes that satisfy these conditions for each subject.

We then selected and manually analyzed at least 10% of the documented classes and methods: columns "Analyzed Classes" and "Analyzed Methods", respectively. To this end, we applied probability proportional to size (PPS) sampling, with a probability of

selecting each class proportional to the number of methods in the class. For each analyzed method in each selected class, we manually determined its ground truth — the correct translation of Javadoc comments to executable method specifications. We did so by reading its Javadoc comment, which is expressed in English, and writing executable specifications that correspond to each `@param`, `@return`, and `@throws` and `@exception` tag. Every translation was reviewed independently by at least two of the authors of this paper.

Sometimes, the text corresponding to a Javadoc tag cannot be expressed as an executable specification. An example is "`@throws` IOException if there is trouble reading the file". For such comments, we do not expect Jdoctor to produce any output. We discarded classes that did not contain any comment that could be translated to executable specification, i.e., do not belong to the list of analyzed classes. This left us with 118 analyzed classes and 829 Javadoc comments for which we manually produced the ground-truth executable specification. Our experiments compare three tools, all of which create executable procedure specifications from programmer-written informal English specifications.

- **@tComment** [50] pattern-matches against predetermined templates for three different types of nullness specifications. We wished to use the @tComment implementation, but were stopped by the limited documentation (for example, its file format is undocumented). We found it easier to reimplement @tComment based on its published description [50]. Our implementation of @tComment achieves similar results to those in its paper and is publicly available for external inspection.[7]
- **Toradocu** [24] generates exceptional postconditions from `@throws` comments by means of a combination of NLP and string matching. We used the Toradocu implementation from GitHub.[8]
- **Jdoctor** is the tool described in this paper.

We did not consider ALICS [42] in our comparison, because we could not make the tool work properly, even with the support of the authors (see Section 8 for more information about this issue). Moreover, ALICS does not produce executable specifications, and a comparison would have been hard.

5.2 Accuracy Results (RQ1, RQ2)

Table 2 reports the accuracy of @tComment, Toradocu, and Jdoctor on the subject classes of Table 1. As mentioned in Section 2, Toradocu does not handle preconditions, and neither Toradocu nor @tComment handle normal postconditions (values *n.a.* in the table). The data in the table show that Jdoctor's precision is comparable with state-of-the-art approaches, and Jdoctor's recall is substantially higher than state-of-the-art approaches.

Preconditions. Jdoctor handles preconditions of different types, while @tComment deals only with null-related checks that it handles with a simple analysis. The slightly higher precision of @tComment than Jdoctor benefits from the specificity of its patterns, such as "may be null", "must not be null", and "`@throws` IllegalArgumentException if … is null". However, the simple analysis of @tComment misses many translations, leading to much lower recall than Jdoctor.

[7] https://github.com/albertogoffi/toradocu/blob/master/src/main/kotlin/tcomment/tcomment.kt
[8] https://github.com/albertogoffi/toradocu/releases/tag/v0.1

Table 2: Accuracy (precision, recall, f-measure) of tools that translate English to executable procedure specifications.

	Precond (`@param`)		Normal postcond (`@return`)		Exceptional postcond (`@throws`)		Overall (all Javadoc tags)		
	Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec	F
@tComment	0.97	0.63	n.a.	0.00	0.80	0.16	0.90	0.24	0.38
Toradocu	n.a.	0.00	n.a.	0.00	0.61	0.39	0.61	0.23	0.33
Jdoctor	0.96	0.97	0.71	0.69	0.97	0.79	0.92	0.83	0.87

Normal postconditions. Jdoctor is the only approach that is expressive enough to handle `@return` comments. Its precision and recall are lower for `@return` comments than for other tags, due to the complexity of common return conditions. For example, the comment "`@return` the sorted array" states a postcondition checking whether the array returned by the routine is sorted. This check involves a loop over the array, and is not supported in the current implementation of Jdoctor. Other conditions difficult to check are, for example, the comment "`@return` n+1 if no overflows occur" from class `FastMath` of Commons Math, whose guard should check whether an overflow occurred. Yet another reason for relatively low precision and recall is that postconditions often include comparisons between two or more elements. Currently, Jdoctor translates subjects and predicates as a whole, assuming that subjects and predicates translate to single Java elements, and cannot handle more complex cases, such as "if the size difference between `a` and `c` is not equal to 1", "if maximal number of iterations is smaller than or equal to the minimal number of iterations", and "if `xval` and `yval` have different sizes".

