The Journal of Systems & Software 181 (2021) 111041

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

MeMo: Automatically identifying metamorphic relations in Javadoc comments for test automation ☆

Arianna Blasi [a,*], Alessandra Gorla [b], Michael D. Ernst [c], Mauro Pezzè [a,d], Antonio Carzaniga [a]

[a] USI Università della Svizzera Italiana, Switzerland
[b] IMDEA Software Institute, Spain
[c] University of Washington, USA
[d] SIT Schaffhausen Institute of Technology, Switzerland

ARTICLE INFO

Article history:
Received 1 December 2020
Received in revised form 3 July 2021
Accepted 7 July 2021
Available online 15 July 2021

Keywords:
Software testing
Test oracle generation
Natural language processing

ABSTRACT

Software testing depends on effective oracles. Implicit oracles, such as checks for program crashes, are widely applicable but narrow in scope. Oracles based on formal specifications can reveal application-specific failures, but specifications are expensive to obtain and maintain. Metamorphic oracles are somewhere in-between. They test equivalence among different procedures to detect semantic failures. Until now, the identification of metamorphic relations has been a manual and expensive process, except for few specific domains where automation is possible. We present MeMo, a technique and a tool to automatically derive metamorphic equivalence relations from natural language documentation, and we use such metamorphic relations as oracles in automatically generated test cases. Our experimental evaluation demonstrates that 1) MeMo can effectively and precisely infer equivalence metamorphic relations, 2) MeMo complements existing state-of-the-art techniques that are based on dynamic program analysis, and 3) metamorphic relations discovered with MeMo effectively detect defects when used as test oracles in automatically-generated or manually-written test cases.

1. Introduction

Oracles are necessary in software testing to reveal implementation defects and errors. An ideal oracles would be complete and generic. Complete means that the oracle would reveal every deviation from the desired application-specific behavior. Generic means that the oracle can be used with any test, including automatically generated ones. However, there is a trade-off between cost to generate oracles and their effectiveness. At one end of the spectrum, there are implicit oracles that check basic consistency rules such as the dereference of null pointers. They are cheap to obtain and totally generic, but are far from complete, as they reveal only relatively simple, program-independent properties (Barr et al., 2015). At the other end of the spectrum are application-specific oracles. These are often written manually by developers for specific test cases. However, this requires a significant effort and is not applicable to automatically generated test suites (Fraser and Arcuri, 2013). Application-specific oracles that can be derived from formal specifications, but generating and maintaining formal specifications can be very expensive (Balcer et al., 1989; Doong and Frankl, 1994), and is considered cost-effective only in some domains.

Metamorphic and differential testing balance cost, completeness, and genericity. Metamorphic testing oracles reveal failures by checking metamorphic relations: application-specific symmetries and equivalences (Chen et al., 1998, 2003). For example, an oracle for a commutative function $sum(a, b)$ is the metamorphic relation $sum(a, b) \equiv sum(b, a)$. Metamorphic relations are partial specifications that are much less onerous to write and maintain than full behavioral specifications. Metamorphic oracles are generic, and yet they can detect application-specific errors and therefore are complementary to the oracles generated (along with test suites) by automatic test case generators such as Randoop (Pacheco et al., 2007) and EvoSuite (Fraser and Arcuri, 2013). Such tools produce test suites that rely mainly on *implicit* oracles or common contract violations,[1] and generate *regression* assertions, that is, oracles that detect differences from a previous version of the software under test.

Metamorphic relations are particularly useful when correctness might be hard to define at a higher level. Identifying metamorphic relations has been for a long time a largely manual task

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail address: arianna.blasi@usi.ch (A. Blasi).

[1] https://randoop.github.io/randoop/manual/#kinds_of_errors.

https://doi.org/10.1016/j.jss.2021.111041

carried out by experts of the application domain. The few techniques to automatically identify metamorphic relations either focus on specific domains, such as model transformations (Troya et al., 2018), or work under strict assumptions, such as dealing only with functions with numeric parameters (Zhang et al., 2019). A few techniques automatically generate composite metamorphic relations by combining simple manually-identified metamorphic relations (Xiang et al., 2019; Liu et al., 2012). Other techniques use either code structure information to train machine learning classifiers (Kanewala and Bieman, 2013; Kanewala, 2014) or dynamic analysis information (Su et al., 2015; Goffi et al., 2014) to identify metamorphic relations.

This paper presents MeMo, a technique and a tool to automatically infer equivalence metamorphic relations from code comments *written in natural language*. Of course, the quality of the results strictly depends on the completeness and correctness of the developers' comments. The benefit lays in the fact that these relations can be readily used to generate test oracles: MeMo can further reduce the cost and increase the effectiveness of metamorphic testing.

The MeMo tool analyzes the Javadoc documentation of a Java class under test. MeMo produces equivalence metamorphic relations in the form of Java assertions that can be used as metamorphic oracles. MeMo consists of four components. The two core ones are a MR finder whose task is to identify sentences that describe metamorphic relations, and a translator that translates such natural-language sentences into executable specifications, or assertions.

Previous techniques derive executable specifications from technical documentation in the form of structured natural language (Tan et al., 2012; Pandita et al., 2012; Motwani and Brun, 2019; Pandita et al., 2016). Some of the authors of this paper previously developed a framework to infer pre- and post-conditions from semi-structured Javadoc comments, such as the `@param` tag that describes one parameter (Blasi et al., 2018; Goffi et al., 2016). We observe that metamorphic relations tend to be described in method summaries, that is, the *unstructured* portion of Javadoc documentation. MeMo is the first approach that uses unstructured technical documentation in natural language to derive metamorphic relations.

SBES (Goffi et al., 2014; Mattavelli et al., 2015) infers relations in Java programs similar to the ones discovered by MeMo. SBES is a dynamic analysis that formulates conditions based on observed behaviors, and cannot distinguish between correct and incorrect behaviors. By contrast, MeMo relies on static information (code comments), which is more general than specific runtime behaviors and more likely to be correct. Our work combines the vision of SBES (Goffi et al., 2014; Mattavelli et al., 2015) (automatically inferring equivalence metamorphic relations), and the power of natural language processing (Blasi et al., 2018; Goffi et al., 2016).

It may seem superfluous to extract metamorphic relations from comments that are in principle already known to developers. MeMo automates the manually intensive process of translating this information into executable specifications that are usable for tasks such as testing and automatic program repair (Carzaniga et al., 2015, 2010, 2013, 2009) Moreover, MeMo can expose bugs — documented relations that do not hold in practice — by translating and executing equivalence metamorphic relations during testing.

We evaluated MeMo on nine mature open-source Java projects, both in isolation to identify metamorphic relations, and in a testing framework by using those relations as oracles. The experimental results (Section 4) indicate that MeMo produces Java executable specifications with a precision of 91% and a recall of 69%. Furthermore, those specifications are useful as test oracles when used in combination with EvoSuite and Randoop, and even with manually-written assertions. MeMo identifies valid relations that SBES does not find, and identifies relations that are generally more complex than those found by SBES.

In summary, this paper makes the following contributions:

- We present a novel technique to automatically infer equivalence metamorphic relations from natural-language sentences found in code comments.
- We implemented our technique in a tool called MeMo.
- Experiments show that MeMo generates relations that can effectively detect implementation defects when used in test cases automatically generated by Randoop and EvoSuite, and in test suites manually-written by developers.

The remainder of this paper is organized as follows. Section 2 describes the problem of deriving metamorphic relations from code comments. Section 3 describes our solution, including the high-level architecture of our MeMo tool and the technical details of its components. Section 4 evaluates MeMo according to three different criteria: (1) effectiveness in both identifying MRs in code comments expressed in natural language and translating them to executable specifications, (2) comparison with the state-of-the-art SBES technique, and (3) detection of implementation defects when using identified MRs as test oracles. Section 5 discusses threats to the validity of our approach. Section 6 reviews related work. Section 7 concludes.

2. Metamorphic relations in Javadoc

Metamorphic relations are properties of the intended behavior of a software component or system. Equivalence relations between operations are among the most studied metamorphic relations (Chen et al., 2018).

Metamorphic relations can serve as oracles in software testing. For instance, a `sum` operation representing a commutative addition operation should produce the same result with any permutation of the input parameters. The metamorphic relation `sum(a,b) == sum(b,a)` must hold for all values of the parameters `a` and `b`.

Because these relations are important and informative, they are often described in code comments. For example, the following Google Guava (guava, 2020) comment states that two different method calls should have the same functional behavior on an object of class `Iterables`.

