Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints

Arianna Blasi
Università della Svizzera italiana
Lugano, Switzerland

Alessandra Gorla
IMDEA Software Institute
Madrid, Spain

Michael. D. Ernst
University of Washington
Seattle, USA

Mauro Pezzè
Università della Svizzera italiana
Lugano, Switzerland
SIT Institute of Technology
Schaffhausen, Switzerland

ABSTRACT

A class may need to obey temporal constraints in order to function correctly. For example, the correct usage protocol for an iterator is to always check whether there is a next element before asking for it; iterating over a collection when there are no items left leads to a *NoSuchElementException*. Automatic test case generation tools such as Randoop and EvoSuite do not have any notion of these temporal constraints. Generating test cases by randomly invoking methods on a new instance of the class under test may raise run time exceptions that do not necessarily expose software faults, but are rather a consequence of violations of temporal properties.

This paper presents *CallMeMaybe*, a novel technique that uses natural language processing to analyze Javadoc comments to identify temporal constraints. This information can guide a test case generator towards executing sequences of method calls that respect the temporal constraints. Our evaluation on 73 subjects from seven popular Java systems shows that *CallMeMaybe* achieves a precision of 83% and a recall of 70% when translating temporal constraints into Java expressions. For the two biggest subjects, the integration with Randoop flags 11,818 false alarms and enriches 12,024 correctly failing test cases due to violations of temporal constraints with clear explanation that can help software developers.

ACM Reference Format:
Arianna Blasi, Alessandra Gorla, Michael. D. Ernst, and Mauro Pezzè. 2022. Call Me Maybe: Using NLP to Automatically Generate Unit Test Cases Respecting Temporal Constraints. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3551349.3556961

1 INTRODUCTION

Test oracles — executable assertions that check relevant properties — greatly enhance test cases [3]. Executable specifications help to automate software engineering tasks such as test case generation and debugging. In software testing, for instance, executable specifications expressing pre-conditions may be used to automatically filter out invalid inputs. Executable specifications expressing post-conditions can be used as test oracles. Both can be used as run-time assertions, aiding in software debugging.

Some approaches in the state of the art automatically generate executable assertions using *Natural Language Processing (NLP)*, based on the observation that many important properties are available as code comments. For example, tagged Javadoc comments informally state input, return, and exception conditions, while Javadoc summaries informally state many general properties. iComment [31], aComment [32], @tComment [33], Toradocu [14], and Jdoctor [5] automatically generate pre-conditions and post-condition from Javadoc tags. Other approaches such as ALICS [24] and MeMo [6] automatically infer specifications from Javadoc summaries. When similar specifications are available to automatic test case generators such as Randoop [22] and EvoSuite [11], they reduce false alarms and increase true alarms in generated test suites, while filtering out invalid test cases.

This paper focuses on *temporal constraints* documented as natural language comments, and defined as "the allowed sequences of method invocations in the API governing the secure and robust operation of client software using the API" [23]. Temporal constraints are also referred to as *function precedence protocols* [27], introduced as protocols that "define ordering relations among function calls in a program", and *method ordering constraints* [25], defined by stating that "not every method can be called at any point during the execution of a program". In a nutshell, temporal constraints specify the correct sequence of method invocations, thus asserting how to properly use software components. For instance, developers who use an iterator on a collection instance in Java should always check whether there is a next element before trying to retrieve an element from the container. As another example, developers should set a daemon before using a thread.

Executing test cases that violate temporal constraints may result in run-time exceptions that are difficult to classify as expected versus faulty behavior for a test generation tool or a human tester. Is the run-time exception the expected behavior? Does it expose a fault in the unit under test? Without a machine-interpretable specification, this classification cannot be done automatically. Moreover, executable specifications expressing temporal properties are also

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556961

useful for automated debugging, as they report violations of the expected constraint.

Several approaches dynamically infer temporal constraints from execution traces [1, 19, 25, 26]. They assess what is the actual behavior of a program, which may differ from the expected behavior. Doc2Spec [37] and ICON [23] can infer some temporal constraints statically from the code documentation, hence predicating on the expected behavior. Doc2Spec relies on specific templates to recognize the constraints inside the text, while ICON relies on machine learning features and some heuristics.

This paper proposes *CallMeMaybe*, a technique to automatically identify temporal constraints of Java classes from Javadoc method summaries expressing the expected order of the operations as natural language sentences. *CallMeMaybe* parses each sentence of a summary to identify temporal constraints, and produces temporal specifications as simple JSON structures that indicate the required order of operations. *CallMeMaybe* JSON structures are serializable and ready to exploit in other tools such as automated test case generators. *CallMeMaybe* translations are accurate and generalize well on different documentation styles, without relying on patterns and needing training data.

We evaluated the effectiveness of our technique as the impact of *CallMeMaybe* temporal constrains on automatically generated test suites. We integrated *CallMeMaybe* with Randoop [22], an open source test case generator for Java, to automatically filter out error test cases that invoke sequences violating *CallMeMaybe* temporal constraints and explain expected exceptions violating the properties in normal test cases. When Randoop classifies a test case that throws a *declared* exception as passing, Randoop enhanced with *CallMeMaybe* reports the violations of any *CallMeMaybe* temporal property, with a clear explanation about the violation that raised the runtime exception. When Randoop flags a test case as potentially exposing a fault, due to a runtime exception that is not explicitly declared, Randoop enhanced with *CallMeMaybe* reports a false positive if the test case violates any *CallMeMaybe* temporal property.

We evaluated *CallMeMaybe* on 73 classes randomly selected from seven popular Java systems in which we manually identified 89 temporal constraints. On this ground truth, *CallMeMaybe* achieves a good accuracy of 83% precision and 70% recall. We also evaluated *CallMeMaybe* on ICON [23]'s dataset, and confirmed that *CallMeMaybe* identifies all the constraints in their Java ground truth. In our experiments with Randoop, *CallMeMaybe* prevented 11,818 false alarms and 12,024 brittle regression tests (false positives).

The remainder of the paper is structured as follows. Section 2 motivates our work with Javadoc comments taken from popular Java systems. Section 3 defines *CallMeMaybe* and its core components. Section 4 presents our experimental setting and discusses our empirical results. Section 5 discusses the related work. Section 6 summarizes the main contribution of this paper.