Exceptional postconditions. Jdoctor has better precision and recall than @tComment. @tComment translates only nullness-related comments, i.e., comments containing the word "null" in specific contexts. For instance, Jdoctor can translate the expression "aString is not empty", while @tComment does not. Jdoctor's sentence analysis is more effective than @tComment's pattern-matching in analyzing complex sentences. Jdoctor can translate sentences composed of many dependent clauses connected with specific grammatical conjunctions, like *if x and y are positive or z is negative*, while @tComment's pattern matching does not. Jdoctor has better precision and recall than Toradocu. The better results of Jdoctor are due to a more precise algorithm to translate subjects and predicates into Java elements and to a wider set of supported comments (see Section 4).

An overall precision of 92% and a recall of 83% support a positive answer to **RQ1**: *Jdoctor is effective and accurate in translating Javadoc comments into procedure specifications.* Our evaluation also supports a positive answer to **RQ2**: *Jdoctor achieves better accuracy than state-of-the-art techniques @tComment and Toradocu.*

5.3 Inconsistent Specifications

We manually checked the accuracy of Jdoctor reported in Section 5.2, by inspecting the output of Jdoctor for each Javadoc comment. The inspection indicates that Jdoctor can produce *correct, but inconsistent specifications.* While currently many of these issues can be found only manually analyzing Jdoctor's output, we

plan to extend the tool to automatically report them. We now describe some of the many inconsistencies that we encountered. We found that many developer-written specifications were logically inconsistent — that is, they could not be implemented, or used by clients. This highlights a side benefit of converting informal English specifications into executable form: The inconsistencies are not as obvious in the informal formulation, and the machine-readable specification can be automatically checked. Below we report six types of inconsistencies and errors that Jdoctor could highlight.

Some specifications translate into expressions that are not always well defined, because their evaluation could throw an exception. An example is `arg.f > 0` when `arg` is null, which can lead to two reasonable but different interpretations: the expression `arg.f` always has a value (i.e., `arg != null && arg.f > 0`), or the condition is true whenever `arg.f` has a value (i.e., `arg != null || arg.f > 0`). Both interpretations occur in practice, although their meaning is quite different, both for clients and for implementers.

Many specifications have conflicting preconditions and postconditions. A simple example from class `CollectionUtils` in Commons Collections is

> * @param a the first collection, must not be null
> * @throws NullPointerException if a is null

where the @throws clause refers to cases that violate the @param clause, leading to two legal albeit contradicting interpretations: (i) the method's domain is all values (the programmer is allowed to pass null), with a promise about what happens when null is passed in, or (ii) the domain is all non-null values, and programmers should not pass null. Both interpretations are plausible, and there is no way to know which the designers intended, but the difference is far for being trivial: the former states that maintainers must not change the implementation in the future, while the latter gives the implementer freedom to change the implementation in the future, and clients who depend on the exception will break.

Some of the specifications indicate multiple outcomes for a given condition. As an example, let us consider

> * @throws NullPointerException if arg1 is null
> * @throws IllegalArgumentException if arg2 is negative

If `arg1` is null and `arg2` is 0, then the routine is required to throw *both* `NullPointerException` and `IllegalArgumentException`, which is impossible in Java. A client who does not notice the inconsistency may experience unexpected behavior. One example of this kind of inconsistency is in class `KShortestPaths` of JGraphT. Similar inconsistencies arise also in postconditions.