Listing 1: Equivalence relation in Guava method summary

```
/** Returns an iterable whose iterators cycle indefinitely over the
    provided
elements.

After remove is invoked on a generated iterator, the removed
    element will no
longer appear in either that iterator or any other iterator created
    from the same
source iterable. That is, this method behaves exactly as
Iterables.cycle(Lists.newArrayList(elements)).
The iterator's hasNext method returns true until all of the original
    elements have
been removed.

Warning: Typical uses of the resulting iterator may produce an
    infinite loop. You
should use an explicit break or be certain that you will
eventually remove all the elements.

To cycle over the elements n times,
use the following: Iterables.concat(Collections.nCopies(n,
Arrays.asList(elements))) */

public static <T> Iterable<T> cycle(T... elements) { ...
```

Method summaries are unstructured text. They use different wordings to describe equivalence relations, with sentences like "…this method behaves exactly as…" or "…it is identical to…", and they mix in code fragments. The challenges that the two core components of MeMo face for inference of equivalence relations from code comments are:

1. Identifying sentences that describe equivalent behaviors, which may be embedded in large text blocks like the one in Listing 1. This requires a technique that understands the semantics of the natural language used by developers in their comments.
2. Identifying the code elements involved in the metamorphic relation. Matching specific terms in the natural language sentence to the corresponding code elements is complicated by the fact that they may be lexically different, and terms in comments may refer to code entities (i.e. methods, classes, fields, and method parameters) from a variety of foreign scopes. This matching is necessary to translate metamorphic relations into valid Java executable statements, so they can be used directly in Java code, for instance as test oracles in unit testing methods.

To illustrate these challenges and our tool MeMo, we list some examples of Javadoc comments and MeMo's output.

Documented method

```
/** Equivalent to newReentrantLock(lockName, false). */
public ReentrantLock newReentrantLock(String lockName) { … }
```

MeMo output

```
methodResultID.equals(receiverObjectClone.newReentrantLock(args
    [0], false))
```

In this simple example, the comment is a single sentence that directly states an equivalence property. MeMo must distinguish the English from the code snippet, interpret the English, and recognize that the first argument in the documentation refers to the method's only parameter, while the second argument is a Boolean literal. In the output produced by MeMo, `methodResultID` refers to the return value of the method call, `receiverObjectClone` refers to a cloned instance of the object on which the method is invoked, and `args[0]` refers to the first actual argument for that invocation (more technical details in Section 3).

Resolving parameter names in the natural language comment may be less straightforward. This next example uses fields of external classes:

Documented method

```
/** Calling this method is equivalent to call composeInverse(r,
RotationConvention.VECTOR_OPERATOR). */
public
applyInverseTo(org.apache.commons.math3.geometry.euclidean.
    threed.FieldRotation<T>
 r) { …
```

MeMo output

```
methodResultID.equals(receiverObjectClone.composeInverse(args[0],
org.apache.commons.math3.geometry.euclidean.threed.
    RotationConvention.VECTOR_OPERATOR))
```

Here is an example from Guava's `com.google.common.primitives.Shorts` class

Documented method

```
/** Returns a fixed-size list backed by the specified array, similar to
Arrays#asList(Object[]). The list supports [...] */
public static List<Short> asList(short... backingArray)
```

MeMo output

```
methodResultID.equals(java.util.Arrays.asList(args[0]))
```

MeMo recognizes that only the first sentence states an equivalence relation, that `Arrays` comes from a different library (`java.util`), and that `short...` is convertible to `Object[]`.

Some methods do not return values that can be compared with `==` or `equals()`. The following example from Guava class `LongAdder` shows how MeMo handles methods with `void` return types.

Documented method

```
/** Equivalent to add(−1). */
public void decrement() { … }
```

MeMo output

```
receiverObjectID.add(−1);
receiverObjectClone.decrement();
assert(receiverObjectClone.equals(receiverObjectID));
```

Through the `receiverObjectClone`, MeMo can compare the states after two separate invocations of the methods involved in the metamorphic relation.

When a metamorphic relation involves multiple methods, MeMo combines them into either chains of method calls or nested calls. Let us consider again the example in Listing 1:

Documented method

```
/** … this method behaves exactly as Iterables.cycle(Lists.
    newArrayList(elements)). … */
```

MeMo output

```
methodResultID.equals(Iterables.cycle(com.google.common.collect.
    Lists.newArrayList(args[0])))
```

Code snippets in Javadoc comments may even include multiple statements, and not only expressions:

Documented method

```
/** For each occurrence of an element e in occurrencesToRemove,
removes one occurrence of e in multisetToModify. Equivalently, this
method modifies multisetToModify so that multisetToModify.count(e
    ) is
set to Math.max(0, multisetToModify.count(e) −
occurrencesToRemove.count(e)). This is not the same as
multisetToModify. Multiset#removeAll removeAll(
    occurrencesToRemove),
which removes all occurrences of elements that appear in
occurrencesToRemove. However, this operation is equivalent to,
    albeit
sometimes more efficient than, the following: for (E e :
occurrencesToRemove) { multisetToModify.remove(e); } */
```

MeMo output

```
methodResultID==[ for (Object e : args[1]) { args[0].remove(e);} ]
```

The above translation is a compact representation of the assertions that MeMo produces in this case. In a nutshell, MeMo declares a new method and includes the code statements in squared parenthesis as its

Finally, some comments describe *conditional* equivalence, such as the following comment in the `com.google.common.collect.Multisets` class:

Documented method

```
/** Removes a number of occurrences of the specified element
    from this
multiset. If the multiset contains fewer than this number of
    occurrences to
```

```
begin with, all occurrences will be removed. Note that if
      occurrences == 1,
this is functionally equivalent to the call remove(element) */
```

MeMo output

```
if (args[1] == 1)
assert(methodResultID==(receiverObjectClone.remove(args[0])));
```

We now describe how MeMo identifies and translates metamorphic equivalence relations into executable specifications.

3. MeMo

MeMo discovers metamorphic relations in Javadoc documentation. It produces executable Java assertions that can be used as test oracles.

The MeMo architecture combines a comment processor, a Metamorphic Relations (MR) finder, a translator, and an executor. The comment processor parses the input source code and produces a cleaned representation of the Javadoc comments of each method. The MR finder identifies, among the extracted sentences, the ones that mention a metamorphic relation. The translator processes each sentence containing a metamorphic relation, interpreting its natural-language parts as well as its embedded code to produce a metamorphic relation as a Java assertion. The executor weaves the produced assertions into existing code (for instance, a test suite), to test whether the metamorphic relation holds at run time.

3.1. Comment processor

A Javadoc method comment consists of a free-text summary description followed by tag blocks (`@param`, `@return`, `@throws`, `@since`, `@version`, etc.). From our manual inspection of Java projects, most metamorphic properties appear in method summaries. This intuition is supported by the official Oracle documentation, which states that the summary should provide a general description of the method, including any interesting semantic property, while the tag blocks should convey more specific, narrower information (Oracle, 2020). Based on this information, MeMo's comment processor ignores tag blocks.

The comment processor also removes formatting information such as HTML markup. It stores text marked with `@code` and `@link` inline tags for later analysis. Content enclosed in `@code` tags may identify a code snippet that MeMo needs later in the translation process. Content inside `@link` tags may be helpful in identifying external dependencies, which in turn are essential to resolve symbols and method calls referenced in the documentation.

The comment processor splits the cleaned summary text of each method into sentences, and it forwards them to the MR finder for further analysis.

Fig. 1 shows how the comment processor behaves on a code snippet excerpted from the Google Guava API. The comment processor analyzes the documentation of method `asList(short... backingArray)` to identify and store information associated to @links and @code tags, information that MeMo uses in later steps of the translation process. Then, the comment processor splits the paragraph into sentences at full stops (periods).

3.2. MR finder

The MR finder decides whether each sentence extracted with the comment processor describes a metamorphic relation. It returns true if the sentence contains one of a set of equivalence phrases, or if the sentence is semantically similar to one that contains one of the equivalence phrases. Both tests require the presence of a method signature in the sentence.

3.2.1. Equivalence phrase search

The equivalence phrase search uses a fixed set of ten equivalence phrases mined from real-world Javadoc documentation. The corpus is a set of 4741 Javadoc sentences randomly chosen from the documentation of seven widely used Java projects: Apache Commons Collections, Apache Commons Math, Apache Hadoop, Apache Lucene, Eclipse Vert.x, Google Guava, and GWT.

We manually classified each sentence as expressing a metamorphic equivalence relation (positive examples) or not, and further manually identified the relevant equivalence phrases used to express the metamorphic relations. The resulting set of equivalence phrases is: *equivalent, similar, analog, like, identical, behaves as, equal to, same as, alternative, replacement for*.

Fig. 2 illustrates how the MR finder retains only one sentence from the running-example of method `asList` from Fig. 1, that is, the one containing one of the equivalence phrases followed by a method signature.

