Automating Test Oracles Generation

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
Alberto Goffi

under the supervision of
Prof. Mauro Pezzè

January 2018

Dissertation Committee

Dissertation accepted on 23 January 2018

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Alberto Goffi
Lugano, 23 January 2018

To all the people who made this thesis possible

Abstract

Software systems play a more and more important role in our everyday life. Many relevant human activities nowadays involve the execution of a piece of software. Software has to be reliable to deliver the expected behavior, and assessing the quality of software is of primary importance to reduce the risk of runtime errors. Software testing is the most common quality assessing technique for software. Testing consists in running the system under test (SUT) on a finite set of inputs, and checking the correctness of the results. Thoroughly testing a software system is expensive and requires a lot of manual work to define test inputs (stimuli used to trigger different software behaviors) and test oracles (the decision procedures checking the correctness of the results).

Researchers have addressed the cost of testing by proposing techniques to automatically generate test inputs. While the generation of test inputs is well supported, there is no way to generate cost-effective test oracles: Existing techniques to produce test oracles are either too expensive to be applied in practice, or produce oracles with limited effectiveness that can only identify blatant failures like system crashes.

Our intuition is that cost-effective test oracles can be generated using information produced as a byproduct of the normal development activities. The goal of this thesis is to create test oracles that can detect faults leading to semantic and non-trivial errors, and that are characterized by a reasonable generation cost.

We propose two ways to generate test oracles, one derives oracles from the software redundancy and the other from the natural language comments that document the source code of software systems.

We present a technique that exploits redundant sequences of method calls encoding the software redundancy to automatically generate test oracles named cross-checking oracles (CCOracles). We describe how CCOracles are automatically generated, deployed, and executed. We prove the effectiveness of CCOracles by measuring their fault-finding effectiveness when combined with both automatically generated and hand-written test inputs.

We also present Toradocu, a technique that derives executable specifications from Javadoc comments of Java constructors and methods. From such specifications, Toradocu generates test oracles that are then deployed into existing test suites to assess the outputs of given test inputs. We empirically evaluate Toradocu, showing that Toradocu accurately translates Javadoc comments into procedure specifications. We also show that Toradocu oracles effectively identify semantic faults in the SUT.

CCOracles and Toradocu oracles stem from *independent* information sources and are *complementary* in the sense that they check different aspects of the SUTs.

Contents

Figures

Tables

Acronyms

CCOracle cross-checking oracle. v, ix, xi, 3, 4, 13, 15–19, 23–32, 63–67

SUT system under test. v, 1–11, 13, 15, 16, 18, 25, 31, 39, 49, 55–58, 63–67

Chapter 1

Introduction

Software systems play a central role in our daily life. Software virtually affects every human activity: working, driving, watching movies, taking pictures, communicating with others, and so on. Over time, software systems are becoming more and more complex, and show more and more complex behaviors that have to be verified. The cost of checking the software behavior can amount up to the 75% of the total development cost [6, 39, 40].

Software verification is an activity that aims to identify mismatches between the *expected* and the *actual* behavior of software systems, and is an essential part of modern development processes. Software verification comprises software testing and formal verification. Formal verification verifies that a system under test (SUT) is correct with respect to some specifications for *any* possible input. Instead, software testing samples the behavior of a SUT and checks the correctness of the *finite set* of sample executions. IEEE defines testing as

> "the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain" [9].

Although formal verification and testing share the same goal, their applicability largely differs. Formal verification is expensive because it requires formal specifications of the properties under test, and those specifications are seldom available. There are several reasons for this: To write formal specifications developers must use unfamiliar languages typically on a different abstraction level of the programming languages they know and use everyday. While there is a plethora of tools supporting developers in programming, there is way less support in writing and debugging formal specifications. Formal verification applies only where its benefit counterbalances its cost, for example in critical software systems and, in general, where the developer craves or needs strong guarantees on the software behavior.

In industry the maturity level of testing activities is higher than formal verification activities, and testing is now well integrated in software development processes, to the point that testing can become the pillar around which a software system is built like in test-driven development (TDD). The central artifact of software testing is the test case. Informally, a test case is composed of a test input and a test oracle. Test input is a set of stimuli—inputs and execution conditions for the software under test. Test oracle is a mechanism that applies a pass/fail criterion to software executions [4, 79]. Test suite (i.e., a collection of test cases) and oracle generation are time-consuming and expensive activity, and automation is a promising research direction to reduce the cost and improve the effectiveness of software testing. Test automation reduce the cost of testing by automating several testing activities (e.g., test case execution, test suite management, regression testing, test case generation). While many testing activities are supported by mature tools and the generation of test inputs is becoming more and more automated, automatically generating effective test oracles is still an open problem. In a recent survey of the literature on test oracles, Barr and colleagues affirm that

> "compared to many aspects of test automation, *the problem of automating the test oracle has received significantly less attention, and remains comparatively less well-solved*. This current open problem represents a significant bottleneck that inhibits greater test automation and uptake of automated testing methods and tools more widely" [5].

After an empirical evaluation of automated oracles, Nguyen and colleagues conclude that

> "the high false positive rates of the existing automated oracles make them cost effective only when manual oracle definition costs are very high (more than 30 times higher, in our rough estimate), as compared to the manual assessment of a failed test case. In practice, *this might prevent any industrial adoption of automated oracles*, unless their false positive rate is dramatically reduced" [69].

Automated oracles are especially important in the context of automatic testing: automatic test case generators produce hundreds, thousands of test cases for each software module under test. Without a cost-effective oracle, developers have to inspect each generated test case, and manually define oracles, for instance assertions, to check the results. The cost of such activity represents a major bottleneck for the adoption of automatic test case generators.

This thesis addresses the problem of *generating cost-effective test oracles*. The cost-effectiveness of a test oracle largely depends on the information used to generate the oracle. The more the available information is formal and complete, the more the oracle is effective and costly. The less the available information is formal and complete, the less the generated oracle is effective and expensive. Finding a good trade-off between cost and effectiveness of test oracles is a major challenge and is the ultimate goal of this thesis.

To be reasonably inexpensive oracles must be generated automatically, from information sources that are usually available as byproduct of the software development and/or inexpensive to produce. To be effective test oracles must be able to identify failures due to semantic faults, going beyond the mere crash of the SUT. In this PhD research, we identify two information sources that are commonly available and that we exploit to generate test oracles: intrinsic software redundancy and natural language comments embedded in the source code of the SUT.

1.1 Research Hypothesis and Contributions

The overall research hypothesis of this PhD thesis is: *Cost-effective test oracles can be automatically generated from different and heterogeneous information sources that are present in software systems as a byproduct of the normal development activities.*

State-of-the-art techniques generate either inexpensive application-independent oracles that miss many failures, or effective but expensive oracles from ad-hoc artifacts that must be produced specifically for the oracle generation. This thesis proposes approaches that use information already present in software systems as a byproduct of the normal software development to produce effective oracles without incurring a high cost.

This dissertation advances the state of the art in test oracle generation by proposing techniques to automatically generate test oracles from information sources already present in software systems, namely intrinsic redundancy and code comments:

Oracles from Intrinsic Redundancy The thesis proposes a technique to generate test oracles from redundancy that is emerging in modern software systems, and that we refer to as *intrinsic software redundancy* [16, 61]. Intrinsic software redundancy has been identified by Carzaniga et al., who exploited this redundancy to design self-healing Web applications [15] and standard Java applications [13]. We demonstrate that intrinsic software redundancy can be mined automatically from software systems and can be used to automatically generate test oracles that we refer to as cross-checking oracles (CCOracles). In this thesis, we propose a technique that generates CCOracles from intrinsic redundancy that can be automatically mined from the SUT, and that automatically deploys the generated oracles into existing test cases. We show that CCOracless are effective in identifying faults when combined with both manually defined and automatically generated test inputs.

Oracles from Code Comments The thesis proposes a technique to generate test oracles from natural language comments embedded in the source code of the SUT. Code comments are pervasive in modern software systems. So far they have been only marginally exploited to generate oracles, even though they contain a substantial amount of information about the expected behavior of the SUT. We propose an approach to translate code comments into procedure specifications that comprise pre- and post- conditions that, in turn, we transform into test oracles. We demonstrate the effectiveness of the approach, both in terms of accuracy of the generated specifications and in terms of effectiveness of the generated oracles.

The two techniques we propose are *different* and *independent*. They are different because they rely on different information sources to generate test oracles. Intrinsic redundancy and comments can be available in different amount or even at different time, so that one technique may be applicable when the other is not. They are also independent in the sense that they produce complementary oracles that can be used together to thoroughly test the SUT.

1.2 Research Methods

Research methods can be categorized in two main classes: quantitative and qualitative methods. In this thesis we use both quantitative and qualitative investigations to validate the research hypothesis. In particular, we support the cost-effectiveness of the oracle-generating techniques that we propose by:

- *Quantitatively* evaluating the effectiveness of the generated oracles. In particular, we measure the effectiveness as the number of failures reported by the oracles.
- *Quantitatively* measuring the accuracy of the generated oracles. The accuracy of an oracle consists in the number of false (spurious) alarms reported by the oracle.
- *Qualitatively* discuss the cost of the approaches. A quantitative study on the cost would be hard to perform and even harder to generalize. The cost of an oracle generating technique is strongly affected be the surrounding environment in which the technique is applied:

development process, skill set of the people using the technique, nature and peculiarities of the SUT, company established practices, and so on. For this reason we opted for a qualitative analysis of the cost.

1.3 Thesis Structure

The thesis is organized as follows:

- Chapter 2 describes the most relevant software engineering approaches to generate test oracles. We survey techniques that generates oracles from different kinds of information, with different levels of cost-effectiveness.
- Chapter 3 presents CCOracles. It describes the information source from which CCOracles are generated, the intrinsic software redundancy; it shows how the intrinsic redundancy can be automatically identified and used to generate CCOracles; it presents the results of a thorough empirical evaluation of CCOracles.
- Chapter 4 describes how we generate test oracles from code comments. It describes how we exploit natural language processing techniques to translate natural language code comments into procedure specifications; it also shows how to generate test oracles from procedure specifications, and how to deploy the generated oracle into test cases. It presents the results of the empirical evaluation that we performed to validate the technique.
- Chapter 5 summarizes the contributions of the thesis and outlines future research directions.

Chapter 2

Automated Oracle Generation

Test oracles are of prime importance to detect faults at testing time, expensive to design manually, and difficult to generate automatically. This chapter overviews the main techniques proposed to automatically generate test oracles, highlights their contributions and strength, and discusses their limitations and weakness. This chapter also emphasizes the characteristics of cost-effective oracles, and the limitations of existing techniques with respect to cost-effectiveness.

Designing effective test oracles is a difficult and expensive task. In their seminal work, Davis and Weyuker discuss the issue of creating oracles to test software systems for which is difficult or even impossible to define the expected behavior [25, 97]. While when designing test cases manually developers usually write both test inputs and oracles (often in the form of assertions), the oracle problem becomes extremely important in the context of the automatic generation of test inputs. Current automatic test case generators produce *inputs* with limited oracles. Generated test inputs with limited oracles can only identify blatant failures, for instance system crashes, and lead to many false alarms and missed alarms. Many results produced by the SUT, when fed with automatically generated test inputs, require manual inspection and assessment. This is a time-consuming and error-prone task: for each result, developers have to understand how automatically generated test inputs exercised the SUT, and figure out what the expected behavior of the SUT for the specific input is to classify the result as either correct or wrong. This process is tedious, expensive and error-prone: The same or analogous behaviors can be exercised by several tests and for each test the developer must assess the validity of the outputs. This factor can be mitigated, but not eliminated, with test suite minimization techniques.

A cost-effective technique to generate test oracles would allow automatic testing to be more effective and less expensive: more effective because a semantically rich oracle can identify not only blatant faults, but also complex wrong behaviors; less expensive because an automatic oracle relieves developers from the burden of manually writing oracles for automatically generated test cases.

We now survey the main techniques proposed so far to generate test oracles and deal with the oracle problem [4, 5, 80, 85]. We borrow the classification and terminology from a recent survey by Barr and colleagues who classify test oracles in three categories: *implicit*, *specified*, and *derived* test oracles [5].

2.1 Implicit Oracles

Implicit oracles are generated from implicit knowledge about the correct behavior of the SUT, do not depend on the SUT semantics, are easy to generate, and are generally applicable to any program. Implicit oracles need to be defined only once, and can be deployed and applied to test any implementation.

