Automatic Generation of Test Oracles from Natural Language Specifications

Doctoral Dissertation submitted to the

Faculty of Informatics of the Università della Svizzera Italiana

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

presented by

Arianna Blasi

under the supervision of

Prof. Mauro Pezzè and Prof. Alessandra Gorla

05 2022

Dissertation Committee

Dissertation accepted on 10 05 2022

[illegible]

Research Advisor

Prof. Mauro Pezzè

Co-Advisor

Prof. Alessandra Gorla

PhD Program Director

The PhD program Director *pro tempore*

[illegible]

Arianna Blasi
Lugano, 10 05 2022

Abstract

This PhD thesis proposes a framework to automatically derive test oracles from natural language specifications. We studied and developed cost-effective techniques to derive oracles from information commonly available in natural language about the code.

Despite previous research in software testing, the oracle problem, that is, the challenge to distinguish correct from incorrect behavior, is still largely open. Contemporary test case generators (TCGs) rely on either simple and incomplete implicit oracles or on regression oracles that refer to the results of running previous versions of the program. Implicit oracles can reveal exceptions and program crashes, but miss semantically relevant issues. Regression oracles can detect deviations from the behavior observed in former versions of the program, but not in new functionalities. Many approaches generate powerful and complete test oracles from formal specifications that are still not the most common practice in software development.

The main goal of this thesis is to define techniques that use natural language annotations, which current approaches largely ignore, to generate effective test oracles without additional human effort. Most software systems are supported by textual information, such as annotations, comments, and wikis. This information is typically informal and unstructured, and often combines natural language expressions with developers' jargon. We also observe that informal artifacts are prone to human mistakes. In a nutshell, informal artifacts are not always reliable and are hard to exploit automatically.

This thesis defines approaches to automatically interpret and translate informal and unstructured information that combines natural language expressions with developers' jargon into actionable test oracles, that is, test cases that can be automatically evaluated, while overcoming human errors that affect their quality. We first automatically verify the consistency between the code and its documentation by modeling both code and documentation with a Bag Of Words (BOW) representation, and signal developers inconsistencies that we detect at a fine-grained level. We then process the pruned specifications to automatically generate test oracles in the form of executable assertions. We designed and developed approaches that process both structured and unstructured Javadoc specifications, to derive both descriptive and prescriptive assertions for the methods of a Java class. We successfully experimented with the approaches on popular, widely-used and open-source Java systems.

This PhD thesis opens new research directions towards the automated exploitation of natural language artifacts beyond Javadoc specification, to derive powerful test oracles. The intuitions and observations our study proposes can be generalized and applied to many other software related artifacts.

Acknowledgements

We would like to thank the Swiss National Science Foundation that partially supported this PhD within the SNF project ASTERIx: Automatic System TEsting of inteRactive software applIcations *(SNF 200021_178742).*

I thank my advisor, Prof. Mauro Pezzè, and my co-advisor, Prof. Alessandra Gorla, for having believed in me since the very first day I entered their office. I will always be grateful to them for having introduced me to the research world.

I cannot praise enough all the great scientists who inspired me during this journey. I thank the impressive women I had the luck to work with: Dr. Nataliia Stulova, an example of rigourous researcher and great humanity at the same time, and Dr. Pooja Rani, a living demonstration of how hard work and grit look like. I thank all my amazing co-authors: Prof. Michael D. Ernst, Prof. Antonio Carzaniga, Prof. Oscar Nierstrasz, Prof. Sebastiano Panichella, Dr. Alberto Goffi, and Dr. Konstantin Kuznetsov. Thanks to the former members of my lab, who were there when I had just started: Prof. Valerio Terragni, Dr. Daniele Zuddas, Dr. Pasquale Salza and Dr. Cristina Monni. In fact, thanks to all the colleagues of the STAR group, both in Lugano and SIT. Also thanks to all the people in Meta that made my summer internship not only valuable but also greatly enjoyable.

Thanks to the students and researchers from the old open space and the Software Institute, with whom I had the possibility to share teaching duties, coffe breaks and chats. In particular, thanks to the colleagues living in Como for the fun we shared outside the university. And of course, thanks to all the people from the IMDEA Software Institute with whom I could live similar experiences.

Thanks to all the people who supported me through these years outside the work environment. I thank the awesome women of the Aerial&Dance Academy in Como, who taught me to fly high both metaphorically and literally. I thank my mental health counselors, especially for the support offered during pandemic lockdowns. I thank the volunteers of Supporto Attivo, Vicini di Strada, Comunità Annunciata, and all the other associations I may or may not have had the pleasure to know in Como which work hard everyday to improve the life of the less privileged.

Last but not least, I thank my immediate family, and the one composed of my dear friends and my loving partner. Actually, in the not-sky-high possibility that it is a PhD student reading this: Sure, science is cool, just do remember that healthiness and worthy human relationships are as well.

Contents

Figures

Tables

Chapter 1

Introduction

This chapter reflects on the importance of testing activities and the need for automation. It presents the main ideas of the thesis.

1.1 Background

Software testing is a fundamental activity to ensure software quality. *Test oracles* are the mechanism to distinguish between correct and incorrect behavior of the software under test (SUT). While there exist many techniques to automatically generate test inputs, there are relatively few approaches to automatically generate test oracles. Some existing approaches can be easily automated, but produce partial oracles that cannot reveal relevant semantic errors. This is the case for implicit oracles that trigger system crashes and unhandled exceptions. Other approaches can reveal relevant semantic errors, but rely on information that is not often available, for example, by generating oracles from formal specifications. This thesis studies and develops new and cost-effective techniques to automatically generate test oracles from information widely available in the form of informal natural language annotations written in English.

Barr et al. classify oracle generation techniques in three main categories Barr et al. [2015]:

- *Specified* test oracles come from formal specifications, which may be given in many different forms and languages. Formal specifications provide a formal encoding of the correct behaviors of the SUT. They are rarely available and are costly to define and maintain.

- *Derived* test oracles are inferred from different artifacts, including informal documentation and system executions. Derived oracles rely on information commonly available but often incomplete, for instance, regression oracles rely on a previous version of the program and can detect failures when the current implementation of the software breaks properties that held in the previous versions, but not failures in new or modified functionality.

- *Implicit* oracles express properties that are generally valid and hold independently from the specific semantics of the software under test, for instance null-dereferences are illegal. Implicit oracles can easily be generated automatically, but can detect only a few kinds of failures.

Automatic test case generators, such as Randoop (Pacheco et al. [2007]) and EvoSuite (Fraser and Arcuri [2011]), rely on both regression and implicit test oracles. This means that they can only detect incorrect behavior with respect to previous versions of the SUT, or generally undesired behavior that does not depend on its intrinsic semantics, and miss many relevant failures. The lack of semantically relevant oracles is a main limitation of such systems.

Developers produce multiple natural language artifacts: documentation, source code comments, wikis, tutorials. Software-related natural language artifacts in natural language are widely and commonly available, , however, they are ofter hard to interpret automatically. The presence of developers' jargon adds to the intrinsic complexity of natural language, and reduces the effectiveness of classic NLP techniques. Moreover, the final artifacts may be victims of developers' mistakes and inaccuracy, which threaten the reliability of what they document about the SUT.

This PhD thesis proposes tools and techniques to *automatically* derive properties of the SUT from natural language specifications in reliable and cost-effective ways. The work shows how informal software specification can look, how to overcome defects in informal specifications, and how to exploit the documented properties to improve testing activities.

1.2 Research Hypothesis and Contributions

The main research hypothesis of this thesis is the following:

> *It is possible to automatically generate executable test oracles from commonly available natural language information.*

The first approaches to automatically generating test oracles from natural language annotations were proposed by Tan et al. who defined @tComment about a decade ago Tan et al. [2012]. @tComment exploits relatively simple techniques and can synthesize only simple conditions. To the best of our knowledge, the seed research idea was not been followed by in-depth studies to either extend the original @tComment approach or argue the infeasibility of *reliably* deriving test oracles from information not formally structured and prone to human errors. Only recently, the software engineering community acknowledged the value of informal natural language artifacts, and started studying their use to improve software artifacts.

This PhD thesis contributes to the state of the art by defining an approach to automatically derive semantically-relevant properties from natural language artifacts that document the SUT. We show how natural language software specifications convey information exploitable for testing, and propose a framework to both overcome the unreliability of informal specifications and automatically generate test oracles by interpreting informal specifications. We observe that the natural language specifications may be flawed, i.e., somewhat unreliable to derive a correct oracle. We propose approaches to automatically check natural language specifications, fix the identified flaws, and translate the corrected information into test oracles. *Automation* is a key point across the whole framework, for its cost effectiveness that we obtain by both requiring little if any human effort and no heavy computational time.

The main results presented in this thesis were published in international venues, and tools and datasets are publicly available. We prsented *JDoctor* Blasi et al. [2018] at the ACM SIGSOFT International Symposium on Software Testing and Analysis. We published *MeMo* Blasi et al. [2021a] in a special issue of the Journal of Systems and Software on metamorphic testing. We initially presented *RepliComment* at the IEEE/ACM International Conference on Program Comprehension Blasi and Gorla [2018], and published an extension in a special issue of the Journal of Systems and Software on code clones Blasi et al. [2021b]. We presented *UpDoc* Stulova et al. [2020] at the International Working Conference on Source Code Analysis and Manipulation. We submitted *CaMeMa* for publication to a software engineering venue. The tools and datasets are available on the STAR lab website [1], Zenodo [2], and on the thesis author's GitHub page [3]. The author of the thesis contributed either wholly or significantly to the novel ideas presented in all papers, to the implementation of the tools, and to the experimental evaluation of the results.

[1] https://star.inf.usi.ch/#/software-data

[2] https://zenodo.org/record/5094318#.Yj3IIjfMLtw, https://zenodo.org/record/1297458#.Yj3IPDfMLtw

[3] https://github.com/ariannab?tab=repositories

1.3 Thesis Structure

This thesis is organized as follows. Section 2 presents the state of the art of both test oracle generation and exploitation of natural language software artifacts. Section 3 overviews our framework by discussing both the characteristics of informal software specification and the challenges of interpreting them to generate test oracles. Section 4 presents a first approach to derive test oracles from semi-structured natural language specifications. Section 5 presents two approaches to generate test oracles from unstructured specifications. Section 6 presents two approaches to detect defects in natural language specifications and suggest fixes. Section 7 summarizes the main results of the thesis and indicates the new research directions that emerge from the results presented in this thesis.

Chapter 2

State Of The Art

This section overviews techniques to automatically generate test oracles in terms of cost-effectiveness. It also discusses studies about the potential and pitfalls of natural language artifacts in software engineering.

2.1 Test Oracle Generation

Software testing is an essential activity for engineering software systems. Testing improves the quality of software systems by revealing faults during the development process. Thorough software testing is expensive, and automatically generating test cases can largely reduce the costs. Automatic test case generation requires producing both inputs that solicit the execution of the program under test and *oracles* that determine the acceptability of the results. Producing good oracles requires knowledge of the semantics of the SUT and high effort that impacts the cost of testing.

The *oracle problem* attracted a lot of research interest since the early work of Davis and Weyuker Davis and Weyuker [1981]; Weyuker [1982], as clearly described in Barr et al.'s excellent survey Barr et al. [2015]. Much recent research efforts study test oracles to tackle modern challenges, such as deep learning systems Nejadgholi and Yang [2019], autonomous driving systems Jahangirova et al. [2021], and the problem of energy consumption of software systems Bruce et al. [2018]; Jabbarvand et al. [2020].

This thesis defines approaches to generate general oracles in the presence of available information about the SUT. Here, we overview the most common techniques to automatically generate test oracles by considering costs and effectiveness. Both dimensions largely depend on the level of semantically-relevant information required to generate the oracle. Figure 2.1 visualizes the core approaches with respect to these dimensions. The ideal approach sits in the bottom right corner of the space: high effectiveness at low cost.

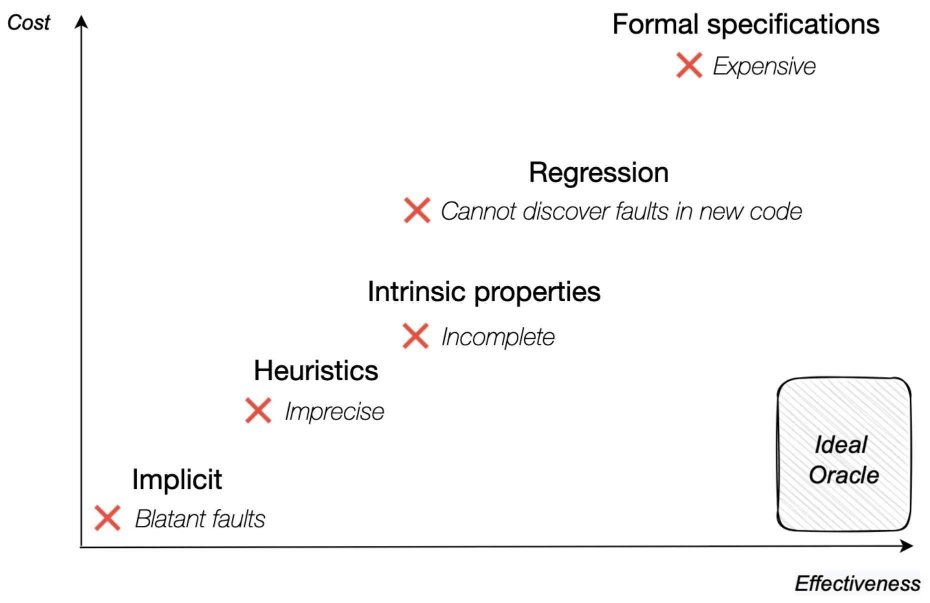

Figure 2.1. Placing test oracles according to Cost and Effectiveness

Implicit oracles sit near the leftmost bottom corner: They can be cheaply generated, but offer limited effectiveness, since they ignore semantics of the SUT. Implicit oracles are essentially "blatant faults" Barr et al. [2015]. They tell developers that something went wrong in the execution, and can be detected without actually formalizing a semantically-relevant oracle, but give little information about the flaw that lead to the crash.

Heuristic oracles improve the effectiveness of implicit oracles with heuristics that assess anomalies such as thrown exceptions. Many techniques reason about exceptions using heuristics Csallner and Smaragdakis [2004, 2005]; Pacheco and Ernst [2005]; Pacheco et al. [2007]; Ma et al. [2015]. As a representative example, Randoop Pacheco et al. [2007] considers a test that throws `NullPointerException` on `null` input as expected behavior. Randoop and similar heuristics-based approaches allow testers to customize such rules to alleviate the false positive rate that may derive from heuristics not tailored to the semantics of the SUT. As a relevant example of heuristics imprecision, JCrasher Csallner and Smaragdakis [2004] and Check 'n' Crash (CnC) Csallner and Smaragdakis

[2005] assume that methods do not raise exceptions unless specifically declared in the method signature, an assumption that can lead to many false alarms. Similarly, Randoop classifies as erroneous a test case leading to an exception if it was not explicitly declared in any signature.

Intrinsic properties-based, regression, and formal specification-based oracles improve effectiveness and increase generation costs by exploiting *intrinsic properties*, *previous versions*, and *formal specification* of the SUT, respectively.

Approaches to generate test oracles from *formal specifications* flourished since the eighties, as several specification languages exist. Formal specification provides a mathematical model of the SUT behavior, making it a good source of test oracles. Algebraic specifications Antoy and Hamlet [2000]; Gannon et al. [1981]; Doong and Frankl [1994] define a software module in terms of its interface, usually by means of first-order logic to prove properties of the specification. Assertions and contracts Araujo et al. [2011]; Cheon [2007]; Meyer [1988]; Rosenblum [1995]; Taylor [1983] are popular means to check the program behavior. Popular programming languages such as Java [1], C++ [2] and Python [3] support assertions by means of the `assert` statement. An assertion is a boolean expression that checks the SUT at run time either in the source or test code. Model-based specifications Day and Gannon [1985]; Fujiwara et al. [1991]; Mcdonald [1998]; Mikk [1995]; Cheon and Leavens [2002]; Gay et al. [2016] support reasoning about a system by operations that can alter the system state. Producing and maintaining a formal specification is expensive, and this limits the applicability of these techniques in practice. Many approaches derive both test input and oracles from semi-formal specification, like UML models Schwarzl and Peischl [2010]; Porres and Rauf [2010]; Wang et al. [2015]; Mai et al. [2018]. Semi-formal specifications are more common than formal specifications; however, the effectiveness of automatically generated oracles is limited to the subset of the SUT semantics formalized with the semi-formal models.

Metamorphic testing generates test oracles by manipulating intrinsic properties of the SUT. Metamorphic oracles reveal failures by checking metamorphic relations that capture application-specific symmetries and equivalences (Chen et al. [1998, 2003]). Simple examples of equivalence metamorphic relations (MRs) are $sum(a,b) \equiv sum(b,a)$, or $\sin(x+2\pi) \equiv \sin(x)$. Metamorphic testing becomes difficult and effort-demanding when the specification of the SUT is not available and hard to define. Similarly, crosschecking oracles Carzaniga et al.

1 https://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html

2 https://www.cplusplus.com/reference/cassert/assert/

3 https://docs.python.org/3/reference/simple_stmts.html#the-assert-statement

[2014] exploit the intrinsic redundancy of SUT, while symmetric testing Gotlieb [2003] exploits permutation relations between program executions. In a nutshell, metamorphic testing, crosschecking oracles, and symmetric testing generate oracles by comparing the results of two behaviors of the SUT that should be equivalent. The effectiveness of these oracles is limited to cases where such properties exist. They may not be always known, available, or easy to exploit.

Regression test oracles offer a good trade-off between cost and effectiveness by relying on previous versions of the SUT. Regression assertions can be easily generated automatically Pacheco et al. [2007]; Fraser and Arcuri [2011] and offer semantically relevant oracles. The cost of regression oracles derives mainly from the need to incrementally monitor and record the executions of the SUT during testing. Not surprisingly, regression testing is one of the most common automated testing practices in industry Ali et al. [2019]; Ziftci and Cavalcanti [2020]. Regression oracles are useful to reveal differences with respect to previous versions (regression of the SUT); however, they cannot reveal faults in new code.

In the last decade, a new research direction has emerged that studies the possibility of automatically generating semantically relevant test oracles from information about the SUT in natural language, and thus with negligible extra effort . Informal natural language information describing software is available in different formats and shapes, making it an appealing candidate for the automatic acquisition and exploitation of knowledge about the SUT. Natural language information is not as easy to exploit and as reliable as formal specification, which explains why work in this direction started evolving only with the improvement of natural language processing techniques . It is fair to affirm that this line of research was promoted by works by Tan et al. Tan et al. [2012]. Their @tComment technique uses *pattern matching* on source code documentation to determine conditions related to nullness of method parameters. A similar work, Toradocu by Goffi et al. Goffi et al. [2016], predicates on exceptional behaviors instead. Such simple rule-based approaches are effective and safe in generating reliable oracles about simple properties. However, their applicability is limited to both the properties they can handle and of the complexity of the natural language information they can exploit. More advanced approaches infer properties of the SUT from code documentation while addressing some natural language ambiguities. ALICS by Pandita et al. Pandita et al. [2012] combines part-of-speech tagging and pattern-matching to generate simple pre- and post-conditions, by exploiting a restricted vocabulary of synonyms. Yet other approaches exploit natural language information to generate test inputs. Both Mai et al. and Wang et al. Wang et al. [2015]; Mai et al. [2018] automatically derive test cases from use case specifications by means of *Semantic Role Labeling*, taking advantage of the simplifications induced by the structure of semi-formal specifications to generate the test inputs. The technique can be effective in the context of use case specifications, where the vocabulary is quite restricted. In a free-form of natural language text, the matching of roles in a sentence would be more difficult to apply.

2.2 Relevance Of Natural Language Artifacts In SE

We conclude this section by discussing the relevance of natural language artifacts in software engineering. System requirements, software documentation, source code comments, programming discussion boards, all employ natural language as the primary mean to convey information concerning software. It is thus not surprising that the software engineering research community looks for ideas to automatically exploit them to support different development activities.

Requirements are often scattered among large documents. Many techniques use natural language processing to automatically extract key aspects and improve traceability Xiao et al. [2012]; Abad et al. [2019]; Shi et al. [2020]; Hey et al. [2021]. Discussion boards, such as Stack Overflow, hold a large volume of collective knowledge about APIs. Natural language processing techniques help in finding and organizing such information, to build knowledge graphs and concept hierarchies Li et al. [2018]; Chen et al. [2019], support code search Rahman and Roy [2018], or opinion mining Lin et al. [2019]. Other work exploits app reviews, which convey improvement suggestions and bug reports Panichella et al. [2016, 2015]; Di Sorbo et al. [2016]. Source code comments document useful information to both program users and developers. They support program comprehension and ease communication among developers Pawelka and Juergens [2015]; Arnaoudova et al. [2016]; Louis et al. [2020]; Nie et al. [2019], and identify self-admitted technical debt da Silva Maldonado et al. [2017]. Some work uses code comments to infer types in non-strongly typed languages Malik et al. [2019], reveal code misuses that lead to program crashes Kechagia et al. [2019], detect code clones Nafi et al. [2019], and generate formal specifications Zhai et al. [2020].

Some research work proposes ways to improve the parsing of software-related natural language information Abebe and Tonella [2010]; Gupta et al. [2013]; Partachi et al. [2019]. Approaches that unveil issues affecting software-related natural language artifacts are also growing in popularity. Both Wen et al. and Aghajani et al. Wen et al. [2019]; Aghajani et al. [2019] bring attention on anti-patterns and inconsistencies impacting software documentation, including source code comments.

Other techniques attempt to automatically detect and possibly fix emerging issues. Ratol and Robillard [2017] propose a technique that detects textual references to identifier renamed during refactoring activities, to update identifiers in related unstructured comments. Liu et al. [2020] propose an approach to update comments as soon as the code changes, to avoid introducing bad comments. This body of research work is particularly relevant for the framework we propose in the thesis. Fluri et al. [2007] notes that comment changes are triggered by corresponding code changes in less than half cases of code changes, and changes may happen quite late (more than three revisions later). Wen et al. [2019], in their large-scale study involving over 1500 open source projects (of which 500 commits manually analyzed) confirm that *"in most of the cases, code and comments do not co-evolve"* simultaneously. Co-evolution occurs roughly in 20% cases. Such studies prove that informal code specification may not always be a reliable source to derive test oracles, which motivates our work on detecting inconsistencies in informal software specification.

Chapter 3

Exploiting Natural Language Specifications

In this chapter, we discuss how to exploit informal software specification expressed in natural language to derive test oracles. We focus on code documentation expressed in English as Javadoc annotations. We discuss the characteristics of informal Javadoc specifications, and show how to take advantage of such characteristics to generate test oracles.

3.1 The Pervasiveness Of Natural Language Specifications

Software can be specified in several ways. Natural language information describing the software behavior can be find in many artifacts, such as software requirements, UML specifications, wikis, user and developer guides, and suchlike.