2 MOTIVATING EXAMPLE

Temporal constraints, sometime referred to also as *call protocols* [27], specify the correct sequences of invocations of method calls. Temporal constraints describe either the effects of some executions in terms of expected sequence of events (hereafter *descriptive constraints*) or the proper usage of a class, to prevent undesired consequences during the program execution (hereafter *prescriptive constraints*).

Listing 1 shows a descriptive constraint from Cern's Colt library. The comment describes the effects of calling method `clear()` on an instance of `AbstractCollection` *after* the return of the call.

Listing 1: Descriptive temporal constraint from class AbstractCollection of the Colt library

```
1 /** The receiver will be empty after this call returns. */
2 public void clear() { ...
```

Listing 2 shows a prescriptive constraint from Apache Commons Collections. The comment states the order of method calls that is required to avoid undesirable behavior, by ruling out any usage of an instance of method `IteratorEnumeration` *before* invoking method `setIterator`.

Listing 2: Prescriptive temporal constraint from class IteratorEnumeration of Apache Commons Collections

```
1 /** Constructs a new IteratorEnumeration that will not function until
2  setIterator(Iterator) is invoked. */
3 public IteratorEnumeration() { ...
```

Developers document temporal constraints both in method and class summaries, and describe temporal constraints in two ways:

(1) by stating state that some specific methods should or should not be *invoked* (*called*, *used*, etc.) in some order relative to other operations or events. Namely, developers explicitly mention the *operations* that the API user should or should not perform (typically method calls, constructor calls, field accesses). We refer to this type of constraint description as *explicit operation*.
(2) by stating some actions that the API user should or should not perform, *implicitly* encoding code identifiers. Namely, developers implicitly refer to the *operations* in the API via natural language English terms. We refer to this type of constraint description as *implicit action*.

We illustrate how the same temporal constraint may be expressed in either ways with the following example that we adapt from a constraint described in the JDK `java.lang.Thread` class:

(1) **Explicit Operation** *"method* `setDaemon` *should be called before method* `start`*"*. This comment explicitly mentions the method names that the temporal constraint involves, `setDaemon` and `start`. The term *call* refers to a legitimate operation the user can perform with the code elements. Listings 3 and 4 are other examples of explicit descriptions, in a method summary and a class summary, respectively.
(2) **Implicit Action** *"the thread should be started after setting the daemon"*. In this comment, the term *started* implicitly refers to an invocation of method `start`, and the phrase *setting the daemon* to method `setDaemon`. The method invocations are suggested in the *propositions* (subject and verb pairs) [9], and not explicitly spelled out.

Listing 3: Temporal constraint that explicitly names methods, from class LoopingListIterator of Apache Commons Collections

```
/** This method can only be called after at least one {@link #next} or
{@link #previous} method call */
public void remove() ...
```

Listing 4: Temporal constraint from GraphStream class summary

```
/**
 * Allows to run a layout in a distinct thread.
 *
 * ...
 *
 * Once you finished using the runner, you must call release() to break the
 * link with the event source and stop the thread. The runner cannot be used
 * after.
 */
public class LayoutRunner extends java.lang.Thread
```

Listing 5 reports the original temporal constraints, which is a mix of implicit and explicit documentation: The JDK Javadoc specification explicitly mentions the operation of `invoking` the documented method, `setDaemon`, and implicitly specifies that the method should be invoked before invoking method `start`, by means of the proposition `(the thread, is started)`. In the following, we refer to propositions expressing temporal constraints as "temporal propositions".

Listing 5: Temporal constraint both explicitly and implicitly referring to methods and operations, from class Thread of the JDK

```
/** This method must be invoked before the thread is started. */
public final void setDaemon(boolean on) ...
```

Class summaries may specify both desirable (*good*) and undesirable (*bad*) *class usages* by means of code snippets. Listing 6 shows a Google Guava summary that indicates a bad usage of a class by combining a snippet with some discouraging statements. The text terms highlighted in red indicate the bad usage, and occur both before and inside the snippet as code comments.

Listing 6: Summary that describes a "bad constraint" from Google Guava

```
/**
 * Overrides the {@link ImmutableMultiset} static methods that lack {@link
 * ImmutableSortedMultiset} equivalents with deprecated, exception-throwing
 * versions. This prevents accidents like the following:
 *
 * {@code
 *     List<Object> objects = ...;
 *     // Sort them:
 *     Set<Object> sorted = ImmutableSortedMultiset.copyOf(objects);
 *     // BAD CODE!
 *     // The returned multiset is actually an unsorted ImmutableMultiset!
 * }
 *
 * ...
 */
abstract class ImmutableSortedMultisetFauxverideShim<E> extends
ImmutableMultiset<E>
```

Listing 7 shows an Apache Commons Collections summary that indicates a good, legitimate usage.

Listing 7: Summary that describes a "good constraint" from Apache Commons Collections

```
/**
 * Defines an iterator that operates over a {@code Map}.
 * ...
 *
 * In use, this iterator iterates through the keys in the map. After each
 * call to {@code next()}, the {@code getValue()} method provides direct
 * access to the value. The value can also be set using {@code setValue()}.
 *
 * {@code
 *     MapIterator<String, Integer> it = map.mapIterator();
 *     while (it.hasNext()) {
 *         String key = it.next();
 *         Integer value = it.getValue();
 *         it.setValue(value + 1);
 *     }
 * }
 *
 */
public interface MapIterator<K, V> extends Iterator<K>
```

2.1 Problem Domain

To define an automatic translator from natural language comments into temporal constraints, we analyze the specific problem domain to explain our goals and outline the challenges we must face:

Prescriptive vs. descriptive constraints. Both prescriptive and descriptive constraints are useful for developers and tools. We focus on prescriptive constraints because they are more useful than descriptive constraint in improving automatically generated test suites: Test case generators are unaware of prescriptive constraints informally described in code comments, and generate many invalid sequences leading to errors, thus resulting in false alarms. They can benefit from temporal constraints obtained from natural language comments to prune invalid test cases. For example, Randoop [22] cannot infer the temporal constraint informally expressed in the method summary shown in Listing 2 that requires an `Iterator` to be set after instantiating an `IteratorEnumeration`. It thus generates many error test cases that instantiate a new `IteratorEnumeration` without invoking `setIterator`, which are actually false alarms. By translating the prescriptive constraint into a temporal constraint, *CallMeMaybe* offers an automatic mean to reduce false alarms.