The most common implicit oracle is the "crash oracle", sometimes also called "null oracle": Crash oracle deems as erroneous any system crash and unhandled exception. Crash oracle is general since it can be applied to any application, given that it ignores the semantics of the SUT. Crash oracle is a heuristic oracle since it signals as erroneous every crash and unhandled exception, while not every crash and unhandled exception are erroneous behaviors. (A system may intentionally crash under some circumstances, for instance to prevent major consequences when a malicious attack is detected.) For such programs, the crash oracle would classify the crash as an erroneous result, while the crash is the *correct and expected* result. The generation of a crash oracle is straightforward: Modern execution environments and testing frameworks like JUnit[1] treat anomalous program termination as failures by default. Crash oracle is used in different testing techniques because of its simplicity and low cost. For example, the crash oracle is commonly used in automatic testing of graphical user interfaces (GUIs) where GUIs under test are stimulated with different strategies, for instance random[2] and reinforcement learning [59]. Crash oracle is combined with test inputs generated with tools like EvoSuite [34, 35], Randoop [70], and GRT [56] to detect faults causing system crashes. Alike those tools, robustness tester tools such as JCrasher [22], Check 'n' Crash [23], and DSD-Crasher [24] exploit the crash oracle to detect faults leading to crashes.

Self-healing techniques often use the crash oracle to detect system failures and apply proper healing strategies [13]. Crash oracle is used in fuzzing, that is the generation of random input in attempt to make the SUT crash [67]. The effectiveness of the crash oracle is limited to those faults that lead to an anomalous termination of the SUT. Shrestha and Rutherford empirically evaluated the fault-finding effectiveness of the crash oracle, and indicate that crash oracle detects about one fifth of faults present in software systems [87].

Randoop [70] generates test cases that combine assertions checking crashes with assertions that check object-oriented contracts.[3] For example, Randoop-generated assertions check that the implementations of the Java method `java.lang.Object#equals` are:

- reflexive: `x.equals(x)==true`;
- symmetric: `x.equals(y)==y.equals(x)`;
- transitive: `x.equals(y)==true && y.equals(z)==true => x.equals(z)`.

These checks are easy to generate in the test suite. Randoop also heuristically classifies test executions. For instance, Randoop deems as faulty those Java methods that trigger `NullPointer-Exception` when none of their arguments is `null`. Although straightforward to generate, the heuristic-based assertions generated by Randoop are not always sound and may generate false alarms.

Implicit oracles are effective in identifying generally wrong behaviors, but cannot identify semantic failures, that is, incorrect results produced by the SUT. In a nutshell, implicit oracles

[1] `http://junit.org`
[2] `https://developer.android.com/studio/test/monkey.html`
[3] `https://randoop.github.io/randoop/manual/index.html#classifying_tests`

can only detect general, blatant errors and fail in identifying wrong results produced by the SUT. Implicit oracles are *partial*, since they only check few properties of the result produced by a program, and cannot assess the overall correctness of the results.

An important aspect of implicit oracles is that they are generally applicable, but not always valid. Crash oracle is a perfect example, since not every crash is an unintended incorrect behavior. Thus, implicit oracles are not always correct and may generate spurious results.

2.2 Specified Oracles

Specified oracles are generated from formal specifications of the expected behavior of the SUT. Formal specifications can be used to automatically generate test oracles by checking if the SUT behaves as specified. Formal specifications can also be used to generate test input, therefore the existence of formal specifications supports the automatic generation of complete test cases that stimulate the SUT and check the correctness of its behavior.

There is a vast availability of languages for creating specifications that, according to Barr and colleagues [5], can be classified in three main categories: model-based specification languages, assertions and contracts, and algebraic specification languages.

Model-based Oracles Model-based specifications describe, either explicitly or implicitly, the different states in which a system can be, and the transitions that alter the state, possibly bringing the system to a different state. Transitions can be constrained, meaning that a transition could be possible only if its preconditions hold.

Several model-based specifications languages have been proposed, the most relevant ones being Z [88], B [53], OCL [96], Alloy [42], PROMELA [41], Final State Machines [54], UML State Machines [11], and Labeled Transition Systems [95].

Peters and Parnas exploit Parnas' seminal work on system specifications by proposing a testing framework in which program documentation is written in a precise formal tabular notation that describes the effects of the program, and therefore enables the automatic generation of test oracles [72, 77].

Assertions and Contracts Assertions and contracts are a popular form of class and method specifications. Assertions are conditions that must hold and can be checked during program execution. The violation of a condition indicates the presence of an error condition in the program. Assertions are supported by many popular programming languages like C, C++, C#, Java, Python.

Languages that implement the design-by-contract methodology (e.g., Eiffel) provide developers with constructs to define contracts, that are pre- and post- conditions. Contracts are exploited in testing to generate test inputs and test oracles [21, 64, 65, 68].

Oracles from Algebraic Specifications Doong and Frankl propose ASTOOT, a technique to automatically generate test cases for object-oriented programs from algebraic specifications [30]. ASTOOT analyzes the algebraic specifications and derives pairs of method invocation sequences and a Boolean tag indicating whether the two sequences should lead to an equivalent state or not. A test case execution consists in executing the two sequences and checking whether the obtained outputs/states are (approximately) observationally equivalent (or not) and if this is consistent with the tag. The ASTOOT approach has been evolved and refined in TACCLE that combines black- and white-box testing for unit testing [17, 18]. TACCLE exploits a black-box testing approach to generate sequences of method calls, and a white-box testing approach to check the observational equivalence (or non-equivalence) of the generated sequences. TACCLE also extends ASTOOT to cluster-level (integration) testing, with a new specification language for modules called Contract. Contract specifications formally describe interactions between different classes and allow the creation of integration tests.

Generating test oracles from formal specifications is not always simple and straightforward. To create oracles in the form of assertions embeddable in a test suite, the language used to write the specifications must have an abstraction level similar to the source code. Also, the specifications must have a semantics that can be converted into assertions using a programming language compatible with the programming language of the SUT. For example, name of variables and constants in specifications and in the actual code of the SUT must be compatible.

Specified oracles are effective in identifying semantic failures corresponding to faults in the implementation of the SUT, but suffer from the cost of producing and maintaining the formal specifications. While modern developers are accustomed to write, inspect, check, and debug source code, familiarity with formal specification languages is still not widespread. Another drawback of formal specifications is the alignment between source code implementation and its formal specification: The alignment must be continuously maintained and, given the fast pace at which the code evolves, this is an expensive task. As a consequence, specified oracles are rarely used in practice for common systems that do not require strong and certified guarantees on their behavior.

2.3 Derived oracles

Derived oracles are generated from artifacts other than formal specifications, such as execution models, informal documentation, invariants, and other versions of the SUT.

Pseudo-oracles Pseudo-oracles rely on multiple alternative, independent implementations of the SUT [25]. They check whether the program under test and the independent implementations produce the same result given the same input. If the different versions agree, that is, they produce the same result, the program under test is considered correct, otherwise pseudo-oracles report a failure. In comparing the results from the different implementations, the comparison mechanism may behave akin to a majority vote algorithm. Pseudo-oracles are conceptually simple. However the comparison procedure, that compares the different outcomes to check whether they are the same, hides potentially complex problems. Comparing integer numbers and Boolean values is easy, while comparing complex outputs is less trivial. Thus, creating pseudo-oracles can be quite difficult, even when multiple implementations of the same functionality are available. The main limitation in pseudo-oracles adoption is the cost of producing multiple alternative (independent) versions of the same functionality. Such high cost is not justifiable for testing the most common applications.

Recently, a kind of pseudo-oracle has been used to test collaborative Web applications, for instance Google Docs and Microsoft Office Online: in any given instant collaborators who are working on the same shared document through their browsers should see the same document, i.e., same content and same style. If the document looks different to the collaborators an error has occurred [8]. In this case, there is no additional cost to generate the oracle: such technique can be applied without incurring any additional development cost.

Metamorphic and Symmetric Testing *Metamorphic testing* exploit metamorphic properties of the program under test to automatically generate test inputs and oracles [19, 20, 85]. A metamorphic relation specifies how a change in the input of a program under test would change its output, and encodes the intrinsic redundancy present in the SUT. For instance, consider the function `cosine(x)` that computes the cosine value of an angle `x`. A metamorphic relation for the cosine function is `cosine(x)`≡`cosine(-x)`. The readers should notice that this is only one of the many metamorphic relations of the function cosine. Given the metamorphic relation and a test input like `x`=π, metamorphic testing generates the following test case:

```
1 result1 = cosine(π)
2 result2 = cosine(−π)
3 assert(result1 == result2)
```

On line 3 the generated test case verifies the validity of the metamorphic relation: If the results of the assumed equivalent operations differ, the assertion fails letting the test case fail. Besides mathematical domains where metamorphic testing has an immediate applicability, metamorphic testing has been applied to different non-numerical domains, like Web services [1, 107], compilers, and computer graphics [85]. The cost of metamorphic testing lies in the generation of initial test inputs and in the definition of the metamorphic relations. While there is a good support for the automatic creation of test inputs, at least in some of the domain where metamorphic testing is applied, the support for an automatic identification of metamorphic relations is at an early stage and is currently possible only in specific domains [47, 48, 55, 102].

Symmetric testing exploits equivalent sequences of method calls to generate test cases with a semantically relevant oracle [38]. Symmetric testing relies on two kind of *symmetries*: *over-values* and *over-variables*. An over-values symmetry describes the relation between two executions of the same functionality executed with different input values. As an example consider the function `add(x,y)` computing the sum of two numbers `x` and `y`. An over-values symmetry for this function is `add(x,y)=add(x+k,y-k)`. An over-variables symmetry describes the relation between two executions of the same functionality executed with a permutation of the same inputs. For example, the symmetry `add(x,y)=add(y,x)` is an over-variables symmetry for the previously introduced function `add`. Symmetries encode correctness conditions. For instance, function `add` is correctly implemented if and only if ∀ x,y,k `add(x,y)=add(x+k,y-k)`. In symmetric testing, symmetries are used to generate test oracles for a given set of test inputs that can be generated with automatic tools or manually defined by developers. Given a test input and a symmetry, symmetric testing verifies that the symmetry holds executing the function under test both with the given input and the input transformed as specified by the symmetry, and then checking whether the symmetry holds comparing the results. If the results differ, a symmetry violation is reported thus highlighting a fault in the implementation of the function under test. The main cost of symmetric testing lies in the manual discovery of symmetries (a common trait shared with metamorphic testing). Manually defining symmetries is a difficult task since many symmetries are nontrivial and they might be difficult to identify for a developer. Symmetries definition is also an extremely important task: the effectiveness of symmetric testing largely depends on the symmetries available for the SUT. To the best of our knowledge no technique supporting the generation of symmetry has been presented so far, although it is conceptually possible to adapt techniques for automatically derive metamorphic relations to the task of symmetries generation.

Oracles from System Executions Test oracles can be generated directly from properties inferred by executing the SUT, from models learned from system execution, and from previous executions (regression oracles).

Quasi-invariants that Daikon generates [31, 32], temporal invariants [7], and final state automata [58] can be used as test oracles.

Researchers have exploited Artificial Neural Nets (ANNs) and machine learning classifiers to *learn* the expected behavior of the SUT from executions and generate oracles [10, 100].

Automatic test case generators like EvoSuite [34, 35] automatically generate test suites with embedded regression oracles, in the form of assertions about the consistency between the results of the current and the previous executions. EvoSuite generates a test input, executes the generated test input, records the output, and generates an assertion that acts as an oracle, for example, `assert(result==42)`.

Generating derived oracles is less expensive than generating specified oracles, because the information needed to generate the oracles is automatically extracted from the system execution. However, an empirical study of Nguyen and colleagues indicates that some derived oracles are still expensive to be applied in practice, since such oracles may present a high false alarm rate that largely impact on their practical applicability [69].

2.4 Cost-effectiveness

Test oracles are characterizable by generation cost and effectiveness. The cost of generating a test oracle roughly amounts to the cost of producing the information needed for the oracle generation, plus the cost of using that information for the oracle generation. The effectiveness of a test oracle is its ability to correctly classify executions of the SUT as correct or wrong. In other words, a test oracle is effective when it is able to precisely identify erroneous behaviors of the SUT and report those (and only those) failures.