Jdoctor automatically identified some errors in the Javadoc, which are likely due to copy-and-paste errors [6]. For example, the documentation of method `CharMatcher.matchesNoneOf` in Guava 19.0 states that it "returns true if this matcher matches **every** character in the sequence, including when the sequence is empty" [bold ours], while it should state that the method matches **no** character in the sequence. Jdoctor correctly translated the typo "matches every character" to a Java expression that uses method `matchesAllOf`, and this assertion failed at run time, highlighting the incorrect Javadoc.

Some procedures' Javadoc refer to incorrect formal parameter names. Common causes are the inheritance of Javadoc from overridden implementations, joint with changes the formal parameter names. A reader of the HTML API documentation would see one set of names in the Javadoc and a different set in the method signature. Oftentimes the correspondence is obvious, but not always, and readers should not have to deduce it.

Jdoctor also automatically reported some typos in the Javadoc. For instance, in class `Node` in the GraphStream project, the Javadoc wrongly says that a method may throw an `IndexOutOfBoundException`, instead of the correct `IndexOutOfBoundsException`. Jdoctor could report the issues since it did not find class `IndexOutOfBoundException` in its classpath.

6 APPLICATION: ORACLE GENERATION

As noted in Section 1, procedure specifications may have many applications. In this section, we evaluate the use of Jdoctor's procedure specifications for improving the generation of test cases.

6.1 Test Classification

Many test case generation tools first generate test cases and then heuristically classify the generated test cases each as[9]

(1) failing (or error-revealing) test cases that reveal defects;
(2) passing (or normal, or expected) test cases that can be used as regression test cases;
(3) illegal (or invalid) test cases that should not be executed; for example, because they either violate some method preconditions or their oracles are incorrect.

The test generation tool outputs both failing tests and passing test cases to the user. The heuristics sometimes misclassify test cases, leading to both false and missed alarms. For example, a test case generator may heuristically classify as invalid a method execution terminating with a `NullPointerException` when the method is invoked with null values. Test generation tools that can access some (partial) specifications, can classify (more) test cases correctly, thus detecting more errors and/or reducing the human effort required to identify false alarms.

We investigate the usefulness of Jdoctor by addressing the following research questions.

RQ3 Do Jdoctor specifications improve the quality of automatically generated tests?

RQ4 Do Jdoctor specifications increase the number of bugs found by an automated test generator?

6.2 Extending Randoop with Jdoctor

Jdoctor outputs a JSON file containing executable Java expressions corresponding to each method preconditions, normal postconditions, and exceptional postconditions. When a method lacks a Javadoc comment or its Javadoc comment lacks a `@param`, `@return`, or `@throws` clause, the JSON file contains no corresponding expression. We extended Randoop [40] to take advantage of this information during test generation, in line with the methodology exploited in previous work [24, 50]. Our implementation, which we refer to as Randoop+Jdoctor, is integrated in the main branch of Randoop's GitHub repository[10].

[9] For simplicity of presentation, we do not consider flaky tests here. The heuristics may also classify a test as one that the tool is unsure about; the tool does not output the test to the user in order to avoid false alarms. For simplicity, we will treat this as if the heuristic classified the test as illegal.

[10] https://github.com/randoop/randoop

Randoop can be thought of as a loop that iteratively creates test cases. Each iteration randomly creates a candidate test by first choosing a method to test, and then choosing arguments from a pool of previously-created objects. Randoop executes the candidate test, and heuristically classifies the test as error-revealing, expected behavior, or invalid based on its behavior. If the test behaves as expected, Randoop places its result in the pool, and continues with its loop. When a time limit has elapsed, Randoop outputs the error-revealing and expected-behavior tests, in separate test suites.