3.2.2. Semantic expansion of MR equivalence phrases

Using just a predefined set of equivalence phrases to identify sentences expressing metamorphic relations would limit MeMo's generalization capabilities, since developers' jargon may vary across projects. For instance, the set of manually-mined equivalence phrases would not find the following equivalence in method `org.graphstream.graph#push` of the `Graphstream` API:

Listing 2: Equivalence relation in Graphstream method summary

```
/** A synonym for add(Edge). */
void push(org.graphstream.graph.Edge edge) { ...
```

The MR finder returns true if a sentence in a code comment contains text that is semantically similar to one of the equivalence phrases in Section 3.2.1. First, MR finder builds a normalized version of the comment sentence by adding an explicit subject when missing and substituting the method signature it refers to with ``that method''. For comment 2 of Section 2, MR finder normalizes the code comment sentence *"A synonym for add(Edge)"* to "`this method is` a synonym for `that method`". (MR finder adds the verb *to be* if there is no verb, as in this case). Second, MeMo builds a dummy sentence for each equivalence phrase in the form of: "`method` ⟨*equivalence phrase*⟩ `that method`". MeMo compares the normalized sentence to each dummy sentence: "this method `is equivalent to` that method" ..."this method `is same as` that method".

The comparison uses Word Mover's Distance (Kusner et al., 2015) to measure the semantic similarity between the sentences, and answer the question: "is the given sentence in the comment also expressing an equivalence?". The MR finder returns true if any of the Word Mover's Distance computations returns a value below 20%, which means a similarity of at least 80%. We set the threshold via experimentation.

3.3. Translator

For each sentence that the MR finder identifies as describing a metamorphic relation, the translator outputs a Java assertion. The translator produces executable Java code that may use the following placeholders: (1) `methodResultID` represents the result of the documented method; (2) `receiverObjectID` is the receiver (the target object) on which the documented method is called; (3) `receiverObjectClone` is a clone of the target object that may be used to invoke the equivalent method or code snippet. MeMo performs method calls only on the cloned instance, to avoid affecting the state of the original object. Also, the cloned object is useful to compare side effects by comparing

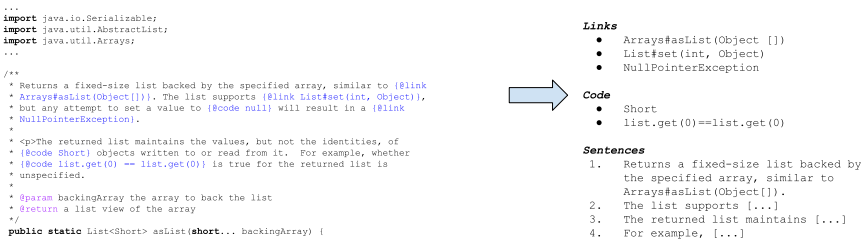

Fig. 1. Information retained by the comment processor.

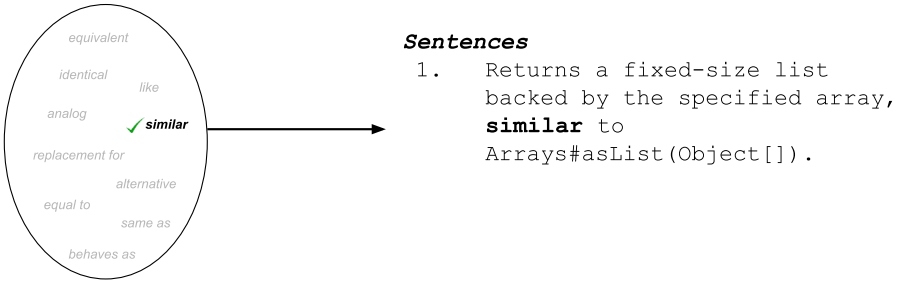

Fig. 2. Sentence retained by the MR finder.

the observable *state* of the cloned and original objects. Section 2 presents examples of the use of all placeholders.

The translator must first identify and resolve all the method signatures mentioned in the sentence, whether in natural language or in a code fragment.

Consider this example:

> ...equivalent to ByteBuffer.allocate(8).putLong(value).array().

There is no direct link to any `ByteBuffer` class in the comment itself nor in the source code. The class this comment belongs to does not *use* the library, which is mentioned in the comment only to highlight a MR. MeMo explores the other project packages and external dependencies to find the right match.

A signature in a comment may use different parameter values, hard-coded or not: In the above example, `8` is hard-coded (MeMo recognizes it is a literal to be interpreted as an integer when translated into code); `value` is the argument name of the documented method (MeMo employs a syntax match to recognize the code element the comment is referring to).

When a comment uses a call chain (as in the above example) rather than a single signature, multiple resolutions must be performed. For the above example, MeMo solves all these tasks to produce the final code translation:

```
methodResultID.equals(java.nio.ByteBuffer.allocate(8).putLong(args
    [0]).array())
```

The next sections describe how MeMo deals with each challenge.

3.3.1. Resolving symbols

MeMo must resolve textual representations of symbols (methods, classes, variables, etc.) to generate code for them.

When processing qualified symbols such as `ClassName.name` and `ClassName#name`, MeMo first tries to find `ClassName` in the same package, then within the whole project under analysis, and finally within external dependencies that appear as `import` statements. Fig. 3 shows an example of symbol resolution. After identifying `java.util.Arrays.asList` as the right candidate from the external dependencies on the left-hand side of the figure, MeMo checks its arguments (second green check, in the center) and the return types of the two equivalent methods (third green check, on the right) for type compatibility. MeMo finds a compatible match, and produces the final translation for the sentence in the blue box: `methodResultID.equals(java.util.Array.asList(arg[0]))`.

For unqualified symbols such as `name` or `#name`, MeMo searches first in the class itself, then in supertypes, then in the entire package, then in the external dependencies. It matches them to method names, fields, formal parameters, and literals.

For chained method calls such as `methodA().methodB()`, MeMo resolves from left to right. It uses the return type of `methodA()` as the preferred scope in which to resolve `methodB`.

3.3.2. Parsing non-trivial code fragments

So far, the examples focused mainly on single method signatures or a chain of method calls. However, there are even more complex mentions inside documentation comments, such as whole code snippets.

Non-trivial code fragments might contain expressions, nested calls, and control structures (`if`, `for`, etc.). MeMo uses a pragmatic approach to deal with non-trivial code. First, MeMo attempts to compile the code using an in-memory compiler (In memory compiler, 2020). The compilation might fail because the code refers to unknown symbols. In that case, MeMo reads the compiler errors and tries to solve the missing symbols (see Section 3.3.1). After solving all missing symbols, and in any case at the end of the synthesis of the specification, MeMo runs the compiler again, to confirm that the output assertion is valid Java code.

3.3.3. Translating conditional equivalence

A metamorphic property does not necessarily hold under all circumstances. A comment might express specific conditions under which a property holds, in the following format: "if [or when, in case, etc.] *condition,* then this call is equivalent to *other method call*". The condition might be directly expressed with code (for instance, "`if collection.isEmpty()`") or in natural language ("if `collection` is empty"). In the first case, MeMo can parse the code expressing the condition directly as with code snippets. In the latter case, MeMo relies on natural language processing. Specifically, MeMo uses the Stanford Parser (Marneffe et al., 2006) to identify the *subject* and *predicate* of the condition. Then, since both subject and predicate must correspond to source code elements, MeMo tries to find their name within the code. Once translated the condition, MeMo proceeds by translating the rest of the metamorphic property as usual.

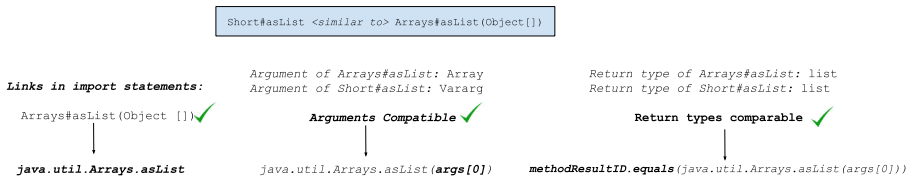

Fig. 3. Example of symbol resolution. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

3.4. Executor

The translator output maps a single method to code fragments (simple method call, chain of calls, or code snippet) for which the metamorphic relation is supposed to hold. Such translations may contain placeholders, which must be properly replaced at run time depending on the scope and context.

MeMo uses aspect-oriented programming for this task. Specifically, MeMo uses an aspect template with a join point *around* the method call for which we have a translation. When a test suite — whether manually or automatically generated — contains a method call for which MeMo is aware of a translation, the executor triggers the aspect and compares the execution of the original and supposedly equivalent code fragment declared in the documentation.