In this thesis, we focus on *documentation of source code* and on API comments in particular. Documentation of source code is particularly interesting because it describes the code at different levels of granularity. Inline comments typically document specific lines of code, while block comments document multiple statements. The most extensive kind of source code comments are API comments, as they serve to document the behavior of whole methods and classes. They fit well the thesis objective, as we are interested in inferring the *expected behavior* of the SUT for testing.

The most popular contemporary programming languages offer tools to effectively generate API documentation. In this thesis, we focus on Javadoc, the de facto standard to write code documentation for the Java language. An analogue is PyDoc for Python. In general, all the findings in the thesis generalize and apply to any similar natural language artifact for documenting code.

3.2 An Overview Of Javadoc Specification

The Javadoc standard was introduced in the early 2000s and has been widely used since then to document Java software. Popular libraries are especially well-documented through Javadoc to aid both users and developers to comprehend, improve and maintain source code. We can get an intuition of the span of Javadoc from Guava, the Google Core Libraries for Java. Guava is currently used by nearly thirty thousand artifacts, according to the Maven Repository [1], and counts over forty thousands stars on GitHub [2]. A simple check of Google Guava's Javadoc indicates the presence of nearly ten thousand sentences, by considering only the documentation of methods.

Javadoc comments can document classes, fields, constructors and method declarations. We can distinguish two main components of Javadoc documentation: Unstructured free-text and semi-structured descriptions. Free-text descriptions are the so-called Javadoc *summaries*, while semi-structured descriptions are Javadoc *block tags*. Any Javadoc comment can be enriched with *Inline tags*, for instance, `@code` and `@link` to highlight code identifiers and hypertext links, respectively. Table 3.1 overviews all Javadoc block and inline tags[3], where Inline tags are within curly brackets.

In a nutshell, informal specifications are rarely if ever composed of mere textual information, and are often annotated with tags that are extremely useful when parsing the documentation. Common natural language parsers would not be able to process, for example, the presence of a method signature mentioned inside a code comment. However, we can easily adapt natural language parsers to consider code identifiers found in text as nouns, by being aware that code identifiers are wrapped within `@code` tags .

We overview the main challenges of interpreting Javadoc comments, by discussing some Javadoc examples that we excerpt from popular libraries. We argue that it is possible to achieve a high precision in automatically generating exe-

[1] `https://mvnrepository.com/artifact/com.google.guava/guava`
[2] `https://github.com/google/guava`
[3] `https://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html#javadoctags`

Table 3.1. Available Javadoc tags

Tag	Introduced in JDK/SDK
@author	1
{@code}	1.5
{@docRoot}	1.3
@deprecated	1
@exception	1
{@inheritDoc}	1.4
{@link}	1.2
{@linkplain}	1.4
{@literal}	1.5
@param	1
@return	1
@see	1
@serial	1.2
@serialData	1.2
@serialField	1.2
@since	1.1
@throws	1.2
{@value}	1.4
@version	1

cutable test oracles from Javadoc specifications by exploiting *syntactic features* of the text. This observation matured during the PhD work (Blasi et al. [2018]), and is sustained in other studies as well (Di Sorbo et al. [2019]). We observe that precision is most important than recall when generating test oracles: Missing a few oracles (low recall) is a much lesser issue than generating wrong oracles (low precision).

Listing 3.1: a simple example of informal specification with low ambiguity level The natural language text of tag `@param` at line 3 states that parameter `fpp` should be both positive and less than a constant double value. To generate an executable test oracle, our technique must be aware that (i) "positive" means greater than zero, i.e., `> 0`, (ii) the text indicates a comparison between a parameter and the constant value `1.0`, (iii) the conjunction "and" implies that both conditions must hold as preconditions for the parameter. As humans, we barely notice the amount of information there is to consider to understand and translate such simple text.

This first example is indeed a basic one in terms of interpretation. The context is still quite explicit, thanks to the `@param` tag, there are no implicit subjects to infer, and the predicate is straightforward. In other words, the level of ambiguity of this piece of informal specification is extremely low.

Listing 3.1. Example from BloomFilter class of Google Guava

```
/**
 * ...
 * @param fpp the desired false positive probability (must be
 *      positive and less than 1.0)
 */
public static <T> BloomFilter<T> create(
Funnel<? super T> funnel,
 int expectedInsertions,
double fpp) { ... }
```

Listing 3.2: an example of ambiguous subjects due to incomplete information A subject may be ambiguous, and thus hard to translate into code, for different reasons. The natural language text of tag `@throws` at line 3 mentioning “either collection” and “the comparator” shows an example of ambiguity that derives from information left implicit in the comment. Our knowledge and intuition suggest us that “collection” in the comment refers to the Iterables in input, and that “comparator” refers to the other parameter. In essence, we infer that “collection” is just another *identifier* for parameters `a` and `b`, as “comparator” is for parameter `c`. Automatic techniques need to infer such knowledge as well. In this example, the parameter names are not helpful at all for machine comprehension (since they are generic `a`, `b` and `c`) – however, our automatic technique can match the subjects of the `@throws` text to the right code candidates from type information.

Listing 3.2. Example from CollectionUtils class of Apache Commons Collections

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

Listing 3.3: an example of ambiguous subjects due to natural language ambiguity Listing 3.3 refer to two parameters, `x` and `y`, and a `@throws` text that states that an exception will be thrown if “the number of points” is less than a

certain value, leaving it unspecified both what "the number of points" is and what "points" refers to. The `@param` text at line 3 indicates that `x` is a "points" array. By observing that this the first noun describing `x`, we can infer that "points" is an *identifier* for `x`. We also know `x` is an array, which becomes yet another identifier for the parameter.

Listing 3.3. Real example from DividedDifferenceInterpolator class of Apache Commons Math

```
 1 /**
 2  * ...
 3  * @param x Interpolating points array.
 4  * @param y Interpolating values array.
 5  * ...
 6  * @throws NumberIsTooSmallException if the number of points
 7  *      is less than 2.
 8  * ...
 9  */

11 protected static double[] computeDividedDifference(final double x[],
12  final double y[]) { ... }
```

Ambiguous predicates Interpreting, matching, and translating predicates can be particularly difficult. While subjects are often stated explicitly by their code names inside textual informal specification, or, at least, recognizable by smart expedients as we saw in the previous examples, predicates may be expressed in a great variety of ways. Some predicates are straightforward expressions and can be easily mapped to code expressions: for example "is positive" can be easily matched against the code expression `> 0`. Some other predicates are essentially *actions* whose code matching is a specific method call. For example a predicate "is empty" that refers to a subject collection `coll` of type `List`, is easily translated as `coll.isEmpty()`. However, variety and challenges can go much further than that, as for example, when developers use *synonyms* of actions to describe predicates.

Listing 3.4: an example of ambiguity due to the use of synonyms The predicate in the `@throws` tag text of Listing 3.4 "is not found in the graph", referring to subject "vertex", does not clearly specify the meaning of a vertex *not found* in a graph. This concept could be interpreted as either "the vertex does not exist in the structure" (thus matching a hypothetical method `vertexExists`),

or "the structure does not contain such vertex" (thus matching the hypothetical `containsVertex`). In either case, an automatic translation technique needs some *semantic understanding* capability to infer the right match between natural language and code.

Listing 3.4. Real example from Graph class of JGrapht

```
/**
 * ...
 * @throws  NullPointerException if vertex is not found in the
 *      graph
 */
Set edgesOf(Object vertex) {  ... }
```

Listing 3.5: an example of wrong informal specification Informal specifications may not be updated synchronously with the code, or it may be simply wrong due to mere human distraction. The example in Listing 3.5 survived from the very first version of the Google Guava library up to version 19.0 before getting fixed:

Listing 3.5. Real example from CharMatcher class of Google Guava

```
/**
 * ...
 * @return  true if this matcher matches every character in the
 *      sequence, including when the sequence is empty
 */
public boolean matchesNoneOf(CharSequence sequence) { ... }
```

The comment in Listing 3.5 comes from the Javadoc of method `matchesNoneOf`. The mismatch between the comment, "true if this matcher matches every character in the sequence" and the name of the method, "matchesNoneOf(CharSequence sequence)" suggests a likely inconsistency between the comment and the code. Guava fixed this error in release 20.0 by substituting the description of `matchesNoneOf`'s return tag with "true if this matcher matches *no characters* in the sequence, including when the sequence is empty", which sounds indeed more consistent.

Listing 3.6: an example of semi-structured and unstructured documentation The example in Listing 3.6 illustrates the differences between semi-structured and unstructured documentation. The semi-structured section of the Javadoc comment is identified with block tags, and it is preceded by an unstructured summary (*Merges the arrays in input [...] in the newly merged array.*). The method summary is a paragraph reporting general information about the behavior of the

method (how it operates, how the result will look like, and so on) in a free-text form. The block tags provide essential information about specific elements of the method, parameters (@param), normal postconditions (@return), exceptional postconditions (@throws).

Listing 3.6. Sample Javadoc specification of a method

```
/**
 * Merges the arrays in input. This method combines two arrays in a single
 * array object that will be returned. The array elements maintain their original
 * order in the newly merged array. The elements of the first array precede the
 * elements of the second array in the newly merged array.
 *
 * @param x the first array, not null
 * @param y the second array, not null
 * @return an array which is the result of the merge.
 *         Empty if both arrays are empty
 * @throws IllegalArgumentEsxception if either array is null
 *
 */
public Object[] merge(Object[] x, Object[] y) throws IllegalArgumentException
 {...}
```

Semi-structured and unstructured informal specification come with different pros and cons. Unstructured free text is often richer than the information embedded in block tags; however it may also contain information that is either not interesting or usable for testing purposes.

With this thesis, we aim to provide a framework that shows how to exploit both kinds of informal specification to derive test oracles.

3.3 From Informal Specifications To Oracles

We now frame the main challenges that emerge when dealing with both structured and unstructured comments in Javadoc documentation:

Challenge 1. *Understanding the code context,* which amounts to identifying which code elements a certain part of the documentation refers to. The broader the scope of a documentation fragment is, the harder this challenge is.

Challenge 2. *Translating the information from natural language into executable oracles:* Test oracles must be encoded in some way that allows the oracles to be evaluated, such as in the form of executable code assertions. To produce compilable and executable assertions, we must find the right bridge between natural and programming language, and solve jargon references often present in Javadoc comments.

Challenge 3. *Dealing with unreliable specification:* Informal specification does not follow strict rules or schemes. Developers are free to write it following their personal taste, making it prone to error. As indicated in examples discusses above in this section, informal specifications are not always aligned to the code. Exploiting flawed specification implies deriving incorrect oracles.

With this thesis, we identify two key characteristics of an informal natural language artifact to successfully address these challenges:

Characteristic 1. ***Artifact format:***

The structure of the natural language artifact can help us address **Challenge 1**. Let us consider for example the Javadoc @param tags: Whatever information may be written after such tags, we can safely assume that it refers to a method parameter (and sometimes its relations with other parameters). It may also help us address **Challenge 2** because, usually, the amount of text written for a single tag is much shorter and self-contained than in some unstructured comments, like class summaries. Finally, a format that favors code understanding opens the possibility to automatically infer fixes for a flawed specification, easing **Challenge 3**.

Characteristic 2. ***Artifact scope:***

The scope of the natural language information can help us address **Challenge 1**. Let us consider for example how a method summary rarely refers to a scope broader than the method itself, while a class summary may refer to multiple properties scattered around the whole class. The scope of a comment may also help us address **Challenge 2** because the amount of text to tackle is different. As per **Challenge 3**, fixing flawed specification at a narrow scope may be easier because the set of potential correct fixes reduces.

In this thesis, we show how we exploit Javadoc different formats and scopes to successfully address the challenges of translating natural language software specification into test oracles. Figure 3.1 outlines the two phase framework that we define to address the problem of automatically generating reliable test oracles from informal specifications. The main characteristics of the framework are:

1. Admitting the possibility of starting with a flawed informal specification, since we cannot deny it is a real and hindering possibility.

2. Exploiting natural language interpretation by taking advantage of the specific Artifact Format and Scope, to automatically identify flaws and suggest fixes . The framework proposes the fixes to the developers who may/may not implement them. Implementing the suggested fixes improves the effectiveness of the generated test oracles.

3. Exploiting natural language interpretation to identify suitable code translations for the natural language text inside the reliable version of the specification.

4. Generating actionable test oracles, e.g., in the form of executable assertions.

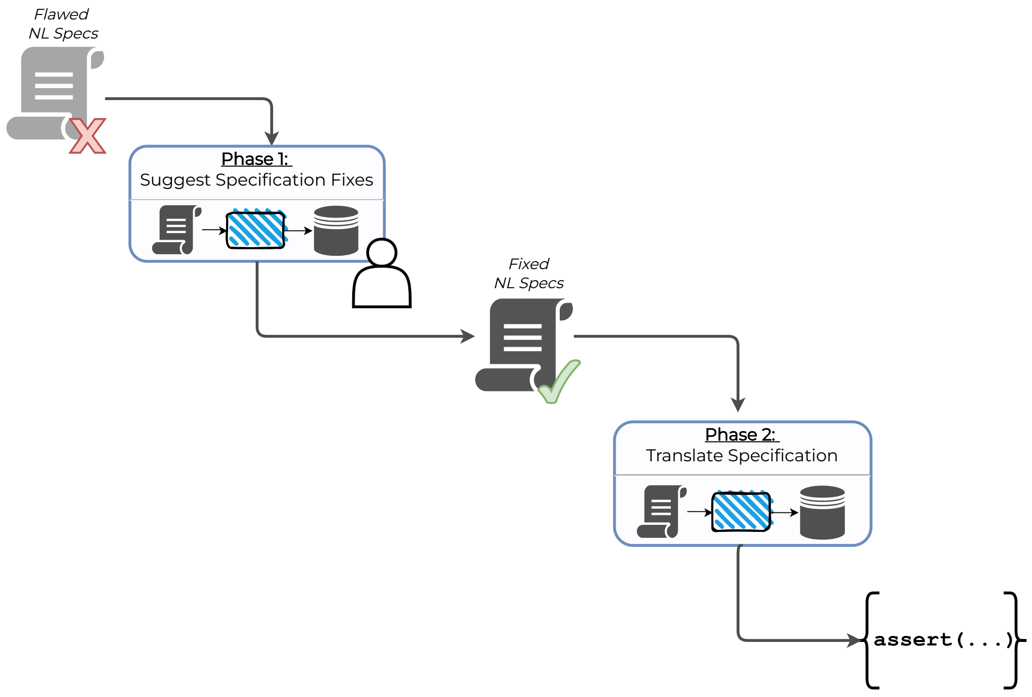

Figure 3.1. From natural language specifications to actionable test oracles

In this thesis, we also show the added value of embedding the generated oracles into test suites either written by developers or automatically generated with automatic test case generators such as Randoop Pacheco et al. [2007] and EvoSuite Fraser and Arcuri [2011]. We, of course, need to define the format of the generated oracles, which we require to be machine readable and automatically actionable .

We already observed that the Javadoc documentation includes information of little use and interest for testing purposes. Before even considering how to translate information from Javadoc documentation to actionable assertions, we need to identify the elements in the documentation that are relevant for testing purposes and are worth translating into actionable assertions. It is also worth noticing that Javadoc documentation contains information that corresponds to different kinds of properties:

Prescriptive properties properties that prescribe what the user of a software should or should not do. They specify the correct usage of the code, for example, in terms of preconditions.

Descriptive properties properties that describe the effect of the software behavior. They specify how the code execution affects the state of the SUT.

Both prescriptive and descriptive properties are relevant for testing purposes, since the correct behavior of a software is defined by indicating both the conditions to execute the code (prescriptive properties) and the effects of executing the code (descriptive properties). In the following chapters, we define approaches to automatically generate both prescriptive and descriptive properties.

Chapter 4

Deriving Test Oracles From Semi-structured Javadoc

In this section, we present a technique and a tool, *JDoctor*, to derive executable assertions from Javadoc block tags. We also report the results of a set of experiments that confirm the high precision and recall of *JDoctor* in deriving pre, post and exceptional-post conditions for Java methods from Javadoc structured specifications. The assertions that *JDoctor* automatically generates improve the test cases generated with automatic test case generators, such as Randoop. The core contributions of this chapter is the main content of a paper that we presented in the technical track of the 2018 International Symposium of Software Testing and Analysis(Blasi et al. [2018]).

Artifact format

Semi-structured: Javadoc tags

Artifact scope

Specific aspects of the documented method: Determined by each tag

JDoctor generates actionable assertions from the semi-structured Javadoc specification at the method level that provides information about the expected behavior of the documented methods. The format of the semi-structured Javadoc specifications provides some useful hints about the code context. The core contribution of *JDoctor* relies in the intuition of exploiting the context provided with the partial format of semi-structured Javadoc specifications to disambiguate the information in natural language and produce actionable assertions. The core challenge that *JDoctor* addresses is the natural language ambiguity of the textual description.

4.1 Translating Pre, Post, Exception-Post Conditions

The semi-structured portion of a Javadoc method documentation offers information about pre-conditions, normal post-conditions, and exception post-conditions. Pre-conditions describe the expected input that the method does or does not accept. For example, the method may not accept null inputs. Normal post-conditions describe what the method returns. For example, the method may return true if some conditions hold, false otherwise. Exception post-conditions describe the exceptions the method throws in case some non-acceptable conditions. For example, the method may throw `NullPointerException` upon receiving null inputs. All of these pieces of information are precious for unit testing, as *assertions* can be derived from them.

4.1.1 New Knowledge To Acquire

Mixing between natural language and developers' jargon An off the shelf natural language parser is not designed to properly interpret all the information that non-formal code specification can contain. We need tweaks to properly parse code identifiers (e.g., method signatures), CS terms (e.g., is `null` an adjective or a noun?) , and arithmetic notation (e.g., `x>0`).

Unveiling the relations between natural language terms and code identifiers Of course, when a code identifier is directly mentioned in a natural language comment, the relation is made explicit. However, developers may use verbs to describe an actions that refers to a specific method call, or even use *synonyms* to describe it.

4.1.2 Overview of *JDoctor*

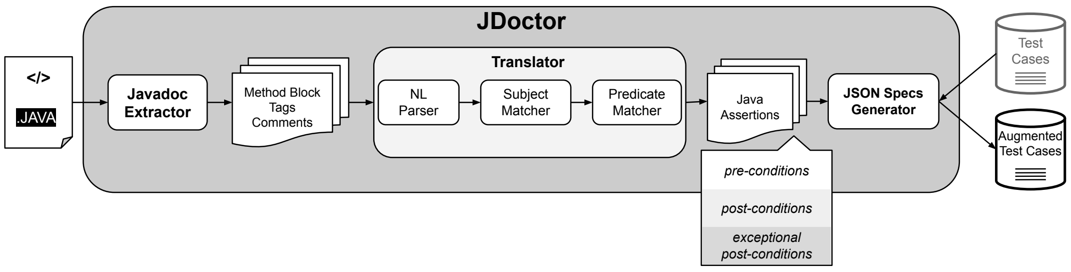

Figure 4.1. JDoctor's workflow

JDoctor works on the Javadoc documentation of methods, using the Javadoc documentation of the declaring class as input. It extracts and analyzes *@param* tags, *@return* tags and *@throws* (or *@exception*) tags. *JDoctor* identifies the presence of translatable oracles inside these tags and translates the natural language information into executable Java assertions, which are the final output of the technique.

Preconditions `@param` tags typically characterize method parameters and state the preconditions that callers must respect . Consider again the example of low specification ambiguity from section 3.2, belonging to the `BloomFilter` class of Google Guava. *JDoctor* transforms this comment into the executable specification that is shown in the box below the Javadoc comment. The clauses are conjoined with the Java conditional operator "and" (&&) to form the complete procedure specification.

```
/**
 * ...
 * @param  expectedInsertions  the  number of expected  insertions
 *       to  the  constructed  BloomFilter;  must be  positive
 * @param  fpp  the  desired  false  positive  probability  (must be
 *       positive  and  less  than  1.0)
 */
public  static  <T> BloomFilter<T> create(
Funnel<? super T> funnel,
 int  expectedInsertions,
double fpp)  {  ...  }
```

expectedInsertions > 0 *fpp > 0 && fpp < 1.0*

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

Exceptional Postconditions `@throws` and `@exception` tags represent postconditions of exceptional executions. Consider the following example, shown in 3.2 to represent a case of subject ambiguity, belonging to `CollectionUtils` class of Apache Commons Collections:

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

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

JDoctor correctly determines that “either collection” refers to parameters `a` and `b`, and “the comparator” refers to parameter `c`.

Again, we borrow an example presented in section 3.2, coming from class `Graph` of JGrapht. This example represents the highest ambiguity possible:

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

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

JDoctor infers that “the graph” is the instance of the `Graph` class itself, which we encode as `receiverObjectID`. Less obviously, it also infers that “not found” is semantically related to the concept of “an element being contained in a container”, thanks to its semantic similarity analysis.

Normal Postconditions `@return` tags typically represent postconditions of regular executions of methods. Below we report an example from method `addEdge` of the `JGraphT` library.

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

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

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

4.1.3 *JDoctor*'s Extractor

The Extractor is the entry point of *JDoctor*. It takes in input the Javadoc documentation of a class under test and extracts the content of the Javadoc tags of each method: *@param*, *@return*, *@throws* and *@exception*. The extractor pre-processes the language text to cleaning it from noise (e.g., HTML tags) and retaining valuable information. The Extractor is not merely responsible of text cleaning. Its main duty is to prepare the text so that it is effectively exploitable by the next component, which will have to translate it into code. The Extractor hence produces an internal representation of the Javadoc comment holding all the useful information that can ease the later translation. In particular:

Punctuation: Jdoctor adds a terminating period when it is absent and removes spurious initial punctuation. Indeed, sometimes developers may (incorrectly) use commas in Javadoc comments to separate parameter or exception names from their descriptions.

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

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

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

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

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

Inner tags: words inside “@code” tags are memorized to be wrapped by placeholders. In later phases, they will be tagged as nouns in the semantic graph produced by the NL parser. Words inside “@link” tags are memorized to be later remembered as possible pointers to useful resources. If a code matching cannot be found inside the class under test, it may be in one of the class linked in the documentation.

4.1.4 *JDoctor*'s Translator

The Translator takes in input the comment representation produced by the Extractor. As we see in the workflow of Figure 4.1, the Translator operates by sub-phases.

Natural Language Parsing Given an English sentence, Jdoctor identifies ⟨subject, predicate⟩ pairs, commonly known as *propositions* Del Corro and Gemulla [2013]. It also identifies conjunctions and disjunctions that connect propositions, if any.

Jdoctor first performs a *partial* POS (Part-Of-Speech) tagging. It marks parameter names as nouns, and inequality placeholders such as `LT0` as adjectives. Jdoctor then completes the POS tagging process by means of the Stanford Parser[1] Marneffe et al. [2006], which produces a semantic graph (i.e., an enriched parse tree)[2] representing the input sentence. In a semantic graph, nodes correspond to the words of the sentence, and edges to grammatical relations between them.