Order of events. The problem of recognizing temporal information in natural language text is a tough computational linguistic challenge [16, 29]. To the best of our knowledge, ICON [23] is the only known approach to automatically identify temporal constraints in software informal specifications without relying on a fixed set of patterns. ICON is mostly based on machine learning and authors report a recall of 60% on their ground truth, noticing that the approach has difficulty in recognizing implicit constraints.

To solve this challenge, we take advantage of the fact that we are operating on a domain (program operations) which is much narrower than the general linguistic problem, and does not require to address totally arbitrary natural language text.

Direction of a constraint. Existing approaches that derive properties involving multiple method calls, for example to identify equivalence metamorphic relations [6], can rely on the fact that such

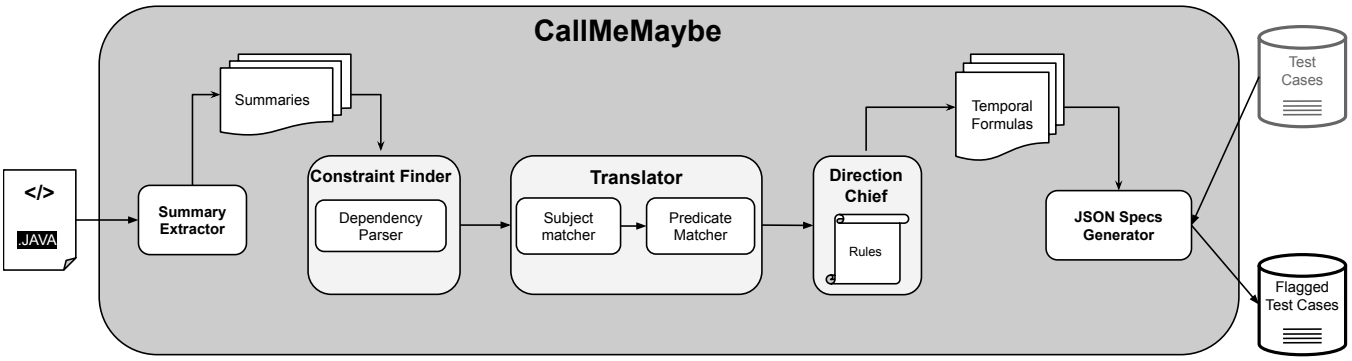

Figure 1: *CallMeMaybe*'s workflow

relations are bi-directional. Temporal constraints must obey precise directions, as an event either follows or precedes the other, and not the contrary. We thus define a new approach to determine the direction of the method calls in the constraint. ICON [23] relies on the heuristic of solely looking at the verb tense, which does not necessarily generalize well for any documentation style.

3 CALL ME MAYBE

CallMeMaybe finds and translates method call constraints by identifying their natural-language descriptions in code comments, and translating them into temporal constraints expressed with precise JSON structures. *CallMeMaybe* is generally applicable to natural language comments. In this paper, we define *CallMeMaybe* referring to Javadoc method summaries.

Pandita et al.'s study on the `java.io` package of the JDK indicates that informal descriptions of temporal constraints are mostly present in summaries and block tags [23]. In particular, the publicly available ICON dataset[1] reports a total of 90 temporal constraints that the authors identified by manually inspecting the package, 34 of which are in summaries, 56 in `@throws`[2], none in other parts of the Javadoc comments.

We decide to consider method summaries and ignore temporal constraints in exception tags. The main reason is that all the constraints reported in the `@throws` text of the `java.io` package prove to refer to exceptions explicitly declared in method signatures, i.e., exceptions that by default would not lead to a false alarm in test case generators when triggered [11, 22]. Beside, these constraints also show to be redundant with respect to what declared already in the summaries anyway.

Figure 1 overviews the *CallMeMaybe* process. *CallMeMaybe* extracts summaries from Javadoc method specifications (*Summary Extractor*), identifies description of constraints on sequences of method calls (*Constraint Finder*), translates the informal descriptions of constraints into temporal constraints (*Translator*), and generates precise JSON structures that can be integrated into test case generators to either augment or prune test suites (*JSON Specs Generator*).

3.1 *CallMeMaybe* Summary Extractor

The *CallMeMaybe* Summary Extractor takes in input the Java source code of a single class and operates on its methods' Javadoc. The Extractor both normalizes the English text of the summaries, by applying simplifications such as removing formatting information (e.g., HTML tags), and identifies the sentences that comprise the cleaned summary text. Each normalized sentence of each summary comprises the input for the next component.

3.2 *CallMeMaybe* Constraint Finder

The *CallMeMaybe* Constraint Finder processes the sentences that it obtains from the *CallMeMaybe* Summary Extractor to identify *propositions* that describe temporal constraints: It builds a semantic graph for the English text with the Stanford Parser [20], traverses the graph to identify propositions relevant to temporal dependencies, and exploits temporal dependencies to build temporal proposition series.

The Constraint Finder looks for specific dependencies inside the graph, namely subject dependencies (typically `nsubj` and `nsubjpass`) along with `advcl` and `advmod` dependencies. These dependencies identify adverbs and adverbial clauses, which fall into several categories, one of them being time[3]. By referring to time dependencies, this component identifies clauses that express the occurrence of an event with respect to another event.

The running example of Listing 5 and the function at line 2 in Algorithm 1 show how the *CallMeMaybe* Constraint Finder exploits temporal dependencies to build temporal proposition series. The *CallMeMaybe* Constraint Finder parses the method summary *"This method must be invoked before the thread is started"* (Listing 5) by generating and traversing the following semantic graph:

```
-> invoked/VBN (root)
        -> method/NN (nsubjpass)
                -> This/DT (det)
        -> must/MD (aux)
        -> be/VB (auxpass)
        -> started/VBN (advcl:before)
                -> before/IN (mark)
                -> thread/NN (nsubjpass)
```

[1] https://sites.google.com/site/temporalspec
[2] We treat every occurrence of `@exception` as `@throws`, since they are synonyms.
[3] https://universaldependencies.org/en/dep/advcl.html

```
                    -> the/DT (det)
      -> is/VBZ (auxpass)
```