Algorithm 1 Executor

```
1:  /** Given the code translation of a metamorphic relation, embeds it within the Aspect template to obtain an executable assertion.*/
2:  function POPULATE-ASPECT-TEMPLATE(translation, receiverObjectID)
3:      if translation contains receiverObjectClone then
4:          CLONE = GENERATE RECEIVER OBJECT CLONE(receiverObjectID)
5:      if translation contains code fragment then
6:          EMBED CODE FRAGMENT IN DUMMY METHOD(code fragment)
7:          CLONE.DUMMY-METHOD()
8:      /** Call to the documented method already existing in the test suite.*/
9:      METHODRESULTID = RECEIVEROBJECTID.DOCUMENTEDMETHODCALL()

10:     if translation contains receiverObjectClone then
11:         ASSERT(RECEIVEROBJECTCLONE.EQUALS(receiverObjectID))

12:     else if translation contains methodResultID then
13:         ASSERT(translation)
```

The executor in MeMo fulfills this task by populating the Aspect template as shown in algorithm 1. Since MeMo uses an *around* pointcut, it can perform some operations both before the method invocation mentioned in the translation (before line 8) and after (from line 9).

If a translation contains object cloning, the clone must reflect the state of the receiver object before the test invokes the documented method. On said clone, the code fragment can be then executed.

After the test suite invokes the documented method, the results of the executions involved in the translation of the metamorphic relation can be compared. Since the comparison is expressed as an assertion (lines 11 and 13), the test case will pass if the metamorphic relation does hold as documented, and will fail otherwise.

4. Evaluation

Our experimental evaluation aims to answer the following research questions:

- RQ1: effectiveness of MeMo. Can MeMo *identify* natural language sentences that express metamorphic properties and *translate* them into executable assertions?
- RQ2: comparison against state of the art. How does MeMo compare with SBES in terms of identified equivalence relations?
- RQ3: usefulness of MeMo assertions as test oracles. Do MeMo assertions improve testing when used as oracles?

4.1. Experimental setting

We evaluated MeMo on a benchmark of 113 classes randomly selected from nine popular Java systems.

One author inspected all 7189 Javadoc sentences and manually translated all those that express a metamorphic relation into a code assertion. Table 1 reports statistics.

We computed precision and recall according to the following formulas:

$$precision = \frac{|Correct|}{|Correct| + |Wrong| + |Spurious|}$$

$$recall = \frac{|Correct|}{|Correct| + |Wrong| + |Missing|}$$

where

Correct = *true positive*: MeMo's output is non-empty and matches the ground truth.

Missing = *false negative*: MeMo's output is empty but the ground truth is not.

Wrong = *non-empty false positive*: MeMo's output is non-empty and does not match the ground truth.

Spurious = *empty false positive*: MeMo's output is non-empty and the ground truth is empty.

We addressed RQ1 by experimenting with all sentences in the benchmark. We addressed RQ2 by experimenting with the subset of sentences that are used in the SBES paper (Mattavelli et al., 2015), that is 792 sentences belonging to 220 methods of 16 classes of the `collect` package of the Google Guava library. We addressed RQ3 by experimenting with the 1274 sentences of the 27 Guava classes, to mitigate the effort required to manually exclude false positives from the mutation analysis, as we further discuss in 4.4.

Table 1
Ground truth: manually-identified metamorphic relations (MR).

Project	URL	Randomly selected classes	Sentences	MRs
Colt	https://dst.lbl.gov/ACSSoftware/colt	9	477	19
ElasticSearch	https://www.elastic.co	10	228	14
GWT	http://www.gwtproject.org	17	448	44
GraphStream	http://graphstream-project.org	3	126	11
Guava	https://github.com/google/guava	33	1558	80
Hibernate	https://github.com/hibernate	5	126	5
JDK	https://github.com/openjdk/jdk	23	3381	72
Math	https://github.com/apache/commons-math	9	653	30
Weka	https://www.cs.waikato.ac.nz/ml/weka	4	192	6
Total		113	7189	281

Table 2
Effectiveness of MeMo on 7189 sentences from 113 classes.

Project	Correct	Missing	Wrong	Spurious	Precision	Recall
Colt	11	8	0	0	1.00	0.58
ElasticSearch	8	6	0	0	1.00	0.57
GWT	12	31	1	1	0.86	0.27
GraphStream	9	2	0	0	1.00	0.82
Guava	62	16	2	2	0.94	0.78
Hibernate	3	2	0	0	1.00	0.60
JDK	59	11	2	6	0.88	0.82
Math	26	4	0	2	0.93	0.87
Weka	3	1	2	0	0.60	0.50
Total	193	81	7	11	0.91	0.69

4.2. RQ1: Effectiveness of MeMo in translating Javadoc comments

Table 2 reports the effectiveness of MeMo in translating Javadoc comments to executable metamorphic relations. MeMo translates JavaDoc sentences into executable assertions with a precision of 91% and a recall of 69%.

Most *missing translations* of MeMo depend on comments that describe parameter values with complex natural language expressions and with little or no code. In Table 2, we see how GWT is the project on which MeMo achieves the poorest recall. A representative example of the reason why is the following comment from method `endsWithRtl` of the `com.google.gwt.i18n.shared.BidiUtils` class.

> */** Like #endsWithLtr(String, boolean), but assumes str is not HTML
> /
> HTML−escaped. */*

that can be translated to the assertion

```
methodResultID.equals(endsWithLtr(args[0], false))
```

where the `false` value for the second argument comes from the intuition that "str is not HTML/HTML-escaped" refers to the second parameter of method `endsWithLtr`, which is the boolean variable `isHtml`. MeMo's NLP techniques do not infer the information required to translate this sentence, and GWT has many comments similar to this one.

The *spurious translations* of MeMo depend on summaries that encompass a considerable amount of information and typically mix mathematical expressions with natural language and code. For such summaries it is not always possible to write a translation, so our ground truth is empty. MeMo, nonetheless, detects the presence of a MR and attempts a translation. A representative example is the summary of method `plus` of class `java.time.YearMonth`:

> */** Returns a copy of this year−month with the specified amount
> added.
> This returns a YearMonth, based on this one, with the amount in
> terms of the
> unit added. If it is not possible to add the amount, because the unit
> is not
> supported or for some other reason, an exception is thrown.
>
> If the field is a ChronoUnit then the addition is implemented here.
> The supported
> fields behave as follows:
>
> MONTHS − Returns a YearMonth with the specified number of
> months added.
> This is equivalent to plusMonths(long).
> YEARS − Returns a YearMonth with the specified number of years
> added. This is
> equivalent to plusYears(long).
> DECADES − Returns a YearMonth with the specified number of
> decades added.
> This is equivalent to calling plusYears(long) with the amount
> multiplied by 10.
> CENTURIES − Returns a YearMonth with the specified number of
> centuries added.
> This is equivalent to calling plusYears(long) with the amount
> multiplied by 100.
> MILLENNIA − Returns a YearMonth with the specified number of
> millennia added.
> This is equivalent to calling plusYears(long) with the amount
> multiplied by 1,000.
> ERAS − Returns a YearMonth with the specified number of eras
> added. Only two
> eras are supported so the amount must be one, zero or minus one.
> If the amount
> is non−zero then the year is changed such that the year−of−era is
> unchanged.
> */*

The few *wrong translations* of MeMo derive either from some mixing between mathematical notations and code or from an incomplete translation. A representative example is the following comment taken from method `forArray(T... array)` of `com.google.common.collect.Iterators` class:

> */** The Iterable equivalent of this method is either
> Arrays#asList(Object[]), ImmutableList#copyOf(Object[])},
> or ImmutableList#of. */*

MeMo's translator does not parse multiple metamorphic relations in a single sentence, thus its output is incomplete:

```
methodResultID.equals(java.util.Arrays.asList(args[0]))
```

When identifying a metamorphic relation inside a comment, the simple syntactic match against the hard-coded set of equivalence phrases achieves a recall of 65% on our dataset. WMD provides a boost of 4% in recall by retrieving further matches (without losing precision). WMD can detect variations of the equivalence phrases, e.g., going from *behaves as* to *"...behaves exactly as..."*, or from *same as* to *"...has the same behavior..."* and to *"...has the same effect as..."*, so that not every variation needs to be hard-coded. WMD can also detect subtle similarities, like the one shown in Listing 2 (i.e., *"A synonym for..."*).

4.3. RQ2: Comparison with SBES

We compared MeMo with Search-Based Equivalent Synthesis (SBES) (Goffi et al., 2014; Mattavelli et al., 2015), a dynamic analysis technique that finds sequences of equivalent method calls through a search-based algorithm.

4.3.1. Comparison of the two techniques

MeMo deduces metamorphic relations from code documentation, and its output is as deterministic and sound as the documentation written by developers. SBES infers relations from executing the code. Its output depends on the initial test suite used in the search-based algorithm. SBES infers *likely* relations, which must be manually confirmed by the user.

MeMo runs much faster than SBES. For example, SBES takes 5 h to analyze the class `java.util.Stack`. MeMo takes a few seconds.

4.3.2. Experimental comparison between SBES15 and MeMo

We compared MeMo and SBES15 (Mattavelli et al., 2015) by executing the corresponding tools on the SBES15 dataset, and by manually intersecting the set of relations produced with the two tools.

MeMo inferred six metamorphic relations from the SBES15 dataset.