Jdoctor identifies the words that comprise the *subject* and the ones that comprise the *predicate* by traversing said graph. Given the single node marked as subject (e.g., by the `NN` POS tag) in the semantic graph, Jdoctor identifies the complete subject phrase by visiting the subgraph with the subject node as root node and collecting all the words involved in a relation of type compound, adverbial modifier, adjectival modifier, determiner, and nominal modifier. Jdoctor builds a predicate by collecting words with the following grammatical relations: auxiliary, copula, conjunct, direct object, open clausal complement, and adjectival, negation, numeric modifiers.

When conjunctions are involved, Jdoctor correctly supports multi-clause sentences by processing the corresponding edges.

Proposition Translation The proposition thus built must be finally translated. When translating propositions into Java expressions (line 12), the goal is to

[1] `http://nlp.stanford.edu/software/lex-parser.html`
[2] `http://universaldependencies.org/u/dep/all.html`

match each subject and predicate to code elements.

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

Jdoctor starts by analyzing the subject of the proposition (line 14) and tries to match the subject to a code element, which may be a parameter of the method, a field of the class, or the class itself. If Jdoctor finds a matching expression for the subject, it proceeds looking for a corresponding matching predicate retrieving the code identifiers of all public methods and fields within the scope of the subject as possible candidates.

JDoctor attempts to translate a predicate according to the most suitable strategy in a cascade-like fashion. (i) It checks whether the predicate matches a set of predefined translations (line 18), (ii) it looks for lexically similar matches (line 20), (iii) it searches for matches according to semantic similarity (line 22) .

Pattern Matching: Jdoctor uses *pattern matching* to map common phrases such as “is positive”, “is negative”, and “is null” to the Java expression fragments `>0`, `<0`, and `==null`, respectively. Pattern matching can efficiently translate common patterns but has of course limited applicability.

Lexical Matching: Jdoctor tries to match a subject or a predicate to the corresponding code element looking at the lexical similarity between words in Javadoc comments and words composing code identifiers. Jdoctor (i) tokenizes code candidates into separate terms according to camel-case convention, (ii) computes the Levenshtein distance between each term and each word in the subject/predicate, and (iii) selects the candidate with the smallest Levenshtein distance, as long as it does not exceed a threshold (with a very small default threshold (i.e. two) to avoid wrong matches as much as possible). Jdoctor uses a lexical matching similar to the one performed by Toradocu Goffi et al. [2016].

Algorithm 1 Comment Translation

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

11: /** Given a proposition (i.e. a pair of subject and predicate), find code elements to match
    subject and predicate. */
12: function MATCH-PROPOSITION(proposition)
13:     subjCandidateList = GET-SUBJECT-CANDIDATES(subject)
14:     matchedSubject = LEXICAL-MATCH(subject, subjCandidateList)
15:     if no match for subject then
16:         return
17:     predCandsList = GET-PREDICATE-CANDIDATES(predicate)
18:     matchedPredicate = PATTERN-MATCHING(predicate, predCandsList)
19:     if no match for predicate then
20:         matchedPredicate = LEXICAL-MATCH(predicate, predCandsList)
21:     if no match for predicate then
22:         matchedPredicate = SEMANTIC-MATCH(predicate, predCandsList)
```

Semantic Matching: The Jdoctor semantic matching approach compensates for cases where nor pattern matchin,g nor syntactical matching can provide a correct code translation. Syntactically different terms can have a close semantics, as we saw in the example from JGraphT with method `containsVertex` in the code and the concept "is not found in the graph" in the comment. Jdoctor's semantic metching exploits *word embedding*, which has been proved to be a powerful approach to represent semantic word relations. It embeds words in a high-dimensional vector space such that distances between words are closely related to the semantic similarities, regardless of the syntactic differences. In particular, Jdoctor uses GloVe,[3] a two-layer neural network model. GloVe is however not sufficient to handle the comparison among whole sentences. Hence, Jdoctor relies on the *Word Mover's Distance (WMD)* algorithm Kusner et al. [2015]. Similarly to what Jdoc-

[3] `https://nlp.stanford.edu/projects/glove/`

tor does for lexical matching, it selects the candidate that has the closest semantic distance up to a given threshold.

Despite offering different matching strategies, Jdoctor resorts only to lexical similarity for subject matching. This approach forces Jdoctor to match subjects to code elements with a very high precision (though it may miss some matches). This conservative decision is vital for the performance of Jdoctor, since subject matching gives the scope to later match the predicate. A wider — and possibly wrong — scope would direct the search for predicate matching towards completely wrong paths. As we stressed since the introductory sections of the thesis, having no oracle would be more desirable than having a wrong oracle.

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

4.1.5 *JDoctor*'s Generator

The last component is responsible for outputting the Java assertions in an actionable format. *JDoctor* exploits a JSON structure that can be be easily read by external tools. In Listing 4.1 we see an excerpt of JSON output for method `load` of class `CacheLoader`, from Google Guava.

Listing 4.1. JDoctor's real example of JSON output

```
[
  ...
  {
  "signature": "load(java.lang.Object key)",
  "name": "load",
  "containingClass": {
    "qualifiedName": "com.google.common.cache.CacheLoader",
    "name": "CacheLoader",
    "isArray": false
  },
  "targetClass": "com.google.common.cache.CacheLoader",
  "isVarArgs": false,
  "returnType": {
    "qualifiedName": "V",
    "name": "V",
    "isArray": false
  },
  "parameters": [
  {
    "type": {
```

```
21        "qualifiedName": "java.lang.Object",
22        "name": "Object",
23        "isArray": false
24      },
25      "name": "key"
26    }
27    ],
28
29    "paramTags": [
30    {
31      "parameter": {
32        "type": {
33          "qualifiedName": "java.lang.Object",
34          "name": "Object",
35          "isArray": false
36        },
37        "name": "key"
38      },
39      "comment": "the non-null key whose value should be loaded",
40      "kind": "PARAM",
41
42      "condition": "args[0]!=null"
43    }
44    ],
45    "returnTag": {
46      "comment": "the value associated with key; must not be null",
47      "kind": "RETURN",
48
49      "condition": "true ? (methodResultID==null)==false"
50    },
51    "throwsTags": [
52    {
53      "exceptionType": {
54        "qualifiedName": "java.lang.Exception",
55        "name": "Exception",
56        "isArray": false
57      },
58      "codeTags": [],
59      "comment": "if unable to load the result",
60      "kind": "THROWS",
61
62      "condition": ""
63    },
64    ...
65  ]
```

At line 28 we find `@param` tags, and at line 41 we find the translation for the condition about the only parameter of the method. Similarly, we find the translation for the `@return` tag at line 48. There is no boolean assertion to translate for the `@throws` tags, hence line 61 reports an empty condition.

4.1.6 Experimental Evaluation of *JDoctor*

The evaluation of *JDoctor* is two-folds. We evaluate it in terms of its translation accuracy, and the usefulness of its assertions in the context of testing.

The translation accuracy evaluation answers two research questions:

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

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

Table 4.1. Subject programs and ground-truth translations. Column "Doc'd Classes" reports the total number of classes with documentation, out of which we selected "Analyzed Classes". "Analyzed Methods" reports the methods with Javadoc tags that the authors of this paper could express in executable form.

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

Precision measures correctness as the proportion of the output that is correct, with respect to missing and wrong outputs. The output is correct (C) when Jdoctor produces a specification that matches the expected specification. The output is missing (M) when Jdoctor does not produce any specification. The output is incorrect when Jdoctor either produces a spurious (S) specification when no specification is expected, or a wrong (W) specification that does not match the expected one. A partially correct translation is still considered wrong. For example, if the comment is "`@throws` Exception if x is negative or y is null", then the translation "`x < 0`" is deemed as wrong.

Precision is defined as the ratio between the number of correct outputs and the total number of outputs:

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

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

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

JDoctor's accuracy is evaluated on the six well-maintained open-source Java systems in table 4.1. Column "Doc'd Classes" reports the number of classes that satisfy these conditions for each subject.

For each analyzed method in each selected class, we manually determined its ground truth — the correct translation of Javadoc comments to executable method specifications.

To answer **RQ2**, *JDoctor*'s performance is compared with the one of two similar tools in the state of the art:

- **@tComment** Tan et al. [2012] pattern-matches against predetermined templates for three different types of nullness specifications.

- **Toradocu** Goffi et al. [2016] generates exceptional postconditions from `@throws` comments by means of a combination of NLP and string matching.

Table 4.2 reports the accuracy of @tComment, Toradocu, and Jdoctor on the subject classes of table 4.1. Toradocu does not handle preconditions, and neither Toradocu nor @tComment handle normal postconditions (values *n.a.* in the table). The data in the table show that Jdoctor's precision is comparable with state-of-the-art approaches, and Jdoctor's recall is substantially higher than state-of-the-art approaches.

Finally, we evaluated the use of Jdoctor's procedure specifications for improving the generation of test cases. We do so by integrating the JSON output of *JDoctor* into the popular test case generator Randoop Pacheco et al. [2007].

As other test case generation tools, Randoop first generates test cases and then heuristically classify each one as:

1. failing (or error-revealing) test cases that reveal defects;

2. passing (or normal, or expected) test cases that can be used as regression test cases;

3. illegal (or invalid) test cases that should not be executed; for example, because they either violate some method preconditions or their oracles are incorrect.

Table 4.2. Accuracy (precision, recall, f-measure) of tools that translate English to executable procedure specifications.

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

The test generation tool outputs both failing tests and passing test cases to the user. The heuristics may misclassify test cases, leading to both *false* and *missed* alarms. By accessing some (partial) specifications, TCGs can improve the classification of generated test cases, thus detecting more errors and/or reducing the human effort required to identify false alarms.

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

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

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

As explained, Jdoctor outputs a JSON file containing executable Java expressions corresponding to each method preconditions, normal postconditions, and exceptional postconditions. When a method lacks a Javadoc comment or its Javadoc comment lacks a `@param`, `@return`, or `@throws` clause, the JSON file contains no corresponding expression. We extended Randoop to take advantage of this information during test generation.

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

We modified Randoop to create Randoop+Jdoctor as follows:

- After choosing the arguments but before creating or executing the candidate test, Randoop+Jdoctor reflectively executes the precondition expressions. If any of them fails, then Randoop+Jdoctor discards the test, exactly as if it had been classified as invalid. In this way, Randoop+Jdoctor avoids the possibility of misclassifying it as an error-revealing or passing test.

- If the test completes successfully, Randoop classifies it as passing. Randoop+Jdoctor reclassifies it as failing if a normal postcondition (a translated `@return` clause) does not hold. Randoop+Jdoctor handles conditional postconditions, such as "@return true if this graph did not already contain the specified edge", because Jdoctor provides information about the conditional.

- While executing the test, Randoop catches any thrown exceptions. If the exception matches one in an exceptional postcondition (a translated `@throws` clause), then Randoop+Jdoctor classifies the test as passing iff the `@throws` condition holds. If the exception does not match, Randoop+Jdoctor falls back to Randoop's normal behavior of heuristically classifying the test.

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

We ran both Randoop and Randoop+Jdoctor on all the 6 programs of table 4.1. To answer **RQ3**, we measured when Randoop+Jdoctor classified a candidate test differently than Randoop (giving to the two tools the same time limit of 15 minutes). There are five possibilities:

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

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

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

Table 4.3. How Jdoctor output (procedure specifications) improves Randoop's test classification. Each cell is the count of candidate tests that were classified differently by Randoop and Randoop+Jdoctor, for one run of Randoop.

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

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

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

A manual inspections of all the 3 *False alarms* and the 20 *Invalid tests* confirms the results. Invalid tests are newly classified as such thanks to Jdoctor specifications on parameters (pre-conditions). For instance, method max() in class com.google.common.primitives.Longs of Guava states that parameter array is “a nonempty array of long values”. Thus any test passing an empty array to max() was correctly classified as invalid since it violates the preconditions.

To answer **RQ4**, we manually inspected some of the 85 *Missed alarms*. Unfortunately, we could spot several incorrect results due to mis-translated comments (precision is not 100% in table 4.2). An example is due to the comment “@throws NullPointerException if the check fails and either @code errorMessageTemplate or @code errorMessageArgs is null” which Jdoctor translated as errorMessageTemplate = = null | | errorMessageArgs = = null, incorrectly missing the part on the failing check. This specification wrongly makes Randoop+ Jdoctor classify tests having any null value as failing if they do not throw a NullPointerException.

Chapter 5

Deriving Test Oracles From Unstructured Javadoc

In this section, we present two techniques and corresponding tools, *MeMo* and *CaMeMa*, to derive metamorphic and temporal specifications from Javadoc summaries. We define the tools and present the experimental results about the precision and recall of *MeMo* and *CaMeMa* in translating text into test oracles. We discuss the integration of *MeMo* and *CaMeMa* within automatic testing, and present the results of experiments involving automatic TCGs. The first part of this chapter reflects the main content of a paper we published in the Journal of Systems and Software (Blasi et al. [2021b]).

Artifact format

Unstructured: Method or class summary

Artifact scope

The whole documented method or class

Moving from semi-structured specification to unstructured specification, we lose the format advantages of the former. The direct bridge that exists, for example, from a `@param` tag to its specific method parameter, is now unavailable. Consider how a Javadoc summary at the method level may report any information concerning the documented method: Again, we are not dealing with formal specification, so there is no strict rule or scheme a developer should follow when writing it. The information the summary reports could be relevant for testing purposes or not, and, even when relevant, it may still not be directly translatable into testing code (e.g., executable assertions).

Luckily, unstructured informal specification is not a completely unknown matter. Some researchers attempted to identify and classify the kind of information we may find in unstructured summaries, both in methods (Monperrus et al. [2012] and in classes (Rani et al. [2021]). Considering for example Monperrus et al. [2012], we can easily see how *State Directives*, *Alternative Directives* and even *Synchronisation Directives* can all be relevant to test the correct behavior of a method. In this thesis, as per method summaries, we focused particularly on discovering and translating *metamorphic relations*, which can be seen as Alternative Directives in Monperrus et al.'s taxonomy, and *temporal constraints*, a kind of State Directives. We then reflected on the possibilities of class usages reported in class summaries, one of the category identified by Rani et al.'s study.

5.1 Discovering And Translating Metamorphic Relations

As introduced in Chapter 2, metamorphic testing resorts to oracles to compare the results of two actions that should be functionally equivalent. Domain experts can identify and write metamorphic oracles for a known program, but the manual task would be onerous. Hence, in recent years, techniques to automatically identify metamorphic relations (MRs) emerged. Such techniques may 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. In this PhD work, we aim to automatically recognize and translate MRs expressed in natural language inside informal documentation.

5.1.1 New Knowledge To Acquire

Unveiling the context in unstructured text Method summaries are completely unstructured, meaning we lose the context information given previously by a Javadoc block tag. We need techniques to recognize the context information that interests us.

Comparing the effects of method calls To fairly compare the effects of the method calls, we must be sure they operate on object instances in the same status.

5.1.2 Overview of *MeMo*

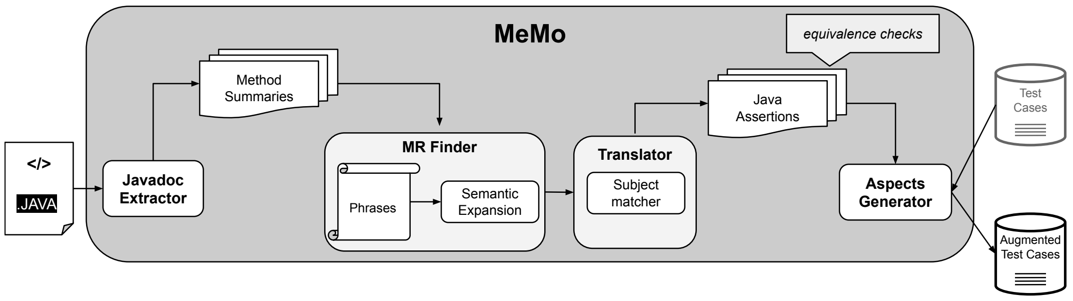

Figure 5.1. MeMo's workflow

The *MeMo* Blasi et al. [2021a] approach operates on Javadoc method *summaries*. A summary is typically composed of multiple sentences, differently from a block tag which is often described by a single sentence. Each of the sentences in a summary may express a property that can or cannot be relevant as test oracle. For this reason, *MeMo*'s architecture augments *JDoctor*'s with a new component, the Finder, that aims to find which sentences of a summary may be relevant as test oracle. Since *MeMo* attempts to find and translate *metamorphic relations* expressed in the text, MRs are the context we look for through the Finder.

In the real example of Listing 5.1 we see the full Javadoc summary of method `Iterables.cycle(T... elements)` in Google Guava's class `Iterables`. In the third sentence, it states that this method call should have the same functional behavior of `Iterables.cycle(Lists.newArrayList(elements))`.

Listing 5.1. 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) { ...
```

In the above example, Google's developers chose to express a MR through the wording "this method behaves exactly as...". Since summaries have a completely free-text format, a MR may be expressed in many other different ways, with sentences like "...this is equivalent to..." or "...it is identical to...". Then, they mix in code fragments. *MeMo* thus needs to first identify sentences that describe equivalent behaviors, which may be embedded in large text blocks like the one in Listing 5.1. Secondly, it needs to recognize the code elements involved in the metamorphic relation. While the latter is a challenge similar to one *JDoctor* already deals with, here the scope widens. Sometimes, the equivalence is said to hold between methods from different classes, or even different libraries.

To better illustrate these challenges, we can look into further real examples of Javadoc summaries expressing MRs. This time, we report only the relevant sentences and not the full summaries for simplicity.

```
/** Equivalent to newReentrantLock(lockName, false). */

public ReentrantLock newReentrantLock(String lockName) { ... }
```

```
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 next example from Guava's `Shorts` class, we see a MR involving calls from a different library, i.e., Oracle's JDK.

```
/** 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) { ... }
```

```
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.

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

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

JDoctor was not comparing the results of method calls, so using object clones is a novelty. Through the `receiverObjectClone`, *MeMo* can compare the states after two separate invocations of the methods involved in the metamorphic relation.

Of course, a metamorphic relation may involve multiple method calls nested at different levels. Considering again the example in Listing 5.1, this is the output *MeMo* produces:

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

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

Code snippets in summaries may even include multiple statements:

```
/**
 * For each occurrence of an element e in occurrencesToRemove,
 * removes one occurrence of e in multisetToModify. [...] this operation is
 * equivalent to, albeit
 * sometimes more efficient than, the following: for (E e :
 * occurrencesToRemove) { multisetToModify.remove(e); }
 */
```

```
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 in its body, thus making the snippet callable.

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

```
/**
* Removes a number of occurrences of the specified element from this
* multiset. [...] Note that if occurrences == 1, this is functionally
* equivalent to the call remove(element)
*/
```

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

We now proceed with the description of each *MeMo* component and how they work together to identify and translate equivalence MRs into executable specifications.

5.1.3 *MeMo*'s Extractor

The Extractor of *MeMo* differs from *JDoctor*'s mainly in extracting documentation summaries while ignoring subsequent tags. From a manual inspection of various Java projects, we observe that sections beyond summaries in Javadoc descriptions are much less likely to express metamorphic properties. This choice also makes sense intuitively, since summaries are supposed to provide general descriptions of the methods, including any interesting semantic property, while the following sections are instead much more specific and narrower in the type of information they convey[1].

5.1.4 *MeMo*'s Finder

The Finder processes the comments as given by the Extractor, and works in two phases. It first splits the comments into *sentences* as text separated by ordinary punctuation (i.e., a period followed by spacing). A metamorphic relation is normally expressed within a single sentence, so we design the second and more semantically rich phase of the Finder to operate on single sentences.

The Finder determines whether each sentence describes a metamorphic relation or not with a binary decision procedure that determines whether a sentence is relevant for our purposes. To recognize the useful sentences, the Finder adopts two strategies in cascade: equivalence phrase search, and, semantic expansion.

[1] https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html#tag

Equivalence phrase search

The equivalence phrase search uses a fixed set of ten equivalence phrases mined from real-world Javadoc documentation. The original was 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. By manually assessing each sentence we isolated the expressions typically used to express equivalence MRs. The resulting set of phrases is: *equivalent, similar, analog, like, identical, behaves as, equal to, same as, alternative, replacement for*. If *MeMo* finds any of this phrases followed by a method signature, it assumes a MR was found in the summary. Otherwise, the Finder tests the second strategy.

Semantic expansion of MR equivalence phrases

While the predefined set of equivalence phrases was derived from a large corpus and would already identify many MRs, restricting *MeMo*'s finding abilities to this would hinder generalization. *We must always assume developers' jargon in informal specification may vary across projects*. As a real example, the set of manually-mined equivalence phrases would not find the following MR in summary of method `push` from the `Graphstream` API:

Listing 5.2. Equivalence relation in Graphstream method summary

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

In this second strategy, the Finder returns true if a sentence in the summary contains text that is *semantically* similar to one of our manually-mined equivalence phrases. It does so by building a normalized version of the comment sentence. This means adding an explicit subject when missing and substituting the method signature it refers to with "`that method`". For the example in Listing 5.2, the Finder normalizes the code comment sentence "*A synonym for add(Edge)* " to "`this method is` *a synonym for* `that method`". Second, *MeMo* builds a dummy sentence for each equivalence phrase in the form of: "`method` ⟨*`equivalence phrase`*⟩ `that method`". Finally, the Finder compares the normalized summary sentence to each dummy sentence built via equivalent phrases: "this method `is equivalent to` that method" ... "this method `is same as` that method".

The comparison quantifies the semantic similarity between the sentences as WMD (Word Mover's Distance Kusner et al. [2015]). *JDoctor* uses WMD to match natural language actions to code names, while *MeMo* uses WMD to answer the question: “is the given sentence in the comment also expressing an equivalence?”.

The Finder returns true if any of the computed WMDs is below 20%, which means a similarity of at least 80%, and passes the sentence to the Translator. We set the threshold experimentally.

5.1.5 *MeMo*’s Translator

Similarly to *JDoctor*’s, *MeMo*’s Translator processes sentences to derive Java assertions that both refer to the correct code components and compile correctly in the applicable context of each assertion. However, *MeMo* does not normally need to identify and match propositions (i.e., identify and match subject and predicate) as *JDoctor*. *MeMo* adopts such a strategy only to translate the *condition* in the case of conditional equivalence, as the one in the example of Listing 5.1.2.

MeMo’s Translator first identifies and resolves the fully qualified names of all the method signatures found in the sentence (contained in whatever code fragment, including whole code snippets). Consider the following real 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* thus needs to explore the other project packages and external dependencies to find the right match for the translation

Then, differently from *JDoctor*, *MeMo*’s Translator already knows at this point that the predicate expresses an equivalence (which would be translated either with the == operator or a call to the equals() method). It can thus directly attempt the correct translation for the code elements that occur in the relation, and it does so via syntax matching (a translation strategy already saw, for example, in *JDoctor*’s subject matching). The Translator then ensures that the results of the documented method and the method (or code) which is said to be equivalent can indeed be compared. For example, assessing whether the primitive return types are of the same type. If they are, and the final assertion compiles, then it is given in output.

5.1.6 *MeMo*'s Generator

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.

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 2 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 2. 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, 13), the test case will pass if the metamorphic relation does hold as documented, and will fail otherwise.

In Listing 5.3 we see an Aspect generated for class `Vector1D` of Apache Commons Math. The aspect surrounds the invocation of the following documented method:

```
/** Calling this method is equivalent to calling: p1.subtract(p2).getNorm() */

public double
org.apache.commons.math3.geometry.euclidean.oned.Vector1D.distance(
  org.apache.commons.math3.geometry.euclidean.oned.Vector1D p1,
  org.apache.commons.math3.geometry.euclidean.oned.Vector1D p2)) {
  ...
}
```

methodResultID == args[0].subtract(args[1]).getNorm()

The equivalence expressed in the comment can be easily verified with a direct comparison (line 27 of method `equivalenceHolds`). If the comment were to report a complex snippet (e.g., code involving loops), the corresponding code would be wrapped inside method `snippetWrapper` (line 32), empty in this case.

Listing 5.3. Real *MeMo*-generated aspect

```
 1 @Aspect
 2 public class Aspect_1 {
 3   @Around("(call(double
 4   org.apache.commons.math3.geometry.euclidean.oned.Vector1D.distance(
 5     org.apache.commons.math3.geometry.euclidean.oned.Vector1D,
 6     org.apache.commons.math3.geometry.euclidean.oned.Vector1D))")
 7   public Object advice(ProceedingJoinPoint jp) throws Throwable {
 8     String output = "Triggered aspect: " + this.getClass().getName() + " (" +
 9     jp.getSourceLocation() + ")";
10     Object target = jp.getTarget();
11     Object[] args = jp.getArgs();
12     Object clonedTarget = new Cloner().deepClone(target);
13     Object result = jp.proceed(args);
14     if (equivalenceHolds(result, target,
15     clonedTarget, args)) {
16       System.err.println(output + " -> Success: Expected equivalence
17       holds");
18     } else {
19       fail(output + " -> Failure: Expected equivalence DOES NOT hold");
20     }
21     return result;
22   }

24   private boolean equivalenceHolds(Object methodResultID, Object
25   receiverObjectID, Object receiverObjectClone, Object[] args) {
26     // Calling this method is equivalent to calling: p1.subtract(p2).getNorm()
27     return ((((Double) methodResultID) ==
28       (((org.apache.commons.math3.geometry.euclidean.oned.Vector1D)
        args [0]). subtract ((( org.apache.commons.math3.geometry.euclidean.oned.Vector1D)
         args [1])). getNorm ())));
  }
  private void snippetWrapper(Object
  receiverObjectClone, Object[] args) {
  }
}
```

5.1.7 Experimental Evaluation Of *MeMo*

Similarly to Jdoctor's, *MeMo*'s evaluation aims to assess both its translation accuracy on MRs, and the usefulness of the generated oracles when applied to testing. Our experimental evaluation aims to answer the following research questions:

- RQ1: Can *MeMo identify* natural language sentences that express metamorphic properties and *translate* them into executable assertions?

- RQ2: How does *MeMo* compare with state-of-the-art technique SBES in terms of identified equivalence relations?

- RQ3: Do *MeMo* assertions improve testing when used as oracles?

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

As we did for *JDoctor*'s ground truth, we inspected all 7189 Javadoc sentences and manually translated all those that express a metamorphic relation into a code assertion. Table 5.1 reports statistics.

Table 5.1. Ground truth: manually-identified metamorphic relations (MR)

Project	Randomly Selected Classes	Sentences	MRs
Colt	9	477	19
ElasticSearch	10	228	14
GWT	17	448	44
GraphStream	3	126	11
Guava	33	1558	80
Hibernate	5	126	5
JDK	23	3381	72
Math	9	653	30
Weka	4	192	6
TOTAL	113	7189	281

Precision and recall are computed according to the same formulas shown in Section 4.1.6.

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 RQ3.

Table 5.2 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%.

We can see how *MeMo* is highly precise, albeit achieving a lower recall than *JDoctor*. 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 5.2, 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

Table 5.2. 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

```
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.

To identify a metamorphic relation within a comment, simple syntactic matching 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 5.2 (i.e., *"A synonym for..."*).

To answer RQ2, 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. *MeMo* deduces metamorphic relations from code documentation, meaning that, when correct, its output is as deterministic and sound as the original specification. SBES infers *likely* relations from executing the code, and its output depends on the initial test suite used in the search-based algorithm, and must be manually confirmed by the user.

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

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]))}
```

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*. *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.

Table 5.3. 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

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 5.3 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 5.3, 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.

Finally, to answer RQ3 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 5.2). 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 seconds for EvoSuite and 100 seconds 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 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

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

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 *MeMo* oracles.

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.

Figure 5.2 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

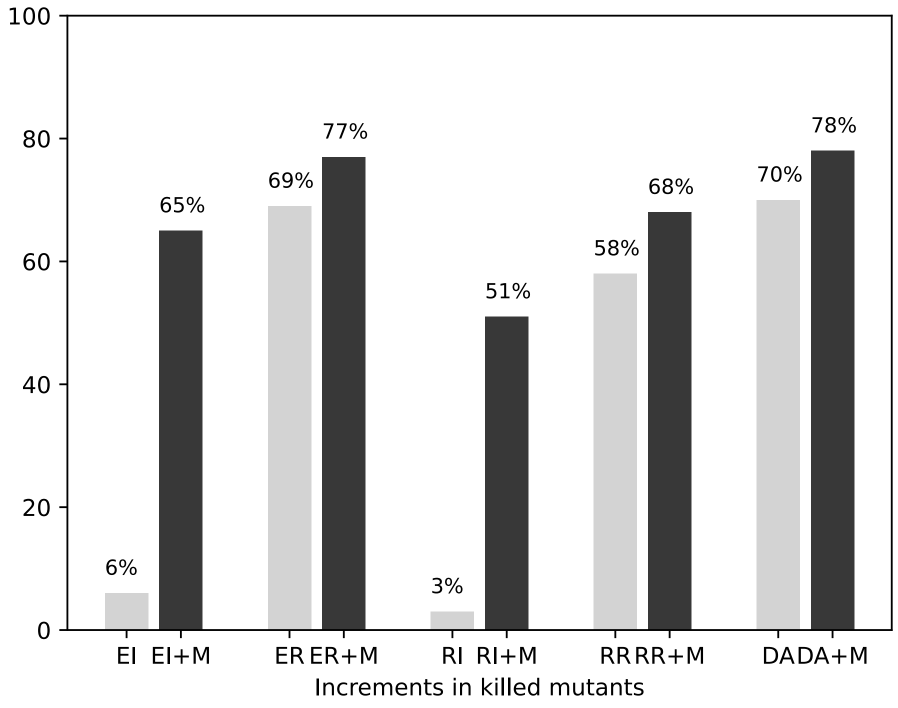

Figure 5.2. Improvement in mutants killed with *MeMo* oracles. Each pair of bars compares a test suite without *MeMo* oracles to one with *MeMo* oracles. EI stands for EvoSuite Implicit oracles, ER for EvoSuite Regression oracles, RI for Randoop Implicit oracles and RR for Randoop Regression oracles. DA means developers' assertions, referring to the developers' test suite. +M indicates augmented test suites with *MeMo* oracles.

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).

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 5.4. 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) { $\ldots$
```

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 5.4: *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`:

```
/** ... 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 5.1.7.

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.

5.2 Discovering And Translating Temporal Constraints

Temporal constraints, also referred to as *call protocols* Ramanathan et al. [2007], specify the correct sequence of method invocations for a proper use of an API. Call protocols are often inferred dynamically, and formalized as finite state machines Ammons et al. [2002]; Pradel et al. [2010] that can be used to support testing activities, as for example by comparing the execution of generated tests against the inferred API protocols Thummalapenta et al. [2009]; Pradel and Gross [2012]. In this thesis, we observe that free-text Javadoc summaries of either methods or classes often document temporal constraints. Thus we can support testing by formalizing the *documented* behavior of a program as specified by API developers. Doc2Spec was the first approach Zhong et al. [2009] to infer temporal constraints on resources following a similar intuition. Doc2Spec infers temporal constraints formalized as FSMs by exploiting a template: "resource creation methods, followed by resource manipulation methods, followed by resource release methods". The more recent ICON approach Pandita et al. [2016] offers a more general technique to recognize temporal constraints based on ML features, by observing that temporal constraints are not necessarily restricted to the Doc2Spec template. ICON was not applied to any testing or software engineering task. In this thesis, we propose a new approach that improves and complements all the above approaches.

To the best of our knowledge, there is no in-depth study on the way informal specifications either express temporal constraints or differentiate among different kinds of temporal constraints. In this section, we discuss *prescriptive* and *descriptive* constraints, overview the challenges in identifying temporal constraints in informal specification, and propose an approach to recognize temporal constraints expressed in informal specification and translate them into automatically exploitable test oracles.

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

A temporal constraint is **descriptive** if it describes the effects of an execution in terms of sequence of events. These constraints describe properties similar to the properties that *JDoctor* recognizes in `@return` and `@throws` block tags. A temporal constraint is **prescriptive** if it describes the proper usage of a class, to prevent undesired consequences during the program execution.

Listings 5.5 and 5.6 show examples of a descriptive constraint from Cern's Colt library and a prescriptive constraint from Apache Commons Collections, respectively. Listing 5.5 describes the effects of calling method `clear()` on an instance of `AbstractCollection`, and it is thus a prescriptive constraint.

Listing 5.5. Descriptive constraint from class AbstractCollection of the Colt library

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

Listing 5.6 enforces an order of method calls that must be respected to avoid undesirable behaviors, and is thus prescriptive. We cannot properly use an instance of `IteratorEnumeration` without invoking `setIterator` first.

Listing 5.6. Prescriptive constraint from class IteratorEnumeration of Apache Commons Collections

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

We observe that developers often document temporal constraints both in method and class summaries, by referring either explicitly to the concept of operation or implicitly to a method name. For example, developers may document a constraint on a `Thread` class that prescribes that a daemon should be set before the thread is started either explicitly or implicitly:

Explicitly referring to the concept of *operation* *EXAMPLE: "method `setDaemon` should be called before method `start`".* In this example, the term *call* refers to a general *operation* that can be performed in a program, similarly to *invocation*, *instantiation*, etc. The constraint is thus expressed by using an execution-related action, and, by explicitly mentioning the signature of the involved methods.

Implicitly referring to a method name *EXAMPLE: "the thread should be started after setting the daemon".* In this example, the term *started* implicitly refers to an invocation of method `start`, and the phrase *setting the daemon* to method `setDaemon`. The constraint is thus expressed by using actions implicitly encoding actual code identifiers.

This example is similar to the real one in Listing 5.7. The JDK Javadoc specification implicitly refers to the action of invoking method `start()` on a thread instance by using the proposition `(thread, is started)`.

Listing 5.7. Constraint that implicitly refers to method, from class Thread of the JDK

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

Listing 5.8 from Apache Commons Collections, instead, refers explicitly to method *calls*, thus falling back to the concept of "operation".

Listing 5.8. Constraint that explicitly refers to the concept of operation (call), from class LoopingListIterator of Commons Collections

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

Listing 5.9 from the Graphstream library is an example of a description in a class summary that both implicitly refers to the action of invoking methods and explicitly refers to method signatures:

Listing 5.9. Constraint from Graphstream class summary

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

Class summaries may also specify desirable (*good*) and undesirable (*bad*) *class usages* via code snippets. Listing 5.10 shows a Google Guava summary that indicates a bad usage through a snippet and discouraging statements. The text terms highlighted in red indicate the bad usage and occur both before and inside the snippet as code comments:

Listing 5.10. Summary that describes a 'bad constraint' from Google Guava

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

Listing 5.11 shows an Apache Commons Collections summary that indicates a good usage:

Listing 5.11. Summary that describes a 'good constraint' from Apache Commons Collections

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

Descriptive temporal constraints indicate useful assertions. However, oftentimes the temporal constraints described in Javadoc summaries are a repetition of constraints described in block tags, as we can infer from the Pandita et al. [2016]'s dataset. Since *JDoctor* already deals fairly well with block tags, descriptive constraints that we can find in summaries do not enrich much the assertions already produced with *JDoctor*. Meanwhile, prescriptive temporal constraints can be very useful in automatic testing: TCGs not aware of prescriptive constraints may easily generate invalid sequences leading to errors, thus resulting in many false alarms. For example, Randoop Pacheco et al. [2007] cannot infer what is expressed in the method summary of 5.6: It instantiates a new `IteratorEnumeration` and attempts to make calls on it without invoking `setIterator` first, leading to errors that are prompted to the user (false alarms).

5.2.1 New Knowledge To Acquire

Describing order of events To the best of our knowledge, ICON (Pandita et al. [2016]) by Pandita and others is the only study focusing on automatically identifying temporal constraints in informal specification. ICON is mostly based in machine learning and has, admittedly, difficulties in recognizing implicit constraints.

The general problem of distinguishing temporal information in natural language text is a tough challenge for the computational linguistics community (Sakaguchi et al. [2018]; Huang et al. [2016]). We take advantage of the characteristics of our domain, which is much narrower than the general linguistic problem, and does not require to address the challenge for any kind of natural language text.

The direction of a temporal constraint *MeMo* already parses method summaries, meaning we can take advantage of some existing knowledge. However, there is an important difference between translating an equivalence MR and a temporal constraint. Equivalences are bi-directional, thus we need only to compare the effects of the calls on the same instance, while ignoring the direction of the calls themselves. Temporal constraints are characterized by a precise direction, as an event must either follow or precede the other, and not the contrary. This introduces a relevant challenge in correctly translating constraints in natural language text.

Method VS Class summaries Class summaries contain constraints that describe either bad or good usages, either via natural language sentences or code snippets. Class summaries have a wider scope and are thus more challenging than method summaries. Beside, a snippet is just a piece of code that contains little descriptive elements. Understanding if a snippet either suggests a proper (good) class usage or discourages a bad one would be yet another issue to tackle.

CaMeMa [4] identifies and translates temporal constraints by extracting constraint descriptions from method summaries, and translating them into actionable temporal constraints. In principle, *CaMeMa* could work also on class summaries. Here we focus on solving the challenge for methods.

5.2.2 Overview Of *CaMeMa*

Figure 5.3 overviews the *CaMeMa* process. It extracts summaries from Javadoc method specifications, identifies descriptions of temporal constraints, translates the informal descriptions into rules, and generate temporal constraints that can be used to either augment or prune test suites.

CaMeMa addresses the new challenges of directional descriptions of temporal constraints by enriching the *MeMo* components with a Direction Chief component, that determines the direction of the method calls in the constraint.

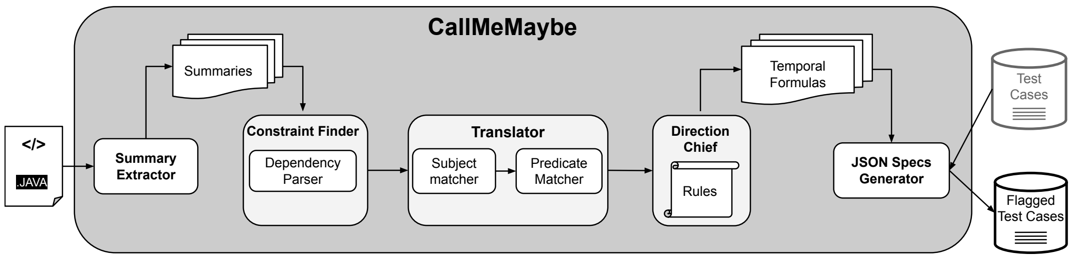

Figure 5.3. CaMeMa's workflow

5.2.3 *CaMeMa*'s Javadoc Extractor

CaMeMa Extractor is built on top of the *MeMo* extractor: It extracts information in method summaries and ignores any kind of Javadoc block tag following the summaries.

5.2.4 *CaMeMa*'s Constraint Finder

The *CaMeMa* Constraint Finder processes the information that it obtains from the Extractor by splitting the comments into *sentences* and identifying the ones describing temporal constraints. The strategy is similar to *JDoctor*, as it involves building a semantic graph and traversing it to build propositions. *CaMeMa* Constraint Finder looks for `advcl` and `advmod` dependencies. These dependencies

[4] Call Me Maybe.

identify adverbs and adverbial clauses, which fall into several categories, one of them being time [5]: We can thus identify clauses that express when an event happens with respect to a point in time, or another event.

By parsing the method summary *"This method must be invoked before the thread is started"* in Listing 5.7, the *CaMeMa* Protocol Finder generates the following semantic graph:

```
 1 -> invoked/VBN (root)
 2         -> method/NN (nsubjpass)
 3                 -> This/DT (det)
 4         -> must/MD (aux)
 5         -> be/VB (auxpass)
 6         -> started/VBN (advcl:before)
 7                 -> before/IN (mark)
 8                 -> thread/NN (nsubjpass)
 9                         -> the/DT (det)
10         -> is/VBZ (auxpass)
```

The dependency advcl:before at line 6 identifies an adverbial clause together with its specific modifier, *before*, thanks to the Enhanced English Universal Dependencies parsing (Schuster and Manning [2016]).

The Finder discovers this advcl:before dependency, along with *subject* ones, and extracts three propositions from them:

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

Finally combining the three in the following proposition series:

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

This final result is passed to *CaMeMa*'s Translator.

5.2.5 *CaMeMa*'s Translator

CaMeMa's Translator transforms temporal proposition series into temporal constraints. It identifies the code identifiers in each proposition, by considering both explicit references to operations and implicit references to method names.

[5] https://universaldependencies.org/en/dep/advcl.html

Recognizing the concept of "operation": The Translator identifies propositions that express actions explicitly referring to the concept of operation by relying on the SO_word2vec model Efstathiou et al. [2018], which is a word2vec model for the software engineering domain. It is pre-trained on over 15GB of textual data from Stack Overflow posts. The Translator starts from a basic golden set of three words: *operation*, *call* and *use*, and queries the model with this 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 Translator assumes the presence of a constraint, with the proposition subject being an explicit reference to a code element.

Recognizing actions encoding code identifiers: *CaMeMa*'s Translator identifies propositions containing actions implicitly encoding method names with a strategy adapted from *JDoctor*: It tries to match both subject and predicate against elements in the code by either Lexical Matching or Pattern Matching (Section 4.1.4)

In the example of Listing 5.7, the *CaMeMa* Translator identifies the subject of the first proposition `(This method, be invoked)` as the documented method ("this method" gets normalized with the corresponding method signature), and the predicate, "invoked", as a concept of interest by means of the word2vec model strategy. It then matches the subject and predicate in the second proposition, `(thread, is started)`, by looking for syntactic similarities with code identifiers. Subject "thread" refers to the instance of the receiving object of class `Thread`. Predicate "is started" refers to a call to method `start()`. This way, it solves all the calls involved in the constraint:

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

To generate an oracle, the *CaMeMa* Direction Chief needs to infer the direction of this temporal constraint from **BEFORE**, the temporal modifier in the proposition.

5.2.6 *CaMeMa*'s Direction Chief

The Direction Chief identifies the constraint directions by looking at the temporal modifier of the proposition. It uses a small set of fixed rules, one for each specific temporal particle. A rule essentially dictates the direction of the link between the two proposition, either a left arrow ←, or a right arrow →. The rules currently encoded in our *CaMeMa* implementation are AFTER, ONCE, BEFORE, PRIOR and UNTIL, as shown in Table 5.4. Such a small set of rules already serves a large number of cases, and the set can be trivially extended to include the application of further rules.

Table 5.4. Rules currently encoded in *CaMeMa*

Modifier	**Arrow**
AFTER	←
ONCE	←
BEFORE	→
PRIOR	→
(NOT) UNTIL	→

The *CaMeMa* Direction Chief solves the full constraint of our example as:

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

The rule-based strategy has the advantage of not relying on potentially deceiving heuristics, making the final translation into oracle reasonably safe. ICON Pandita et al. [2016] uses a strategy which looks at the verb tense: The past tense indicates a method call that must happen before, and vice-versa. This strategy does not generalize well to all temporal sentences. For instance, it would be difficult to asses the direction of a constraint such as *"`start` should be invoked after `setDaemon` is invoked"* just by considering the sentence tense.

5.2.7 *CaMeMa*'s Generator

The *CaMeMa* Generator produces temporal constraints as an actionable output. It models the constraints with a simple JSON structure, similarly to *JDoctor*. The structure indicates 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 *CaMeMa* ready to exploit by other tools such as automated TCGs. Listing 5.12 shows the excerpt of JSON output for our running example.

Listing 5.12. *CaMeMa*'s real example of JSON output

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

5.2.8 Experimental Evaluation Of *CaMeMa*

Similarly to the procedure we followed for *JDoctor* and *MeMo*, *CaMeMa*'s evaluation assesses its translation accuracy on constraints, and the usefulness of the generated oracles when applied to testing. The experimental evaluation aims to answer the following research questions:

- RQ1: Can *CaMeMa identify* natural language sentences that express temporal constraints and *translate* them into temporal formulas?

- RQ2: Do *CaMeMa* constraints reduce the human effort to assess false alarms and expected exceptions in automated testing when used as oracles?

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

As we did for *JDoctor* and *MeMo*, to produce the ground truth we inspect all the Javadoc sentences in the dataset and manually translate the ones expressing temporal constraints into temporal formulas. Tab 5.5 reports statistics.

Table 5.5. Ground truth: manually-identified temporal constraints

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

Table 5.6. Effectiveness of *CaMeMa* on 73 classes

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

To answer RQ1 we measure precision and recall. Table 5.6 reports the results. *CaMeMa* achieves good precision and recall, of 83% and 70% respectively.

To answer RQ2, we feed *CaMeMa* specifications to Randoop Pacheco et al. [2007]. We know that Randoop heuristics classify a test that throws a checked exception as expected behavior, and that it outputs the error-revealing and expected-behavior tests in separate test suites. Randoop+*CaMeMa* operates on both of them:

Inside error-revealing tests: Randoop adds a line of comment above the statement that raised exception explaining the cause. It does so by implementing some internal checks. We modify Randoop so to have *CaMeMa*-specific checks on error-revealing tests. That is, when Randoop detects some error for a method call for which we have a constraint specification, we run our additional *CaMeMa* checks. If the constraint was violated, the error test case is still given in output, but we enrich Randoop's default comment with information about the violation of a constraint. This way, the user knows such a error test case may actually be a false alarm.

Inside regression test suite: Randoop reports the message of the checked exception that was thrown, which it wraps in a `try/catch` block. Inside the `catch` clause, it also adds a comment stating "// Expected exception". In case we have a *CaMeMa* oracle specifying this behavior, we enrich the comment by reporting the natural language constraint as documented by the developer. This way, the user knows precisely what went wrong, and how a legitimate sequence should look like. Moreover, we can assess whether

Randoop heuristics were correct in their assessment.

Our experiments compare the original Randoop test generation tool with Randoop+*CaMeMa*, which extends Randoop with *CaMeMa*-generated temporal specifications. We run both Randoop and Randoop+*CaMeMa* on the programs reported in Table 5.5. To answer RQ2, we measure the quantity of test cases that Randoop+*CaMeMa* enriched with information about violated constraints. We leave the default time limit of 100 seconds for each class.

Five projects out of seven in Table 5.5 have constraints concerning file systems operation and other domains that make it hard for Randoop to generate satisfying test suites, at least without external suggestions for the inputs to be fed. With that said, we have the highest number of constraints for Apache Commons Collections and for the JDK, for which it is easy to automatically generate inputs, so we experiment with Randoop+*CaMeMa* using their classes. In Table 5.7 we report statistics.

Table 5.7. Randoop+*CaMeMa* signals, in terms of number of false alarms detected in error test cases, and enriched expected exceptions in regression test suites

Project	False Alarms	Expected Exceptions
Commons Collections	10948	8298
JDK	870	3726
TOTAL	11818	12024

We see how Randoop+*CaMeMa* is able to warn about hundreds of false alarms for the JDK, and thousands for Commons Collections, leading to a total of 11818 false alarms detected. Regression test suites are enriched by *CaMeMa* constraint specifications in thousands of test cases for both projects, precisely 12024 in total. Although a manual assessment of all the results would be hardly feasible, we here report some relevant examples.

Consider the constructor of class `IteratorChain`, documented in Listing 5.13.

Listing 5.13. Prescriptive constraint from class IteratorChain of Apache Commons Collections

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

CaMeMa translates the constraint in Listing 5.13 as the following temporal formula:

org.apache.commons.collections4.iterators.IteratorChain.addIterator(java.util.Iterator<? extends E>) ← receiverObjectID.org.apache.commons.collections4.iterators.IteratorChain()

In Listing 5.14, we see an excerpt of Randoop automatically generated test case for class `IteratorChain`.

Listing 5.14. Real example of Randoop+*CaMeMa* behavior for Commons Collections