The *CallMeMaybe* Constraint Finder observes that the dependency advcl:before (in bold in the graph above) identifies an adverbial clause together with its specific modifier, *before*, thanks to the Enhanced English Universal Dependencies parsing [30]. It discovers this advcl:before dependency, along with *subject* dependencies (lines 4 and 5 of Algorithm 1), and extracts three propositions from them:

nsubjpass: (This method, be invoked)
nsubjpass: (thread, is started)
advcl:before: (invoked, started)

As a final step, the *CallMeMaybe* Constraint Finder combines the three propositions in the following proposition series (line 6):

(This method, be invoked) **BEFORE** (thread, is started)

3.3 *CallMeMaybe* Translator

The *CallMeMaybe* Translator transforms temporal proposition series into temporal constraints, by recognizing the code identifiers involved in the temporal proposition that it obtains from the Constraint Finder (Algorithm 1, from line 9). The Translator finds code identifiers that correspond to both *explicit operation* and *implicit action*, the two constraint description styles that we discuss in Section 2, and substitutes the matching in the input proposition to generate a temporal constraint.

Explicit operations The *CallMeMaybe* Translator recognizes the presence of legitimate operations by relying on the SO_word2vec model [10], which is a word2vec model for the software engineering domain, pre-trained on over 15GB of textual data from Stack Overflow posts. The *CallMeMaybe* Translator starts from a basic golden set of three words (*operation*, *call*, and *use*) and queries the model with these positive examples to get a whole new set of related concepts that include words such as *invocation* or *return*. If any of the verbs in the proposition belongs to this set, the *CallMeMaybe* Translator assumes the presence of a temporal operation, with the proposition subject being an explicit reference to a code element and the verb being a legitimate operation. If it fails in recognizing an operation as legitimate, the Translator looks for the presence of an implicit action instead.

Implicit Actions The *CallMeMaybe* Translator identifies actions by relying on lexical similarities between the English words in the proposition and the code identifiers. It first tries to match the subject, by looking for a code identifier with very close lexical similarity with respect to the English subject. If it finds a promising candidate for the subject, it generates a list of suitable candidates for the predicate, by looking for invocations that match the code subject. Likewise, it assesses the list of candidate predicates according to the lexical similarity with respect to the English text. If it fails finding a match for the subject or the predicate, the Translator aborts the translation.

We illustrate the *CallMeMaybe* Translator by referring to the running example of Listing 5. The *CallMeMaybe* Translator identifies the subject of the first proposition (`This method, be invoked`) as the documented method (Algorithm 1, line 10), it normalizes "this method" with the corresponding method signature (Algorithm 1, line 12), and identifies the predicate, "invoked" as a concept of interest by means of the word2vec model strategy, thus successfully completing the matching of the first proposition (`This method,`

Algorithm 1 Temporal constraint finder and translator

```
1:  /** Find temporal constraints inside the comment text. Given the English
    text, the function traverses its semantic graph and builds a temporal
    proposition series.*/
2:  function FIND-TEMPORAL-CONSTRAINTS(natural language comment)
3:      semantic-graph =BUILD-SEMANTIC-GRAPH(natural language com-
    ment)
4:      subj-propositions = IDENTIFY-SUBJECT-RELATIONS(semantic-graph)
5:      adv-propositions = IDENTIFY-ADVERBIAL-RELATIONS(semantic-
    graph)
6:      proposition-series = CONNECT-PROPOSITIONS-WITH-MODIFIER(subj-
    propositions, adv-propositions)
7:  end function

8:  /** Translate identified propositions into Java expressions. Given the
    list of propositions in English, the function matches each part against
    code elements in the Java source code. */
9:  function TRANSLATE(proposition-series)
10:     if verb is explicit operation then
11:         if subject refers to documented method then
12:             NORMALIZE-PROPOSITION-SUBJECT(proposition-subject)
13:         else
14:             MATCH-PROPOSITION-SUBJECT(proposition-subject)
15:         end if
16:     else
17:         if verb is implicit method call then
18:             MATCH-PROPOSITION-SUBJECT(proposition-subject)
19:             MATCH-PREDICATE(matched-subject, proposition-verb)
20:         end if
21:     end if
22: end function

23: /** Given an English subject, finds the correct code elements to match
    it looking at lexical similarity. */
24: function MATCH-PROPOSITION-SUBJECT(proposition-subject)
25:     subjCandidateList = GET-SUBJECT-CANDIDATES(subject)
26:     matchedSubject = LEXICAL-MATCH(subject, subjCandidateList)
27:     if no match for subject then
28:         abort
29:     end if
30: end function

31: /** Given the matched subject, finds the correct code elements to match
    the predicate looking at lexical similarity. */
32: function MATCH-PREDICATE(matched-subject, proposition-predicate)
33:     predCandsList = GET-PREDICATE-CANDIDATES(matched-subject,
    proposition-predicate)
34:     matchedPredicate = LEXICAL-MATCH(proposition-predicate, pred-
    CandsList)
35:     if no match for predicate then
36:         abort
37:     end if
38: end function
```

be invoked). The *CallMeMaybe* Translator then matches the subject and predicate expressed as implicit operations in the second proposition, (thread, is started), by looking for syntactic similarities with code identifiers (Algorithm 1, line 17). It successfully matches subject *thread* to the instance of the receiving object of class Thread, and predicate *is started* to a call to method start().

The *CallMeMaybe* Translator completes the translation by instantiating the final proposition (This method, be invoked) BEFORE (thread, is started) with the identified matches to obtain the temporal constraint:

receiverObject.setDaemon(args[0]) **BEFORE** receiverObject.start()

The only part that still needs a translation is the specific temporal modifier (*BEFORE* in our running example).

3.4 *CallMeMaybe* Direction Chief

The *CallMeMaybe* Director Chief infers the direction of the temporal constraint from the temporal modifier of the proposition it receives from the *CallMeMaybe* Translator. This inference happens according to a small set of fixed rules, one for each temporal particle. The rules dictate the direction of the link between the two proposition: either a left ←, or a right → arrow. Table 1 shows the rules currently encoded in our *CallMeMaybe* implementation, that is, rules for AFTER, ONCE, BEFORE, PRIOR and UNTIL. The current *CallMeMaybe* implementation does not deal either with cycles or with temporal constraints involving more than two method calls. Yet, this small set of rules already proves to serve a large number of cases, and the set can be trivially extended to include the application of further rules.