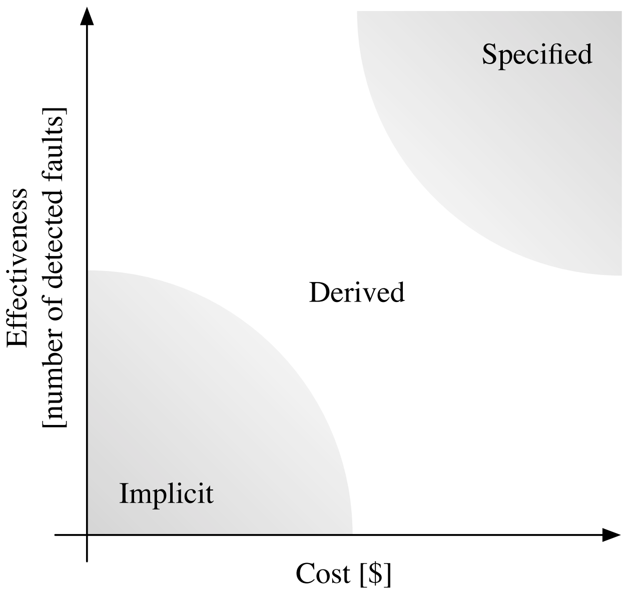


Figure 2.1. Test oracles cost-effectiveness

Let us consider the intuitive and informal landscape of the cost-effectiveness of implicit, specified, and derived test oracles, where the oracle cost only amounts to the cost of its generation (see Fig. 2.1). On one side of the spectrum there are implicit oracles: inexpensive to generate, but with a low effectiveness. Implicit oracles are generated from inexpensive information that is usually available and does not have to be provided ad-hoc for the oracle generation. At the same time, implicit oracles are only capable of detecting a small portion of the wrong behaviors the SUT might show. On the other side of the spectrum, there are specified oracles that are expensive to generate and effective in finding semantic faults. The high generation cost of specified oracles stems from the fact that the specifications from which they are produced are difficult and costly to define. The effectiveness of specified oracles comes from the fact that they are able to check every specified aspect of the SUT. In this sense, the effectiveness of specified oracles largely depends on the quality of the specifications from which oracles are generated. The more the behavior of the SUT is specified, the more the specified oracles are effective in identifying failures. In between implicit and specified oracles there are derived oracles that can be less expensive than specified oracles and more effective than implicit oracles. Derived oracles save developers from writing formal specifications and are able to detect faults at testing time that would be overlooked by simple implicit oracles.

Existing techniques to generate derived oracles show significant limitations. All the approaches that derive oracles from executions of the SUT are convenient during regression testing but difficult to apply for standard (i.e., non-regression) testing. Oracles generated from executions of the SUT capture the behavior of the "golden" version and then test the new version to check that golden and new version behave the same. However, this approach is not applicable in normal testing, when new functionalities are added to the system (or existing functionalities are changed). Pseudo-oracles require multiple implementations of the same functionality resulting in an expensive alternative to specified oracles (but applicable even when it is difficult to define the expected behavior of a system, and formal specification would be hard or impossible to write). Symmetric/metamorphic testing are *testing techniques* rather than oracle generation techniques and requires symmetric/metamorphic relations to be present, discovered, and formalized. All of this implies a cost that counterbalances the effectiveness of such derived oracles. Besides the information source used to generate the oracles, the rate of false alarms of derived oracles is a critical issue [69]. To be truly cost-effective derived oracles must provide a decent level of accuracy, i.e., few false alarms must be generated. Clearly, the level of accuracy provided by specified oracles is difficult to match, still the number of false alarms reported by derived oracles should not be so high to prevent the adoption of derived oracles in practice.

The goal of this thesis is to design techniques to generate cost-effective test oracles that are more effective than implicit oracles and current derived oracles, and that can be generated with an acceptable cost. To achieve the goal, test oracles have to be generated from semantically relevant information that is specific to the SUT, and that is commonly present in software systems (i.e., that has not to be produced ad-hoc for the oracle generation). This leads to the generation of oracles that are truly cost-effective in most situations.

Chapter 3

Oracles from Intrinsic Redundancy

Modern software systems are *intrinsically* redundant, that is, they contain different implementations of the same functionality. Thus, intrinsically redundant systems can provide the same functionality through different executions. The intrinsic redundancy of software systems is a consequence of the software life cycle, and does not derive from explicit decisions to add redundancy, as in the case of safety critical systems where redundancy is explicitly added to increase reliability.

We observe that intrinsic software redundancy can be used to automatically generate test oracles, by checking that two intrinsically redundant functionalities produce the same result when executed with the same input. In this chapter, we propose a novel kind of test oracles, named CCOracles, that compares the outcomes of two redundant functionalities, and identifies executions that lead to non-equivalent results, thus violating the redundancy assumption.

The chapter introduces CCOracles, presents an approach to automatically generate CCOracles from intrinsic software redundancy, and discusses the results of an extensive empirical evaluation about the effectiveness of CCOracles in identifying synthetic faults in several systems under test. The chapter also describes how intrinsic redundancy can be automatically identified to further reduce the cost of generating CCOracles.

Modern software systems are *intrinsically redundant* in the sense that they can provide the same functionality through different executions. Intrinsic software redundancy is not added explicitly to systems, for example to meet reliability requirements. Rather, intrinsic redundancy is a consequence of independent design and process decisions: The intrinsic redundancy of software systems stems from different design practices like design for reusability, backward compatibility, and performance optimization.

Design for Reusability Current software systems mix and are composed of different components whose functionalities may overlap. For example, there are a lot of Java libraries whose functionalities overlap, at least partially, with the standard Java library: Trove[1] that provides high-performance collections, Guava[2] that contains various functionalities from collections to

[1]`http://trove.starlight-systems.com`
[2]`http://github.com/google/guava`

```java
public boolean put(K key, V value) {
  return get(key).add(value);
}
```

```java
public boolean putAll(K key,Iterable values){
  checkNotNull(values);
  if (values instanceof Collection) {
    Collection valueCollection = (Collection) values;
    return !valueCollection.isEmpty()
      && get(key).addAll(valueCollection);
  } else {
    Iterator valueItr = values.iterator();
    return valueItr.hasNext()
      && Iterators.addAll(get(key), valueItr);
  }
}
```

Figure 3.1. Methods `put` and `putAll` of the `AbstractMultimap` class from Google Guava

graphs and strings manipulation, SLF4J[3] which is an API for logging that can be bound at runtime with many back ends. The usage of reusable components is fostered by design for reusability practices.

Backward compatibility During the system evolution functionalities are modified, updated and replaced. Removing and replacing functionalities may significantly impact on the overall system behavior, since it may affect any module that depends on the removed or replaced function. To mitigate the impact of such changes, developers often keep the old functionality in the system, sometimes as “deprecated” functions. This phenomenon is evident in every major release of Java, where many methods are deprecated, leaving different alternatives to achieve the same goal inside the Java library itself.[4]

Performance optimization Classic examples of redundancy due to performance optimization are algorithms that work well for specific cases and poorly in others, for instance algorithms that work well in terms of time but badly in terms of memory consumption. Consider the case of sorting: there exist several algorithms to sort an array and, depending on the array size and the memory available, an implementation can be better than the others. Therefore, having multiple implementations of the same functionality makes sense in some contexts to build more efficient software systems.

Software is redundant at different levels, from single instructions to functions and methods to entire subsystems. For examples, systems can contain different subsystems taking care of logging, and may contain different methods to sort collections and arrays. So far, most studies have focused on the use of intrinsic redundancy at method call or function level for procedural systems [13, 14, 16, 38, 61, 85].

Intrinsic redundancy at method call level manifests itself in the form of equivalent sequences of method calls that produce the same effect, even though their executions differ. Figure 3.1 shows an example of intrinsic redundancy at method call level. Methods `put(K key, V value)` and `putAll(K key, Iterable values)` of the class `AbstractMultimap` from the Google Guava project[5] add key-value pair(s) to the map. The method `put` adds a new single key-value pair to a map, while `putAll` adds multiple key-value pairs, one for each element of the `Iterable` object

[3]http://www.slf4j.org

[4]https://docs.oracle.com/javase/9/docs/api/deprecated-list.html#method

[5]https://github.com/google/guava

`values`. Although they provide the same functionality, that is, they add key-value pair(s) to the map, their implementation is largely different: `put` directly invokes method `add`, `putAll` does not. Even if it is true that at some point the execution of `putAll` could trigger the execution of the method `add`, still the implementations differ: `putAll` is not simply implemented as multiple calls to `put`, at least, not directly. Therefore, `put` and `putAll` are redundant sequences of method calls.

The presence of intrinsic redundancy in modern software systems has been empirically studied in the last years. Carzaniga et al. studied the documentation of popular JavaScript libraries looking for intrinsically redundant method calls to generate workarounds in self-healing approaches for runtime failures. A workaround for a given method/function consists of an alternative sequence, that is, a redundant sequence, of methods/functions. Carzaniga and colleagues mined about 300 redundant sequences from Google Maps, JQuery, and YouTube JavaScript libraries [14, 15, 37]. Mattavelli's PhD thesis reports an exhaustive investigation of the presence of intrinsic redundancy in software systems [61]. In the thesis, Mattavelli reports 5813 workarounds, that is, redundant method call sequences, from 961 classes of 12 open source Java projects (Apache Ant, Apache Commons Lang, Google Guava, Oracle JDK, Joda Time among the others). These data provide strong evidence of the presence of intrinsic redundancy at method call level in software systems (at least in those written in Java and JavaScript). The intrinsic redundancy is not equally distributed across different software systems. Some systems contain way more equivalent sequences than others. For example, Google Guava contains about 15 equivalent sequences per class, while Apache Ant less than 4.

Intrinsically redundant functionalities are substantially different from code clones: intrinsically redundant functionalities are different implementations with the same functional behavior, while code clones are logically or structurally similar code fragments with slightly different functionality. Code clones are well known in the research community [43, 73, 81], and mainly regarded as the consequence of bad design and development practices, aimed to be avoided or fixed. Intrinsically redundant functionalities have been investigated only recently, can stem from good practices and can be fruitfully exploited for different purposes. In recent years, intrinsic redundancy has been used to add self-healing capabilities to Web and Java applications by means of workarounds that are automatically activated upon failures [14, 37, 76]. In this thesis, we propose a new technique for automatically generating semantically relevant test oracles by exploiting the *intrinsic redundancy of software systems at method call level*, that is, equivalent sequences of method calls.

3.1 Cross-Checking Oracles

We propose to use intrinsic software redundancy to create test oracles that do not require additional and expensive-to-define information (e.g., formal specifications) beyond the information already present in the SUT itself. In particular, we exploit intrinsic redundancy at method call level to automatically derive oracles that we call CCOracles. CCOracles are rooted in the idea of pseudo-oracles prosed by Davis and Veyuker [25] (see Section 2.3). However, instead of cross-checking the executions of multiple, independently-developed versions of a functionality, CCOracles exploit the intrinsic redundancy contained in the SUT encoded as *equivalent sequences of method calls*. Referring to Figure 3.1, an example of an equivalence rule that encodes intrinsic redundancy is `AbstractMultimap.put(key,value)` ≡ `AbstractMultimap.putAll (key,List.of(value))`. The rule states that methods `put` and `putAll` shall provide the same functionality for any actual concrete value assigned to arguments `key` and `value`.

The key intuition underlying CCOracles is that two redundant sequences of method calls are supposed to behave equivalently, but their actual behaviors may diverge because of a fault in their implementation. Therefore, if there is an input for which the behavior of the two redundant sequences diverges, the SUT (from which the method sequences come from) is faulty. In other words, given the same input, two redundant sequences should always produce the same output. When two redundant sequences produce different outputs for a specific input, there is a bug somewhere in the code of the SUT.

Figure 3.2 shows the conceptual schema of a CCOracle: Given two reundant sequences $\texttt{Seq}_1$ and $\texttt{Seq}_2$ and given an input for the sequences, the execution of $\texttt{Seq}_1$ and $\texttt{Seq}_2$ produces $\texttt{Output}_1$ and $\texttt{Output}_2$ respectively. The outputs are then checked with an equivalence-check procedure that verifies the equivalence of the outputs. If the outputs are equivalent, the equivalence-check passes, while it fails otherwise. A failure in the equivalence check indicates the presence of a fault in the code.