```
 1 org.apache.commons.collections4.iterators.IteratorChain<java.io.Serializable>
 2  serializableItor0 = new
 3 org.apache.commons.collections4.iterators.IteratorChain<java.io.Serializable>();
 4 // The following exception was thrown during execution in test generation
 5 try {
 6    serializableItor0.remove();
 7   org.junit.Assert.fail("Expected exception of type
 8   java.lang.IllegalStateException; message: Iterator contains no
 9   elements");
10 } catch (java.lang.IllegalStateException e) {
11   // Expected exception.
12   /* CaMeMa constraint violation:
13   "Construct an IteratorChain with no Iterators.
14   You will normally use {@link #addIterator(Iterator)} to add some
15    iterators after using this constructor." */
16 }
```

In the first line, Randoop instantiates a new `IteratorChain`. Before doing anything else, at line 6, it attempts to invoke a removal operation on the newly instantiated iterator. This raises an expected `IllegalStateException` exception, which default messages explains that “Iterator contains no elements”. *CaMeMa* correctly recognized and translated the constraint for the constructor, and can thus report (right after the comment at line 11) that a correct sequence would first invoke method `addIterator` on the newly instantiated iterator.

Let us now consider the documentation of method `end()` of class `Deflater` in Listing 5.15.

Listing 5.15. Prescriptive constraint from class Deflater of the JDK

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

CaMeMa translates the constraint in Listing 5.15 as the following temporal formula (where the `receiverObjectID` preceded by the logical negation operator `!` means no further invocation should happen on the receiver object):

$$\boxed{\text{receiverObjectID.end()} \rightarrow \text{!receiverObjectID}}$$

Listing 5.16 shows the violation of this *CaMeMa* constraint in the error test suites for the JDK.

Listing 5.16. Real example of Randoop+*CaMeMa* behavior for the DJK

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

In the first line, Randoop instantiates a new `Deflater` object. A few statements later, at line 5, the test sequence calls operation `end()`. After this call, Randoop still attempts some operation on the object, i.e. a call at line 13 that raises a NullPointerException “in error” (line 7). However, *CaMeMa* knows this is probably a false alarm, given the violation of the constraint it reports at line 8.

Chapter 6

Automatically Improving Informal Specification

In this section we show how the imperfection of natural language specification can be overcome, to prevent it from hindering our test oracle generation workflow. We present two studies and tools, *UpDoc* and *RepliComment*, which prove to effectively detect defects in flawed Javadoc specification while suggesting fixes. *RepliComment* was originally presented at the IEEE/ACM International Conference on Program Comprehension Blasi and Gorla [2018], and later extended in an article published in the Journal of Systems and Software Blasi et al. [2021b]. *UpDoc* was presented at the International Working Conference on Source Code Analysis And Manipulation Stulova et al. [2020].

While natural language artifacts relating to software are nowadays commonly available (in the form of documentation, comments, wikis), they are prone to human error and imprecision. They may be poorly written, or not always being updated together with the code Fluri et al. [2007]; Wen et al. [2019]; Aghajani et al. [2019]. Correct information is an important requirement for the effectiveness of our automatically generated test oracles: Wrong oracles are even less desirable than no oracles. In the PhD work, we studied different kinds of issue affecting code documentation and proposed ways to automatically detect and overcome them.

6.1 Inconsistency Between Code And Documentation

When the code implementation changes, the corresponding informal specification is not always correspondingly updated. Studies show that documentation may be updated some time after the code changed, even months or several commits later Fluri et al. [2007] Aghajani et al. [2019]. In the worst case, documentation may get never updated.

To assess the correctness of source code during evolution there exist many automatic tools such as parsers, analyzers, compilers and linters. On the other hand, maintaining the correctness of its corresponding informal specification is largely responsibility of the programmer. The outcome is that either existing documentation becomes outdated, or, it is not written in the first place to avoid the problem altogether Fluri et al. [2007]; Wen et al. [2019]. This leads to many undesirable effects, such as the hindering of program comprehension (Aghajani et al. [2019]; Arnaoudova et al. [2016]). In the context of this PhD thesis and the framework it proposes, a possible undesirable consequence would be to derive wrong test oracles.

As a real example of long-lived code-doc inconsistency, consider the `AdaptiveIsomorphismInspectorFactory` class of the JGraphT library in Listing 6.1. In matching colors, we can see the code and the corresponding parts of the Javadoc documentation that relate to program fragments. Notice how the documentation mentions method parameters twice: implicitly in the summary (line 2: "`one of the graphs`") and explicitly with the `@param` tags (lines 5-6).

Listing 6.1. Method body with its doc comment

```
1  /**
2  * Checks if one of the graphs isfrom unsupported graph type and throws
3  * IllegalArgumentException if it is. The current
4  unsupported types
5  * are graphs with multiple-edges.
6  * @param graph1
7  * @param graph2
8  * @throws IllegalArgumentException
9  */
10 protected static void assertUnsupportedGraphTypes(
11         Graph graph1,
12         Graph graph2)
13         throws IllegalArgumentException{
14         Graph [] graphArray = new Graph [] {
15          graph1, graph2
```

```
        };
        for (int i = 0; i < graphArray.length; i++) {
                Graph g = graphArray[i];
                if ((g instanceof Multigraph)
                || (g instanceof DirectedMultigraph)
                || (g instanceof Pseudograph)) {
                        throw new IllegalArgumentException(
                        "graph type not supported for the graph"
                        + g);
                }
        }
}
```

Now take into consideration two revisions of the above code, namely `b4805f5` and `a68071b`. The first one modifies both the signature and the body of the method, but not the respective Javadoc, introducing code-documentation inconsistency:

```
- protected static void assertUnsupportedGraphTypes(
- Graph graph1,
- Graph graph2)
+ protected static voidassertUnsupportedGraphTypes(Graph g)
throws IllegalArgumentException {
        - Graph [] graphArray = new Graph [] {
        - graph1, graph2
        - };
        - for (int i = 0; i < graphArray.length; i++) {
        - Graph g = graphArray[i];
```

The second revision updates the documentation, but only at the `@param` tags level. We still read "`one of the graphs`" in the summary, although the method now takes only one argument:

```
- * @param graph1
- * @param graph2
+ * @param g
```

These two revisions are separated by 7.5 years. This is not surprising: In the next sections, we see with further examples how code-documentation inconsistencies do tend to survive the test of time.

6.1.1 Fine-grained Detection Of Code-Doc Inconsistency

During the PhD work we implemented a technique, *UpDoc*, that attempts to *automatically detect inconsistencies* between code and documentation Stulova et al. [2020]. The tool aims to detect inconsistency by relying on a fine-grained mapping between source code and documentation. *UpDoc* operates directly on the source files, by statically analyzing the information from text and code, without requiring training data. The same authors propose other techniques that detect code-comments inconsistencies via deep learning Panthaplackel et al. [2020, 2021] and require training. The most recent work of Panthaplackel et al. Panthaplackel et al. [2021] detects inconsistencies before they are committed to a repository, while *UpDoc* compares different committed versions of the source code, hence detecting already present errors.

6.1.2 Overview Of *UpDoc*

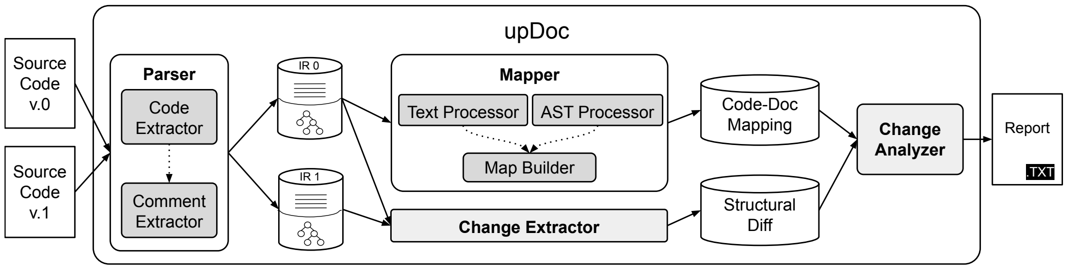

Figure 6.1. UpDoc's workflow

We schematically illustrate the full *UpDoc* architecture and workflow in Figure 6.1.

At a high level, the core of *UpDoc* takes in input the AST of a Java method code and its Javadoc documentation, and produces an internal mapping between them by representing both with bag of words. The technique then computes the similarity between code and documentation by means of cosine similarity and Word Mover's Distance. Such a technique generalizes to any programming language and its respective format of documentation, and does not require training nor extra human effort. The tool consists of four principal modules: Parser, Change Extractor, Mapper and Change Analyzer.

6.1.3 *UpDoc*'s Parser

The Parser is *UpDoc*'s entry point. It takes two revisions of a Java class source code, and extracts tuples of method bodies and their doc comments into an intermediate representation. For the documentation part, *UpDoc* extracts both the unstructured summary part and the sentences from the usual block tags `@param`, `@exception`, `@throws`, and `@return`. We disregard sentences in other block tags as they typically contain information that is not related to the implementation. In *UpDoc*, a single documentation sentence is the smallest unit the technique works on, either from summaries or block tags. For the code part, the Parser extracts the method signature AST nodes to store the method name, return type name, thrown exception type names, and parameter names together with respective type names for each of them.

6.1.4 *UpDoc* Change Extractor

The Change Extractor compares the two representations produced by the Parser for both source code versions. The goal is essentially to create a diff aware of any structural *change* happened between the two versions.

Working with AST-based representation of the source code, *UpDoc* filters out any purely syntactic changes (both for documentation and code), such as whitespaces and formatting edits. The Change Extractor focuses in detecting and holding AST nodes that have been *deleted* and *modified* in the change, as they are likely to require a matching change in the existing documentation text. Indeed, code addition is not strongly associated with comment changes, but rather with new comment addition if any Fluri et al. [2007]. The Change Extractor is built around the source code differencing functionalities of the `GumTreeDiff` framework Falleri et al. [2014] to benefit from the AST-based diffs of the source code under change. This allows the component to hold an internal representation of a commit as a list of changes happened on specific AST nodes. An AST node is identified by its range inside a specific source file (row+column position) and other information, such as its type (e.g., `SignatureNode` for a method signature).

6.1.5 *UpDoc*'s Mapper

The Mapper creates a mapping between the structural units of the source code and the documentation for each method of the code version *before* the change, i.e. SourceCode v.0 in Figure 6.1. The goal is to compute the level of consistency we originally had between code and documentation before the changes happened. The subcomponents of the Mapper are:

Text Processor The Mapper produces a bag-of-words (BoW) representation of each documentation sentence by (a) splitting all code identifiers into constituents with regular expressions that allow for camelCase and special character splitting, (b) expanding abbreviations (incorporating the dataset of Newman et al.

[2019]), (c) reducing each word to its stem, and (d) filtering out stop-words with the “Short English stopwords list” [1].

AST Processor We use AST-based representations of source code, just as Change Distiller Fluri et al. [2007] tool and the GumTree framework Falleri et al. [2014]. AST-based representation allows us to vary the granularity of the source code elements (as AST nodes at different depths) mapped to the comment text. We build the bag-of-words representation of AST nodes similarly to how we do it for the comments.

Map Builder Finally, the Mapper produces a many-to-many mapping between the AST nodes of source code and sentences of the comments.

The mapping reflects how *similar* code is to its relative natural language documentation comment. This depends on the vocabularies of the units of these two elements. For a code and a comment unit to be related and included into the mapping the similarity score of their BoW representations must be higher than a predefined threshold. There are various metrics to assess the similarity between two texts. Some of them, like *cosine similarity*, can only assess lexical similarity. Others, such as WMD we know well by now, have a *semantic* understanding of natural language words.

To illustrate the advantage of a semantic understanding of text over a more naive one, consider once again the code example of the Listing 6.1. In Table 6.1 we report similarity scores of the BoW representations of the comment text (which did not change), and the method signature before and after the change (BC and AC respectively). Using WMD *UpDoc* can relate the method pa-

Table 6.1. Similarity Measures Sensitivity

Comment	[1:illeg,1:argument,1:throw,1:one,1:except,1:check,1:unsupport,1:type,2:graph]		
		WMD	Cosine Sim
Method (BC)	[1:illeg,1:argument,1:void,1:assert,1:except,1:unsupport,1:type,4:graph,1:first,1:second]	**70%**	**75%**
Method (AC)	[1:illeg,1:argument,1:void,1:assert,1:g,1:except,1:unsupport,1:type,2:graph]	**66%** (-4%)	**75%** (no change)

rameters `graph1` and `graph2` to the independent clause of the first sentence of the method doc comment (“`Checks if one of the graphs is from unsupported graph type`”). When `Graph graph1` and `Graph graph2` disappear to leave only `Graph g`, the semantic understanding detects a decrease of the similarity with

[1] `https://www.ranks.nl/stopwords.`

respect to the comment. With cosine similarity *UpDoc* does not detect this inconsistency as the vocabulary is still too syntactically similar, despite the change. We implement both cosine similarity and WMD metrics in upDoc.

6.1.6 *UpDoc*'s Change Analyzer

The last component of *UpDoc* takes in input both the output from the Change Extractor and from the Mapper.

Iterating through the related sentences of the node under change, the Change Analyzer checks if all related sentences are present in the diff, and warns the programmer if not. In case of code modification (or addition), this component issues a warning if the new code does not have any relation to the comment text (e.g., common identifiers or domain terms). In Listing 6.2 we see an excerpt of report for our running example:

Listing 6.2. Excerpt of real *UpDoc* final report

```
WARNING: Consistency between doc and code decreased!

Original documentation:
Checks if one of the graphs is from unsupported graph type and throws
IllegalArgumentException if it is. [0.69]
The current unsupported types are graphs with multiple-edges. [0.58]
graph first [0.81]
graph second [0.80]
illeg argument except [0.48]

Documenting node:
| in original source sub-element SimpleName in
lines [219..219]:
graph1
| inside of MethodSignature in lines [218..222]:
protected static void assertUnsupportedGraphTypes(
Graph graph1,
Graph graph2)
throws IllegalArgumentException

Current documentation:
Checks if one of the graphs is from unsupported graph type and throws
IllegalArgumentException if it is. [0.66]
The current unsupported types are graphs with multiple-edges. [0.57]
graph first [0.47]
graph second [0.45]
illeg argument except [0.46]

Documenting node:
| changed  into sub-element
SimpleName in lines [217..217]:
g
| inside of MethodSignature in lines [217..219]:
protected static void assertUnsupportedGraphTypes( Graph g )
throws IllegalArgumentException
```

The report warns the user (line 1) and specifies what change brought the decrease in the code-documentation consistency. It does so both at a fine-grained level and coarser-grained level (lines 12–14, and 30–32). Looking at the report, we immediately know that the original parameter's simple name, `graph1`, was changed into `g`. We know the precise line ranges, and, what is the coarser node the parameter belongs to (i.e., its `MethodSignature`). We, as well, see the original and updated similarity scores with respect to the documentation, in all its parts (lines 5 and 23 for the summary, lines 7–8 and 25–26 for the `@param` tags). It is thus easy for the developer to asses where and how a fix should happen.

6.1.7 Experimental Evaluation Of *UpDoc*

The goal of our experimental evaluation was to assess the efficacy of the Mapper, the very core component of *UpDoc* on which all the inconsistency assessments depend. The evaluation involved 67 comment changes from the dataset of Wen et al. Wen et al. [2019]. As *UpDoc* evaluates the code-comment consistency based on similarity scores, our research hypothesis is that *the Mapper would report higher similarity scores in a mapping for the fixed version*.

We focused on the first 50 entries of the dataset, and we manually analyze the results of *UpDoc* runs. We discarded 8 of them because none of the changes involved comment lines (these are false positives admittedly included in the dataset Wen et al. [2019]). We further discard entries that do not involve comments that *UpDoc* focuses on: 4 documentation changes affecting only class-level doc comments (either general class or field descriptions), and 18 entries that involve changes only in comments within method bodies. This leaves us with 20 commits, each involving one or more changes in method doc comments which we use to evaluate mappings produced by *UpDoc*.

For these 20 commits, which amount to a total of 67 changes, we follow this evaluation protocol:

1. We run *UpDoc* on the version right before the commit, and we store the similarity score for each method and corresponding comment affected by the change.

2. We run *UpDoc* on the supposedly fixed version.

3. We compare the similarity score for each method signature and each corresponding doc comment sentence, before and after the change.

The manual analysis of the results revealed that:

- In **50 cases** the similarity **scores improve** as expected. Nearly half of these cases (24 out of 50) are trivial cases where there was no documentation and developers added some. 26 cases are instead actual fixes or significant semantic improvements to the comment text.
- In **10 cases** the similarity **scores did not change**. Manual analysis revealed that all these changes are either minor formatting fixes (e.g., adding/removing white spaces and new lines) or other minor edits that do not change the semantics of the documentation. The results produced by *UpDoc* are thus expected, since these changes do not address a real code-comment inconsistency.
- In **7 cases** *UpDoc* reports **unexpected decreases** in the similarity scores. The manual analysis shows that these are all due to current limitations of the prototype, which at the moment analyzes only method signatures. Details included in the documentation indeed could match similar elements in the method body, but they do not always match the signature alone.

In conclusion, *UpDoc*'s mapping between methods and doc comments accurately reflects inconsistencies in 90% of the cases.

6.2 Harmful Code Documentation Clones

As a relevant instance of code-documentation inconsistency, we focus on documentation clones. With the PhD work we demonstrated that code documentation suffers for harmful clones, just as the code itself. Developers may copy the API documentation referring to a method and paste it into another method, forgetting to revise the text according to the target method. Not surprisingly, such an issue generates confusion in the documentation readers Arnaoudova et al. [2016].

In our thesis framework, this is yet another obstacle to the generation of correct test oracles. It was exactly through our techniques that we had spot the first case of harmful documentation clone. The example we had shown previously in Listing 3.5 caused *JDoctor* to produce a false positive: An oracle correct according to the documentation, but not according to the implementation. The generated output was the following:

```
/**
* ...
* @return  true  if  this  matcher matches every character  in  the
*      sequence, including  when the sequence is empty
```

```
*/
public boolean matchesNoneOf(CharSequence sequence) { ... }
```

```
⟶ target.matchesAllOf(args[0])?result==true
```

Thus implying that the result of the invocation of method `matchesAllOf` and method `matchesNoneOf` should have been the same (obviously not true). *JDoctor* correctly recognized via semantic understanding of the text that "to match every character" corresponds to the invocation of `matchesAllOf`, but the failing assertion rather revealed an issue in the informal specification.

6.2.1 Automatic Detection And Fix Suggestion For Doc Clones

Our original study and technique, *RepliComment*, was the first approach that focused on harmful documentation clones. It showed that the issue occurs even in widely-used, well-developed and well-documented systems Blasi and Gorla [2018]. The study started by focusing on the detection of comment clones. The findings of the tools were analyzed manually. This preliminary study allowed to distinguish three different kind of documentation clones:

Legitimate clones. Some comments are legitimately cloned and do not create problems. As an example, cloning the same method summary in the case of method overloading produces a correct comment. Hence, such cases should not alarm the developer.

Poor information clones. Comments may get cloned because they are so vague that could document a large number of code elements, without providing useful information. This is the case for example of @throws tags documented with *"on error"*.

Copy-and-paste mistakes. These clones represent the worst case scenario, such as the Guava example described above. For whatever reason, a comment gets copied from its legitimate method to another one that has little to do with it. The method victim of this erroneous paste operation ends up being documented with a piece of text that does not describe its actual behavior.

In an expanded, more recent study Blasi et al. [2021b], we heavily improved *RepliComment* to make it automatically classify the documentation clones. For the copy-and-paste mistakes, *RepliComment* also suggests the more appropriate fix thanks to the integration with *UpDoc*.

Code clones are by far the most studied kind of source clones. The Software Engineering community is rather familiar with the topic, so it is worth it to highlight commonalities and differences. According to the state of the art taxonomy Roy and Cordy [2007], code clones can be instances of Type I, i.e. exact copy-and-paste clones, up to Type IV clones, i.e. semantically equivalent code snippets. Comment clones can be classified according to the same taxonomy as follows:

Type I comment clone: The comment of a code element, i.e. a method, a class or a field, is an exact copy of the comment of another code element except for whitespace and other minor formatting variations.

Type II comment clone: The comment of a code element is an exact copy of the comment of another code element except for identifier names.

Type III comment clone: The comment of a code element is an exact copy of the comment of another code element except for some paragraphs. For instance, the Javadoc comment of a method has the very same free text of another method, but the `@param` , `@throws` , or `@return` tag descriptions differ. Conversely, tag descriptions may be the same, and Javadoc comments may differ in the free text description of the method.

Type IV comment clone: The comment of a code element is lexically different, but semantically equivalent to the comment of another code element.

RepliComment aims to find *problematic* Type I and Type III comment clones affecting methods and fields within the same class, across classes within the same hierarchy, or across classes within the whole project. *RepliComment* does not report Type II clones since comments differ in identifiers, and therefore likely document their corresponding piece of software correctly. Interestingly, we will see in later sections that documentation clones do not seem correlated to code clones, thus highlighting that different approaches must be adopted to tackle the two issues.

6.2.2 Overview Of *RepliComment*

We start by presenting some real examples of comment clones that *RepliComment* can deal with.

A critical comment clone is that of a comment that is copied from a correctly documented method or field, and erroneously pasted to another code entity whose functionality differs completely. One example of this issue exists in the Google Guava project in release 19:[2]

In this example (see Sample 1), the Javadoc `@return` tag of method `matchesNoneOf` is a clone of method `matchesAllOf`, offered by the same class `CharMatcher`. It is easy to see that the return comment of the second method does not match the semantics of its name, while it does match the semantics of `matchesAllOf`. This clone is clearly an example of a copy-and-paste error. It is conceivable that the developers first implemented method `matchesAllOf`, and later implemented `matchesNoneOf` starting from a copy of the first method. The two methods have a similar purpose, i.e. to filter a collection of elements, however in the first case the filter returns all elements matching a given pattern, while in the second case it returns those that do *not* match the given pattern.