We modified Randoop to create Randoop+Jdoctor as follows:

- After choosing the arguments but before creating or executing the candidate test, Randoop+Jdoctor reflectively executes the precondition expressions. If any of them fails, then Randoop+Jdoctor discards the test, exactly as if it had been classified as invalid. In this way, Randoop+Jdoctor avoids the possibility of misclassifying it as an error-revealing or passing test.
- If the test completes successfully, Randoop classifies it as passing. Randoop+Jdoctor reclassifies it as failing if a normal postcondition (a translated `@return` clause) does not hold. Randoop+Jdoctor handles conditional postconditions, such as "@return true if this graph did not already contain the specified edge", because Jdoctor provides information about the conditional.
- While executing the test, Randoop catches any thrown exceptions. If the exception matches one in an exceptional postcondition (a translated `@throws` clause), then Randoop+Jdoctor classifies the test as passing iff the `@throws` condition holds. If the exception does not match, Randoop+Jdoctor falls back to Randoop's normal behavior of heuristically classifying the test.

6.3 Methodology

Our experiments compare the original Randoop test generation tool with Randoop+Jdoctor, which extends Randoop with Jdoctor-generated procedure specifications.

We ran both Randoop and Randoop+Jdoctor on all the 6 programs of Table 1. The experiments of Section 5 used only a subset of the classes in the programs, because of the human effort of manually determining the ground truth, which is the correct translation of an English Javadoc comment into an executable procedure specification. The experiments of this section are executed with the entire programs and also the Javadoc for their supertypes in Apache Commons RNG and OpenJDK 8.

6.4 Test Classification Changes

To answer **RQ3**, we measured when Randoop+Jdoctor classified a candidate test differently than Randoop (giving to the two tools the same time limit of 15 minutes). There are five possibilities:

Same Randoop and Randoop+Jdoctor classify the test in the same way, which might be "failing", "passing", or "invalid".

False alarm Randoop+Jdoctor classifies as passing a test that Randoop classifies as failing. Randoop's output requires manual investigation by the programmer, but Randoop+Jdoctor's does not.

Missed alarm Randoop classifies the test as passing, but Randoop+Jdoctor classifies it as failing. Randoop misses a bug, but Randoop+Jdoctor reveals it to the programmer.

Table 3: How Jdoctor output (procedure specifications) improves Randoop's test classification. Each cell is the count of candidate tests that were classified differently by Randoop and Randoop+Jdoctor, for one run of Randoop.

Subjects	Same	False alarm	Missed alarm	New test	Invalid test
Collections	7527	0	0	2	0
Math	3893	0	0	1	0
Guava	10821	0	34	4	20
JGrapht	4843	0	0	0	0
Plume-lib	4253	0	48	0	0
Graphstream	12454	3	3	8	0
Total	43791	3	85	15	20

Invalid test Randoop classifies the test as passing, but Randoop+Jdoctor classifies it as invalid. Randoop's output contains a meaningless test (e.g. because it violates the preconditions of a method) that may fail at any time in the future, but Randoop+Jdoctor 's does not.

New test Randoop+Jdoctor generates the test that Randoop does not generate, since it classifies it as invalid, leading to better coverage and better regression testing.

Table 3 shows the result. The Jdoctor specifications reduce human effort ("false alarms" column), improve error detection ("missed alarms" column), and reduce invalid tests. We manually inspected the results, and we could confirm all the 3 *False alarms* and the 20 *Invalid tests*. Invalid tests were all newly classified as such thanks to Jdoctor specifications on parameters. For instance, method max() in class com.google.common.primitives.Longs of Guava states that parameter array is "a nonempty array of long values". Thus any test passing an empty array to max() was correctly classified as invalid since it violates the preconditions. We were expecting to find many invalid tests in other projects beside Guava, since Jdoctor could correctly handle many comments on preconditions, as shown in Section 5. We believe we did not see any improvement because Randoop may have achieved low coverage of such projects.