MR 1 corresponds to the comment

```
/** ... This method is equivalent to tailMultiset(lowerBound,
    lowerBoundType).headMultiset(upperBound, upperBoundType)
    . */
```

in method `TreeMultiset.subMultiset()`. MeMo translates it to

```
methodResultID.equals(receiverObjectID.tailMultiset(args[0],args[1]).
    headMultiset(args[2],args[3]))
```

which states that the result of method `subMultiset` is the same as calling method `tailMultiset` with the first two arguments of `subMultiset`, followed by method `headMultiset` with the last two arguments.

MR 2 corresponds to comment:

```
/** Equivalent to size() == 0, but can in some cases be more
    efficient. */
```

in `ArrayListMultimap` and 5 more classes regarding method `isEmpty()` that MeMo translates as:

```
methodResultID == (receiverObjectID.size() == 0)
```

which means that the result of the documented method `isEmpty` should be the same of comparing the result of method `size` on the receiver object to the value 0.

MR 3 corresponds to comment:

```
/** ... Equivalent to (but expected to be more efficient than): for (V
    value : values) { put(key, value); } */
```

in `ArrayListMultimap` and 5 more classes on method `putAll()` that MeMo translates as:

```
methodResultID==[ for (V value : args[1]) { receiverObjectID.put(args
    [0], value); } ]
```

which means that the effect of invoking the documented method `putAll` should be the same obtained by the code snippet in squared parenthesis.

MR 4 corresponds to comment:

```
/** ... If values is empty, this is equivalent to removeAll(key). */
```

in class `ArrayListMultimap` and 5 more on method `replaceValues()` that MeMo translates as:

```
if(!args[1].iterator().hasNext())
 {methodResultID.equals(receiverObjectID.removeAll(args[0]))}
```

Table 3
MeMo's performance on SBES15 dataset considering documented MR: *Both* means that such MRs are found by both MeMo and SBES15. *SBES15-only* means such MRs are found by SBES15 but missed by MeMo. *MeMo-only* are the MRs found by MeMo and missed by SBES15.

Discovered documented MRs			
Both	SBES15-only	MeMo-only	Total
8	5	20	33

MeMo understands that there is a condition that must hold for the documented method `replaceValues` to be comparable to calling method `removeAll` with the first argument. Notice that the second argument, `values`, is an iterator, thus the emptiness condition is verified via `!values.iterator().hasNext()`.

MR 5 corresponds to comment:

```
/** ... Note that if occurrences == 1, this method has the identical
    effect to #add(Object). This method is functionally equivalent (
    except in the case of overflow) to the call addAll(Collections.
    nCopies(element, occurrences)), which would presumably
    perform much more poorly. */
```

on `ConcurrentHashMultiset` and 4 more classes regarding method `add()` that MeMo translates as:

```
if (args[1] == 1) {
    receiverObjectID.add(args[0]);
    receiverObjectClone.add(args[0],args[1]);
    assert(receiverObjectClone.equals(receiverObjectID));
}
&&
methodResultID==(receiverObjectID.addAll(java.util.Collections.
    nCopies(args[1],args[0])))
```

This is the most complicated MR for MeMo, as it combines different features explained in Section 3.2. MeMo uses `&&` to combine two metamorphic properties expressed in two different sentences. The first property is conditional, similarly to relation 4. The second property presents nested calls, with the innermost being in a different system (Java standard library).

MR 6 corresponds to comment:

```
/** ... Note that if occurrences == 1, this is functionally equivalent
    to the call remove(element). */
```

on ConcurrentHashMultiset and 4 more classes on method `remove()`, and MeMo translates it as follows:

```
if (args[1] == 1) {
    receiverObjectID.remove(args[0]);
    receiverObjectClone.remove(args[0],args[1]);
    assert(receiverObjectClone.equals(receiverObjectID));
}
```

This case is similar to relation 4. The first difference is that the condition is expressed as code inside the comment, rather than in natural language. The second difference is that the result of the documented method and the equivalent one are not directly comparable, since one returns `int` and the other `boolean`. Thus, the invocations must be done on two separate, cloned instances of the same receiver object to later compare their statuses.

Comparing MeMo and SBES15 results. In comparing MeMo with SBES15, we take into account that MeMo can only infer metamorphic relations that are documented. On the other hand, SBES may infer properties that are not documented, while missing those that are. Table 3 summarizes the results for the documented relations.

As for Documented properties, that is, properties which MeMo can actually identify and translate, we have:

Both : of the reported 8 equivalences found both by MeMo and SBES15, five are instances of MR 4. SBES15, however,

missed the same property on one class (ImmutableListMultiMap). Two other sequences are instances of MR 6, which, again, actually exists on multiple classes. By relying on the static information of the Javadoc documentation, MeMo, can synthesize the property correctly for all the classes involved. The last sequence, instead, corresponds to the first part of the composed MR 5. As in the case before, SBES15 missed some classes for the first past, detecting it only on one class. The second part of MR 5 was never found by SBES15.

SBES-only : the reported 5 sequences refer to the same comment: *...so, values().size() == size()*. Differently from the heuristics MeMo uses, this comment directly reports some code without preceding it with any keyword that could suggest the presence of an equivalence.

MeMo-only : MeMo found 20 relations missed by SBES15. MR 1 (one instance), MR 3 (six instances), and MR 2 (six instances) were never found by SBES15. The others (MR 4, MR 5, and MR 6) were found only partially by SBES15.

Clearly, not all the MR of the SBES15 dataset are documented. In total, SBES15 finds 188 true positive equivalent sequences. Of these, as per Table 3, 33 are documented: SBES15 found 40% of them, while MeMo 85%. This confirms our hypothesis that the two techniques complement each other, and the amount of true positives increases when they are used in combination.

4.4. RQ3: Usefulness of MeMo assertions as test oracles

We measure the amount of mutants (artificial bugs) detected by test cases augmented with MeMo assertions. This is a proxy measure of the quality of the oracles (strength of the assertions). We use test suites automatically generated with both EvoSuite (Fraser and Arcuri, 2013) and Randoop (Pacheco et al., 2007), and the original developers' test suite.

To mitigate the effort required to manually exclude false positives from the mutation analysis, our experiment uses only Guava as the program under test. Guava represents 1/3 of MeMo's correct translations, with few spurious results (Table 2). Guava comes with a solid manually-written test suite of 5681 test cases, a challenging competitor for MeMo's assertions.

The experiment proceeded in three phases:

Phase 1: Generating test suites. We retrieved the developers' test suite from GitHub and Maven[2] repositories, and automatically generated test suites with EvoSuite and Randoop. We use the respective default timeout, that is, 60 s for EvoSuite and 100 s for Randoop. Randoop and EvoSuite generate different test suites depending on the initial seed. This paper reports the mean of the results of three generations with different seeds. Our goal is to compare the original Randoop and EvoSuite test suites with the test suites augmented with MeMo oracles. We compare two different variants of each original test suite: test suites with only *implicit* oracles, and test suites with both implicit and *regression* oracles.

We discard classes for which the generator either cannot produce a test suite or does not contain method calls to which MeMo can attach assertions via Aspects. For example, for class `com.google.common.collect.ArrayListMultimap` EvoSuite only outputs 5 test cases, none of which covering methods for which MeMo as assertions. For this evaluation. This leaves ten classes for EvoSuite and ten for Randoop. The two sets of classes are not the same. Only Randoop generates tests for `com.google.collect.Multiset` and `com.google.collect.Multimap`, and only EvoSuite generates tests for `com.google.concurrent.RateLimiter` and `com.google.base.CharMatcher`.

Phase 2: Enhancing test suites with MeMo assertions. We invoked MeMo on the subject classes to infer the metamorphic relations and insert assertions within all test cases as additional test oracles.

We executed the augmented test suites, and manually inspected each failing test case to discard any Aspect that raises a failure, to avoid biases in mutation analysis (the next phase). To be clear, no test gets eliminated: We only prevent the attachment of faulty Aspects to them. In this way, we eliminate assertions leading to failures from the analysis, and we assure that subsequent failures are due to assertions that kill mutants.

A few failures are due to equality checks being too strict, for instance, classes that do not override the default Java equality implemented by `Object.equals()`. These are false positives for our oracles. In some cases, test failures may reveal information about the implementation that is not explicit from the documentation. For example, the documentation of method `asList()` implemented by most classes of `primitives` package in Guava asserts the equivalence of method `asList()` to the same method implemented in class `java.util.Arrays`, but this is true only under specific conditions: The implementations produce the same results when methods are invoked with parameter `vararg`, but produce different data structures, albeit with the same data, when invoked with an array. EvoSuite does not invoke the methods using `vararg` parameters, while Randoop does.