```
/**
 * @return true if this matcher matches every character in the
 * sequence, including when the sequence is empty.
 */
public boolean matchesAllOf(CharSequence sequence) { +\ldots+ }
```

```
/**
 * @return true if this matcher matches every character in the
 * sequence, including when the sequence is empty.
 */
public boolean matchesNoneOf(CharSequence sequence) { +\ldots+ }
```

***Sample 1**: Comment clone due to copy-and-paste error.*

Comment clones may also be examples of poor documentation that could be improved to offer a better understanding for developers. See the following example from a non-public class in the Apache Hadoop project release 2.6.5:

```
/**
 * @return true or false
 */
@InterfaceAudience.Public
 @InterfaceStability .Evolving
public synchronized static boolean isLoginKeytabBased() throws IOException
{ +\ldots+ }
```

```
/**
 * @return true or false
 */
public static boolean isLoginTicketBased ()  throws IOException { +\ldots+ }
```

***Sample 2**: Comment clone of poor information.*

[2] `http://google.github.io/guava/releases/19.0/api/docs/com/google/common/base/CharMatcher.html`

These two methods offered by class `UserGroupInformation` have exactly the same comment regarding the postcondition. It states that the methods return either true or false, which is correct. However, the documentation is uninformative, since any boolean method obviously returns either true or false. A more useful documentation should state what the boolean value represents, e.g. whether it is a system component status, or the result of a conditional check. Such clones are symptoms of documentation that could be improved, and thus *RepliComment* aims to report them as well.

Not all comment clones are necessarily an issue to report to developers. They may occur for legitimate reasons, such as when two methods offer the same functionality. The following example comes from class `SolrClient` of the Apache `solr` library release 7.1.0:[3]

```
/**
 * Deletes a single document by unique ID
 * @param collection the Solr collection to delete the document from
 * @param id the ID of the document to delete
 */
public UpdateResponse deleteById(String collection , String id) { +\ldots+ }
```

```
/**
 * Deletes a single document by unique ID
 * @param id the ID of the document to delete
 */
public UpdateResponse deleteById(String id) { +\ldots+ }
```

***Sample 3**: Legitimate comment clone due to method overloading.*

The clone in this case affects the free text in the Javadoc comments. Methods `deleteById()`, however, are an example of function overloading. Given that they have similar purposes, it is legitimate for their method descriptions to be identical. The difference between these two methods, which lies in their parameter lists, is properly documented through the custom `@param` tags.

Let's proceed to understand how practically *RepliComment* can detect such clones and propose fixes. We show the workflow of *RepliComment* in fig. 6.2.

[3] `https://lucene.apache.org/solr/7_1_0//solr-solrj/org/apache/solr/client/solrj/SolrClient.html#deleteById-java.lang.String-java.lang.String-`

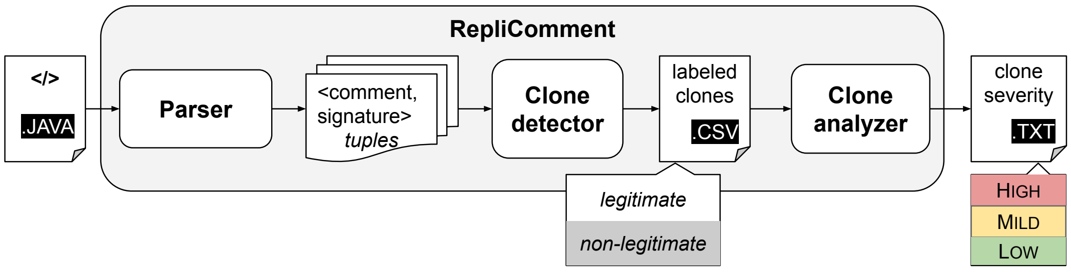

Figure 6.2. *RepliComment*'s workflow

6.2.3 *RepliComment*'s Parser

The Parser component of *RepliComment* takes as input a single Java file, identifies the list of declared methods and field, and stores all method signatures and field names. For each method and field it then identifies the corresponding Javadoc documentation and parses it, extracting the summary and the usual block tags, similarly to *UpDoc*.

The Parser outputs a list of tuples of field names and method signatures, and their respective pre-processed Javadoc comments.

6.2.4 *RepliComment*'s Clone Detector

The Clone Detector aims to identify likely comment clones and distinguish the legitimate and non-legitimate clones. It loops through all the method and field declarations identified by the Parser and looks for Type I clones of whole (summary and block tags altogether) Javadoc comments. It then proceeds to detect Type III clones, i.e. clones of specific comment parts across different methods. Indeed, a single comment part may be cloned while the rest of the comment is not. A single comment part is the smallest clone unit *RepliComment* looks for, and it may be the summary or each one of the block tags.

The Clone Detector would flag a potential comment clone if two methods (or fields) use the same comment to describe the method (or field), either entirely or just in some parts. However, such a naive check is prone to false positives. Hence, this component uses several heuristics to filter out false positives and only flag real clone suspects. The Clone Detector operates in two main steps:

1. It takes the tuples produced by the Parser. For a method, it compares each comment block with the same type of comment blocks of all the other methods within the same file or across files, according to the desired scope. First, it compares the whole Javadoc documentations to check whether there are

whole comment clones documenting methods. Then, it proceeds with the comparison of single *comment parts*: it compares each `@param` tag comment with other `@param` comments and so on. As for fields, the comment always consists of a single free-text summary.

2. When the Clone Detector finds that two or more clones of Javadoc comment, it checks whether the clone might be legitimate or non-legitimate. *RepliComment* never considers whole Javadoc comment clones to be legitimate, as we will explain later.

   Legitimacy of a documentation clone is established if it satisfies any of the following heuristics:

   - the clone is found in methods with the same (overloaded) names
   - the comment describes the same exception type
   - the clones affect parameters that have the same name
   - an exception comment consists of at least 4 words and does not match a generic exception description pattern (recognized via a regex). We have observed that three words are insufficient to express the conditions under which an exception is thrown; furthermore certain generic patterns, such as *"@throws exception for any kind of error,"* are common
   - a `@return` tag clone describes methods with the same, non-primitive return type. This is useful for filtering out APIs with methods that always update the class instance and return it, for which it is legitimate to have comments such as *"@return a reference to this."*
   - constructors without parameters may have cloned comments, since they can have very generic comments, according to the official Oracle guide to writing good Javadoc documentation[4]
   - comment clone describing fields with same name in different classes.

Finally, clones processed by the heuristics are stored in a `csv` report file as tuples with the following items:

- the fully qualified name of the class,
- the signature of the first method or field,

[4] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#defaultconstructors`

- the signature of the second method or field,
- the type of cloned Javadoc comment part (i.e. whole, free-text, `@param` , `@return` or `@throws`),
- the cloned text, and
- a value indicating if the clone is considered legitimate or rather non-legitimate by the Clone Detector.

The `csv` report is the input to the next component, which performs an analysis of the clone suspects to determine their severity level.

6.2.5 *RepliComment*'s Clone Analyzer

The Clone Analyzer core is implemented upon *UpDoc*. This component takes as input the `csv` file produced by the Clone Detector and performs an analysis only on comment clones flagged as non-legitimate. Clones flagged as legitimate are ignored, trusting the judgment of the heuristics described in section 6.2.4. This way, the heuristics act as a filter on all the possible cases of comment clones that can be encountered in a Java project and may contain a high number of false positives. Since the Clone Analyzer needs to perform a careful analysis on each suspect, the heuristic filter helps to significantly reduce the computational effort.

Clone analysis algorithm Listing algorithm 3 shows the pseudo-code of the code analyzer analysis, specifically referring to method comments since they are the most complex to deal with. When dealing with field names instead of method signatures, the reasoning about similarity thresholds is the same.

As we see in line 3 of 3, the Clone Analyzer first checks whether the clone under analysis is a whole Javadoc comment clone. Such types of clones need special consideration. As the official Oracle guide to the Javadoc tool explicitly specifies, developers should *"write summary sentences that distinguish overloaded methods from each other"*.[5] Hence, when a *whole* Javadoc comment is cloned, *RepliComment* assumes there is some sort of issue no matter if the methods are overloaded or not. In other words, whole Javadoc comment clones are never considered legitimate by the Clone Detector, and are never labeled as Low severity issue by the Clone Analyzer. In case of overloading, the Clone Analyzer flags such an issue as Mild severity, and *RepliComment* will report the problem suggesting the developer to correctly document the difference in the parameters.

[5] `https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#doccommentcheckingtool`

Algorithm 3 Clone analyzer

```
1:  /** Given a pair of method signatures and the cloned Javadoc comment, return the severity score of the clone as a
    warning */
2:  function ANALYZE-COMMENT-CLONES(methodSignature1, methodSignature2, clonedJavadoc)
3:      if clonedJavadoc is of type WHOLE_JAVADOC_BLOCK then
4:          if IS-OVERLOADING(methodSignature1, methodSignature2) then
5:              REPORT(Please document parameter)
6:              WARN(MILD_SEVERITY) & EXIT
7:          else
8:              REPORT(Not overloading: fix these comments)
9:              WARN(HIGH_SEVERITY) & EXIT
10:     m1Sim = COMPUTE-SIMILARITY(methodSignature1, clonedJavadoc)
11:     m2Sim = COMPUTE-SIMILARITY(methodSignature2, clonedJavadoc)
12:     if m1Sim < MIN-THRESHOLD and m2Sim < MIN-THRESHOLD then
13:         REPORT(Please fix poor info comment)
14:         WARN(MILD_SEVERITY)
15:     if m1Sim > 0.50 and m2Sim > 0.50 then
16:         REPORT(This looks like a false positive)
17:         WARN(LOW_SEVERITY)
18:     if | m1Sim - m2Sim| > DIFF-THRESHOLD then
19:         REPORT(Please fix method with lowest sim score)
20:         WARN(HIGH_SEVERITY)
21:     else
22:         REPORT(Fix these comments)
23:         WARN(HIGH_SEVERITY)
```

Otherwise, the Clone Analyzer flags the issue as HIGH severity. We assume that there are major issues to fix if unrelated methods have the same comment.

In lines 10 and 11 of 3, the Clone Analyzer computes the similarity scores between the cloned comment and each of the involved methods (we explain the details of this computation below). The similarity scores are used to determine whether the clone is a LOW, MILD or HIGH severity issue:

- Both methods can achieve a very low similarity score with respect to the cloned comment (line 12): the assumption is that the comment is so generic that it does not document well enough either of the methods. We set the MIN-THRESHOLD value to 0.25, based on empirical evidence that this value is the best balance to detect correct matches, while limiting false positives. This is a MILD severity issue, and the Clone Analyzer requires the developer to add more detail to the comment for those methods.

- Both methods can achieve a very high similarity score with respect to the cloned comment (line 15): in this case the comment looks good enough for both. These cases were not filtered out by the heuristics of the Clone Detector in section 6.2.4, but look like false positives nonetheless. Thus, they are reported to be LOW severity issues by the Clone Analyzer.

- If none of the above cases hold, then first we consider the case where one method achieves a significantly better similarity score than another. The method that achieves the highest similarity score is assumed to be the real owner of the comment, while the other is reported to be the victim of a mistaken copy-paste. We set the DIFF-THRESHOLD value to 0.1, once again due to empirical evidence. If both methods have very close similarity scores, both comments are reported as needing correction. Comment clones for which the owner is clearly distinguishable tend to be Type III clones, such as the one in Sample 6.2.2. Indistinguishable comments, instead, mostly belong to Type I clones, i.e. whole comment clones. Such comments are not overly generic, but at the same time, they are not informative enough to highlight the distinction between two different code elements. This case is reported as a HIGH severity issue, urging the developer to fix the wrongly-documented method(s).

We now expand on the description of how a similarity score between a method and its comment is computed.

Method-comment similarity computation We take the full method signature and the part of the method comment marked by *RepliComment* as a likely clone and compute the similarity between them based on natural language cues present in each of them. Our underlying assumption here is that both the comment text and the identifiers in the signature (method name, parameter names, type identifiers , etc.) are written in the same language. This allows us to rely on natural language processing (NLP) techniques to extract vocabularies of each entity, and use the similarity of vocabulary-based representations as a proxy for method-comment similarity.

The first step in the similarity computation is source text processing. For text in comment parts it means identifying full period-terminated sentences using the Stanford CoreNLP toolkit [6], in case the comment consists of more than one sentence. Next, for each sentence we split all source code identifiers present into their individual constituents and expand all detected abbreviations. Finally, we reduce each word to its stem, and we filter out common English stop words. After this step we transform the resulting text into a bag-of-words (BoW) representation. For the method signatures the pre-processing steps are similar, though in this case we start directly with identifier splitting.

After we have obtained two bag-of-words representations, we evaluate their similarity based on the occurrence of common words, for which we employ the

[6] `https://stanfordnlp.github.io/CoreNLP/coref.html`

cosine similarity measure. For a pair of BoWs we consider them to be related if the similarity measure value is above a threshold of 0.25 (MIN-THRESHOLD value in the Algorithm 3), on a scale from 0 (no similarity at all) to 1 (exact similarity).

Clone severity computation After computing the similarity scores, *RepliComment* assigns a degree of severity to the issue (LOW, MILD, or HIGH) as described previously. Finally, *RepliComment* exports the results of its evaluation to a text (`.txt`) report file with a separate entry for each issue category. Each file reports:

1. the record in the `csv` file of clone suspects
2. the specific Java class the clone is from
3. a description of the issue(s) encountered
4. fix suggestions, which differ depending on the type of issue:
   (a) in the case of a HIGH severity issue, *RepliComment* points out which field or method is the one more related to the cloned comment, suggesting to fix the documentation of the other field or method
   (b) in the case of a MILD severity issue, *RepliComment* warns the user that the comment cloned across different fields or methods seems too generic, hence suggesting to fix each comment by providing more detail
   (c) in the case of a LOW severity issue, *RepliComment* warns the user of the clone found, but specifies that she may want to ignore the issue because it is likely a false positive (legitimate clone)

We see how a portion of the `txt` file reporting HIGH severity issues looks like in Listing 6.3:

Listing 6.3. RepliComment Results file example

```
Record 53 in file: log4j.csv
In class: org.apache.log4j.lf5.LogRecord

1) The comment you cloned:"(@return)The LogLevel of this record."
seems more related to <LogLevel getLevel()> than <Throwable
getThrown()>

It is strongly advised to document method <Throwable getThrown()> with
a different, appropriate comment.
```

```
Record 152 in file: hadoop-hdfs.csv
In class: org.apache.hadoop.hdfs.util.LightWeightLinkedSet

1) The comment you cloned:"(@return)first element"
seems more related to <T pollFirst()> than <List pollN(int n)>