To answer **RQ4**, we manually inspected some of the 85 *Missed alarms*. Unfortunately we could spot several incorrect results due to mis-translated comments (precision is not 100% in Table 2). An example is due to the comment "@throws NullPointerException if the check fails and either @code errorMessageTemplate or @code errorMessageArgs is null" which Jdoctor translated as errorMessageTemplate == null || errorMessageArgs == null, incorrectly missing the part on the failing check. This specification wrongly makes Randoop+Jdoctor classify tests having any null value as failing if they do not throw a NullPointerException.

Randoop+Jdoctor could also detect some real defects in the documentation of these well tested projects. Class NumberFormat of the JDK wrongly documents thrown exception types in two methods.

7 THREATS TO VALIDITY

Our subject programs, and the classes selected in Section 5, might not be representative, which may limit the generalizability of the results. This is mitigated by the fact that the portion of classes that we manually analyzed is statistically representative of all the documented classes according to the number of comments and methods.To evaluate the accuracy of Jdoctor, its results have been

compared with a ground truth that the authors of this paper defined. Errors in the ground truth influence the accuracy measures reported in Section 5. To mitigate such risk, each specification in the ground truth has been independently produced and later reviewed by at least two of the authors of this paper.[11] Section 5 compared Jdoctor with our (publicly available) implementation of the @tComment approach [50]. The two @tComment implementations might produce different results for some input comments, even though the precision obtained with our implementation is close to what the original paper reports: 90% vs 98%. Recall values are different (24% vs 97%) because in the paper presenting @tComment, recall is computed considering only null-related comments — exactly the case that @tComment's pattern matching is tuned to — whereas our experiments consider all the comments that can be translated to executable form.

8 RELATED WORK

Section 2 already discussed the most closely related work: @tComment, ALICS, and Toradocu. Our experiments compare to Toradocu and a re-implementation of @tComment, but not ALICS. We tried to use ALICS, but it contains no documentation and is incomplete. We reached out to all the ALICS authors, who confirmed that it is incomplete, and none of them could run the tool to reproduce their own experiments. The ALICS paper [41] does not contain enough information to permit us to re-implement it.

Recently Phan et al. employed a statistical machine translation technique to generate behavioral exception documentation for any given code and viceversa [43]. This approach only works for exceptional comments, which are easier to handle than other comments in our experience. It also relies on a trained model that would not resolve comments of new domains requiring semantic knowledge (as for JGraphT). We believe it could nicely complement Jdoctor, but we consider this direction as future work.

Zhou et al. recently proposed an approach that combines natural language parsing, pattern matching, and constraint solving to detect errors, i.e., code-comments inconsistencies, in Javadoc documentation expressing parameter usage constraints [55]. (To the best of our knowledge, the implementation is not publicly available, even though used patterns are.) The approach extracts preconditions and exceptional postconditions from the Javadoc documentation of a procedure, and analyzes the corresponding source code to derive what exceptions are raised and for which conditions, and then verifies that the conditions stated in the documentation are correct by means of an SMT solver. The approach can achieve good precision and recall, but it works only for four kind of procedure preconditions: (non-)nullness of parameters, type restriction, and range limitation. Instead, Jdoctor supports also non-exceptional postconditions and generates procedure specifications written as Java code that can be readily used by other tools.

iComment [48] and aComment [49] are two additional techniques to extract some form of specifications from code comments. iComment applies NLP techniques to extract function usage rules from the code comments to detect code–comment inconsistencies in C code. aComment analyzes source code and comments to detect concurrency bugs in C code. Although both iComment and aComment generate specifications, they apply to unstructured comments in C code (rather than semi-structured Javadoc comments as Jdoctor). Moreover, iComment and aComment focus on narrow specifications (usage rules and concurrency related specifications), while Jdoctor generates general pre- and postconditions.