Table 1: Rules currently encoded in *CallMeMaybe*

Modifier	Direction
AFTER	←
ONCE	←
BEFORE	→
PRIOR	→
(NOT) UNTIL	→

By referring back to out running example, the *CallMeMaybe* Direction Chief solves the full constraint of our example as:

receiverObject.setDaemon(args[0]) → receiverObject.start()

We selected a rule-based strategy over a heuristic-based strategy to reduce the risks of unsafe translations. ICON [23]'s heuristic relies on the tense of the verb: A past tense indicates a method call that must happen before, and vice-versa. This strategy does not generalize well to all temporal sentences. For instance, it would be difficult to asses the direction of a temporal constraint such as "start *should be invoked after* setDaemon *is invoked*" just by considering the sentence tense, while a rule-based strategy can safely deal with cases like this.

3.5 *CallMeMaybe* Generator

The *CallMeMaybe* Generator produces temporal constraints as simple JSON structures that indicate whether an operation should either precede or succeed any other operation. This JSON format has the advantage of being serializable and machine readable, making the output of *CallMeMaybe* ready to exploit by other tools such as automated test case generators. Listing 8 shows the excerpt of JSON output for our running example.

Listing 8: *CallMeMaybe*'s real example of JSON output

```
[
. . .
  {
    "signature": "java.lang.Thread.setDaemon(boolean)",
    "name": "setDaemon",
    "containingClass": {
      "qualifiedName": "java.lang.Thread",
      "name": "Thread",
      "isArray": false
  },
  . . .
  "mustPrecede": "receiverObject.start()",
  "mustFollow": "",
  . . .
]
```

The code of *CallMeMaybe* is open source. A complete replication package is available at:

https://github.com/ariannab/callmemaybe

4 EVALUATION

Our experimental evaluation addresses the following research questions:

- RQ1: Can *CallMeMaybe identify* natural language sentences that express temporal constraints and *translate* them into temporal formulas?
- RQ2: Do *CallMeMaybe* constraints reduce the manual effort of assessing false alarms and expected exceptions in automated testing when used as oracles?
- RQ3: How do *CallMeMaybe* constraints recognition capabilities compare with the state of the art technique ICON [23]'s?

We evaluated *CallMeMaybe* on a benchmark of 73 classes randomly selected from seven popular Java systems.

For each project, we randomly selected a sample comprising at least 10% of their documented classes, and then manually inspected their Javadoc, keeping only classes containing at least one temporal constraint. From this random sample, it seems the projects with the most prevalent number of constraints are the JDK and Apache Commons Collections. This is partly confirmed by Pandita et al.'s study which concerned the JDK specifically.

We produced the ground truth by inspecting all Javadoc sentences in the dataset and manually translating the sentences expressing temporal constraints into temporal formulas. Table 2 reports the considered projects, the number of randomly selected classes and temporal constraints that we manually identified for each project.

4.1 (RQ1) Translation Accuracy

We answer RQ1 by measuring precision and recall of *CallMeMaybe* on the 73 considered classes. Precision is defined as the ratio

Table 2: Ground truth: manually-identified temporal constraints

Project	Selected Classes	Temporal Constraints
Colt	9	9
Commons Collections	10	11
GraphStream	5	6
Guava	3	3
JDK	32	43
Lucene	7	10
Weka	7	7
TOTAL	73	89

between the number of correct outputs and the total number of outputs:

$$precision = \frac{|C|}{|C| + |S| + |W|}$$

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:

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

CallMeMaybe output is Correct (**C**) when the temporal constraint matches exactly our expected translation. It is Wrong (**W**) when the tool gives in output a constraint that does not match the one we are expecting. It is Spurious (**S**) when we are not expecting any output for a given natural language sentence, but *CallMeMaybe* produces one anyway in error. Finally, it is Missing (**M**) when we expect a constraint in output, but the output is not present. Table 3 reports the number of Correct, Missing, Wrong and Spurious temporal constraints that *CallMeMaybe* automatically identifies. The table also reports the good precision and recall that *CallMeMaybe* achieves for each project, with an average precision over all projects of 83% and an average recall of 70%. A prominent observation is that *CallMeMaybe* does not produce Spurious translations in this ground truth. That is, *CallMeMaybe* never considers a natural language sentence as reporting a legitimate temporal constraint when in fact it does not. This is not surprising, as *CallMeMaybe* Finder and Translator operate different checks before concluding that a natural language sentence is legitimately expressing a temporal constraint on Java elements. In the perspective of generating test oracles, this is a good result, since having a wrong oracle would be worse than having none. Nonetheless, *CallMeMaybe* produces 13 Wrong constraints with respect to our expectations. Some constraints can be indeed difficult to translate automatically, such as the one in Listing 9 from the JDK:

Listing 9: Temporal constraint that *CallMeMaybe* correctly identifies but wrongly translates

```
1 /** Removes all of this collection's elements that are also contained in the
2  * specified collection (optional operation). After this call returns, this
3  * collection will contain no elements in common with the specified collection
4  */
5 public void removeAll(java.util.Collection<?> c) ...
```

CallMeMaybe wrongly translates the above as:

```
receiverObject.contains(args[0]) <- receiverObject.removeAll(args[0])
```

While a correct translation would be:

```
Collections.disjoint(receiverObject, args[0]) <-
    receiverObject.removeAll(args[0])
```

CallMeMaybe also misses 14 translations our ground truth would expect. In particular, it misses all 3 we expect from Google Guava. While this library is well-documented, the documentation may express constraints using relatively complex concepts, hard for *CallMeMaybe* to automatically translate into code expressions. For example, a translation *CallMeMaybe* misses is one for the comment "The specified map [...] should not be accessed directly after this method returns". The concept of "accessing a map" does not matches a clear and specific method call, thus the current implementation of *CallMeMaybe* does not know how to deal with it.

4.2 (RQ2) Randoop Enhancment

We answer RQ2 by integrating *CallMeMaybe* with Randoop into a new tool, Randoop+*CallMeMaybe*, that enhances the output of Randoop [22] with the *CallMeMaybe* outputs.

When a Randoop test throws an exception, Randoop heuristically classifies the test as (1) error-revealing test, which requires manual inspection, (2) normal behavior, which Randoop labels as a regression test and proposes for manual inspection only in the case of future failures, or (3) invalid test, which Randoop discards.