It is strongly advised to document method <List pollN(int n)> with
a different, appropriate comment.
```

6.2.6 Experimental Evaluation Of *RepliComment*

We evaluate *RepliComment* on different dimensions. One goal was to understand the accuracy of *RepliComment* in *identifying and categorizing* comment clone issues. We also conducted a qualitative analysis of the results to investigate whether the issues reported as HIGH severity, which are supposed to be the most worrisome comment clones, are indeed critical documentation issues that developers should fix. Finally, we compared the clone issues reported by *RepliComment* and by a code clone detection tool to study the correlation between code and comment clones.

For our empirical evaluation we select and analyze 10 projects among the most popular and largest repositories on GitHub, as listed in Table 6.2. Specifically, in our study we include projects developed in Java, since *RepliComment* targets this programming language, and these projects include a considerable number of classes documented with Javadoc. We selected these projects because they belong to different companies and developers (*e.g.*, Google, Apache, Eclipse), and thus the study is not biased towards specific documentation styles.

Evaluation Protocol and Research Questions The evaluation of *RepliComment* answers the following research questions:

- *RQ1: Are comment clones prevalent in popular Java projects?* We perform a quantitative study on all the classes of all the projects listed in Table 6.2 to motivate this work. We report the numbers of HIGH, MILD and LOW severity cases that we find in each subject, and we report the results in section 6.2.6.

- *RQ2: How accurate is* RepliComment *at differentiating* legitimate *and* non-legitimate *comment clones?* It is essential that *RepliComment* be able to differentiate between clones that developers should analyze and fix (*non-legitimate* clones), and clones that are *legitimate*. We manually analyze

Table 6.2. Subjects used for the evaluation of *RepliComment*. For each subject we report the number of implemented classes, the lines of Java code and the stars on GitHub as of July 2020

Project	Classes	LOC	Github ★
elasticsearch-6.1.1	2906	300k	50k
hadoop-common-2.6.5	1450	180k	11k
vertx-core-3.5.0	461	48k	11k
spring-core-5.0.2	413	36k	38k
hadoop-hdfs-2.6.5	1319	262k	11k
log4j-1.2.17	213	21k	718
guava-19.0	469	70k	38k
rxjava-1.3.5	339	35k	43k
lucene-core-7.2.1	825	103k	4k
solr-7.1.0	501	50k	4k
Total	1665	1105k	

225 samples of the HIGH, MILD and LOW severity cases that *RepliComment* reports as *non-legitimate* to assess whether they are false positives. Moreover, we manually analyze 200 samples among the cases that *RepliComment* flags as *legitimate* to assess if they are false negatives. We report the results of this evaluation in section 6.2.6.

- *RQ3: How effective are the newly-introduced heuristics at filtering our* legitimate *cases?* RepliComment-V1 Blasi and Gorla [2018] did not include all the heuristics and further improvements that we now implement. We evaluate how effective they are at reducing the number of false positives against the RepliComment-V1 implementation, and we present these results in section 6.2.6.

- *RQ4: How accurate is* RepliComment *at classifying the severity of* non-legitimate *comment clones?* We examine the manually analyzed samples of the previous research question, focusing on how accurate *RepliComment* is at flagging HIGH, MILD and LOW severity cases as such. The results of this evaluation appear in section 6.2.6

- *RQ5: Can* RepliComment *correctly identify the cloned vs. the original comment?* When *RepliComment* finds an instance of a *non-legitimate* comment clone due to a copy-paste error, it reports which comment of the pair is the one that should likely be fixed. We evaluate how accurate this information is in section 6.2.6.

- *RQ6: To what extent do comment clones detected by* RepliComment *correlate with code clone issues?* We investigate how often *RepliComment* reports comment clone issues for methods that are detected as clones by code clone detection tools, and report our findings in section 6.2.6.

Overall, we manually analyze over *500 cases* of comment clones.

RQ1: Prevalence of Comment Clones Table 6.3 shows the complete quantitative data that *RepliComment* outputs for the *method* comment clone search. We report the number of comment clones by type of clone (CP — comment part, WC — whole comment) and severity of the issue (LOW, MILD or HIGH). For each project, the first row reports the results of running *RepliComment* with default scope search (*i.e.*, INTRA-CLASS); the second row (HIERARCHY) reports the additional clones with class hierarchy scope; and the last row (INTER-CLASS) the additional clones with INTER-CLASS search scope.

RepliComment reports a total of 11,368 method comment clones considered to be potential issues, and discards 61,459 comment clones considered to be *legitimate*. For the hierarchy search, *RepliComment* reports 325 additional potentially harmful clones, while it flags 2494 additional *legitimate* clones. Finally, for the inter-class search, *RepliComment* reports 49,255 additional clones, while 145,719 more clones are labeled as *legitimate*.

legitimate We can see that the vast majority of the comment clones are not harmful. The total of 209,672 comment clones labeled as *legitimate* by the *Clone detector* heuristics are not subsequently analyzed by the *Clone analyzer*, and therefore are not reported to developers. 60,948 are left to be analyzed, namely, 23% of the total reported issues.

LOW In the intra-class search, 3,973 cases, *i.e.*, 35% of the 11,368 *non-legitimate* reported issues, are considered to be LOW severity issues, and they all come from comment part clones. In hierarchy search, this is the case for 91 cases of 325 (or 28%), 15 for comment part clones and 76 for whole comment clones. For inter-class search, 31746 (1946 comment parts, 29,800 whole comments) are LOW severity issues over a total of 49,255 (or 64%). This means that the *Clone analyzer* component of *RepliComment* thinks all those cases might be false positives, despite overcoming the filtering heuristics of the *Clone detector* (subsection 6.2.4). Thus, *RepliComment* is able to prune additional clones thanks to the analysis phase.

Table 6.3. Quantitative results of the **method** comment clones reported by *RepliComment* on each analyzed project.

Project	Low		Mild		High		Tot. issues	Legit
	CP	WC	CP	WC	CP	WC		
elasticsearch	111	0	23	567	30	184	**915**	2221
Hierarchy	+4	+39	+2	+21	0	+6	**+72**	+51
Inter-class	+924	+28857	+138	+82	+117	+899	**+31017**	+4323
hadoop-common	100	0	75	173	28	4	**380**	3859
Hierarchy	+2	+15	0	0	0	+1	**+18**	+97
Inter-class	+64	+84	+569	+17	+55	+6	**+795**	+2314
vertx-core	33	0	139	53	795	4	**1024**	17433
Hierarchy	0	+1	+2	0	0	+3	**+6**	+378
Inter-class	+368	+115	+1636	0	+5579	+13	**+7711**	+109558
spring-core	46	0	78	83	15	6	**228**	2089
Hierarchy	+1	0	0	+3	+1	0	**+5**	+75
Inter-class	+192	0	+5	+8	+11	0	**+216**	+964
hadoop-hdfs	23	0	184	13	7	13	**240**	1198
Hierarchy	+1	+11	+12	0	+1	+1	**+26**	+71
Inter-class	+19	+608	+1131	+10	+12	+3	**+1783**	+897
log4j	1	0	3752	437	1	18	**4209**	16689
Hierarchy	0	+2	0	0	0	0	**+2**	+1434
Inter-class	+16	+6	+3752	+9	+1	+4	**+3788**	+18615
guava	75	0	63	215	77	63	**493**	1122
Hierarchy	+2	+1	+127	+44	0	+4	**+178**	+79
Inter-class	+16	+9	+2066	+49	+20	+6	**+2166**	+4091
rxjava	3558	0	12	15	48	4	**3637**	11533
Hierarchy	0	0	0	0	0	0	**0**	0
Inter-class	+2	+3	+13	+12	+5	0	**+35**	0
lucene-core	25	0	84	65	1	50	**225**	1062
Hierarchy	+5	+6	+4	0	0	+2	**+17**	+295
Inter-class	+345	+118	+516	+710	+6	+46	**+1741**	+4268
solr	1	0	3	9	2	2	**17**	4253
Hierarchy	0	+1	0	0	0	0	**+1**	+14
Inter-class	0	0	0	+2	+1	0	**+3**	+689
Total,Intra-class	3973	0	4413	1630	1004	348	**11368**	61459
Additional,Hierarchy	15	76	147	68	2	17	**325**	2494
Additional,Inter-class	1946	29800	9826	899	5807	977	**49255**	145719

Mild In the intra-class search, 53% of the 11,368 issues, consisting of 4,413 clones of comment parts, and 1,630 clones of whole comments, are considered to be Mild severity issues by *RepliComment*. The same applies in the hierarchy search in 66%, and in inter-class search in 22% of the times, respectively.

This means that large proportions of problematic comment clones are considered to be due to poor information quality in the documentation. This is not surprising to us, as our initial hypothesis was that code comment clones are mostly due to lack of proper information rather than oblivious copy-and-paste errors.

Table 6.4. Quantitative results of the **field** comment clones reported by *RepliComment* on each analyzed project.

Project	Low	Mild	High	Tot. issues	Legit
elasticsearch	2	1	0	**3**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+19
hadoop-common	1	21	0	**22**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	+1	0	+1	+6
vertx-core	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	+2	+1	0	+3	+14
spring-core	6	0	0	**6**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+7
hadoop-hdfs	1	3	1	**5**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+4
log4j	0	3	0	**3**	0
Hierarchy	0	0	0	**0**	+2
Inter-class	+1	0	0	**0**	+65
guava	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+6
rxjava	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	+3
lucene-core	1	4	0	**5**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	4	0	+4	+10
solr	0	0	0	**0**	0
Hierarchy	0	0	0	**0**	0
Inter-class	0	0	0	**0**	0
Total, Intra-class	11	32	1	**44**	0
Additional, Hierarchy	0	0	0	**0**	+2
Additional, Inter-class	+3	+6	0	+9	+134

HIGH Finally, in the intra-class search, *RepliComment* reports that 12% of the 11,368 issues, consisting 1,004 cases of clones in comment parts and 348 cases of whole comment clones, are HIGH severity issues. In the hierarchy search this happens only for a small proportion of 6% of cases, and, in the inter-class search, of 14% of cases. Overall, 8155 cases over a total of 60,948 analyzed ones (13%) are considered to be HIGH severity issues. These are the issues that *RepliComment* considers to need an urgent fix.

Table 6.4 shows all clones that *RepliComment* reports for *field* comment clones. Since field comments have no tags, there is no distinction between comment parts and whole comment clones.

In the intra-class search, *RepliComment* reports a total of 44 field comment clones considered to be potential issues, while none is considered *legitimate* right away. In the hierarchy search, *RepliComment* reports no additional potentially harmful clones, while it flags only 2 additional *legitimate* clones. Finally, in the inter-class search, *RepliComment* reports 9 additional problematic clones, while it labels 134 additional ones as *legitimate*. The overall number of potential issues is 53:

LOW A total of 14 issues, hence 26% of the total, are considered of LOW severity.

MILD Most of the issues, *i.e.*, 38 (72% of the total), are considered to be of MILD severity, hence providing poor information.

HIGH Only a single issue is considered to be a HIGH severity one, and it is detected through an intra-class search.

Given the results of this experiment, we conclude that comment clones are prevalent even in popular Java projects. The results of the search with different scopes seem to show that *RepliComment* should better be used either with INTRA-CLASS or HIERARCHY scopes, as looking for comment clones with INTER-CLASS scope reports too many method comment clones to be analyzed by developers, despite the ability of *RepliComment* to filter out many legitimate cases.

RQ2: Accuracy of *RepliComment* at differentiating *legitimate* and *non-legitimate* clones We *manually* analyze some samples of the clones that *RepliComment* identifies as *legitimate* or not to establish the rate of false positives and false negatives. We first present the results regarding method comments, separating clones of comment parts and whole comment clones. We then proceed with the results of field comments.

Method comment clones

False positives We manually inspect all the entries in table 6.3 to ensure a fair sampling, and we remove duplicates to ensure that sampling catches the largest variety of comments. For this purpose, we consider a case to be a duplicate if the comment is exactly the same, but affects multiple method instances. This is likely to happen when developers write generic `@throws` comments such as *"on error"* for all the documented exceptions, for instance. Note that we draw this distinction for manual analysis, but in reality comment clones affecting multiple methods should all be addressed by developers.

Table 6.5. Clones of comment parts and whole comments after duplicate removal

	Comment part clones				Whole comment clones			
Project	Low-CP	Mild-CP	High-CP	Total	Low-WC	Mild-WC	High-WC	Total
elasticsearch	111	15	6	**132**	0	377	103	**480**
Hierarchy	+4	+2	0	**+6**	+7	+2	+6	**+15**
Inter-class	+461	+119	+33	**+613**	+2	+10	+503	**+515**
hadoop-common	34	34	13	**81**	0	0	0	**0**
Hierarchy	+2	0	0	**+2**	+15	0	+1	**+16**
Inter-class	+64	+221	+24	**+309**	+84	+3	+6	**+93**
vertx-core	27	15	14	**56**	0	13	6	**19**
Hierarchy	0	+2	0	**+2**	+1	0	+3	**+4**
Inter-class	+368	+13	+2	**+383**	+3	0	+1	**+4**
spring-core	46	20	12	**78**	0	46	20	**66**
Hierarchy	+1	0	+1	**+2**	0	+3	0	**+3**
Inter-class	+36	+5	+11	**+52**	0	+8	0	**+8**
hadoop-hdfs	23	28	7	**58**	0	13	11	**24**
Hierarchy	+1	+12	+1	**+14**	+11	0	+1	**+12**
Inter-class	+19	+895	+12	**+926**	+6	+10	+3	**+19**
log4j	1	1	1	**3**	0	15	3	**18**
Hierarchy	0	0	0	**0**	+2	0	0	**+2**
Inter-class	+16	+1	+1	**+18**	+6	+9	+4	**+19**
guava	57	24	9	**90**	0	132	48	**180**
Hierarchy	+2	+127	0	**+129**	+1	+1	+4	**+6**
Inter-class	+16	+7	+9	**+32**	+9	+39	+6	**+54**
rxjava	23	7	3	**33**	0	15	2	**17**
Hierarchy	0	0	0	**0**	0	0	0	**0**
Inter-class	+2	+13	+5	**+20**	+3	+1	0	**+4**
lucene-core	25	21	1	**47**	0	65	24	**89**
Hierarchy	+5	+4	0	**+9**	+6	+2	0	**+8**
Inter-class	+345	+516	+6	**+867**	+25	+6	+16	**+47**
solr	1	3	2	**6**	0	9	2	**11**
Hierarchy	0	0	0	**0**	+1	0	0	**+1**
Inter-class	0	0	+1	**+1**	0	+2	0	**+2**
Total,Intra-class	1690	2105	174	**3969**	182	781	773	**1736**
Additional,Hierarchy	15	147	2	**164**	44	8	15	**395**
Additional,Inter-class	1327	1790	104	**3221**	138	88	539	**7207**

Table 6.5 lists the unique comment clone instances after duplicates removal, reporting comment part clones and whole comment clones separately.

We sample entries of Table 6.5 by selecting *at least* 10% of the cases for each category (Low, Mild, High for intra-class, hierarchy and inter-class search). We sample 225 issues for intra-class, 63 for hierarchy, and 124 for inter-class search, for a total of 412 issues.

Regarding **intra-class search**, we find:

- For *comment parts*, we have 50 Mild issues and 30 High issues. We disagree on a total of 33 issues, 26 Mild and 7 High. In particular, all 7 High issues are false positives, so such clones are actually legitimate. Among the 26 Mild cases, 22 of them are false positives (the rest should have been considered HIGH severity issues). Thus *RepliComment* produces **29 false positives** for clones of comment parts.

- For *whole comment* clones, we have 70 MILD issues and 25 HIGH issues. We disagree on a total of **12** issues, 10 MILD and 2 HIGH, and **all of them are false positives**. A common reason why whole clones of comments can still be considered *legitimate* is that an API class is not supported anymore, and its method documentation states so (advising to avoid using the method and pointing to another class, *etc.*).
- In conclusion, *RepliComment* reports 45 false positives for a total of 175 samples for intra-class search, which suggests a precision of 74% of *RepliComment* in intra-class search.

Regarding **hierarchy search**, we have:

- For *comment parts*, we never disagree with *RepliComment* in the additional sampled 17 issues (15 MILD and 2 HIGH ones).
- For *whole comment* clones, we never disagree on the assessment made on 10 MILD, while we do disagree for 11 HIGH ones.
- In conclusion, *RepliComment* achieves a precision of 71% for hierarchy search.

Listing 6.4 show an example of a HIGH-severity comment part clone found while exploring a class hierarchy. The same clone was found during an intra-class search (see Listing 6.3): Bad clones existing in one class may be replicated in its subclasses, thus perpetuating the issue.

Listing 6.4. Hierarchy high-severity issue (*RepliComment* report)

```
———— Record #4 file:2020_JavadocClones_h_hadoop—hdfs.csv ————
In class: org.apache.hadoop.hdfs.util.LightWeightLinkedSet
And its superclass: org.apache.hadoop.hdfs.util.LightWeightHashSet

1) The comment you cloned:"(@return)first element"
seems more related to <T pollFirst()> than <List pollN(int n)>
```

Finally, for **inter-class search**, we have that:

- For *comment parts*, we disagree with 4 *RepliComment* assessments over a total of 31 (16 MILD and 15 HIGH).
- For *whole comment* clones, we disagree with 2 assessments over a total of the 65 (10 MILD and 55 HIGH) issues sampled.

- In conclusion, *RepliComment* reports 6 false positives over a total of 96 issues, achieving a precision of 94%.

As an example, consider Listing 6.5. The interesting fact is that the two different classes across which the whole comment was cloned are not in the same hierarchy, and in general have little in common: they do not even belong exactly to the same package.

Listing 6.5. Inter-class high-severity issue (*RepliComment* report)

```
——— Record #6 file:2020_JavadocClones_cf_hadoop—hdfs.csv ———
In class: org.apache.hadoop.hdfs.tools.offlineEditsViewer.XmlEditsVisitor
And class:
org.apache.hadoop.hdfs.tools.offlineImageViewer.TextWriterImageVisitor

You cloned the whole comment for methods
< XmlEditsVisitor(OutputStream out)> and
< TextWriterImageVisitor(String filename, boolean printToScreen)>

The comment you cloned:"(Whole)Create a processor that writes to the
 file  named and may or may not also output to the screen, as specified.
@param Name of file to write output to @param Mirror output to screen?"
seems more related to <TextWriterImageVisitor(String filename, boolean
printToScreen)> than  <XmlEditsVisitor(OutputStream out)>
```

False negatives Our heuristics could wrongly flag as *legitimate* some clones that actually represent real issues. Cases marked as *legitimate* are filtered out in the first phase, *i.e.*, they are not analyzed further. Thus, in the case of a false negative, the issue would never be revealed. It is hence important to check that false negatives are not pervasive.

RepliComment marks as *legitimate* the comment clones reported in Table 6.3. We do not distinguish between comment parts and whole comments because a whole comment clone can never be considered *legitimate*.

We randomly sample 20 cases for each project and each type of search. If the total number is less than 20 then we analyze all cases. We manually analyze each of the *572 comment clones* to check whether it should indeed be considered to be *legitimate* (*i.e.*, we agree with *RepliComment* heuristics) or *non-legitimate* (*i.e.*, it is a false negative).

Table 6.6. Total of clones considered legitimate by the heuristics

Project	Agree (legit)	Disagree (non-legit)	Precision
elasticsearch-6.1.1	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
hadoop-common-2.6.5	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
vertx-core-3.5.0	20	0	100%
HIERARCHY	19	1	95%
INTER-CLASS	20	0	100%
spring-core-5.0.2	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
hadoop-hdfs-2.6.5	19	1	95%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
log4j-1.2.17	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
guava-19.0	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
rxjava-1.3.5	20	0	100%
HIERARCHY-	-	-	-
INTER-CLASS	20	0	100%
lucene-core-7.2.1	20	0	100%
HIERARCHY	20	0	100%
INTER-CLASS	20	0	100%
solr-7.1.0	20	0	100%
HIERARCHY	14	0	100%
INTER-CLASS	20	0	100%
Total	572	2	99.7%

Table 6.6 shows that we disagree with the classification as *legitimate* in two comment clones over 572 randomly selected in total. This means that we find *only two false negatives in our random sampling*. In particular, one is a case of a very generic exception comment that *RepliComment*'s heuristics miss. The second is the case of parameters documented with the same name (for which a comment clone is tolerated), having however, different non-primitive types.

Field comments

False positives Since *RepliComment* reports a relatively low number of issues for field comments, namely 38 MILD and only one HIGH, we analyze them all. Most of the MILD severity issues, namely 21, are all from hadoop-common. These clones would probably be considered legitimate by developers, since the comment states: *"This constant is accessible by subclasses for historical purposes. If* *you don't know what it means then you don't need it."* Hence, we consider these instances to be false positives. We also flag as false positives 3 instances from hadoop-common: in this case, field names are not parsable correctly due to multiple words being merged into a single one (*e.g.*, DFS_DATATRANSFER_SERVER_VARIABLEWHITELIST_FILE). We agree with the remaining 14 MILD ones, as well as with the single HIGH severity issue (see Listing 6.6). This suggests a precision of 39%.

Listing 6.6. Only high-severity issue existing for field clones (*RepliComment* report)

```

———— Record #7 file:2020_JavadocClones_fields_hadoop—hdfs.csv ————
In class: org.apache.hadoop.hdfs.shortcircuit.ShortCircuitCache


1) The comment you cloned:"(Field)The executor service that runs the
cacheCleaner."
seems more related to <cleanerExecutor> than <releaserExecutor>
```

Listing 6.6 shows the only HIGH-severity issue *RepliComment* finds when exploring field clones, along with its assessment. The clone exists within the same class.

False negatives We sample 20 instances from the 136 total *legitimate* field-level clones, and we confirm that we do agree with all of *RepliComment*'s assessments.

This analysis shows that *RepliComment*'s heuristics can be trusted to filter out many legitimate comment clones, and the rate of false positives is acceptable for practical use.

RQ3: Improvement of Heuristics over RepliComment-V1 We assess how well new heuristics implemented in the clone detector filter out further false positives in *RepliComment* compared to RepliComment-V1. To compare the effectiveness of the heuristics, we take the intersection of comment clones that RepliComment-V1 and *RepliComment* identify, and we compare their classification results. Table 6.7 presents the percentage of clones that RepliComment-V1 and *RepliComment* report as non-legitimate. The ability to report *fewer* issues is positive given the fact that in section 6.2.6 we assessed that heuristics do not cause false negatives. The table highlights the following results:

- In half of the projects (marked in bold font) the decrease of clones marked as *non-legitimate* by the heuristics is significant, going from a minimum reduction of -7% (spring-core-5.0.2) to a maximum of -29% (vertx-core-3.5.0);

Table 6.7. Samples of clones marked as non-legitimate before and after new heuristics application

Project	Old heuristics	New Heuristics
elasticsearch-6.1.1	49%	29%
hadoop-common-2.6.5	10%	9%
vertx-core-3.5.0	35%	6%
spring-core-5.0.2	17%	10%
hadoop-hdfs-2.6.5	9%	17%
log4j-1.2.17	20%	20%
guava-19.0	31%	31%
rxjava-1.3.5	38%	24%
lucene-core-7.2.1	19%	18%
solr-7.1.0	16%	0.5%
Average	24%	16%

- In four projects the reduction was close to non-existent, which means that some false positives are potentially retained, but no new ones are introduced;
- In only one project (hadoop-common-2.6.5) did the number of clones marked as *non-legitimate* increase by +8% instead of diminishing, potentially leading to an increase in the number of false positives.

RQ4: Accuracy of *RepliComment* at Classifying *legitimate* Comment Clones
We manually evaluate *RepliComment*'s assessment for each entry in the samples to determine its accuracy at classifying HIGH, MILD and LOW clones. Results report if our manual evaluation agrees or disagrees with *RepliComment*'s assessment. If we disagree, it means that *RepliComment* assigns the wrong category to one case, for example reporting it as a MILD severity when it is actually a LOW one. Conversely, if we agree it means we would assign the same level of severity to the case.

Method-level analysis Overall, we manually inspect and assess *412* reported issues. Table 6.8 reports the analysis for clones of comment parts. Results show that:

- *RepliComment* is very effective at classifying both LOW (>80%) and HIGH (>70%) severity issues in all kinds of search (intra-class, hierarchy, inter-

Table 6.8. Manual analysis of *RepliComment* assessment for clones of Javadoc parts (summary, @param, @return or @throws)

Category		Sample	Agree	Disagree	Precision
INTRA-CLASS	Low-CP	50	42	8	84%
	Mild-CP	50	24	26	48%
	High-CP	30	23	7	77%
HIERARCHY	Low-CP	15	15	0	100%
	Mild-CP	15	15	0	100%
	High-CP	2	2	0	100%
INTER-CLASS	Low-CP	14	14	0	100%
	Mild-CP	16	16	0	100%
	High-CP	15	11	4	73%
Total		207	162	45	
Average precision					87%

class). This means *RepliComment* can highlight the most critical clones (copy-paste issues) that developers should focus on.

- On the other hand, *RepliComment* often fails at identifying MILD severity issues as such, since *RepliComment* analysis fails nearly half of the times during intra-class search. We carefully analyzed the wrong classifications to give an explanation to this discrepancy: it appears to be a problem of linguistic *semantics*. *RepliComment*, in the current implementation, is neither aware of synonyms nor particular developer jargon. For example, our manual analysis reveals that oftentimes developers refer to a primitive parameter (being it int, long, char, *etc.*) generically as *"the value"*. *RepliComment*'s bag of words representations do not map such an expression to any portion of the method signature, since typically parameters have a specific name and type that differ from *"value"*. Hence, the analysis concludes that the cloned comment does not relate enough either to the first method or to the second one, maybe because it is too generic. Unfortunately such cases are false positives (LOW severity). By tackling synonyms correctly, *RepliComment* would not report as an issue most of the wrongly classified cases.

Table 6.9 reports the analysis for clones of whole comments:

Table 6.9. Manual analysis of *RepliComment* assessment for whole Javadoc clones

Category		Sample	Agree	Disagree	Precision
INTRA-CLASS	Low-WC	0	0	0	0%
	Mild-WC	70	60	10	86%
	High-WC	25	23	2	92%
HIERARCHY	Low-WC	10	10	0	100%
	Mild-WC	10	10	0	100%
	High-WC	11	0	11	0%
INTER-CLASS	Low-WC	14	14	0	100%
	Mild-WC	10	10	0	100%
	High-WC	55	53	2	96%
Total		205	180	25	
Average precision					75%

Precision of *RepliComment* in classifying both MILD and HIGH severity issues in all kinds of search for whole comment clones tends to be very high (∼90%), except for hierarchy search. In general, if a whole comment is copied for an overloaded method, it most likely means that the developer simply forgot to document the difference in the parameters, which would be a MILD severity issue. On the other hand, if a whole comment is copied across methods that are not overloaded, something is likely to be off. We report a particular example of this in Listing 6.7:

Listing 6.7. RepliComment HIGH severity whole comment clone example

```
———— Record #519  file:2020_JavadocClones_elastic.csv ————
In class: org.elasticsearch.common.collect.ImmutableOpenMap
1) You cloned the whole comment for methods
<Iterator keysIt()> and
<Iterator valuesIt()>

This is not an overloading case. Check the differences among the two
methods and document them.