We now briefly survey alternative approaches for generating test oracles. The main approaches for generating test oracles (1) exploit heuristics to validate thrown exceptions [11, 12, 32, 39, 40], (2) rely on previous versions of the software under test to validate the results (regression oracles) [20], (3) benefit from various kinds of specifications: algebraic specifications [2, 17, 22], assertions [3, 9, 36, 45, 52], Z specifications [35, 38], context-free grammars [13], JML annotations [10] and finite state machines [21], (4) manipulate properties of the system under test, as in the case of metamorphic testing [8] and symmetric testing [25], and (5) exploit some form of redundancy of software systems (cross-checking oracles) [7].

Heuristic-based oracles are imprecise; regression-based oracles are useful only in the context of regression testing; specification based oracles require the availability of some form of specifications; metamorphic, symmetric, and cross-checking oracles rely on properties of the system under test and of its implementations that may not be always available or easy to exploit. Jdoctor complements all these techniques by generating procedure specifications from Javadoc comments. The Jdoctor procedure specifications can be used to generate test oracles.

9 CONCLUSIONS

This paper presents Jdoctor, a technique to automatically generate executable procedure specifications from Javadoc comments. Jdoctor enables the application of techniques that require procedure specifications when such specifications are not available, while Javadoc comments are. Jdoctor produces procedure specifications consisting of preconditions, postconditions, and exceptional postconditions, by means of a mix of natural language processing, pattern matching, lexical matching, and semantic similarity analysis. Experimental results show that Jdoctor is effective in producing correct and complete specifications, with overall precision and recall of 92% and 83%, respectively. Jdoctor outperforms existing state-of-the-art techniques, with better precision *and* recall. When procedure specifications generated by Jdoctor are fed to an automatic test case generator such as Randoop [40], the generator produces better test cases. As future work we plan to automatically detect inconsistent specifications, extend Jdoctor to unstructured documentation, and possibly focus on integration and system tests [16, 26, 33].

ACKNOWLEDGMENTS

This work was partially supported by the Spanish project DEDETIS, by the Madrid Regional project N-Greens Software (n. S2013/ICE-2731), and by the swiss project ASTERIx: Automatic System TEsting of inteRactive software applIcations (SNF-200021_178742). This material is also 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.

[11] Data publicly available for external revision: https://github.com/albertogoffi/toradocu/tree/master/src/test/resources/goal-output

REFERENCES

[1] Gabor Angeli, Melvin Johnson Premkumar, and Christopher D Manning. 2015. Leveraging Linguistic Structure for Open Domain Information Extraction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL '15)*. Association for Computational Linguistics.

[2] Sergio Antoy and Dick Hamlet. 2000. Automatically Checking an Implementation against Its Formal Specification. *IEEE Transactions on Software Engineering* 26, 1 (2000), 55–69.

[3] Wladimir Araujo, Lionel C. Briand, and Yvan Labiche. 2011. Enabling the Runtime Assertion Checking of Concurrent Contracts for the Java Modeling Language. In *Proceedings of the International Conference on Software Engineering (ICSE '11)*. 786–795.

[4] Marc J. Balcer, William M. Hasling, and Thomas J. Ostrand. 1989. Automatic Generation of Test Scripts from Formal Test Specifications. In *Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV3 '89)*. ACM, 210–218.

[5] Boris Beizer. 1990. *Software Testing Techniques* (2 ed.). Van Nostrand Reinhold Co., New York, NY, USA.

[6] Arianna Blasi and Alessandra Gorla. 2018. RepliComment: Identifying Clones in Code Comments. In *Proceedings of the International Conference on Program Comprehension (ICPC'18)*. ACM.

[7] Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. 2014. Cross-checking Oracles from Intrinsic Software Redundancy. In *Proceedings of the International Conference on Software Engineering (ICSE '14)*. ACM, 931–942.

[8] Tsong Y. Chen, F.-C. Kuo, T. H. Tse, and Zhi Quan Zhou. 2003. Metamorphic Testing and Beyond. In *International Workshop on Software Technology and Engineering Practice (STEP '03)*. IEEE Computer Society, 94–100.