Randoop+*CallMeMaybe* improves these heuristics by flagging each test that violates a *CallMeMaybe* temporal constraint, significantly reducing the effort for manually inspecting error revealing and regression tests.

Randoop error-revealing tests Let us consider a test that Randoop classifies as revealing an error because some method call that violates a temporal constraint throws an exception. Randoop would present the output for manual inspection with no further information. The manual inspection would reveal the violation of the temporal constraint and discard the test. Randoop+*CallMeMaybe* automatically identifies the violation of the temporal constraint and clearly flags the test as a false alarm, thus reducing the effort of manually inspecting the test results.

Randoop regression tests Let us consider a test that violates some temporal constraints and that Randoop classifies as expected behavior, being not aware of the temporal constraint informally described in the method summary (it is irrelevant if the call does or does not throw an exception). It would be error-prone to output a regression test that requires the current behavior (the specific value returned or the specific exception thrown), because such a regression test is brittle. In general, an invalid call may result in arbitrary behavior, so an implementation change could result in a different exception, or no exception, being thrown. The brittle regression test would fail after such an implementation change, wasting the developers' time. In some cases, a method might be specified to behave in a certain way (say, throw a certain exception) when a temporal constraint is violated. Discarding the test

Table 3: Effectiveness of *CallMeMaybe* on 73 classes

Project	Correct	Missing	Wrong	Spurious	Precision	Recall
Colt	9	0	0	0	1.00	1.00
Commons Collections	10	1	0	0	1.00	0.91
GraphStream	5	1	0	0	1.00	0.83
Guava	0	3	0	0	0.00	0.00
JDK	32	6	5	0	0.86	0.84
Lucene	4	3	3	0	0.57	0.57
Weka	2	0	5	0	1.00	0.29
TOTAL	62	14	13	0	0.83	0.70

is not incorrect in such circumstances, but it does make the regression test suite smaller. If a developer desires such tests despite their potential brittleness, then at the developer's option, Randoop+*CallMeMaybe* could output such tests, but with a descriptive comment. If the test fails in the future, the comment tells the developer what went wrong and how a legitimate sequence should look.

We generated test cases with both Randoop and Randoop+*CallMeMaybe* for the Commons Collections and JDK projects of Table 2. This accounts for most of the discovered constraints. We omitted the other projects in Table 2 because they have temporal constraints on file systems operation and on other domains that reduce the ability of Randoop to generate tests without some external suggestions for the inputs to be fed (like the names of files in the file system, to be passed to the open routine).

We ran both Randoop and Randoop+*CallMeMaybe* with a time limit of 100 seconds per class. Table 4 reports the number of test that Randoop erroneously classifies as error-revealing and normal tests, and that Randoop+*CallMeMaybe* flags as violating temporal constraints that *CallMeMaybe* automatically identifies in the method summaries. Randoop+*CallMeMaybe* flags 11,818 false alarms that Randoop would report as failing tests ("error-revealing" column), and 12,024 brittle regression tests that might have confusingly failed in a regression test suite for manual inspection with no useful information ("normal" column).

Table 4: Invalid tests generated by Randoop but prevented by Randoop+*CallMeMaybe*. The columns indicate how Randoop classified the tests.

Project	Error-revealing	Normal
Commons Collections	10948	8298
JDK	870	3726
TOTAL	11818	12024

We spot-checked 100 random tests, and confirm that Randoop+*CallMeMaybe* looks consistently correct in its assessments. Considering the high number of correct temporal constraints *CallMeMaybe* can deliver and the absence of spurious translation(RQ1), this is not surprising. We discuss two representative cases:

`IteratorChain` *example*. Let us consider the constructor of class `IteratorChain` presented in Listing 10.

Listing 10: Prescriptive constraint from class IteratorChain of Apache Commons Collections

```
1 /** Construct an IteratorChain with no Iterators.
2 You will normally use addIterator(Iterator) to add some iterators after using
3 this constructor. */
4 public IteratorChain() { ...
```

CallMeMaybe translates the protocol in Listing 10 into the following temporal constraint:

IteratorChain.addIterator(java.util.Iterator<? extends E>) ⟵ receiverObject.IteratorChain()

Listing 11 is an excerpt of a Randoop-generated test case for class `IteratorChain`.

Listing 11: Randoop generated test for class `IteratorChain` of Commons Collections

```
1  IteratorChain<Serializable> serializableIter0 =
2      new IteratorChain<Serializable>();
3
4  // The following exception was thrown during execution in test generation
5  try {
6    serializableIter0.remove();
7    org.junit.Assert.fail("Expected exception of type
8    java.lang.IllegalStateException; message: Iterator contains no
9    elements");
10 } catch (java.lang.IllegalStateException e) {
11   // Expected exception.
12   /* CallMeMaybe Constraint violation:
13   "Construct an IteratorChain with no Iterators.
14   You will normally use {@link #addIterator(Iterator)} to add some
15    iterators after using this constructor." */
16 }
```

The Randoop test instantiates a new `IteratorChain` at line 1, and invokes a removal operation on the newly instantiated iterator at line 6 with no actions in-between. *CallMeMaybe* correctly recognizes and translates the constraint for the constructor, reporting that a correct sequence would first invoke method `addIterator` on the newly instantiated iterator. In this case, the `IllegalStateException` exception is expected, as specified by `remove`, though a comprehensive explanation on how to avoid such behavior is not present. Randoop+*CallMeMaybe* correctly flags the test indicating the violation of the temporal constraints.

`Deflater` *example*. Let us consider the documentation of method `end()` of class `Deflater` in Listing 12.

Listing 12: Prescriptive constraint from class Deflater of the JDK

```
1 /** Closes the compressor and discards any unprocessed input. This
2 method should be called when the compressor is no longer being used, but
3 will also be called automatically by the finalize() method. Once this method
4 is called, the behavior of the Deflater object is undefined. */
5 public void end() { ...
```

CallMeMaybe translates the temporal constraint informally described in Listing 12 into the following temporal constraint, where the logical negation operator `!` before `receiverObject` indicates that no further invocation should happen on the receiver object):

receiverObject.end() → !receiverObject

Listing 13 shows a test that Randoop generates for class `Deflater` and Randoop+*CallMeMaybe* correctly flags as containing a violation of a temporal constraint that *CallMeMaybe* successfully identifies.