Phase 3: Mutation analysis. We generated mutants for the classes under test with Major (Just et al., 2011). We executed all Randoop and EvoSuite test cases on the mutants with different oracles: implicit oracles only, regression oracles, and both implicit and regression oracles augmented with MeMo oracles. We performed analogous steps with the developers' test suite: we first ran the test suite as-is (i.e., with developers' manually written assertions), and then augmented with MeMo assertions.

Our analysis considers only mutants relevant for the studied assertions: That is, mutations of methods executed by at least one test case that contains a MeMo assertion.

Fig. 4 presents the results of our experiments. MeMo's automatically generated assertions complement both automatically generated test suites and developers' test suites.

Improvement over implicit oracles. MeMo's automatically generated assertions are much more effective than implicit oracles: automatically generated test suites reveal many more mutants when augmented with MeMo assertions.

Improvement over developers' oracles. Developers' assertions alone kill 269 mutants. MeMo kills 40 of these mutants. Most importantly, 34 times MeMo assertions kill more mutants than developers' assertions. A total of 81 mutants survive both developers' and MeMo assertions: Some may be equivalent mutants, some others may be mutants that are not exercised by the test suite, a few others may be defective mutants that do not compile.

Improvement over regression oracles. EvoSuite and Randoop kill 233 and 230 mutants, respectively, when executed with regression assertions. 67 and 69 of those mutants are killed equally well by MeMo oracles, meaning that MeMo assertions are as effective as EvoSuite and Randoop regression assertions in 29% and 30% of the cases, respectively. Adding MeMo assertions to regression test suites brings 25 additional kills to EvoSuite and 35 to Randoop. Some mutants are not killed by either regression or MeMo oracles (76 for EvoSuite and 112 for Randoop, on average).

[2] https://mvnrepository.com/artifact/com.google.guava/guava-tests/19.0.

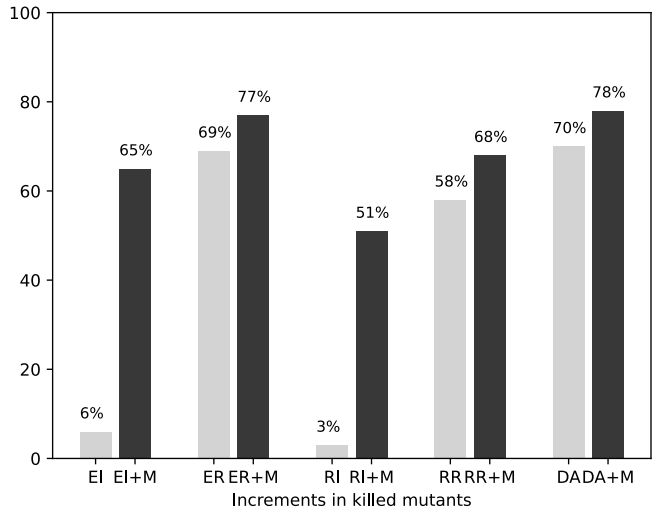

Fig. 4. Improvement in mutants killed with MeMo oracles. Each pair of bars compares a test suite without MeMo oracles to one with MeMo oracles. EI stands for EvoSuite Implicit oracles, ER for EvoSuite Regression oracles, RI for Randoop Implicit oracles and RR for Randoop Regression oracles. DA means developers' assertions, referring to the developers' test suite. +M indicates augmented test suites with MeMo oracles.

Analysis of MeMo's kills. MeMo's assertions kill not only mutants that affect the interface level, as may be expected, but also deeper methods under test, as illustrated by the following two examples.

The first example comes from method `com.google.common.primitives.Longs.fromByteArray`:

Listing 3: Equivalence relation in com.google.common.primitives.Longs method summary

```
/** Returns the long value whose byte representation is the given 8 bytes, in
big−endian order; equivalent to Longs.fromByteArray(new byte[] { b1, b2, b3, b4,
b5, b6, b7, b8}). */
static long fromBytes(byte b1, byte b2, byte b3, byte b4, byte b5, byte b6, byte
b7, byte b8) { ...
```

Method `fromByteArray` calls method `fromBytes` after spitting the array into single bytes. Major mutates `fromByteArray` with the following mutant:

```
166:LVR:0:POS:com.google.common.primitives.Longs@fromByteArray(byte[]):295:0
 |==> 1
```

This mutant is killed by a developer test case augmented with the assertion that MeMo automatically generates from the MR informally described in Listing 3: *Returns the long value whose byte representation is the given 8 bytes, in big-endian order; equivalent to Longs.fromByteArray*. The same test case does not, however, kill the mutant without MeMo's oracles. We observe that the bug is not a simple intra-method issue, but involves the invocation of two methods.

The second example comes from `com.google.common.math.DoubleMath`, a class with many dependencies. Major mutates both the class itself *and* its dependencies. In particular, Major seeds several bugs into class `com.google.common.math.DoubleUtils`.

MeMo correctly identifies the MR informally described in the comment of method `DoubleMath`:

Listing 4: Equivalence relation in com.google.common.math.DoubleMat method summary

```
/** ... This is equivalent to, but not necessarily implemented as, the
expression !Double.isNaN(x) && !Double.isInfinite(x) && x == Math.rint(x). */
public static boolean isMathematicalInteger(double x) { ...
```

and produces an executable assertion. Method `isMathematicalInteger` invokes some methods of class `DoubleUtils`, such as `isFinite(double)`. Major seeds bugs in the body of the methods invoked in `isMathematicalInteger` leading to several mutants like:

```
187:ROR:<=(int,int):==(int,int):com.google.common.math.DoubleUtils@isFinite(double):75:getExponent(d)
 <= MAX_EXPONENT |==> getExponent(d) == MAX_EXPONENT
```

that successfully get killed by MeMo's assertion derived from Listing 4.

These two examples illustrate how MeMo's assertions can kill mutants that alter both the interfaces and the intra-methods calls, even ones that may survive developers' oracles.

We conclude our analysis of MeMo's kills, with some data about the effectiveness of MeMo's oracles for different kinds of mutations. We inspected the mutants that tests do not kill with either regression or developers' oracles alone, but kill with MeMo's assertions, and classified them according to the mutation operators. We observe that MeMo is particularly effective in killing mutants generated with `LVR` (Literal Value Replacement) and `OR` (Operator replacement) mutation operators,[3] which constitute 80% of MeMo's mutants killing. The remaining 20% are mutants generated with `EVR` (Expression Value Replacement) and `STD` (Statement deletion) mutation operators.

5. Threats to validity

The manually-written ground truth may be prone to human error. The translations targets nine different Java projects, coming from different teams and companies to avoid overfitting the capabilities of the translator on specific styles of comment.

The projects selected to extract the MR finder equivalence phrases might not contain all the possible types of metamorphic properties that can be expressed in Javadoc comments. In an attempt to generalize as much as possible, we labeled more than four thousand sentences belonging to seven different projects from diverse ecosystems (Google, Apache, Eclipse), and then evaluated the same features on four more projects from yet different ecosystems. The sentences might still not represent all the possible ways MRs are expressed in comments: MeMo's MR Finder semantic expansion aims to address this challenge. Manual labeling sentences can be prone to human error. However, the high percentage of correct translations compared the relatively low number of missing and spurious translations indicates that possible errors are limited.

6. Related work

Test oracles from metamorphic relations. The very first intuition of exploiting what we today call *metamorphic properties* is found in the work of Davis and Weyuker (1981) and Weyuker (1982). The authors proposed the idea of a pseudo-oracle to test programs for which a testing oracle is not available (for example, because developers have only a vague idea of the correct result). Essentially, a pseudo-oracle is an independent copy of the original program; it is written in a different way, but according to the

[3] http://mutation-testing.org/doc/major.pdf.

same specification. When the program and its independently developed copy are executed on the same inputs, the tester can infer the correctness of the original program if their outputs coincide. It is however with the work of Chen et al. (1998) that the term *metamorphic* testing is introduced. The authors described the idea as an approach to derive new test cases from other existing ones, with the aim to uncover errors that exist in similar applications. The same authors later defined the concept of metamorphic relations, used to derive the new test cases (Chen et al., 2003). From that moment on, metamorphic testing started gaining popularity, and by today it has been applied in different contexts (Segura et al., 2016; Chen et al., 2018), including debuggers (Tolksdorf et al., 2019), machine learning algorithms (Murphy et al., 2008; Xie et al., 2011; Xu et al., 2018), optimization programs (Chen et al., 2012; Merkel et al., 2011), image processing (Jameel et al., 2017), web search engines (Chen et al., 2012; Zhou et al., 2012), and embedded software (Chen et al., 2012; Kuo et al., 2011).

Approaches to automatically infer metamorphic relations aim to reduce the effort required to derive them manually. Kanewala (2014) proposed the first attempt to automatically infer MRs, by applying machine learning prediction models in the context of scientific software. Troya et al. (2018) implemented an approach to automatically infer likely MRs for model transformations, in the context of Model-Driven Engineering. Xiang and others recently proposed an approach to combine simple MRs to derive complex and efficient MRs (Xiang et al., 2019) in the context of programs with numerical input values only.