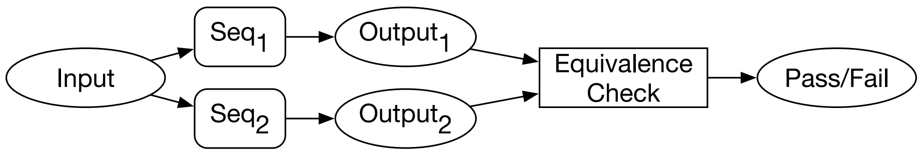


Figure 3.2. Conceptual schema of a CCOracle.

A CCOracle needs two redundant method call sequences, that is two call sequences that are *equivalent* (i.e., they produce the same result) and *different* (i.e., they are not exactly the same sequence). More precisely, a CCOracle relies on the redundancy between a single method call `m` that we refer to as original method call, and a sequence of method calls $\texttt{m}_0 \ldots \texttt{m}_n$. A CCOracle is applied to a test input, that is a valid input for both `m` and the sequence $\texttt{m}_0 \ldots \texttt{m}_n$, and that can be either manually-written or automatically generated by automatic test case generator like EvoSuite [35] and Randoop [70].

As an example, consider the redundant sequences of Figure 3.1. Starting from a given input, for instance an empty map, adding a single key-value pair by means of method `map.put(key, value)` and adding a collection with a single key-value pair with the call `map.putAll(key, List.of(value))` must produce an equivalent map, that is, a map containing a single key-value pair. The equivalence of the resulting maps must be checked by the equivalence check decision procedure.

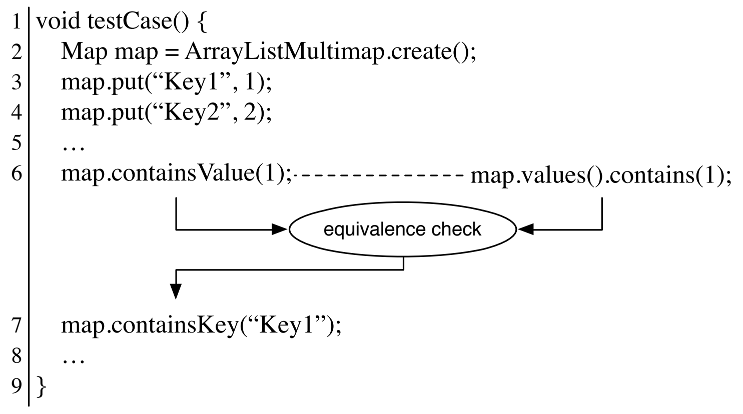
CCOracles are composed of three main ingredients:

1. *Cross-check execution*: The mechanism allowing the execution of two method sequences starting from the same input.

2. *Equivalence check*: The decision procedure that verifies the equivalence of the outputs produced by the sequences.

3. *Deployment*: The way CCOracles are applied to existing test inputs.

The general idea of CCOracles can be applied in many testing contexts. We implemented the idea for the unit testing of Java programs and in the remainder of this chapter we refer to such specific implementation of CCOracles.



```
void testCase() {
    Map map = ArrayListMultimap.create();
    map.put("Key1", 1);
    map.put("Key2", 2);
    ...
    map.containsValue(1);------------- map.values().contains(1);
    map.containsKey("Key1");
    ...
}
```

Figure 3.3. A test case with an embedded CCOracle

To illustrate the ingredients of a CCOracle, let us consider the more concrete example reported in Figure 3.3. The example consists of an excerpt of a unit test case for the class `ArrayListMultimap` from Google Guava. The test case is augmented with a CCOracle exploiting the following, given equivalence: `map.containsValue(x)` ≡ `map.value().contains(x)`. The equivalence states that method `map.containsValue(x)` and the method call chain `map-.values().contains(x)` are two equivalent ways to check whether a specific element `x` is a value contained in the map. The execution of the test case proceeds normally up to the invocation of the method `containsValue(1)`. Then, right before the execution of `containsValue(1)`, the cross-check execution mechanism executes both `containsValue(1)` and `value().contains(x)` starting from the same initial map, obtaining two maps. Then the equivalence check verifies whether the two maps are equivalent. If so, the execution of the test case continues, otherwise the CCOracle reports a failure. The automatic deployment mechanism decides where to execute a cross-checking oracle within a test case.

A key element for the success of CCOracles is the ability of automatically identifying redundant method call sequences. The next sections describe how redundant method call sequences can be automatically identified and the three ingredients of a CCOracle.

3.1.1 Automatic Identification of Redundant Sequences

Redundant method call sequences are relations between a method call and a sequence of method calls, and encode the intrinsic redundancy present in a software system. Redundant sequences have been exploited to add self-healing capabilities to software systems [13, 14]. CCOracles exploit redundant sequences to automatically generate test oracles.

In this section, we discuss a search-based approach to automatically identify relations $m \equiv s$, between a method m of a class, and a method calls sequence s. m and s execute differently, but produce equivalent results. (The approach is briefly discussed in this thesis since it is not one of the author's main contributions. The approach is described at length by Mattavelli in his Ph.D. thesis [61].)

Checking the full equivalence of two programs (sequences) on every input is a well-known undecidable problem. We propose a method that refers to a tractable notion of equivalence based on *testing equivalence* [27]: two sequence are testing equivalent—hereafter *equivalent* for short—if they are equivalent with respect to a finite (and thus tractable) set of inputs. We designed a search-based approach for Java programs called SBES (Search-Based Equivalent Sequences) [36, 61, 62]. SBES takes as input a method call m and a set of test input I for m, and generates redundant sequences in two iterative phases. SBES first generates a set of *likely-equivalent candidates*, by exploiting a genetic algorithm (GA) to synthesize a candidate sequence c that is testing equivalent to m, that is, it reaches the same state and produces the same return value of m on every test input in I. SBES checks that two states or return values are the same by recursively computing the distance between the two objects. SBES computes the distance of primitives fields as the absolute mathematical difference between the numeric fields and the Levenshtein distance of string fields. It computes the distance between arrays as the sum of the distance of their elements, and replacing missing elements with the maximum possible value for the array elements type, when processing arrays with different length. It considers objects as “infinitely” distant from `null`. SBES is implemented as a modified version of the GA implemented in EvoSuite [34, 35]. If SBES cannot synthetize a candidate sequence c within a given search (time) budget, it terminates. Otherwise, SBES validates the computed *candidate c* to understand whether c is effectively equivalent to m. SBES tries to synthetize a test input t for which the behavior of m and c diverges, with a given search (time) budget. If SBES succeeds in finding a test input t for which c produces different output or state than m, SBES discards the candidate sequence c as non-equivalent, adds t to the set of test inputs I, and iterates to select a new candidate, otherwise SBES returns c as a redundant sequence of m.

Notice that, although SBES generates redundant sequences, it does not check to what extent the identified sequences are redundant. Therefore, SBES may find equivalences like $m \equiv s$ where m and s contain exactly the same statements. Such equivalences do not affect the effectiveness of CCOracles even though they are not helpful in identifying functional faults of the SUT.

SBES can synthetize redundant method call sequences in Java classes with good precision and recall. We compute precision as the ratio between the number of correct equivalent sequences synthesized by SBES and the total number of equivalent sequences reported by SBES (correct and non-correct):

$$precision = \frac{correct}{correct + wrong}$$

We compute recall as the ration between the number of correct equivalent sequences synthesized by SBES and the total number of equivalent sequences of a given class:

$$recall = \frac{correct}{correct + wrong + missing}$$

SBES synthesizes correct equivalences when the generated equivalence holds for every possible input. On the contrary, if the synthesized sequence does not hold for every possible input, we deem the equivalence as wrong. Missing sequences are those sequences that SBES failed to synthesize. We evaluated the precision and recall of SBES as follows:

1. We selected a class from a library.

2. We manually derived its redundant sequences, each in the aforementioned form $m \equiv s$. Note that manually derived equivalences are “minimal”. An equivalence is minimal if any statement in the right-hand side s cannot be removed without altering the effect produced by the sequence s. Of course a minimal sequence can be indefinitely extended, for instance by adding adding statements with an overall null effect. For example, considering

sequences from a Java list, the sequence `X.add(42)` can be extended as `X.add(42); X.add(42); X.remove(42); ...`

3. For each equivalence, we ran SBES to generate the equivalent sequences of m (the left-hand side of the equivalence). SBES is nondeterministic due to the random nature of GA that SBES exploits to generate call sequences. Thus, we run SBES 20 times per target method m, with a search budget of 3 minutes for the candidate generation phase and 6 minutes for the candidate validation phase.

4. We inspected each equivalent sequence that SBES produces and we classify it as correct and wrong.

5. We measured precision and recall according to their definitions.

Table 3.1 summarizes the subject classes we use in our experimental evaluation. For each subject class, the table reports the number of methods provided by the class (column Available), the number of methods considered in our evaluation (column Target), and the number of equivalent sequences we manually derived for the class (column Equivalences). Target methods are those methods that appear as left-hand side of manually-derived equivalences. Subject classes are selected from three popular, open-source Java libraries: Google Guava, GraphStream, and the Java standard library. GraphStream and Guava libraries are also used for the CCOracles evaluation and are described in Section 3.2.1. In addition, we consider the `Stack` class from the standard Java Class Library.[6] In total, we evaluated SBES on 266 target methods from 23 Java classes, with a total of 421 equivalences.

For each subject, Table 3.2 shows the aggregate results of the 20 runs of SBES for each subject. Table 3.2 reports the number of: manually derived equivalences (column Equiv.), equivalences that SBES synthesizes correctly (column Correct), equivalences that SBES wrongly synthesizes (column Wrong), precision and recall. SBES has an overall precision of 82%, meaning that more than 4 out of 5 synthesized equivalences are correct. With an overall recall of 74%, SBES can identify 3 out of 4 equivalences present in a Java class. In summary, SBES can effectively identify many redundant method call sequences for Java programs.

3.1.2 Cross-Check Execution

The cross-check execution mechanism executes both method call sequences with the same input. In general, the input consists of every external element that could affect the execution. For instance, the input of a method call consists of the state of the program and the arguments provided to the method call. The state of a program in itself includes many aspects: the state of the receiver objects, the state of global variables, the state of input/output operations, etc. In an object oriented system, the state is usually composed of interconnected objects referring each other through their field members.

A CCOracle executes two method call sequences on the same object, by guaranteeing the mutual independence of the two executions, and the equivalence check compares the results. The test case execution following the execution of a CCOracle shall not be affected by the execution of the CCOracle.

We explored several techniques to execute pairs of redundant sequences without interferences between their executions.

[6] `http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html`

Table 3.1. SBES Empirical Evaluation Subjects

		Methods		
Library	**Class**	**Available**	**Target**	**Equivalences**
Graphstream	Path	31	2	5
	Edge	36	9	20
	SingleNode	72	5	12
	MultiNode	76	5	12
	Vector2	29	5	21
	Vector3	39	6	22
Guava	ArrayListMultimap	25	15	18
	ConcurrentHashMultiset	27	16	16
	HashBasedTable	25	16	13
	HashMultimap	24	15	13
	HashMultiset	26	16	19
	ImmutableListMultimap	30	11	20
	ImmutableMultiset	32	8	20
	LinkedHashMultimap	24	15	13
	LinkedHashMultiset	26	16	19
	LinkedListMultimap	24	24	17
	Lists	17	8	16
	Maps	32	9	12
	Sets	30	10	25
	TreeBasedTable	27	15	17
	TreeMultimap	26	14	12
	TreeMultiset	35	20	34
Java	Stack	50	15	45
Total		**778**	**266**	**421**

Table 3.2. SBES Empirical Evaluation Results

			Synthesized Equiv.			
Library	**Class**	**Equiv.**	**Correct**	**Wrong**	**Precision**	**Recall**
GraphStream	Path	5	5	2	0.71	1.00
	Edge	20	20	1	0.95	1.00
	SingleNode	12	12	0	1.00	1.00
	MultiNode	12	12	0	1.00	1.00
	Vector2	21	21	3	0.87	1.00
	Vector3	22	22	4	0.84	1.00
Guava	ArrayListMultimap	18	12	3	0.80	0.67
	ConcurrentHashMultiset	16	6	2	0.75	0.38
	HashBasedTable	13	2	8	0.20	0.15
	HashMultimap	13	13	1	0.92	1.00
	HashMultiset	19	19	5	0.79	1.00
	ImmutableListMultimap	20	2	0	1.00	0.10
	ImmutableMultiset	20	3	0	1.00	0.15
	LinkedHashMultimap	13	12	3	0.80	0.92
	LinkedHashMultiset	19	19	6	0.76	1.00
	LinkedListMultimap	17	11	0	1.00	0.65
	Lists	16	15	1	0.94	0.94
	Maps	12	8	0	1.00	0.67
	Sets	25	21	0	1.00	0.84
	TreeBasedTable	17	3	10	0.24	0.18
	TreeMultimap	12	8	2	0.80	0.67
	TreeMultiset	34	34	10	0.78	1.00
Java	Stack	45	32	7	0.82	0.71
Total		**421**	**312**	**68**	**0.82**	**0.74**

First, we explored a solution based on the *fork* of the execution of the test case into two separated Java virtual machine [49]. Cloning the Java virtual machine is the ideal solution since it supports completely independent executions, but this solution is complex and, to the best of our knowledge, there is no readily usable implementation of a cloneable JVM. Unfortunately, the fork primitive provided by operative systems cannot fork standard JVM implementations, since the fork operation does not support multithreaded processes (the forked process would have only one active thread that is the duplication of the thread from which the fork operation is invoked). Another major shortcoming of this solution is that the output of the two executions are in two different virtual environments. This complicates the comparison, since at least one of the two results must be serizalized to be sent outside its environment, and serialization is time-consuming and potentially a source of problems, for example when only part of an object state can be serialized, not to mention when entire objects cannot be serialized out of the box, that is, their classes do not implement `java.io.Serializable` interface.