Listing 13: Real example of Randoop+*CallMeMaybe* behavior for the JDK

```
1 Deflater deflater2 = new Deflater((-1), true);
2 long long3 = deflater2.getBytesWritten();
3 deflater2.setLevel(2);
4 deflater2.end();
5
6 /* during test generation this statement threw an exception of type
7 java.lang.NullPointerException in error
8 But, CallMeMaybe Constraint violated too:
9 "Closes the compressor and discards any unprocessed input. This method
10 should be called when the compressor is no longer being used, but will also
11 be called automatically by the finalize() method. Once this method is called,
12 the behavior of the Deflater object is undefined." */
13 long long7 = deflater2.getBytesWritten();
```

The Randoop test instantiates a new `Deflater` object at line 1, calls `end()` at line 4, and invokes `getBytesWtitten` on the `Deflater` object at line 13. This raises a NullPointerException "in error" (line 7), while the correct usage of the class dictates that no further invocation should happen after `end()`, as Randoop did instead. Radoop+*CallMeMaybe* correctly identifies a constraint violation and flags the false alarm.

4.3 (RQ3) Comparison with ICON

We answer RQ3 by comparing *CallMeMaybe* with ICON [23] on the ICON's public dataset. As reported in the original publication[4], the ICON project website reports the natural language sentences manually labeled as temporal constraints or not, and annotated with a summary of the results of ICON with respect to this ground truth. The ICON website does not include either the implementation of the tool or the translations it produces, so we run *CallMeMaybe* on the ground truth of ICON as provided by the authors.

ICON produces temporal formulas using a formal language. Its Java ground truth comprises temporal constraints coming from `@throws` Javadoc tags and Javadoc summaries. *CallMeMaybe* operates on Javadoc summaries, thus we experimented on the Javadoc summaries in the ICON's dataset, which are all from the classes of package `java.io` of the JDK, and include 34 temporal constraints that the authors of ICON's paper manually identified.

CallMeMaybe correctly identifies all the 34 temporal constraints, thus achieving an accuracy of 100% with respect to the ground truth reported in the ICON's dataset. ICON's paper and dataset do not indicate the amount of temporal constrains that ICON identifies in the summaries of the `java.io` package alone, thus we cannot directly compare this data. *CallMeMaybe* improves over ICON by identifying legitimate exceptions that automatic test case generators would not, as discussed in Section 4.2, while ICON mostly identifies exceptions declared in the method's signatures, which automatic test case generators easily recognize as legitimate. *CallMeMaybe* achieves an overall precisions of 83% and a recall of 70%, as reported in Section 4.1, both higher than the precision and recall reported in the ICON's paper, 79% and 60%, respectively. We would like to notice that the data are not homogeneous, being computed on different data sets.

5 RELATED WORK

Temporal constraints, or *function precedence protocols* [27] or *method ordering constraints* [25] in the literature, are often inferred dynamically [2, 4, 12, 17, 18, 35, 36] and formalized as finite state machines [1, 25]. The inferred specification can be used as a target for verification (e.g., model checking) [7, 8, 13, 15, 21] or to support testing activities, for example by comparing the execution of generated tests against inferred API protocols [26, 34]. Our work is motivated by the observation that free-text Javadoc summaries of methods and classes often document temporal constraints. This implies that we can support testing by formalizing the *documented* behavior of a program as specified by API developers, and doing so in a static fashion.

Doc2Spec is the first approach [37] to infer temporal constraints on resources following a similar intuition. Specifically, Doc2Spec infers temporal constraints formalized as finite state machines following a specific template: "resource creation methods, followed by resource manipulation methods, followed by resource release methods". The more recent ICON by Pandita et al. [23] correctly observes that temporal constraints are not necessarily restricted to such a template, and attempts to offer a more general technique to recognize them based on specific machine learning features. However, to the best of our knowledge, ICON was not applied to any specific task in testing or other software engineering activities. *CallMeMaybe* is a new approach that improves and complements all the above.

The intuition of deriving executable oracles from natural language specification is exploited by different techniques in the state of the art. The iComment [31], aComment [32], @tComment [33], Toradocu [14] and Jdoctor [5] approaches all extract some form of specifications from code comments. These approaches all rely on semi-structured natural language documentation, meaning they cannot deal with unstructured text that documents properties in Javadoc summaries. Other similar approaches such as ALICS [24] and MeMo [6] deal with unstructured documentation, as *CallMeMaybe* does. However, they do not recognize and model temporal constraints.

[4] https://sites.google.com/site/temporalspec

6 CONCLUSION

CallMeMaybe derives temporal constraints from natural language specification of Java systems. *CallMeMaybe* generates machine-parsable sequences of Java expressions that can improve the results of automatic test case generators by flagging false alarms and explaining expected exceptions. *CallMeMaybe* analyzes *unstructured* natural language text and recognizes informal descriptions of temporal constraints in free text. It then finds a suitable translation into Java expressions, recognizing the right *direction* of the method calls involved.

CallMeMaybe proves to be accurate in its temporal constraint translations, with a precision of 83% and a recall of 70% on a ground truth of 89 expected Java specifications across 73 classes of different Java systems. In this paper, we integrate *CallMeMaybe* into Randoop to show how to use *CallMeMaybe* temporal constraints to improve automatic test case generation, while state-of-the-art approaches, like ICON, do not exploit temporal constraints for software engineering activities. Randoop+*CallMeMaybe* correctly flags thousands false alarms [22], thus substantially reducing the effort of manual inspection. *CallMeMaybe* is the first approach that aids automatic testing by relying of natural language information with temporal constraints, while state-of-the-art approaches focus on preconditions, postconditions, exceptions [5, 14, 24, 33] and metamorphic relations [6].

The results that we document in this paper suggest further improvements of *CallMeMaybe*. *CallMeMaybe* translations can be used to improve the complex temporal relations modeled with finite state machines [1, 25, 27] by using *CallMeMaybe* constraints to verify the correctness of dynamically inferred models. Indeed, while *CallMeMaybe* derives constraints based on developers' specifications of *expected* behavior, dynamically inferred models represent the *actual* program behavior. In this paper we used *CallMeMaybe* to infer temporal constraints from method summaries; *CallMeMaybe* can be straightforwardly extended to other kinds of informal specifications. Constraints in class summaries would deserve a study on their own, as class documentation tends to differ from method summaries, and is not yet widely explored in the state of the art [28]. Class summaries are often expressed by means of code snippets (as our examples in Section 2 show), thus opening new opportunities to automatically distinguish code snippets that represent good and bad usages of the class.