SBES (Goffi et al., 2014), the technique to which we compare, automatically derives likely equivalent method calls in Java systems, useful in the scope of metamorphic testing. It is a search-based technique that executes multiple times all the methods in the search space to first find candidate properties, and then confirm they hold. Other techniques can infer metamorphic properties dynamically for the use in specific domains (such as machine learning algorithms Su et al., 2015 and numerical programs Zhang et al., 2019), or dynamically infer specific kind of metamorphic properties (such as polynomial MP Zhang et al., 2014).

Oracles from natural language artifacts. Some previous research derives via test oracles via natural language processing to some kind of natural language documentation of the software under test, although to the best of our knowledge, MeMo is the first to discover and translate metamorphic relations. It is worth mentioning that Monperrus et al. (2012), in their taxonomy of knowledge embedded in API documents, report the existence of "Alternative directives", which are a subset of the properties discovered by MeMo.

Similarly to MeMo, other work derives test oracles from Javadoc tags (`@param`, `@return`, `@throws`). Such work includes inference of assertions about nullness of parameters by Tan et al. (2012) and derivation of executable specifications (preconditions, post-conditions, and exceptional post-condition) by Goffi et al. (2016) and Blasi et al. (2018). Motwani and Brun (2019) later applied the same ideas to the documentation of the JavaScript language. Differently from MeMo, all these approaches take advantage of structured natural language; none of them attempts to analyze more complex, unstructured text. A distinctive feature of MeMo is that it infers assertions from unstructured text, without requiring the implicit information encoded in Javadoc tags, and as such takes advantage of useful information that these techniques cannot process to generate useful oracles. Pandita et al.'s work analyzes API documents to generate code contracts (Pandita et al., 2012), and discovers temporal constraints using a ML classifier (Pandita et al., 2016). Differently from Pandita et al.'s approaches, MeMo infers test oracles in the form of executable assertions, and focuses on the discovery of metamorphic relations leveraging NLP and word embeddings.

Automatically generated test oracles. Several techniques attempt to automatically derive test oracles. The main approaches in the state of the art generate regression oracles (Fraser and Arcuri, 2013), taking advantage of prevision versions of the program under test. Other techniques generate oracles according to some heuristics (Csallner and Smaragdakis, 2004, 2005; Pacheco and Ernst, 2005; Pacheco et al., 2007; Ma et al., 2015), for example asserting that a NullPointerException without a null input is likely to indicate a bug. Of course, oracles can be derived from formal specifications and similar artifacts. It is worth mentioning techniques that exploit algebraic specifications (Antoy and Hamlet, 2000; Gannon et al., 1981; Doong and Frankl, 1994), assertions and contracts (Araujo et al., 2011; Cheon, 2007; Meyer, 1988; Rosenblum, 1995; Taylor, 1983), context-free grammars (Day and Gannon, 1985), and finite state machines (Fujiwara et al., 1991). However, approaches like ours that exploit natural language artifacts pose a further challenge, since they have to deal with the translation of a non-formal, non-standard specification into an oracle.

7. Conclusions

MeMo derives metamorphic relations from natural language specification. MeMo generates Java executable assertions from Javadoc summaries that can act as oracles in the context of software testing. Differently from previous work (Tan et al., 2012; Goffi et al., 2016; Blasi et al., 2018; Motwani and Brun, 2019; Pandita et al., 2012), MeMo focuses on equivalent metamorphic relations and is not limited to semi-structured text, but analyzes *unstructured* natural language text, thus facing the challenge of recognizing the right topic of a piece of text. MeMo works by means of two main components, a MR finder that recognizes metamorphic properties inside Javadoc summaries, and a translator that produces executable Java assertions corresponding to what is expressed in the comment. MeMo properties are not limited in the scope of the unit under test, but potentially involve broader scopes such as the whole system under test or even units of external dependencies.

MeMo proves to be accurate in synthesizing metamorphic properties, with a precision of 91% and a recall of 69% on a ground truth of 281 expected Java specifications. MeMo can infer specifications that the state of the art (Goffi et al., 2014; Mattavelli et al., 2015) cannot discover. MeMo assertions prove to be useful when applied to testing, by killing more mutants than implicit oracles, regression oracles and developer-written assertions. Further improvements for MeMo are possible. While the translator component is fairly precise, its recall suffers from 81 missing translations. As explained in Section 4, many missing translations could only be addressed with a relatively advanced understanding of the natural language. Also, the equality checks of the executor component could be tailored on the specific objects' fields, to make the aspects more reliable. In general, MeMo may benefit from special handling of peculiar implementation cases, such as when comparing computations performed on `varargs` rather than on `arrays` (as explained in Section 4).

MeMo is open source and available at https://github.com/ariannab/MeMo

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work is partially supported by the Swiss SNF project ASTERIx: Automatic System TEsting of InteRactive software applications (SNF 200021_178742).

References

Antoy, Sergio, Hamlet, Dick, 2000. Automatically checking an implementation against its formal specification. IEEE Trans. Softw. Eng. 26 (1), 55–69, 2000.

Araujo, Wladimir, Briand, Lionel C., Labiche, Yvan, 2011. Enabling the runtime assertion checking of concurrent contracts for the java modeling language. In: Proceedings of the International Conference on Software Engineering (ICSE '11), pp. 786–795.

Balcer, Marc J., Hasling, William M., Ostrand, Thomas J., 1989. Automatic generation of test scripts from formal test specifications. In: Proceedings of the Symposium on Software Testing, Analysis, and Verification (TAV3 '89). ACM, pp. 210–218.

Barr, Earl T., Harman, Mark, McMinn, Phi., Shahbaz, Muzammil, Yoo, Shin, 2015. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525, 2015.

Blasi, Arianna, Goffi, Alberto, Kuznetsov, Konstantin, Gorla, Alessandra, Ernst, Michael D., Pezzè, Mauro, 2018. Translating code comments to procedure specifications. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '18). ACM.

Carzaniga, Antonio, Gorla, Alessandra, Mattavelli, Andrea, Pezzè, Mauro, Perino, Nicolò, 2013. Automatic recovery from runtime failures. In: Proceedings of the International Conference on Software Engineering (ICSE '13). IEEE Computer Society, pp. 782–791.

Carzaniga, Antonio, Gorla, Alessandra, Perino, Nicolò, Pezzè, Mauro, 2010. Automatic workarounds for web applications. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '10). ACM, pp. 237–246.

Carzaniga, Antonio, Gorla, Alessandra, Perino, Nicolò, Pezzè, Mauro, 2015. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. ACM Trans. Softw. Eng. Methodol. 24 (3), 16, 2015.

Carzaniga, Antonio., Gorla, Alessandra., Pezzè, Mauro., 2009. Handling software faults with redundancy. In: de Lemos, R., Fabre, J., Gacek, C., Gadducci, F., ter Beek, M. (Eds.), Architecting Dependable Systems VI. Springer, pp. 148–171.

Chen, Tsong Y., Cheung, Shing-Chi, Yiu, Shiu Ming, 1998. Metamorphic Testing: A New Approach for Generating Next Test Cases. Technical Report, Department of Computer Science, Hong Kong University of Science and Technology.

Chen, Tsong Yueh, Kuo, Fei-Ching, Liu, Huai, Poon, Pak-Lok, Towey, Dave, Tse, TH, Zhou, Zhi Quan, 2018. Metamorphic testing: A review of challenges and opportunities. ACM Comput. Surv. 51 (1), 4, 2018.

Chen, Tsong Yueh, Kuo, Fei-Ching, Towey, Dave, Zhou, Zhi Quan, 2012. Metamorphic testing: Applications and integration with other methods: Tutorial synopsis. In: Proceedings of the International Conference on Quality Software. IEEE Computer Society, pp. 285–288.

Chen, Tsong Y., Kuo, F.-C., Tse, T.H., Zhou, Zhi Quan, 2003. Metamorphic testing and beyond. In: International Workshop on Software Technology and Engineering Practice (STEP '03). IEEE Computer Society, pp. 94–100.

Cheon, Yoonsik, 2007. Abstraction in assertion-based test oracles. In: Proceedings of the International Conference on Quality Software (QSIC '07), pp. 410–414.

Csallner, Christoph, Smaragdakis, Yannis, 2004. Jcrasher: an automatic robustness tester for java. Softw. - Pract. Exp. 34 (11), 1025–1050, 2004.

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

Davis, Martin D., Weyuker, Elaine J., 1981. Pseudo-oracles for non-testable programs. In: Proceedings of the ACM '81 Conference (ACM '81). ACM, pp. 254–257.

Day, J.D., Gannon, J.D., 1985. A test oracle based on formal specifications. In: Proceedings of the Conference on Software Development Tools, Techniques, and Alternatives (SOFTAIR '85), pp. 126–130.