We then investigated the use of a *checkpoint and rollback* approach whose conceptual schema is shown in Figure 3.4. In this approach, the initial state of the system is saved (`init-state`) with a checkpoint operation so that both sequences can be executed starting from `init-state`. The redundant sequence is then executed and the output `output1` is saved. Then, the system state is rolled back to previously saved `init-state` before the execution of the original method invocation to produce output `output2`. The `restore` operation retrieves the output of the redundant sequence `output1` previously saved. Finally, the equivalence check procedure `compare` verifies that `output1` and `output2` are equivalent. Even this solution requires complex checkpoint-rollback mechanisms, either based on some form of serialization or other mechanisms like software transactional memories [86]. Thus, a checkpoint-rollback technique can show the same shortcomings of the fork-based solution described before.

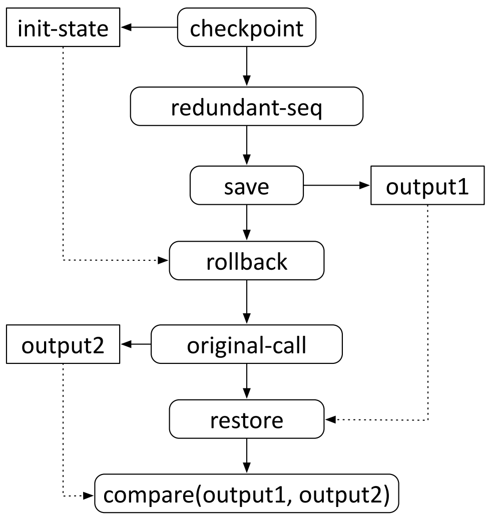
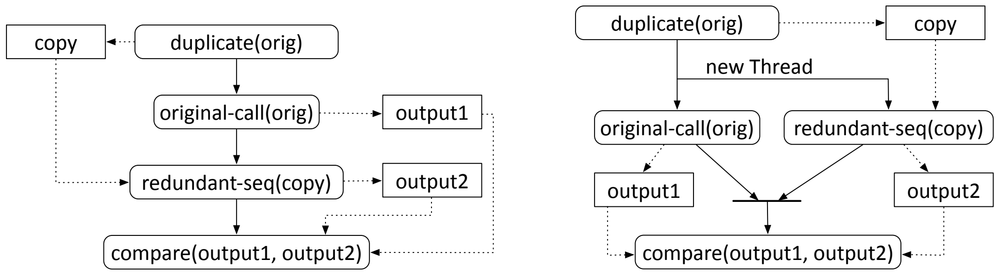


Figure 3.4. Cross-check execution by means of a checkpoint and rollback mechanism

We resorted to a simple, clone-based solution. Figure 3.5 shows a sequential and a parallel version of this solution. In this approach the original method call, the redundant sequence, and the equivalence check execute in the same virtual machine. Without any isolation between executions provided by the platform, it is crucial to create a mechanism to allow the execution



(a) Sequential. *(b)* Parallel.

Figure 3.5. Cross-check execution based on object duplication

of the CCOracle without interfering with the original test case execution. Our solution relies on the duplication of the receiver object of the original call: in this framework the original call is executed on the original receiver object, while the redundant sequence is executed *on a copy* of the original receiver object. Optionally, also the arguments of the original method call can be deep-cloned to ensure a higher level of isolation, especially for testing classes whose methods may have side-effects on the input parameters. For the object duplication we exploit an open-source and publicly-available library,[7] since it deep-clones objects, even instances of classes not implementing the interface `java.lang.Cloneable`, that is, the standard implemented interface to make a class cloneable. We modified the library in the way collections are duplicated: By default, the cloning library uses methods `putAll` or `addAll` of collections APIs to insert elements into the newly created clone object. Instead, we tweaked the library to deep-clone the exact internal structure of collections.

To optimize the check for the consistency of the created clones, we added preliminary checks that verify the result of the cloning procedure. The first check verifies that the created copy is consistent with the original receiver object out of which the copy is created. Then, a second check ensures that the execution of the original method call on two copies produces the same consistent outcome. Only if the preliminary checks pass, the cross-check execution proceeds with the execution of the original method on the original receiver object and the execution of the redundant sequence on a copy of the original receiver object.

Even a perfect duplication mechanism cannot prevent all side-effects. In fact, interferences may still happen through the shared state like global static variables. Although this may happen, we observed that in the context of unit testing and with the additional checks, we are able to obtain accurate test oracles.

Figure 3.5 shows two different versions of the solution based on objects duplication: sequential and parallel. The sequential approach, in which original call and redundant sequence are executed one after the other sequentially in the same thread, is the one we adopted in our prototype implementation of CCOracles. In the parallel alternative, original call and redundant sequence are executed in parallel in two different threads that synchronize before the execution of the equivalence check. Although this solution may be more efficient in some circumstances, we adopted the more simple, linear approach.

[7]`https://github.com/kostaskougios/cloning`

3.1.3 Equivalence Check

The equivalence check verifies that the original method call and the redundant sequence are *semantically* equivalent, meaning that they produce the same output given the same input. In the context of object-oriented languages, like Java, the output of a method invocation is often structured, and is composed of the state of the receiver object, the return value, the state of the method arguments, the shared state like global variables, and the state of I/O operations. We simplify the equivalence check by considering only a subset of the output: We consider the state of the receiver object and the state of the return value, thus ignoring the state of the parameters, I/O and global state. This means that our equivalence check procedure may be unsound, judging equivalent two sequences producing outputs that are actually different, but whose difference lies, for example, into state of I/O operations or global state.

Comparing the state of two objects is not trivial. The simplest way is to check their identity, that is, to check if two references point to the exact same object in memory. This approach is too strict: it distinguishes two absolutely identical objects. Moreover, identity check would not work in CCOracles since the two references, inputs of the equivalence check procedure, are indeed pointing to two different objects. Sometimes, developers rely on the method `java-.lang.Object#equals(Object o)` to verify the equivalence of two objects. Method `equals` is inherited by every Java class and, by default, performs an identity check: It considers equivalent only two references pointing to the same object in the *heap*, that is, it returns `true` if and only if the receiver object and the argument of the `equals` invocations are two references to the same object. The `equals` method is often overridden to provide a more semantic equality comparison, but even when the method `equals` is correctly implemented for a given class, it may trigger the execution of the identity check defined in `java.lang.Object#equals(Object o)`. This can happen when the `equals` implementation at some point relies on the `equals` defined in `java.lang.Object`, for example, to check the equality of referenced objects of a type that does not provide a reimplementation of the `equals` method. Therefore, the `equals` method may, in some circumstances, perform an undesired identity check.

To overcome the limitations of the `equals` method, we designed a generic procedure to verify the equality of two objects based on the notion of observational equivalence. The procedure is generic, that is, it applies to *any* object type, does not require any knowledge about the objects to compare, and does not require any implementation of specific interfaces by the classes of the objects to compare. Thus, the procedure is fully compatible with instances of existing classes. Our observational equivalence check tries to infer that the outputs are indistinguishable, by probing the two objects with identical sequences of public methods, and by comparing the return values. If the return values of the probing sequences are primitives types (or wrappers), then they are directly compared. If return values are objects, they are recursively compared with the observational equivalence procedure. If the procedure observes a difference, there is at least one sequence of calls (a counterexample) that, if applied to the two objects, leads to two different results, and thus the results of the original and the redundant calls differ. If the procedure fails in identifying a difference, the equivalence check deems the two objects as equivalent. Ideally, the procedure would deem as equivalent two objects that cannot be distinguished through any sequence of method calls, be it finite or infinite. For practical purposes, the probing sequence is bounded and then the result produced by the equivalence check may not be always correct.

The CCOracles equivalence check blends `equals` method with observational equivalence: It exploits the `equals` method when there is an implementation overriding the identity check provided by `java.lang.Object#equals(Object o)`, and relies on the observational equivalence when there is no override of `equals` available or when the override implementation at some point triggers the execution of method `java.lang.Object#equals(Object o)`. In this second case, the CCOracles equivalence check substitutes the `equals` method defined in `java.-lang.Object` with the observational equivalence, so that, every time there is an invocation of such method, the observational equivalence actually executes. The blended approach presents a major advantage over a pure observational equivalence, since it uses the equality notion provided by developers when available. We implemented the CCOracles equivalence check by modifying the implementation of the `equals` method in the `java.lang.Object` class to trigger the execution of the observational equivalence instead of the identity check implemented in the `equals` method. We expect such equality implementation to be both precise and efficient in most of the cases. Thus, the blended approach is more efficient than a pure observational equivalence check.

3.1.4 Automatic Deployment

The automatic deployment *automatically* embeds CCOracles into a test suite. Given a test case, S composed of $n+1$ statements $s_0, \dots, s_n$, and a list of equivalences (redundant method call sequences) $ES = \{m_0 \equiv eqSeq_0, \dots, m_x \equiv eqSeq_x\}$ where m_i is a single method invocation, the automatic deployment selects where to deploy CCOracless to correctly trigger them during the test case execution.

We implement oracles deployment by means of aspect-oriented programming (AOP) [51].[8] We translate each equivalence $m_i \equiv eqSeq_i \in ES$ into an *aspect*. Every aspect defines one single *advice* that performs cross-check execution and equivalence check. An advice executes every time a test case execution reaches a *join point* corresponding to a *pointcut*. Given an equivalence $m_i \equiv eqSeq_i \in ES$, the aspects we create defines:

- a pointcut that triggers the advice execution *in place of* the execution of m_i for every statement $s_i \in S$ that invokes m_i;
- an advice the cross-check the execution of m_i and $eqSeq_i$.

We use aspects to instrument the bytecode of the input test suite, either statically with an ad-hoc compiler or dynamically at load time with a mechanism called *load-time weaving*. Advices are executed instead of the method calls originally present in test cases. In other words, by means of aspects, CCOracless change the normal control flow of the input test suite, embedding CCOracles checks during test case executions. In this respect, CCOracless are more akin to embedded assertions rather than assertions checking the overall result of a test case execution.

3.2 CCOracles Evaluation

We empirically evaluated CCOracles to investigate to what extent the intrinsic software redundancy at method call level can be used to automatically generate cross-checking oracles and to what extent such oracles can identify non-blatant failures of the SUT [12].

Previous work already shows that modern software systems are intrinsically redundant, and that such redundancy can be practically exploited to add self-healing capabilities to software systems [14, 37, 61, 76]. Our experiments aim to evaluate the *effectiveness* of CCOracles that

[8] `https://eclipse.org/aspectj`

we measure in terms of fault-finding ability of the oracles, that is, the amount of faults that CCOracless can reveal by signaling a failure. The oracles effectiveness depends and is affected by the test input. In our evaluation we consider both manually and automatically generated test suites. In detail, we answer the following research questions:

RQ1 To what extent are CCOracles effective in revealing failures when deployed within *manually-written* test cases? How do CCOracles compare with oracles (assertions) defined by developers in terms of fault-revealing effectiveness?