REFERENCES

[1] Glenn Ammons, Ras Bodik, and James R Larus. 2002. Mining Specifications. In *Proceedings of the Symposium on Principles of Programming Languages*. ACM, 4–16.

[2] Glenn Ammons, Rastislav Bodík, and James R. Larus. 2002. Mining specifications. In *POPL 2002: Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Portland, Oregon, 4–16.

[3] Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. 2015. The Oracle Problem in Software Testing: A Survey. *IEEE Transactions on Software Engineering* 41, 5 (2015), 507–525.

[4] A. W. Biermann and J. A. Feldman. 1972. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.* C-21, 6 (June 1972), 592–597.

[5] 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 the International Symposium on Software Testing and Analysis (ISSTA '18)*. ACM.

[6] Arianna Blasi, Alessandra Gorla, Michael D Ernst, Mauro Pezze, and Antonio Carzaniga. 2021. MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation. *Journal of Systems and Software* 181 (2021), 111041.

[7] E. M. Clarke, Orna Grumberg, and Doron Peled. 1999. *Model Checking*. MIT Press.

[8] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. 2013. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodology* 22, 3 (July 2013), 25:1–25:46.

[9] 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.

[10] Vasiliki Efstathiou, Christos Chatzilenas, and Diomidis Spinellis. 2018. Word embeddings for the software engineering domain. In *Proceedings of the Working Conference on Mining Software Repositories*. ACM, 38–41.

[11] 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.

[12] Mark Gabel and Zhendong Su. 2008. Javert: Fully automatic mining of general temporal properties from dynamic traces. In *FSE 2008: Proceedings of the ACM SIGSOFT 16th Symposium on the Foundations of Software Engineering*. Atlanta, GA, USA, 339–349.

[13] Dimitra Giannakopoulou and Corina S. Păsăreanu. 2009. Interface generation and compositional verification in JavaPathfinder. In *FASE 2009: Fundamental Approaches to Software Engineering*. York, UK, 94–108.

[14] Alberto Goffi, Alessandra Gorla, Michael D. Ernst, and Mauro Pezzè. 2016. Automatic Generation of Oracles for Exceptional Behaviors. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16)*. ACM, 213–224.

[15] Gerard J. Holzmann. 1997. The Model Checker SPIN. *IEEE Transactions on Software Engineering* 23, 5 (May 1997), 279–295. Special Issue: Formal Methods in Software Practice.

[16] Ruihong Huang, Ignacio Cases, Dan Jurafsky, Cleo Condoravdi, and Ellen Riloff. 2016. Distinguishing Past, On-going, and Future Events: The EventStatus Corpus. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics.

[17] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. 2015. General LTL Specification Mining. In *ASE 2015: Proceedings of the 30th Annual International Conference on Automated Software Engineering*. Lincoln, NE, USA.

[18] David Lo and Siau-Cheng Khoo. 2006. SMArTIC: Towards building an accurate, robust and scalable specification miner. In *FSE 2006: Proceedings of the ACM SIGSOFT 14th Symposium on the Foundations of Software Engineering*. Portland, OR, USA, 265–275.

[19] Davide Lorenzoli, Leonardo Mariani, and Mauro Pezzè. 2008. Automatic Generation of Software Behavioral Models. In *30th International Conference on Software Engineering (ICSE)*. IEEE Computer Society.

[20] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations (ACL '14)*. Association for Computational Linguistics, 55–60.

[21] Kenneth L. McMillan. 1993. *Symbolic model checking*. Kluwer Academic Publishers.

[22] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. 2007. Feedback-Directed Random Test Generation. In *Proceedings of the International Conference on Software Engineering (ICSE '07)*. ACM, 75–84.

[23] Rahul Pandita, Kunal Taneja, Laurie Williams, and Teresa Tung. 2016. ICON: Inferring temporal constraints from natural language api descriptions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE Computer Society, 378–388.

[24] 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.

[25] Michael Pradel, Philipp Bichsel, and Thomas R Gross. 2010. A framework for the evaluation of specification miners based on finite state machines. In *2010 IEEE International Conference on Software Maintenance*. IEEE, 1–10.

[26] Michael Pradel and Thomas R Gross. 2012. Leveraging test generation and specification mining for automated bug detection without false positives. In *Proceedings of the International Conference on Software Engineering*. IEEE, 288–298.

[27] Murali Krishna Ramanathan, Ananth Grama, and Suresh Jagannathan. 2007. Path-sensitive inference of function precedence protocols. In *Proceedings of the International Conference on Software Engineering*. IEEE, 240–250.

[28] Pooja Rani, Sebastiano Panichella, Manuel Leuenberger, Andrea Di Sorbo, and Oscar Nierstrasz. 2021. How to identify class comment types? A multi-language approach for class comment classification. *Journal of Systems and Software* 181 (2021), 111047.

[29] Tomohiro Sakaguchi, Daisuke Kawahara, and Sadao Kurohashi. 2018. Comprehensive Annotation of Various Types of Temporal Information on the Time Axis.

In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA).

[30] Sebastian Schuster and Christopher D Manning. 2016. Enhanced english universal dependencies: An improved representation for natural language understanding tasks. In *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC'16)*. 2371–2378.

[31] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. 2007. /* iComment: Bugs or Bad Comments? */. In *Proceedings of the Symposium on Operating Systems Principles (SOSP '07)*. ACM, 145–158.

[32] 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.

[33] 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.

[34] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. 2009. MSeqGen: Object-Oriented Unit-Test Generation Via Mining Source Code. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE '09)*. ACM, 193–202.

[35] Westley Weimer and George Necula. 2005. Mining temporal specifications for error detection. In *TACAS 2005: Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Edinburgh, UK, 461–476.

[36] Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: Mining Temporal API Rules from Imperfect Traces. In *ICSE 2006, Proceedings of the 28th International Conference on Software Engineering*. Shanghai, China, 282–291.

[37] Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. 2009. Inferring Resource Specifications from Natural Language API Documentation. In *Proceedings of the International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, 307–318.