2) The comment you cloned:"(Whole)Returns a direct iterator over the
keys."
seems more related to <Iterator keysIt()> than <Iterator valuesIt()>
```

As for the hierarchy search, *RepliComment* misclassifies constructor comments. Overall, it reports a low number of HIGH severity issues, but unfortunately they all look like false positives. To properly tackle constructor comments, more advanced assessments may be needed.

Field-level analysis We analyze 14 LOW-severity issues, 38 MILD-severity issues and only one HIGH-severity issue. We consider correct all LOW-severity issues, which include 11 clones identified during intra-class search, and 3 additional clones identified during inter-class search. Regarding MILD-severity issues, we believe 24 are wrongly classified, since they should probably be labeled as LOW. We consider correct the only HIGH-severity issue coming from an intra-class analysis of hadoop-hdfs. This yields a precision of 100% for LOW and HIGH severity issues, and of 39% for MILD severity issues.

The results of this experiment show that *RepliComment* is effective at differentiating comment clones, so developers can effectively focus on the most critical ones first.

RQ5: Ability to Identify Cloned and Original Comments The ultimate goal of *RepliComment* is to support developers in pointing out *which* comment to fix, when the clone is due to a copy-and-paste error. In this section we evaluate how good *RepliComment* is at distinguishing the original and the cloned comment.

Method-level analysis

Intra-class clones To answer this question, we examine *RepliComment*'s assessment for the same 30 entries of HIGH-CP in Table 6.8, and the 25 HIGH-WC entries in Table 6.9.

- For HIGH-CP, we exclude the seven entries for which we disagree, since according to our manual inspection they are not real copy-paste issues. Our manual analysis confirms the correctness of *RepliComment* in pointing out the comment that was cloned for all the remaining 23 cases out of 30. Thus, the tool correctly suggests to the developer which method needs a documentation fix with a precision of 77%.

- Similarly, for HIGH-WC, we exclude the two entries for which we disagree. Our manual analysis reveals that we are unsure about three suggestions out of 23, and we do not agree with one out of 23 because we can infer that the two methods are actually equivalent in behavior (*RepliComment* in such a case should suggest that each of the methods is similarly related to the comment, meaning that neither of them appears better than the other).

We completely agree with the suggestions for the remaining 19 out of 23 cases, which yields a precision of 83% in suggesting the right fix to the developer.

Hierarchy clones We examine *RepliComment*'s assessment for the two entries of HIGH-CP in Table 6.8 and the eleven HIGH-WC entries in Table 6.9.

- For HIGH-CP, we do agree with both *RepliComment*'s picks. It is interesting to note that one is an example already found via intra-class analysis of hadoop-hdfs, which was replicated in the hierarchy.
- We exclude HIGH-WC, since we disagreed with all of their assessments.

Inter-class clones We examine *RepliComment*'s assessment for the 15 entries of HIGH-CP in Table 6.8 and the 55 HIGH-WC entries in Table 6.9.

- For HIGH-CP, we exclude the four instances for which we disagree with *RepliComment*. We do agree with all the remaining ones.
- Similarly, for HIGH-WC, we exclude two instances. As for the remaining 53 ones, it is worth noting that 49 of them seem to arise from the same elastic patterns of documentation. For example, the developers tend to write comments like *"Sets the minimum score below which docs will be filtered out"* both for actual setter methods and methods which are not actually setters, or at least, methods which perform some extra operations beside setting a value. Hence, *RepliComment* is justified in picking the setter method as the right owner of the comment. That said, those are probably voluntary habits accepted by the project's developers, and not actual copy-and-paste slips. Excluding such instances, we are left with four, which do look like oblivious copy-and-paste mistakes and for which we agree with *RepliComment*'s pick.

Field-level analysis As for field-level analysis, we only have a single instance of HIGH severity issue, for which we confirm the assessment of *RepliComment*.

This experiment confirms that *RepliComment* can actually support developers in highlighting which comments are the original ones and which ones are copied, and therefore should be fixed.

RQ6: Correlation with code clones Comment clones may be the result of copy-and-paste practice on entire method implementations. If this was the case, comment clones would appear only when their corresponding method implementations are clones as well. To understand if this is the case, we compare clone issues reported by *RepliComment* and by NiCad 2.6 code clone detector Cordy and Roy [2011]. We follow this comparison protocol for each of the projects:

- We extract class-qualified signatures of methods for which *RepliComment* reports HIGH severity issues in Javadoc comments for both comment parts and whole comments in all three analysis modes (within the same file, within the class hierarchy, and across all classes of the project);
- We extract class-qualified signatures of methods which NiCad reports as type III (near-miss blind renamed) clones with first over 70% and then only with exactly 100% similarity using the default configuration (clones sized between 10 and 2500 LOC, the near-miss difference threshold set to at most 30% different lines); We use the default code clone similarity threshold of NiCad clone detector as a baseline in our experiments. The difference of 30% is already quite liberal in the context of code clones, and previous studies on human judgment of code clones suggest that it is not trivial to agree on when a clone becomes a legitimate method with just a similar structure Kapser and Godfrey [2006].
- We pipe GNU core utilities sort and comm to sort outputs of both tools and compare them line by line, respectively.

Additionally, we collect the statistics of how many methods reported as code clones by NiCad have Javadoc comments. Table 6.10 presents such data both for exact and non-exact code clones.

We can see from the statistics collected that code clones seem to be fairly well-documented, with a minimum percentage of commented methods of 15% in elasticsearch and a maximum percentage of 92% in hadoop-hdfs. The remaining eight projects can be further split into two groups, where in the first group the rate of documented code clones is around 30%, and in the other group this rate is closer to 60%.

However, across the 10 projects we have detected only a few cases for which both *RepliComment* and NiCad tools reported clone issues in the same methods.

RepliComment reported whole comment clones in the same file, the first clone tuple consisting of two methods in the rxjava project, and the second clone tuple of three methods in the lucene project, where both clone tuples consist of exact code clones (code similarity 100%).

Table 6.10. Code clones statistics

Project	Code clones exact			Code clones 70%+ similar		
	All	Commented	Matching	All	Commented	Matching
elasticsearch-6.1.1	153	43 (28%)	0	1248	193 (15%)	**29**
hadoop-common-2.6.5	155	95 (61%)	0	1047	364 (34%)	0
vertx-core-3.5.0	23	6 (28%)	0	202	56 (27%)	0
spring-core-5.0.2	22	17 (77%)	0	143	89 (62%)	0
hadoop-hdfs-2.6.5	422	389 (92%)	0	5764	2093 (36%)	0
log4j-1.2.17	18	10 (55%)	0	90	40 (44%)	0
guava-19.0	84	37 (44%)	0	417	224 (53%)	**3**
rxjava-1.3.5	35	10 (28%)	**2**	332	102 (30%)	**2**
lucene-core-7.2.1	73	24 (32%)	**3**	592	175 (29%)	**3**
solr-7.1.0	129	25 (19%)	0	528	84 (16%)	0

Additionally, when lowering code clone similarity threshold to 70% *RepliComment* and NiCad report matching issues in two additional projects: in the elasticsearch project 29 code clones distributed over 7 different clone classes with in-class similarity varying from 70% to 91% are also reported by *RepliComment* as methods with inter-class whole comment clones, and in the guava project 3 code clones distributed over 1 clone class with in-class similarity of 72% are also reported by *RepliComment* as methods with intra-class comment part clones.

Our findings indicate that critical comment clones issues cannot necessarily be well-detected by code clone detection tools, as in most cases the clones in comments were considered to be legitimate by *RepliComment*.

Chapter 7

Conclusions

This thesis proposes a framework to automatically generate cost-effective test oracles from informal software documentation in natural language, aiming to overcome the core limitations of state-of-the-art approaches. Current approaches to automatically generate test oracles either produce oracles with limited capabilities, as implicit and regression oracles, or rely on seldom available artifacts, as oracles automatically generated from formal specification.

This thesis takes advantage of the observation that most professional software systems are commonly available with documentation in the form of text enriched with jargon expressions, such as code annotations, comments, and wikis. The thesis defines approaches to effectively generate oracles in the form of executable assertions from the information available as code annotations, after pruning common human mistakes. In this way, our approach automatically generates semantically relevant oracles without requiring expensive human effort.

7.1 Contributions

The thesis contributes to the state of the art by defining both two approaches, *UpDoc* and *RepliComment*, to detect inconsistencies in informal specification, and three approaches, *JDoctor*, *MeMo* and *CaMeMa*, to automatically derive prescriptive and descriptive test oracles. *UpDoc* automatically detects fine-grained inconsistencies between code and Javadoc specification. *RepliComment* integrates *UpDoc* to detect harmful clones in the documentation and suggest correct fixes to developers. *JDoctor* derives oracles from semi-structured Javadoc specification. *MeMo* and *CaMeMa* generate oracles from unstructured Javadoc summaries.

UpDoc *UpDoc automatically detects inconsistencies* between code and documentation Stulova et al. [2020]. *UpDoc* produces an internal mapping between the code of Java methods and the corresponding Javadoc documentation represented as bag of words. It then computes the similarity between code and documentation, by means of cosine similarity and Word Mover's Distance. *UpDoc* operates at a fine-grained level, relying on single AST nodes. The technique generalizes to any programming language and documentation format, and does not require either training or extra human effort. Our experimental evaluation on 67 comment changes from the dataset of Wen et al. [2019] indicates that *UpDoc* 's mapping between methods and documentation comments accurately reflects inconsistencies in 90% of the cases.

RepliComment *RepliComment* is the first approach that specifically focuses on harmful documentation clones. We define harmful documentation clones, show that they occur even in widely-used, well-developed and well-documented systems Blasi and Gorla [2018], and propose *RepliComment*, an approach that assesses the severity of harmful clones, and suggests suitable fixes for high severity clones to the developers Blasi et al. [2021b]. Our study indicates that documentation clones are not correlated to code clones, and highlights the need of different approaches to tackle the two issues.

JDoctor *JDoctor* Blasi et al. [2018] generates executable assertions from the Javadoc documentation of Java methods. *JDoctor* extracts and analyzes three classes of Javadoc tags: *@param* tags that express properties about the preconditions of the documented method; *@return* tags that express properties about the normal postcondition of the method; *@throws* and *@exception* tags that express the exceptional postconditions of the method. *JDoctor* translates the information present in these tags as natural language text into executable Java assertions. Our experimental evaluation indicates that *JDoctor* translates informal textual annotations into executable specifications with high precision and recall, largely improving over state-of-the-art approaches. When applied to automated testing, such as by integrating the specifications in Randoop Pacheco et al. [2007], *JDoctor* assertions reduce false positives and false negatives by correctly classifying test cases.

MeMo *MeMo* Blasi et al. [2021a] identifies metamorphic relations in Javadoc method *summaries*, a kind of unstructured informal specification, and generates metamorphic oracles. Our experimental evaluation indicates that *MeMo* accurately identifies metamorphic properties, and nicely complements state-of-the-art techniques such as SBES Mattavelli et al. [2015]. When used as test oracles jointly with both regression and developers' assertions, *MeMo*'s assertions improve the mutation score.

CaMeMa *CaMeMa* identifies both prescriptive and descriptive temporal constraints informally described in Javadoc method summaries, and produces temporal formulas in a machine-readable format. *CaMeMa* produces temporal specifications with good precision and recall. When integrated with Randoop Pacheco et al. [2007], *CaMeMa* 's specifications enrich generated error and regression test suites by revealing false alarms and validating expected exceptions.

7.2 Open Research Directions

By showing the possibility of automatically generating test oracles through the machine translation of text from Javadoc comments, this thesis opens new research directions towards the further exploitation of natural language artifacts to improve software testing. We identify three main research directions open with this thesis: *Informal annotations beyond Javadoc*, *Testing artifacts beyond oracles*, *Sentiment analysis beyond NLP*.

Informal annotations beyond Javadoc In this thesis, we show how to generate different test oracles from Javadoc natural language specifications both in semi-structured and unstructured form. There are many more natural language artifacts documenting software, such as wikis, tutorials, and the like. Such documentation refers to software systems at many different granularity levels, and describes different types of properties. The results presented in this thesis indicate enormous opportunities to adapt the approaches here defined, and devise new ones to automatically extract useful information to improve testing activities.

Testing artifacts beyond oracles In this thesis, we show how to automatically generate executable assertions, metamorphic relations and temporal specifications from natural language annotations to be used as test oracles. Our evaluation of *JDoctor*, *MeMo* and *CaMeMa* in the context of automated test case generation indicates that natural language annotations provide information that can be used to automate the testing process beyond test oracles generation.

Sentiment analysis beyond NLP Our preliminary observations about Javadoc class summaries suggest that analyzing the sentiment of information written by developers may help us better classify the conveyed information. For example, a documented class usage may be discouraged, hence enforcing prescriptive properties. As shown with *CaMeMa*, this is different from documenting purely descriptive usages. The potential of sentiment analysis in this direction are worth investigating.

Bibliography

Zahra Shakeri Hossein Abad, Vincenzo Gervasi, Didar Zowghi, and Behrouz H Far. Supporting analysts by dynamic extraction and classification of requirements-related knowledge. In *Proceedings of the International Conference on Software Engineering*, pages 442–453. IEEE, 2019.

Surafel Lemma Abebe and Paolo Tonella. Natural language parsing of program element names for concept extraction. In *Proceedings of the International Conference on Program Comprehension*, pages 156–159. IEEE, 2010.

Emad Aghajani, Csaba Nagy, Olga Lucero Vega-Márquez, Mario Linares-Vásquez, Laura Moreno, Gabriele Bavota, and Michele Lanza. Software documentation issues unveiled. In *Proceedings of the International Conference on Software Engineering*, pages 1199–1210. IEEE, 2019.

Nauman Bin Ali, Emelie Engström, Masoumeh Taromirad, Mohammad Reza Mousavi, Nasir Mehmood Minhas, Daniel Helgesson, Sebastian Kunze, and Mahsa Varshosaz. On the search for industry-relevant regression testing research. *Empirical Software Engineering*, 24(4):2020–2055, 2019.

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

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

Wladimir Araujo, Lionel C. Briand, and Yvan Labiche. Enabling the runtime assertion checking of concurrent contracts for the java modeling language. In *Proceedings of the International Conference on Software Engineering*, ICSE ’11, pages 786–795, 2011.

Venera Arnaoudova, Massimiliano Di Penta, and Giuliano Antoniol. Linguistic antipatterns: what they are and how developers perceive them. *Empirical Software Engineering*, 21:104–158, February 2016.

Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

Arianna Blasi and Alessandra Gorla. Replicomment: Identifying clones in code comments. In *Proceedings of the International Conference on Program Comprehension*, ICPC'18. ACM, 2018.

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

Arianna Blasi, Alessandra Gorla, Michael D Ernst, Mauro Pezze, and Antonio Carzaniga. Memo: Automatically identifying metamorphic relations in javadoc comments for test automation. *Journal of Systems and Software*, 181:111041, 2021a.

Arianna Blasi, Nataliia Stulova, Alessandra Gorla, and Oscar Nierstrasz. Replicomment: identifying clones in code comments. *Journal of Systems and Software*, 182:111069, 2021b.

Bobby R Bruce, Justyna Petke, Mark Harman, and Earl T Barr. Approximate oracles and synergy in software energy search spaces. *IEEE Transactions on Software Engineering*, 45(11):1150–1169, 2018.

Antonio Carzaniga, Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè. Cross-checking oracles from intrinsic software redundancy. In *Proceedings of the International Conference on Software Engineering*, ICSE '14, pages 931–942. ACM, 2014.

Hui Chen, John Coogle, and Kostadin Damevski. Modeling stack overflow tags and topics as a hierarchy of concepts. *Journal of Systems and Software*, 156: 283–299, 2019.

Tsong Y. Chen, Shing-Chi Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. Technical report, Department of Computer Science, Hong Kong University of Science and Technology, 1998.

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

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

Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '02, pages 231–255, 2002.

James R Cordy and Chanchal K Roy. The nicad clone detector. In *Proceedings of the International Conference on Program Comprehension*, pages 219–220. IEEE Computer Society, 2011.

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

Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proceedings of the International Conference on Software Engineering*, ICSE '05, pages 422–431. IEEE Computer Society, 2005.

Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017.

Martin D. Davis and Elaine J. Weyuker. Pseudo-oracles for non-testable programs. In *Proceedings of the ACM '81 Conference*, ACM '81, pages 254–257. ACM, 1981.

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

Luciano Del Corro and Rainer Gemulla. Clausie: Clause-based open information extraction. In *Proceedings of the International Conference on World Wide Web*, WWW '13, pages 355–366. ACM, 2013.

Andrea Di Sorbo, Sebastiano Panichella, Carol V Alexandru, Junji Shimagaki, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. What would users change in my app? summarizing app reviews for recommending software changes. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 499–510. ACM, 2016.

Andrea Di Sorbo, Sebastiano Panichella, Corrado Aaron Visaggio, Massimiliano Di Penta, Gerardo Canfora, and Harald C Gall. Exploiting natural language structures in software informal documentation. *IEEE Transactions on Software Engineering*, 2019.

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

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

Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the International Conference on Automated Software Engineering*, pages 313–324, 2014.

Beat Fluri, Michael Wursch, and Harald C Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)*, pages 70–79. IEEE, 2007.

Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM, 2011.

Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

Susumu Fujiwara, Gregor von Bochmann, Ferhat Khendek, Mokhtar Amalou, and Abderrazak Ghedamsi. Test selection based on finite state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Transactions on Programming Languages and Systems*, 3(3):211–223, 1981.

Gregory Gay, Sanjai Rayadurgam, and Mats PE Heimdahl. Automated steering of model-based test oracles to admit real program behaviors. *IEEE Transactions on Software Engineering*, 43(6):531–555, 2016.

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

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

Arnaud Gotlieb. Exploiting symmetries to test programs. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '03, pages 365–374. IEEE Computer Society, 2003.

Samir Gupta, Sana Malik, Lori Pollock, and K Vijay-Shanker. Part-of-speech tagging of program identifiers for improved text-based software engineering tools. In *Proceedings of the International Conference on Program Comprehension*, pages 3–12. IEEE Computer Society, 2013.

Tobias Hey, Fei Chen, Sebastian Weigelt, and Walter F Tichy. Improving traceability link recovery using fine-grained requirements-to-code relations. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 12–22. IEEE, 2021.

Ruihong Huang, Ignacio Cases, Dan Jurafsky, Cleo Condoravdi, and Ellen Riloff. Distinguishing past, on-going, and future events: The EventStatus corpus. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2016.

Reyhaneh Jabbarvand, Forough Mehralian, and Sam Malek. Automated construction of energy test oracles for android. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 927–938, 2020.

Gunel Jahangirova, Andrea Stocco, and Paolo Tonella. Quality metrics and oracles for autonomous vehicles testing. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 194–204. IEEE, 2021.

René Just, Franz Schweiggert, and Gregory M. Kapfhammer. 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, pages 612–615. IEEE Computer Society, 2011.

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

Upulee Kanewala and James M. Bieman. Using machine learning techniques to detect metamorphic relations for programs without test oracles. In *ISSRE*, ISSRE '13, pages 1–10. IEEE Computer Society, 2013.

Cory J Kapser and Michael W Godfrey. Supporting the analysis of clones in software systems. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(2):61–82, 2006.

Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. Effective and efficient api misuse detection via exception propagation and search-based testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 192–203. ACM, 2019.

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

Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. Improving api caveats accessibility by mining api caveats knowledge graph. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 183–193. IEEE, 2018.

Bin Lin, Fiorella Zampetti, Gabriele Bavota, Massimiliano Di Penta, and Michele Lanza. Pattern-based mining of opinions in q&a websites. In *Proceedings of the International Conference on Software Engineering*, pages 548–559. IEEE, 2019.

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

Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *Proceedings of the International Conference on Automated Software Engineering*, pages 585–597. IEEE Computer Society, 2020.

Annie Louis, Santanu Kumar Dash, Earl T Barr, Michael D Ernst, and Charles Sutton. Where should i comment my code? a dataset and model for predicting locations that need comments. In *Proceedings of the 42nd International Conference on Software Engineering (New Ideas and Emerging Results)(ICSE NIER 2020)*. ACM, 2020.

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

Phu X. Mai, Fabrizio Pastore, Arda Goknil, and Lionel C. Briand. A natural language programming approach for requirements-based security testing. In *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '18, pages 58–69. IEEE Computer Society, 2018.

Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. Nl2type: inferring javascript function types from natural language information. In *Proceedings of the International Conference on Software Engineering*, pages 304–315. IEEE Computer Society, 2019.

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

Andrea Mattavelli, Alberto Goffi, and Alessandra Gorla. Synthesis of equivalent method calls in Guava. In *Proceedings of the 7th International Symposium on Search-Based Software Engineering*, SSBSE '15, pages 248–254. Springer, 2015.

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

Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1st edition, 1988.

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

Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? an empirical study on the directives of api documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

Kawser Wazed Nafi, Tonny Shekha Kar, Banani Roy, Chanchal K Roy, and Kevin A Schneider. Clcdsa: cross language code clone detection using syntactical features and api documentation. In *Proceedings of the International Conference on Automated Software Engineering*, pages 1026–1037. IEEE Computer Society, 2019.

Mahdi Nejadgholi and Jinqiu Yang. A study of oracle approximations in testing deep learning libraries. In *Proceedings of the International Conference on Automated Software Engineering*, pages 785–796. IEEE, 2019.

C. Newman, M. J. Decker, R. S. AlSuhaibani, A. Peruma, D. Kaushik, and E. Hill. An open dataset of abbreviations and expansions. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 280–280, 2019.

Pengyu Nie, Rishabh Rai, Junyi Jessy Li, Sarfraz Khurshid, Raymond J Mooney, and Milos Gligoric. A framework for writing trigger-action todo comments in executable format. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 385–396, 2019.

Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '05, pages 504–527. Springer Berlin Heidelberg, 2005.

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

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

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

Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A Visaggio, Gerardo Canfora, and Harald C Gall. How can i improve my app? classifying user reviews for software maintenance and evolution. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 281–290. IEEE Computer Society, 2015.

Sebastiano Panichella, Andrea Di Sorbo, Emitza Guzman, Corrado A. Visaggio, Gerardo Canfora, and Harald C. Gall. Ardoc: App reviews development oriented classifier. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 1023–1027. ACM, 2016.

Sheena Panthaplackel, Pengyu Nie, Milos Gligoric, Junyi Jessy Li, and Raymond Mooney. Learning to update natural language comments based on code changes. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, pages 1853–1868, 2020.

Sheena Panthaplackel, Junyi Jessy Li, Milos Gligoric, and Raymond J Mooney. Deep just-in-time inconsistency detection between comments and source code. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 427–435, 2021.

Profir-Petru Partachi, Santanu Dash, Christoph Treude, and Earl T Barr. Posit: Simultaneously tagging natural and programming languages. In *Proceedings of the International Conference on Software Engineering*, 2019.

Timo Pawelka and Elmar Juergens. Is this code written in english? a study of the natural language of comments and identifiers in practice. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 401–410. IEEE, 2015.

Ivan Porres and Irum Rauf. From nondeterministic uml protocol statemachines to class contracts. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, pages 107–116. IEEE, 2010.

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

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

Mohammad Masudur Rahman and Chanchal Roy. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 473–484. IEEE, 2018.

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

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

Inderjot Kaur Ratol and Martin P Robillard. Detecting fragile comments. In *Proceedings of the International Conference on Automated Software Engineering*, pages 112–122. IEEE Computer Society, 2017.

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

Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of Computing TR*, 541(115):64–68, 2007.

Tomohiro Sakaguchi, Daisuke Kawahara, and Sadao Kurohashi. Comprehensive annotation of various types of temporal information on the time axis. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation*. European Language Resources Association (ELRA), 2018.

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

Christian Schwarzl and Bernhard Peischl. Generation of executable test cases based on behavioral uml system models. In *Proceedings of the International Workshop on Automation of Software Test*, pages 31–34, 2010.

Lin Shi, Mingyang Li, Mingzhe Xing, Yawen Wang, Qing Wang, Xinhua Peng, Weimin Liao, Guizhen Pi, and Haiqing Wang. Learning to extract transaction function from requirements: an industrial case on financial software. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1444–1454, 2020.

Nataliia Stulova, Arianna Blasi, Alessandra Gorla, and Oscar Nierstrasz. Towards detecting inconsistent comments in java source code automatically. In *International Working Conference on Source Code Analysis and Manipulation*, 2020.

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

Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tComment: Testing Javadoc comments to detect comment-code inconsistencies. In *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, pages 260–269. IEEE Computer Society, 2012.

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

Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. MSeqGen: Object-oriented unit-test generation via mining source code. In *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202. ACM, 2009.

Javier Troya, Sergio Segura, and Antonio Ruiz-Cortés. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software*, 136:188–208, 2018.

Chunhui Wang, Fabrizio Pastore, Arda Goknil, Lionel Briand, and Zohaib Iqbal. Automatic generation of system test cases from use case specifications. In *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, pages 385–396. ACM, 2015.

Fengcai Wen, Csaba Nagy, Gabriele Bavota, and Michele Lanza. A large-scale empirical study on code-comment inconsistencies. In *Proceedings of the International Conference on Program Comprehension*, pages 53–64. IEEE, 2019.

Elaine J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4):465–470, 1982.

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

Xusheng Xiao, Amit Paradkar, Suresh Thummalapenta, and Tao Xie. Automated extraction of security policies from natural-language software documents. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 1–11, 2012.

Juan Zhai, Yu Shi, Minxue Pan, Guian Zhou, Yongxiang Liu, Chunrong Fang, Shiqing Ma, Lin Tan, and Xiangyu Zhang. C2s: translating natural language comments to formal program specifications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 25–37, 2020.

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

Hao Zhong, Lu Zhang, Tao Xie, and Hong Mei. Inferring resource specifications from natural language API documentation. In *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, pages 307–318. IEEE Computer Society, 2009.

Celal Ziftci and Diego Cavalcanti. De-flake your tests: Automatically locating root causes of flaky tests in code at google. In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pages 736–745. IEEE Computer Society, 2020.