RQ2 To what extent are CCOracles effective in revealing failures when deployed within *automatically generated* test inputs? How do CCOracles compare with implicit oracles automatically generated in terms of fault-revealing effectiveness?

3.2.1 Evaluation Setup

To answer the research questions, we conducted an empirical evaluation on subjects, we selected classes from the following three open-source, publicly available Java libraries:

Guava is a core library developed by Google and used in many of their Java projects. Guava offers classes supporting caching, concurrency, string processing, and many other tasks. Guava also extends the set of collections and data structures provided by the Java standard library.[9]

Joda-Time is a library that improves in many respects the support for dates and times provided by the standard Java library.[10]

GraphStream is a library supporting the creation, manipulation, and analysis of dynamic graphs.[11]

From each library we selected concrete classes. From Guava and Joda-Time we selected classes for which we had a set of equivalences that was derived in previous research work [13]. We selected two more classes from the GraphStream library. For the selected class for which we had no equivalences, we manually derived a set of equivalences. Table 3.3 lists the subject classes along with the number of methods and of equivalences we found for every class. Overall, we selected 18 classes from which we identified 529 equivalences.

For each subject, we considered both manually-written and automatically generated test suites, to address RQ1 and RQ2, respectively. We generated test suites with the test case generator Randoop [70]. Both *manually-written* and *generated* test suites contain oracles. Manually-written test suites have assertions defined by developers to check the outcome of test case executions. Generated test suites contain Randoop-generated implicit oracles that check the validity of general object-oriented properties, for instance, method `equals` must be reflexive, symmetric, and transitive.

To evaluate the effectiveness of CCOracles we seeded artificial faults into the subject classes,with systematic mutation analysis [29, 79]. Although mutants might not strongly resemble real faults, recent studies indicate a positive correlation between real fault detection and mutation detection effectiveness of a test suite independently of code coverage [45]. In

[9]https://github.com/google/guava
[10]http://www.joda.org/joda-time
[11]http://graphstream-project.org

Table 3.3. CCOracles Evaluation Subjects and Number of Available Equivalences

Subject Class		**Methods**	**Equivalences**
Guava	ArrayListMultimap	25	29
	ConcurrentHashMultiset	27	32
	HashBiMap	20	16
	HashMultimap	24	29
	HashMultiset	26	30
	ImmutableBiMap	25	19
	ImmutableListMultimap	30	34
	ImmutableMultiset	32	50
	LinkedHashMultimap	24	30
	LinkedHashMultiset	26	31
	LinkedListMultimap	24	29
	TreeMultimap	26	28
	TreeMultiset	35	37
Joda-Time	DateMidnight	118	20
	DateTime	153	27
	Duration	44	6
GraphStream	SingleGraph	107	41
	MultiGraph	107	41

other words, mutation analysis is considered an effective technique to evaluate and compare the effectiveness of different test suites. We generated mutants of each subject class by means of Major, an automatic framework for mutation analysis [44, 46]. With Major, we mutated each subject class obtaining several mutants. We then tested each mutant with both the manually-written and the Randoop-generated test suites, and recorded the test suites and oracles that *kill* mutants.

In summary, for each selected subject we investigated RQ1 and RQ2 using the following process:

- We seeded faults mutating the subject class (and its superclasses) with Major. We mutated also super-classes to reflect the fact that faults could be located in inherited methods. Thus, we obtained a set of mutants (versions).
- We measured the effectiveness of CCOracles with *manually-written* test suites as follows:
  - Among all the mutants, we selected the mutants whose seeded mutation is *covered* (executed) by the execution of at least one test case of the manually-written test suite.
  - We tested each selected mutant with the manually-written test suite with and without CCOracles. We recorded the mutants killed by executing the test suite, and the oracles,

either manually-written or CCOracle, that contribute in killing the mutant.

- We measured the effectiveness of CCOracles with *generated* test suites as follows:
  - We generated a test suite for each mutant with Randoop, specifying the subject class as the test target to Randoop, and by setting 90 seconds as time budget for the generation. We discarded mutants for which Randoop fails to generate a test suite.
  - Among all the mutants, we selected the mutants whose seeded mutation is *covered* (executed) by the execution of at least one test case of the generated test suite.
  - We tested each selected mutant with the generated test suite with and without CCOracles. We recorded the mutants killed by executing the test suite, and the oracles, either Randoop generated or CCOracle, that contribute in killing the mutant.

Since we are interested in the fault-finding effectiveness of *the oracles*, we discard mutants whose corresponding mutations are not covered (executed) by the test cases, since the amount of not covered mutations relates to limitations of the test suites, and not with the effectiveness of the oracle that cannot detect faults that are not executed with the test suite.

During test suites execution, we activated an aspect (corresponding to an equivalence) at time, to better measure the effectiveness of the oracles, by identifying which oracle kills which mutant.

3.2.2 Evaluation Results

Table 3.4 reports the results of the empirical evaluation. For each subject class, and for both the manually-written and automatically generated test suites, the table reports the number of selected mutants, the number of mutants killed by implicit or CCOracles, and the number of mutants killed by both implicit and CCOracles.

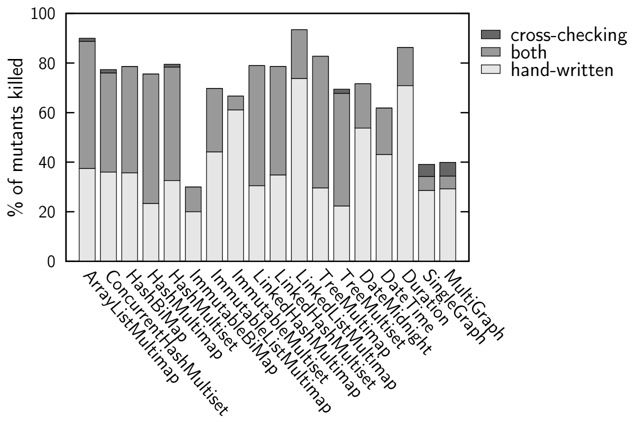


Figure 3.6. Mutants killed with manually-written tests

To answer *RQ1* we embedded CCOracles into manually-written test suites. As shown by results in Table 3.4 and more clearly depicted by Figure 3.6, CCOracles are effective when combined with manually-written test cases. More precisely, CCOracles correctly kill 28% mutants on average, while CCOracles kill 31% mutants. We do not expect CCOracles to spot all the

Table 3.4. CCOracles Empirical Evaluation Results

	Hand-Written Tests				Generated Tests			
		mutants killed by				mutants killed by		
Subject Class	**selected mutants**	**hand-written**	**cross-checking**	**both**	**selected mutants**	**implicit**	**cross-checking**	**both**
ArrayListMultimap	80	30	1	41	30	7	11	0
ConcurrentHashMultiset	150	54	2	60	89	0	64	0
HashBiMap	14	5	0	6	12	4	1	0
HashMultimap	90	21	0	47	40	7	12	0
HashMultiset	83	27	1	38	66	0	35	0
ImmutableBiMap	30	6	0	3	13	2	1	1
ImmutableListMultimap	43	19	0	11	22	1	8	0
ImmutableMultiset	72	44	0	4	49	6	5	0
LinkedHashMultimap	95	29	0	46	26	11	10	0
LinkedHashMultiset	89	31	0	39	53	0	33	0
LinkedListMultimap	61	45	0	12	24	2	4	0
TreeMultimap	81	24	0	43	35	1	10	0
TreeMultiset	121	27	2	55	113	0	37	0
DateMidnight	134	72	0	24	81	2	11	0
DateTime	181	78	0	34	148	0	8	0
Duration	152	119	0	26	146	2	26	0
SingleGraph	248	71	12	14	243	0	10	0
MultiGraph	253	74	14	13	254	0	11	0
Total	**1977**	**776**	**32**	**516**	**1444**	**45**	**297**	**1**

seeded faults, since not every method has a redundant equivalent method to cross-check its execution, and the equivalence check may still fail to identify the difference in the mutant's execution produced by the mutation. Still, CCOracles kills almost one- third of the mutants. Therefore, we can affirm that CCOracless are effective in identifying seeded faults. Unsurprisingly, compared with manually-written assertions, CCOracles are less effective. In fact, developers assertions identify 66% of the mutants (70% on average). This was expected, since not every facet of the outcome of a test can be checked with a CCOracle, and the intrinsic redundancy (although present) may not be enough to identify some failures. However, CCOracles kills 32 mutants (2% of the total) that go unnoticed by manually-written assertions. Thus CCOracless well complement manually-written oracles, being able to kill mutants that assertions written by expert developers do not kill. This means that CCOracless can increase the fault-finding effectiveness of manually-written test suites.

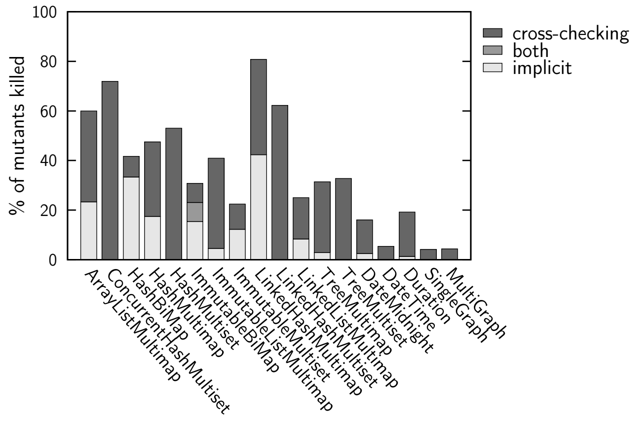


Figure 3.7. Mutants killed with generated tests

To answer *RQ2* we augmented Randoop-generated test suites with CCOracles. The results reported in Table 3.4 and graphically illustrated in Figure 3.7 show that CCOracless are effective when combined with automatically generated test suites. In fact, CCOracles kills 21% mutants, 27% on average, while implicit oracles kill only 3%, 10% on average. Therefore, CCOracles substantially improve automatically generated test suites that rely on state-of-the-art implicit oracles, and improves automatic testing, without requiring extra effort. During the experiments with generated test suites, CCOracles detected a real bug in the implementation of the class `SingleGraph` of the GraphStream library. The bug has been reported to and confirmed by the developers.[12] Methods `SingleGraph.removeNode(Node node)` and `SingleGraph.removeNode(int nodeId)` are supposed to be equivalent, both removing a node from a graph given a reference to the node to be removed or its identification number. The two methods were actually not behaving equivalently on the following test input:

```
SingleGraph graph1 = new SingleGraph("graph1");
Node node1 = graph1.addNode("node1");
SingleGraph graph2 = new SingleGraph("graph2");
Node node2 = graph2.addNode("node2");
graph1.removeNode(node2);
```

[12]`https://github.com/graphstream/gs-core/issues/109`

The test creates two graphs (`graph1` and `graph2`) with two nodes (`node1` and `node2` respectively). Then the test tries to remove `node2` from `graph1`. On line 5 a CCOracle is deployed checking the equivalence between `graph1.removeNode(node2)` and `graph1.removeNode(node2.getId())`. The actions should obviously have no consequences on `graph1` since `node2` is not a node of `graph1`. Instead, `graph1.removeNode(node2)` wrongly removes `node1` from `graph1` while `graph1.removeNode(node2.getId())` does not. The difference in the state of `graph1` is detected by the CCOracle that thus reports a failure. The fact that `SingleGraph.removeNode(Node node)` deletes the wrong node from the receiver graph is clearly an implementation bug.

The fact that CCOracles can detect real faults confirms their effectiveness when used in combination with generated test suite.

All the results reported in Table 3.4 and in Figures 3.6 and 3.7 are true alarms, meaning that CCOracles correctly signaled failures when killing mutants. We inspected all the failure reports produced by CCOracles during the evaluation. Out of 846 mutants killed by CCOracles, only 2 (2‰) are false alarms, that is, the CCOracle generated spurious alarms when the test cases should pass. In both cases, false alarms are generated for the subject class `ImmutableMultiset` of Google Guava. The reason for such false alarms is that cloned object tree contains object nodes with different hash codes. Different hash codes can cause differences in the behavior of the equivalent sequences, differences that are revealed by the equivalence check procedure of the cross-checking oracles. In essence, CCOracles may in rare cases produce spurious results when applied to objects whose behavior depends on hash code.