Doong, Roong-Ko, Frankl, Phyllis G., 1994. The ASTOOT approach to testing object-oriented programs. ACM Trans. Softw. Eng. Methodol. 3 (2), 101–130, 1994.

Fraser, Gordon, Arcuri, Andrea, 2013. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291, 2013.

Fujiwara, Susumu, Bochmann, Gregor von, Khendek, Ferhat, Amalou, Mokhtar, Ghedamsi, Abderrazak, 1991. Test selection based on finite state models. IEEE Trans. Softw. Eng. 17 (6), 591–603, 1991.

Gannon, John, McMullin, Paul, Hamlet, Richard, 1981. Data abstraction, implementation, specification, and testing. ACM Trans. Program. Lang. Syst. 3 (3), 211–223, 1981.

Goffi, Alberto, Gorla, Alessandra, Ernst, Michael D., Pezzè, Mauro, 2016. Automatic generation of oracles for exceptional behaviors. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '16). ACM, pp. 213–224.

Goffi, Alberto, Gorla, Alessandra, Mattavelli, Andrea, Pezzè, Mauro, Tonella, Paolo, 2014. Search-based synthesis of equivalent method sequences. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14). ACM, pp. 366–376.

guava, 2020. Google guava project. https://github.com/google/guava.

In memory compiler, 2020. In memory compiler. https://github.com/trung/InMemoryJavaCompiler.git.

Jameel, Tahir, Lin, Mengxiang, Chao, Liu, 2017. Metamorphic relations based test oracles for image processing applications. pp. 892–906, 2017.

Just, René, Schweiggert, Franz, Kapfhammer, Gregory M., 2011. MAJOR: An efficient and extensible tool for mutation analysis in a java compiler. In: Proceedings of the International Conference on Automated Software Engineering (ASE '11). IEEE Computer Society, pp. 612–615.

Kanewala, Upulee, 2014. Techniques for automatic detection of metamorphic relations. In: Proceedings of the International Conference on Software Testing, Verification and Validation Workshop. IEEE Computer Society, pp. 237–238.

Kanewala, Upulee, Bieman, James M., 2013. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In: SSRE (ISSRE '13). IEEE Computer Society, pp. 1–10.

Kuo, Fei-Ching, Chen, Tsong Yueh, Tam, Wing K., 2011. Testing embedded software by metamorphic testing: A wireless metering system case study. In: Proceedings of the Conference on Local Computer Networks. IEEE Computer Society, pp. 291–294.

Kusner, Matt J., Sun, Yu, Kolkin, Nicholas I., Weinberger, Kilian Q., 2015. From word embeddings to document distances. In: Proceedings of the International Conference on International Conference on Machine Learning (ICML '15), pp. 957–966.

Liu, Huai, Liu, Xuan, Chen, Tsong Yueh, 2012. A new method for constructing metamorphic relations. In: Proceedings of the International Conference on Quality Software (QSIC '12). IEEE Computer Society, pp. 59–68.

Ma, Lei, Artho, Cyrille, Zhang, Cheng, Sato, Hiroyuki, Gmeiner, Johannes, Ramler, Rudolf, 2015. GRT: Program-analysis-guided random testing. In: Proceedings of the International Conference on Automated Software Engineering (ASE '15). ACM, pp. 212–223.

Marneffe, Marie-Catherine, MacCartney, Bill, Manning, Christopher, 2006. Generating typed dependency parses from phrase structure parses. In: Proceedings of the International Conference on Language Resources and Evaluation (LREC '06). European Language Resources Association (ELRA), pp. 449–454.

Mattavelli, Andrea, Goffi, Alberto, Gorla, Alessandra, 2015. Synthesis of equivalent method calls in guava. In: Proceedings of the 7th International Symposium on Search-Based Software Engineering (SSBSE '15). Springer, pp. 248–254.

Merkel, Robert, Wang, Daoming, Lin, Huimin, Chen, Tsong Yueh, 2011. Automatic verification of optimization algorithms: a case study of a quadratic assignment problem solver. Int. J. Softw. Eng. Knowl. Eng. 21 (11), 289–307, 2011.

Meyer, Bertrand, 1988. Object-Oriented Software Construction, first ed. Prentice Hall.

Monperrus, Martin, Eichberg, Michael, Tekes, Elif, Mezini, Mira, 2012. What should developers be aware of? An empirical study on the directives of API documentation. Empir. Softw. Eng. 17 (6), 703–737.

Motwani, Manish, Brun, Yuriy, 2019. Automatically generating precise oracles from structured natural language specifications. In: Proceedings of the International Conference on Software Engineering. IEEE Computer Society, pp. 188–199.

Murphy, Christian, Kaiser, Gail E., Hu, Lifeng, 2008. Properties of machine learning applications for use in metamorphic testing. p. 867, 2008.

Oracle, 2020. Oracle java doocumentation. https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag.

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

Pacheco, Carlos, Lahiri, Shuvendu K., Ernst, Michael D., Ball, Thomas, 2007. Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering (ICSE '07). ACM, pp. 75–84.

Pandita, Rahul, Taneja, Kunal, Williams, Laurie, Tung, Teresa, 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, pp. 378–388.

Pandita, Rahul, Xiao, Xusheng, Zhong, Hao, Xie, Tao, Oney, Stephen, Paradkar, Amit, 2012. Inferring method specifications from natural language API descriptions. In: Proceedings of the International Conference on Software Engineering (ICSE '12). IEEE Computer Society, pp. 815–825.

Rosenblum, David S., 1995. A practical approach to programming with assertions. IEEE Trans. Softw. Eng. 21 (1), 19–31, 1995.

Segura, Sergio, Fraser, Gordon, Sanchez, Ana B., Ruiz-Cortés, Antonio, 2016. A survey on metamorphic testing. IEEE Trans. Softw. Eng. 42 (9), 805–824.

Su, Fang-Hsiang, Bell, Jonathan, Murphy, Christian, Kaiser, Gail E., 2015. Dynamic inference of likely metamorphic properties to support differential testing. In: Zhu, Hong, Hao, Dan, Mariani, Leonardo, Subramanyan, Rajesh (Eds.), Proceedings of the International Workshop on Automation of Software Test. IEEE Computer Society, pp. 55–59.

Tan, Shin Hwei, Marinov, Darko, Tan, Lin, Leavens, Gary T., 2012. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: ICST 2012: 5th International Conference on Software Testing, Verification and Validation. Montreal, Canada, pp. 260–269. http://dx.doi.org/10.1109/ICST.2012.106.

Taylor, Richard N., 1983. An integrated verification and testing environment. Softw. - Pract. Exp. 13 (8), 697–713, 1983.

Tolksdorf, Sandro, Lehmann, Daniel, Pradel, Michael, 2019. Interactive metamorphic testing of debuggers. In: Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '19). ACM, pp. 273–283.

Troya, Javier, Segura, Sergio, Ruiz-Cortés, Antonio, 2018. Automated inference of likely metamorphic relations for model transformations. J. Syst. Softw. 136, 188–208, 2018.

Weyuker, Elaine J., 1982. On testing non-testable programs. Comput. J. 25 (4), 465–470.

Xiang, Zhenglong, Wu, Hongrun, Yu, Fei, 2019. A genetic algorithm-based approach for composite metamorphic relations construction. Information 10 (12), 392.

Xie, Xiaoyuan, Ho, Joshua WK., Murphy, Christian, Kaiser, Gail, Xu, Baowen, Chen, Tsong Yueh, 2011. Testing and validating machine learning classifiers by metamorphic testing. J. Syst. Softw. 84 (4), 544–558.

Xu, Liming, Towey, Dave, French, Andrew P., Benford, Steve, Zhou, Zhi Quan, Chen, Tsong Yueh, 2018. Enhancing supervised classifications with metamorphic relations. In: Proceedings of the International Conference on Software Engineering. ACM, pp. 46–53.

Zhang, Jie, Chen, Junjie, Hao, Dan, Xiong, Yingfei, Xie, Bing, Zhang, Lu, Mei, Hong, 2014. Search-based inference of polynomial metamorphic relations. In: Crnkovic, Ivica, Chechik, Marsha, Grünbacher, Paul (Eds.), Proceedings of the International Conference on Automated Software Engineering. ACM, pp. 701–712.

Zhang, Bo, Zhang, Hongyu, Chen, Junjie, Hao, Dan, Moscato, Pablo, 2019. Automatic discovery and cleansing of numerical metamorphic relations. In: Proceedings of the IEEE International Conference on Software Maintenance and Evolution. IEEE Computer Society, pp. 235–245.

Zhou, Zhi Quan, Zhang, ShuJia, Hagenbuchner, Markus, Tse, T.H., Kuo, Fei-Ching, Chen, Tsong Yueh, 2012. Automated functional testing of online search services. Softw. Test. Verif. Reliab. 22 (4), 221–243, (6 2012).