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

Arianna Blasi  
Università della Svizzera italiana  
Lugano, Switzerland

Michael D. Ernst  
University of Washington  
Seattle, USA

Alessandra Gorla  
IMDEA Software Institute  
Madrid, Spain

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.

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
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 predating 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 temporal 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 temporal 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 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 adopt 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
1| /** This method can only be called after at least one [@link #next] or [@link #previous] method call */
2| public void remove() ... 

Listing 4: Temporal constraint from GraphStream class summary
1| /***
2| * Allows to run a layout in a distinct thread.
3| *
4| *
5| *
6| * Once you finished using the runner, you must call release() to break the
7| * link with the event source and stop the thread. The runner cannot be used
8| * after ...
9| */
10| 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
1| /*** This method must be invoked before the thread is started. */
2| 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 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
1| /***
2| * Overrides the [@link ImmutableMultiset] static methods that lack [@link ImmutableSortedMultiset] equivalents with deprecated, exception-throwing versions. This prevents accidents like the following:
3| *
4| *
5| *
6| * BAD CODE!
7| * The returned multiset is actually an unsorted ImmutableMultiset!
8| */
9| 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
1| /***
2| * Defines an iterator that operates over a [@code Map].
3| *
4| *
5| * In use, this iterator iterates through the keys in the map. After each
6| * call to [@code code.next ()], the [@code code.getValue ()] method provides direct
7| * access to the value. The value can also be set using [@code code.setValue ()].
8| *
9| * [...]
10| * MapIterator<String, Integer> it = map.mapIterator();
11| * while (it.hasNext ()) {
12| *     System.out.println (it.next () + " = " + it.getValue () + "\n"
13| * }
14| */
15| 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
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 admod 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:

\[^1\]https://sites.google.com/site/temporalspec
\[^2\]We treat every occurrence of @exception as @throws, since they are synonyms.
\[^3\]https://universaldependencies.org/dep/advcl.html
As a final step, the CallMeMaybe Constraint Finder combines the three propositions in the following proposition series (line 6):

```
| nsubjpass: (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, invocan

---

**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-GRAH(natural language comment)
4:  subj-propositions = IDENTIFY-SUBJECT-RELATIONS(semantic-graph)
5:  adv-propositions = IDENTIFY-ADVERBINAL-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, predCandsList)
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>
<thead>
<tr>
<th>Modifier</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFTER</td>
<td>←</td>
</tr>
<tr>
<td>ONCE</td>
<td>↔</td>
</tr>
<tr>
<td>BEFORE</td>
<td>→</td>
</tr>
<tr>
<td>PRIOR</td>
<td>←</td>
</tr>
<tr>
<td>NOT) UNTIL</td>
<td>→</td>
</tr>
</tbody>
</table>

By referring back to our 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 assess 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
1 [
2   ...
3   {
4     "signature": "java.lang.Thread.setDaemon(boolean)",
5     "name": "setDaemon",
6     "containingClass": {
7       "qualifiedName": "java.lang.Thread",
8       "name": "Thread"
9     }
10   },
11   ...
12   "mustPrecede": "receiverObject.start()",
13   "mustFollow": "",
14   ...
15 ]
```

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
with respect to our expectations. Some constraints can be indeed difficult to translate automatically, such as the one in Listing 9 from the JDK:

```
receiverObject.removeAll(java.util.Collection<?> c)
```

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) Randoope Enhancement

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

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

**Randoope error-revealing tests** Let us consider a test that Randoope classifies as revealing an error because some method call that violates a temporal constraint throws an exception. Randoope 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. Randoope+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.

**Randoope regression tests** Let us consider a test that violates some temporal constraints and that Randoope 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 differst 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 2: Ground truth: manually-identified temporal constraints**

<table>
<thead>
<tr>
<th>Project</th>
<th>Selected Classes</th>
<th>Temporal Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colt</td>
<td>9</td>
<td>9</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>GraphStream</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Guava</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>JDK</td>
<td>32</td>
<td>43</td>
</tr>
<tr>
<td>Lucene</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>Weka</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td><strong>TOTAL</strong></td>
<td><strong>73</strong></td>
<td><strong>89</strong></td>
</tr>
</tbody>
</table>

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

\[
\text{precision} = \frac{|C|}{|C| + |S| + |W|}
\]

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:

\[
\text{recall} = \frac{|C|}{|C| + |W| + |M|}
\]

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**

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

CallMeMaybe wrongly translates the above as:

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

CallMeMaybe correctly translates the above as:

```
receiverObject.removeAll(args[0])
```

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.

CallMeMaybe wrongly translates the above as:

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

While a correct translation would be:

```
receiverObject.removeAll(java.util.Collection<?> c)   
```

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.

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

\[
\text{precision} = \frac{|C|}{|C| + |S| + |W|}
\]

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:

\[
\text{recall} = \frac{|C|}{|C| + |W| + |M|}
\]

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**

\[
\text{precision} = \frac{|C|}{|C| + |S| + |W|}
\]

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:

\[
\text{recall} = \frac{|C|}{|C| + |W| + |M|}
\]

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**

```java
/** Removes all of this collection’s elements that are also contained in the
 * specified collection (optional operation). After this call returns, this
 * collection will contain no elements in common with the specified collection
 */
```

```
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:

```
receiverObject.removeAll(java.util.Collection<?> c)   
```

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.

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

```java
/** Removes all of this collection’s elements that are also contained in the
 * specified collection (optional operation). After this call returns, this
 * collection will contain no elements in common with the specified collection
 */
```
Table 3: Effectiveness of CallMeMaybe on 73 classes

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

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.

<table>
<thead>
<tr>
<th>Project</th>
<th>Error-revealing</th>
<th>Normal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commons Collections</td>
<td>10948</td>
<td>8298</td>
</tr>
<tr>
<td>JDK</td>
<td>870</td>
<td>3726</td>
</tr>
<tr>
<td>TOTAL</td>
<td>11818</td>
<td>12024</td>
</tr>
</tbody>
</table>

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.

```java
// CallMeMaybe translates the protocol in Listing 10 into the following temporal constraint:
IteratorChain.addIterator(java.util.Iterator<? extends E>) ← receiverObject.IteratorChain()
```

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

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

```java
Listing 11: Randoop generated test for class IteratorChain of Commons Collections
1 IteratorChain<Serializable> serializableItor0 =
2 new IteratorChain<Serializable>();
3 try {
4 serializableItor0.remove();
5 org.junit.Assert.fail("Expected exception of type java.lang.IllegalArgumentException ; message: Iterator contains no elements");
6 } catch (java.lang.IllegalArgumentException e) {
7 // Expected exception .
8 // CallMeMaybe Constraint violation : "Construct an IteratorChain with no Iterators ."
9 You will normally use @link #addIterator ( Iterator ) to add some iterators after using this constructor : */

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 IllegalArgumentException 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 // during test generation this statement threw an exception of type
6 java . lang . NullPointerException in error
7 if (false) { . . .
8 But, CallMeMaybe Constraint violated too :
9 CallMeMaybe achieves an overall precision of 83% and a recall of 70%,
10 as reported in Section 4.1, both higher than the precision and recall
11 reported in the ICON’s paper, 79% and 60%, respectively. We would
12 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.
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