Overall we can affirm that the additional checks we perform in the cross-check execution phase of CCOracles do not suffer from false alarms (see Section 3.1.2). Thus, CCOracles are automatically generated oracles that can be applied without high false alarm rate.

3.3 Limitations and Threats to Validity

The effectiveness of CCOracles naturally depends on the amount of intrinsic redundancy (encoded as redundant sequences) that the SUT contains. We did not evaluate the presence of intrinsic redundancy in software, as it has been proven by existing research work [16, 61]. The empirical evaluation demonstrates that, whenever the SUT is intrinsically redundant at method call level, such redundancy can be used to automatically derive effective test oracles.

As a technique, the main limitation of CCOracles is the kind of faults that CCOracles aim to detect. CCOracles focus on *functional faults*, and they are not intended and designed to detect non-functional faults. This stems from the nature of redundant sequences, from which CCOracles are generated. Redundant sequences do not capture non-functional properties. In other words, two sequences that produce equivalent results in different ways (for example with different execution time or memory consumption) are considered equivalent and then redundant. Therefore, CCOracles ignore differences in the non-functional behavior of sequences, and they do not identify faults causing performance degradation, memory leaks and similar. The use of intrinsic redundancy to detect non-functional errors has not been explored yet and is outside the scope of this thesis.

The current prototype has limitations that could affect its effectiveness depending on characteristics of the SUT. The cross-check execution mechanism does not execute the two redundant sequences in complete isolation. Interferences between the executions of the redundant sequences may happen, for instance, through the static shared state or shared resources like files and directories. In practice, the execution of one sequence may interfere with the other invalidating the pass/fail answer produced by the oracle. In our evaluation we did not witnessed interferences, but they may appear when CCOracles are used to test different classes.

Another limitation of the current implementation is the lack of support for I/O operations. More in detail, CCOracles do not check the result of I/O operations during the equivalence check. This may lead to missed alarms when two supposedly redundant sequences produce different outcomes and the outcomes are results of I/O operations. For example, consider the case of two redundant sequences producing a file as result. Being redundant, the output files have to be identical. If the files have different content, it means there is an implementation fault invalidating the redundancy. The CCOracle checking the sequences would not be able to detect the difference and therefore would not report the failure.

The results of the empirical evaluation we performed have limitations that impact both their internal and external validity. Threats to the internal validity may derive from implementation errors in the mechanisms that constitute a CCOracle. Errors in the automatic deployment, cross-check execution or equivalence check may have introduced spurious outcomes affecting the overall results of the evaluation. To mitigate the threat we carefully tested the implementation and we manually investigated the outcomes of the empirical evaluation and we fixed all the errors we were able to find in our implementation. Furthermore, a complete replication package available with the prototype implementation of CCOracles and the experimental infrastructure is publicly available.[13] Sampling bias is a threat to the external validity of the results. In our evaluation, subject classes are not randomly selected from projects that have similar characteristics: they are open-source, written in Java, and extensively used as building blocks in many other projects. Thus, subject classes may not represent well the entire population of Java classes. This, and the rather limited number of subject classes, could limit the generalizability of the results that is CCOracles may show a different fault-finding effectiveness when applied on other Java classes with a similar presence of intrinsic redundancy and faults.

Another threat to validity is the fact we used synthetic faults (mutants) to validate the effectiveness of CCOracles. Just et al. show with an empirical evaluation that synthetic faults are good representative of real faults in the sense that a statistically significant correlation exists between fault- and mutant-detection effectiveness of a test suite [45]. Although this correlation exists, some real faults do not have a synthetic counterpart. This means that CCOracle may show a lower fault-finding effectiveness on real faults than the one shown with synthetic faults.

[13]`http://star.inf.usi.ch/star/cross-check`

Chapter 4

Oracles from Natural Language Documentation

Software systems are commonly documented with code comments. Many modern languages provide ways to document functions, method, classes, and modules with semi-structured comments that can be used to generate a nicely formatted documentation. Such comments are commonly present in software systems.

We propose a technique to derive test oracles by exploiting code comments. In particular we propose an approach called Toradocu that translates semi-structured comments, written in natural language, into procedure specifications. Toradocu uses the procedure specifications to generate test oracles and deploy them into existing test suites. The chapter presents Toradocu and discusses the results of an extensive empirical evaluation of the effectiveness of Toradocu.

Software systems are documented at different levels, from code comments embedded into source code, to procedure documentation and user guides. Code comments embedded in source code and procedure documentation are commonly encoded in markup languages, like Javadoc, Doxygen and Sphinx. Javadoc[1] is the standard way to document Java code and is included in Java since its first release in 1995. Doxygen[2] is the most common way to document C/C++ source code, but it also supports other languages like Fortran, Java, and Python. Sphinx[3] is used to document Python code and also supports C/C++ and JavaScript among other languages. Code comments are widespread and commonly used. Virtually every non-trivial project is documented with such kind of documentation.

Documentation captures the design goals, and frequently describes the expected behavior of the code fragment it documents. In general, a comment may document different code elements: statements, fields, methods, classes, packages. Java APIs (constructor and methods) are commonly documented with Javadoc comments, whose typical structure is shown in Figure 4.1. Javadoc comments are particularly interesting because they represent a sort of informal procedure specification that expresses the expected behavior of the method they document. More precisely, Javadoc comments express the behavior that a user of the documented API should expect.

[1]`http://docs.oracle.com/javase/9/javadoc/javadoc.htm`
[2]`http://www.doxygen.org`
[3]`http://www.sphinx-doc.org`

```
 1  /**
 2   * Returns the index of the first matching BMP character in a character sequence, starting from a
 3   * given position, or {@code -1} if no character matches after that position.
 4   *
 5   * <p>The default implementation iterates over the sequence in forward order, beginning at {@code
 6   * start}, calling {@link #matches} for each character.
 7   *
 8   * @param sequence the character sequence to examine
 9   * @param start the first index to examine; must be nonnegative and no greater than {@code
10   *   sequence.length()}
11   * @return the index of the first matching character, guaranteed to be no less than {@code start},
12   *   or {@code -1} if no character matches
13   * @throws IndexOutOfBoundsException if start is negative or greater than {@code sequence.length()}
14   */
15  public int indexIn(CharSequence sequence, int start) { ... }
```

Figure 4.1. Javadoc comment of method `CharMatcher#indexIn(CharSequence, int)` from Google Guava library

Figure 4.1 shows the Javadoc comment of method `CharMatcher#indexIn(CharSequence, int)` (we omit the implementation of the method for brevity). Method `indexIn` belongs to class `CharMatcher` from Google Guava library (version 23.2). The Javadoc comment is composed of two optional blocks: The description block highlighted in gray in Figure 4.1, and a list of block tags highlighted in light blue in Figure 4.1. A description block is a free-form natural language text that generally describes the behavior of the commented method. The text can contain HTML formatting tags like <p> or inline tags like `{@code start}` and `{@link #matches}` that respectively identify source code fragments and links to other documentation fragments, resources, and documents. A Javadoc comment like the one reported in Figure 4.1 describes in detail the behavior of the documented method `indexIn`. For example, lines 9–10 indicate that the integer value used as second argument must be greater or equal zero and less than the length of the sequence used as first argument. This "pre-condition" is also described on line 13, where block tag `@throws` specifies the type of the exception that is thrown when the pre-condition is not satisfied.

Program documentation is a valuable source of information, and software engineering researchers have studied comments with different goals.

Comment Quality Analysis The quality of comments has a direct impact on developers who have to understand, use, modify, and improve the commented code. JavadocMiner automatically assesses the quality of Javadoc comments by computing several metrics related to both the quality and readability of the comment text and the consistency between comment and documented source code [50]. JavadocMiner detects low quality comments, to focus the efforts in improving code documentation. With the same goal, Steidl and colleagues propose two metrics to evaluate the quality of Javadoc comments [90]: coherence and length. The coherence between a method description and the documented method name is defined as the percentage of words in common between the method name and the method description. The length is the number of words in a comment. These metrics identify critical method comments. Pawelka and Juergens investigated the natural language used for source code identifiers (for example, variable names, method names, etc.) and comments [74], and found that open-source projects are consistently written in English, while closed-source projects are more mixed, combining English with other languages.

Source Code Generation Source code generation approaches aim to produce methods from the Javadoc documentation. Zhai et al. use Javadoc method comments to provide a testing equivalent implementation of the commented method that is simpler than the original [101]. The generated implementation can be useful when the original implementation is not available but required for specific kinds of analysis. Also, the generated implementation is generally simpler, meaning that analysis techniques should be more efficient on the generated implementation than the original one. The generated implementation may inaccurately mimic the original implementation.

Code Summarization Code summarization aims to generate natural language description of the behavior of a given piece of code. Sridhara et al. detect high level actions within a Java method and derive a natural language (English) documentation for the method describing the method behavior [89]. The approach considers only the source code of the to-be-commented method, thus ignoring the context, i.e., how the to-be-commented method is generally invoked and used. The approach generates summaries that include information about how the summarized method is used and the context [63].

Specification Mining and Bug Finding Using documentation to detect errors in the documented software system is a quite old idea. The early attempts exploited documentation written in formal languages for testing and verification purposes [30, 72, 77, 78]. More recently researches focused on natural language documentation that is way more common than formal specifications. Specifications inferred from natural language documentation are sometimes used to find inconsistencies between the documentation and the implementation of a software system. An inconsistency stems from an error in either the documentation or the code (or both). A recent empirical study shows that the most common repair action to fix a bug in Java systems is changing the documentation [104].

API documentation often includes code examples and refers to code elements such as classes, interfaces, and so on. It may happen that the documentation is not updated after a code change, leading to confusing, inconsistent and useless API documentation. DOCREF automatically detects wrong "code names" in comments, such as a non-existing class name, by extracting code names from comments (even in code snippets embedded in code comments) and checking whether the mentioned code names are correct, i.e., they correspond to existing code elements like types, methods, fields, parameters, etc [103].

Text2Policy derives Access Control Policies (ACP) from natural language software documentation. Text2Policy transforms English sentences like "The Health Care Personnel (HCP) does not have the ability to edit the patient's security question and password." to a formal specification in eXtensible Access Control Markup Language (XACML). Text2Policy helps developers in identifying inconsistencies between ACP and functional requirements expressed in natural language like use cases [99].

INDICATOR analyzes the API documentation and the implementation of a web service to derive constraints on parameters. INDICATOR validates derived constraints by means of testing with an average precision of 94.4% and an average recall of 95.5% [98].

Doc2Spec infers resource specifications from Javadoc comments. Specifications inferred by Doc2Spec describe how resources should be manipulated. For example, for a file resource, Doc2Spec could infer that *read* actions should always happen before a *close* action and that no actions can be performed after the *close* action. The precision of the specification created by

Doc2Spec depends on the quality of the documentation of the resource usage model [105].

Tan and colleagues exploit code comments to detect bugs: Their approach detected several unknown lock-related bugs in Linux [92].

Rubio-González and Liblit's approach detects mismatches between possible error codes produced by a Linux (file-related) system call and the list of documented error codes that system call is supposed to produce. The technique exploits a very simple pattern-based approach to extract the list of possible, documented error codes from Linux manual pages [83].

iComment detects both bad comments and bugs by analyzing code comments in programs written in C/C++. iComment focuses on lock-related issues. To understand comment semantics, iComment combines natural language processing (NLP) with (supervised) machine learning. Comments are classified according to a small set of rules that represent lock-related constraints. iComment detects code-comments inconsistencies by statically analyzing the code to check whether an inferred rule is violated [91].

aComment reports concurrency bugs related to the interrupt context in OS code inferring the correct expected behavior from the analysis of code comments and source code. More in detail, aComment analyzes from comments and code the requirements [93].

ALICS infers method specifications from Javadoc comment exploiting NLP techniques, by generating specifications as first order logic expressions. For example, given the block tag comment `@param x cannot be null`, ALICS produces the predicate `cannot_be(x, null)`. ALICS defines the semantics of generated FOL expressions only for some of them. In particular, ALICS supports `String` and `Integer` classes, `null` checks, `@return` and `@throws` block tag comments. This means that specifications produced by ALICS are not readily usable, but they must be manually translated in a suitable format. For instance, the predicate `cannot_be(x, null)` may be translated into the Java code `x != null` [71].