[9] Yoonsik Cheon. 2007. Abstraction in Assertion-Based Test Oracles. In *Proceedings of the International Conference on Quality Software (QSIC '07)*. 410–414.

[10] Yoonsik Cheon and Gary T. Leavens. 2002. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '02)*. 231–255.

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

[12] Christoph Csallner and Yannis Smaragdakis. 2005. Check 'n' Crash: Combining static checking and testing. In *ICSE 2005, Proceedings of the 27th International Conference on Software Engineering*. St. Louis, MO, USA, 422–431.

[13] J. D. Day and J. D. Gannon. 1985. A Test Oracle Based on Formal Specifications. In *Proceedings of the Conference on Software Development Tools, Techniques, and Alternatives (SOFTAIR '85)*. 126–130.

[14] Luciano Del Corro and Rainer Gemulla. 2013. ClausIE: Clause-based Open Information Extraction. In *Proceedings of the International Conference on World Wide Web (WWW '13)*. ACM, 355–366.

[15] Nelly Delgado, Ann Quiroz Gates, and Steve Roach. 2004. A taxonomy and catalog of runtime software-fault monitoring tools. *IEEE Transactions on Software Engineering* 30, 12 (Dec. 2004), 859–872.

[16] Giovanni Denaro, Alessandra Gorla, and Mauro Pezzè. 2008. Contextual Integration Testing of Classes. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE '08)*. Springer, 246–260.

[17] Roong-Ko Doong and Phyllis G. Frankl. 1994. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodology* 3, 2 (1994), 101–130.

[18] Anthony Fader, Stephen Soderland, and Oren Etzioni. 2011. Identifying Relations for Open Information Extraction. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP '11)*. Association for Computational Linguistics, 1535–1545.

[19] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '11)*. ACM, 416–419.

[20] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[21] Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. 1991. Test Selection Based on Finite State Models. *IEEE Transactions on Software Engineering* 17, 6 (1991), 591–603.

[22] John Gannon, Paul McMullin, and Richard Hamlet. 1981. Data Abstraction, Implementation, Specification, and Testing. *ACM Transactions on Programming Languages and Systems* 3, 3 (1981), 211–223.

[23] Alejandra Garrido and Jose Meseguer. 2006. Formal Specification and Verification of Java Refactorings. In *Proceedings of the IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '06)*. 165–174.

[24] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic generation of oracles for exceptional behaviors. In *ISSTA 2016, Proceedings of the 2016 International Symposium on Software Testing and Analysis*. Saarbrücken, Genmany, 213–224.

[25] Arnaud Gotlieb. 2003. Exploiting Symmetries to Test Programs. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE '03)*. IEEE Computer Society, 365–374.

[26] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based system testing: high coverage, no false alarms. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '12)*. ACM, 67–77.

[27] A. Hartman. 2002. Is ISSTA research relevant to industry?. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*. Rome, Italy, 205–206.

[28] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-guided Component-based Program Synthesis. In *Proceedings of the International Conference on Software Engineering (ICSE '10)*. ACM, 215–224.

[29] James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. In *Proceedings of the International Conference on Software Engineering (ICSE '02)*. ACM, 467–477.

[30] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger. 2015. From Word Embeddings to Document Distances. In *Proceedings of the International Conference on International Conference on Machine Learning (ICML '15)*. JMLR.org, 957–966.

[31] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. 2007. Contract Driven Development = Test Driven Development - Writing Test Cases. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '07)*. ACM, 425–434.

[32] Lei Ma, Cyrille Artho, Cheng Zhang, Hiroyuki Sato, Johannes Gmeiner, and Rudolf Ramler. 2015. GRT: Program-Analysis-Guided Random Testing. In *Proceedings of the International Conference on Automated Software Engineering (ASE '15)*. ACM, 212–223.

[33] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2012. AutoBlackTest: Automatic Black-Box Testing of Interactive Applications. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, 81–90.

[34] Marie-Catherine Marneffe, Bill MacCartney, and Christopher Manning. 2006. Generating Typed Dependency Parses from Phrase Structure Parses. In *Proceedings of the International Conference on Language Resources and Evaluation (LREC '06)*. European Language Resources Association (ELRA), 449–454.

[35] Jason Mcdonald. 1998. Translating Object-Z Specifications to Passive Test Oracles. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM '98)*. 165–174.

[36] Bertrand Meyer. 1988. *Object-Oriented Software Construction* (1st ed.). Prentice Hall.

[37] Bertrand Meyer, Ilinca Ciupa, Andreas Leitner, and Lisa Ling Liu. 2007. Automatic Testing of Object-Oriented Software. In *Proceedings of the 33rd Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM '07)*. Springer, 114–129.

[38] Erich Mikk. 1995. Compilation of Z Specifications into C for Automatic Test Result Evaluation. In *Proceedings of the 9th International Conference of Z Users (ZUM '95)*. 167–180.

[39] Carlos Pacheco and Michael D. Ernst. 2005. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*. Glasgow, Scotland, 504–527.

[40] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-directed random test generation. In *ICSE 2007, Proceedings of the 29th International Conference on Software Engineering*. Minneapolis, MN, USA, 75–84.

[41] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring method specifications from natural language API descriptions. In *ICSE 2011, Proceedings of the 34th International Conference on Software Engineering*. Zürich, Switzerland, 815–825.

[42] Rahul Pandita, Xusheng Xiao, Hao Zhong, Tao Xie, Stephen Oney, and Amit Paradkar. 2012. Inferring Method Specifications from Natural Language API Descriptions. In *Proceedings of the International Conference on Software Engineering (ICSE '12)*. IEEE Computer Society, 815–825.

[43] Hung Phan, Hoan Anh Nguyen, Tien N. Nguyen, and Hridesh Rajan. 2017. Statistical Learning for Inference Between Implementations and Documentation. In *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track (ICSE-NIER '17)*. IEEE Press, Piscataway, NJ, USA, 27–30. https://doi.org/10.1109/ICSE-NIER.2017.9

[44] Research Triangle Institute. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. NIST Planning Report 02-3. National Institute of Standards and Technology.

[45] David S. Rosenblum. 1995. A Practical Approach to Programming With Assertions. *IEEE Transactions on Software Engineering* 21, 1 (1995), 19–31.

[46] Michael Schmitz, Robert Bart, Stephen Soderland, Oren Etzioni, et al. 2012. Open language learning for information extraction. In *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL '12)*. Association for Computational Linguistics, 523–534.

[47] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. From Program Verification to Program Synthesis. In *Proceedings of the Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '10)*. ACM, 313–326.

[48] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /*iComment: Bugs or Bad Comments?*/. In *SOSP 2007, Proceedings of the 21st ACM Symposium on Operating Systems Principles*. Stevenson, WA, USA, 145–158.

[49] Lin Tan, Yuanyuan Zhou, and Yoann Padioleau. 2011. aComment: Mining Annotations from Comments and Code to Detect Interrupt Related Concurrency Bugs. In *Proceedings of the International Conference on Software Engineering (ICSE '11)*. 11–20.

[50] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST '12)*. IEEE Computer Society, 260–269.

[51] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. 2012. @tComment: Testing Javadoc Comments to Detect Comment-Code Inconsistencies. In *ICST 2012: Fifth International Conference on Software Testing, Verification and Validation (ICST)*. Montreal, Canada, 260–269.

[52] Richard N. Taylor. 1983. An Integrated Verification and Testing Environment. *Software: Practice and Experience* 13, 8 (1983), 697–713.

[53] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. 2003. Model Checking Programs. *Automated Software Engineering* 10, 2 (2003), 203–232.

[54] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 3 (February 2002), 183–200.

[55] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the International Conference on Software Engineering (ICSE '17)*. IEEE Computer Society, 27–37.