In line with ALICS, @tComment extracts method specifications from Javadoc comments [94]; differently from ALICS, @tComment focuses on four specific classes of method behaviors that relate with null arguments, and can infer the following properties for a method that receives as input `null` as argument: (i) Null Normal—no exception is expected; (ii) Null Any Exception—an exception is expected; (iii) Null Specific Exception—a specific exception type is expected; (iv) Null Unknown—expected behavior is unknown. For example, from the comment `@param x cannot be null`, @tComment infers `x==null => exception`. @tComment works heuristically: Comments do not predicate on the behavior of the method, rather state preconditions. The authors of @tComment found out that the proposed heuristic commonly matches developers' intention. @tComment is relatively simpler than ALICS: It does not exploit complex NLP techniques, rather, it relies on pattern matching. Specifications generated with @tComment are expressed in Java and can be directly used without any additional translation. @tComment embeds @Randoop, a modified version of the Randoop test case generator [70] to generate test inputs and oracles that check the properties inferred from the Javadoc documentation.

Zhou et al. address the problem of inconsistency between code and documentation by proposing a technique to automatically detect errors in the Javadoc documentation of methods, assuming correct method implementations [106]. Zhou et al.'s approach detects the following types of errors in comments about constraints on method inputs relying on pattern matching: (i) nullness not allowed—none of the arguments can be null; (ii) nullness allowed—arguments can be null; (iii) type restriction—arguments must be of specific types; (iv) range limitation—arguments must be within a specific range. The approach first infers properties of these type by statically analyzing the source code, and by analyzing the Javadoc comments using NLP techniques such as POS tagging and dependency parsing, and then checks the consistency between formulas derived from code and documents with the Z3 SMT solver [26].

Summing up, documentation is commonly produced during development to express the intended behavior of the program, and there exist several techniques to automatically derive the developers' intentions from the documentation, and report inconsistencies between the documentation and the implementation. However, existing techniques are limited. Approaches that infer specific properties like @tComment [94], Zhou et al. [106], aComment [93], and iComment [91] focus on a small set of properties and do not deal with many properties that documentation may express. General approaches lack semantics, for example ALICS generates specifications whose semantics is defined only in few cases [71], thus ALICS specifications are more akin to a comment in a different language than a specification describing a software behavior.

We overcome such limitations, by providing a technique that generates test oracles from Javadoc method documentation and that (i) is more general than existing ones, i.e., it is able to understand more of the entire comment semantics, and (ii) produces specifications with a defined semantics that can readily be used for the oracle generation (among many other usages such as program comprehension or refactoring).

4.1 Javadoc Documentation

Source code is commonly documented with natural language comments. In particular, procedure (method) comments usually describe the procedure input, output and behavior. In other words, procedure comments—also called APIs documentation—describe preconditions, postconditions and exceptional postconditions of procedures in natural language.

Javadoc is the de facto standard to document Java code, and is similar to many other tools for documenting programs written in different programming languages like Doxygen for C/C++ and Sphynx for Python. Figure 4.1 shows the typical structure of a Javadoc method comment that document preconditions and postconditions in block tag comments: `@param` for preconditions, `@return` for postconditions, `@throws` and `@exception` for exceptional postconditions.[4]

`@param` comments describe the semantics of the input parameters of the documented code, and document constraints on the input. Figure 4.2 shows a simple example of a Javadoc comment that indicates whether a `null` value is acceptable for the parameters (Figure 4.2, line 5). The comment corresponds to the precondition `a != null` (Figure 4.2, line 10). Another common example of conditions expressed in Javadoc are comments that constrain numerical inputs, as the comments at lines 5–7 of Figure 4.3 that correspond to the preconditions at lines 12 and 13 of Figure 4.3. In general, `@param` comments list preconditions of the documented method, that is, conditions that should hold when method is invoked to obtain a correct behavior.

`@return` comments describe the expected outputs of the methods, and correspond to postconditions. For example, the comment at line 6 of Figure 4.2 asserts that the return value of the method cannot be null, and corresponds to the postcondition reported at line 11 of Figure 4.2. As another example, the comment at line 4 of Figure 4.4 says that the return value must be an *empty* map, and corresponds to the postcondition at line 8 of Figure 4.4. Sometimes `@return` comments express postconditions as relations between method inputs and outputs. For example, the `@return` comment in Figure 4.5 asserts that the method returns `true` if the receiver object graph does not already contains the edge given as input.

`@throws` comments document the exceptions that a method can throw and the condition to throw the exceptions. For example, the comment at line 5 of Figure 4.5 states that a

[4] `@exception` is an alias for `@throws`. Hereafter, we mention only `@throws` block comments. Everything we discuss equally applies to `@exception` comments.

```
/**
 * Returns a representation of {@code a} as an instance of type {@code B}. If {@code a} cannot be
 * converted, an unchecked exception (such as {@link IllegalArgumentException}) should be thrown.
 *
 * @param a the instance to convert; will never be null
 * @return the converted instance; <b>must not</b> be null
 */
protected abstract B doForward(A a);

PRE: a != null
POST: result != null
```

Figure 4.2. Javadoc comment of method `Converter#doForward` from Google Guava library and its preconditions and postconditions.

```
 /**
  * [...]
  *
  * @param funnel the funnel of T's that the constructed {@code BloomFilter} will use
  * @param expectedInsertions the number of expected insertions to the constructed {@code BloomFilter};
  *    must be positive
  * @param fpp the desired false positive probability (must be positive and less than 1.0)
  * @return a {@code BloomFilter}
  */
 public static <T> BloomFilter<T> create(Funnel<? super T> funnel, int expectedInsertions, double fpp)

PRE: expectedInsertions > 0
PRE: fpp > 0 && fpp < 1.0
```

Figure 4.3. Javadoc comment of method `BloomFilter#create` from Google Guava library and its preconditions and postconditions.

`NullPointerException` is expected when one of the method arguments is `null`, corresponding to the postcondition reported at line 11 of Figure 4.5.

In the examples, we express pre- and post-conditions (both normal and exceptional) as Java expressions. The formalism used to encode specifications has a direct impact on the usability of the specifications. While first-order logic expressions à la ALICS are not directly usable [71], Java specifications are readily usable since their semantics is known.

In this chapter we propose Toradocu, an approach to automatically infer Java procedure specifications from Javadoc comments (in particular constructor and method comments), and automatically generate test oracles from such specifications. The specifications in Figure 4.2, Figure 4.3, Figure 4.4, Figure 4.5 are examples of specifications that Toradocu is able to infer. In

```
/**
 * [...]
 *
 * @return an empty Bag
 */
public static <E> Bag<E> emptyBag()

POST: result.equals(target.EMPTY_BAG) == true
```

Figure 4.4. Javadoc comment of method `BagUtils#emptyBag` from Apache Commons Collections library and its postcondition.

```
/**
 * [...]
 * @return {@code true} if this graph did not already contain the specified edge.
 * [...]
 * @throws NullPointerException if any of the specified vertices is {@code null}.
 * [...]
 */
boolean addEdge(V sourceVertex, V targetVertex, E e)

POST: !receiver.containsEdge(sourceVertex, targetVertex) → result == true
EXC_POST: sourceVertex == null || targetVertex == null → java.lang.NullPointerException
```

Figure 4.5. Javadoc comment of method `Graph#addEdge` from JGraphT library and its postcondition

a nutshell, Toradocu translates Javadoc natural language comments into Java procedure specifications. Such specifications can be used for different purposes, like program comprehension and verification. We use Toradocu specification to automatically generate test oracles, the main theme of this Ph.D thesis.

4.2 Toradocu

Toradocu derives procedure specifications from `@param`, `@return` and `@throws` Javadoc comments, and generates test oracles from the derived specifications.

Figure 4.6 shows the overall architecture schema of the approach. Given as input the source code of the SUT, the binaries of the SUT, and a set of test cases for the SUT, Toradocu augments the test cases with oracles generated from the Javadoc comments of the input source code. Toradocu works in two phases. In the first phase, Toradocu infers specifications from the Javadoc comments: the *Javadoc extractor* identifies Javadoc comments from existing source given as input to Toradocu, and the *comment translator* converts the identified comments into specifications. In the second phase, the Toradocu *oracle generator* generates test oracles from the inferred specifications, and deploys the oracles into the input test cases. The next sections describes in detail the Toradocu components.

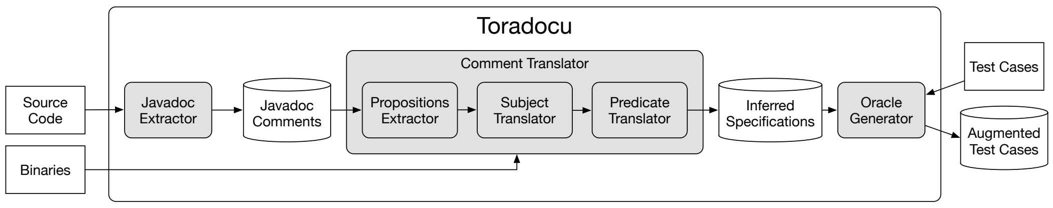


Figure 4.6. Toradocu Architecture

4.2.1 Javadoc Extractor

Given the source code of the SUT and a target class, the Javadoc extractor identifies `@param`, `@return`, and `@throws` Javadoc comments. More specifically, the Javadoc extractor identifies all the `@throws` comments declared in the documented methods, or inherited from the target class.

Toradocu ignores Javadoc description block and tags different than `@param`, `@return`, `@throws`. For example, when processing the comment `@throws NullPointerException if any of the specified vertices is @code null` (Figure 4.5, line 11), the Javadoc extractor identifies the following information:

method: addEdge(V sourceVertex, V targetVertex, E e)
exception: java.lang.NullPointerException
comment: if any of the specified vertices is {@code null}

The Javadoc extractor infers the fully qualified name of the expected exception from the input source code. The Javadoc extractor is implemented as a custom parser of the Java input source code built on top of the JavaParser library.[5]

4.2.2 Comment Translator

The comment transaltor translates the `@param`, `@return`, `@throws` Javadoc comments identified by the Javadoc extractor into procedure specifications. As an example, by processing the natural language comment `@throws [...]  if any of the specified vertexes is {@code null}` (Figure 4.5, line 5), the comment translator derives the specification `sourceVertex==null||targetVertex == null → [...]` (Figure 4.5, line 11).

The comment translator works in three phases:

1. *Proposition extraction*: after a simple preprocessing step, Toradocu identifies propositions (subject-predicate pairs) in the natural language comments by means of NLP techniques.

2. *Subject translation*: Toradocu translates the subject of the identified proposition into a source code elements, by identifying the Java code elements corresponding to the item identified in the natural language comment.

3. *Predicate translation*: Similarly, Toradocu translates the predicate of the identified proposition into source code elements.

Algorithm 1 shows the pseudocode of the algorithm we designed to translate Javadoc comments into boolean expressions. Algorithm 1 first preprocesses the input comment text (Algorithm 1, line 3). The comment translator preprocesses `@return` comments, by removing the initial "if" from the comment, and transforming a dependent clause into a main clause. It preprocesses the other types of comments by adding end-of-sentence periods where missing, being standard and recommended practice not to end comments with period punctuation mark if the comment consists of a single sentence.[6] The preliminary preprocessing facilitates the parsing of the sentence.

The comment translator differentiates between `@return` and other kind of comments, since often `@return` comments aggregate up to three distinct concepts of information that Toradocu translates independently. This is illustrated with the comment "return true if this graph did not already contain the specified edge" (line 3 of Figure 4.5) that expresses three concepts that we call *guard*, *true property*, and *false property* (sometimes implicit). True and false properties predicate on the return value, by commenting about the return values. In the example, the true property is "true", meaning that the result should be `true`, and the false property is expressed implicitly as the negation of the true property ("false"). The guard is a boolean condition that

[5] `http://javaparser.org`
[6] `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`

Algorithm 1 Comment Translator

```
    // Translates the English text of a comment extracted by the Javadoc extractor into a (pre- or post-) specification.
 1: INPUT: comment (Javadoc block tag comment), method (method commented by comment)
 2: function TRANSLATE-COMMENT(comment, method)
 3:     preprocessedComment = PREPROCESS-COMMENT(comment)
 4:     if c.kind = @return then
 5:         guard = GET-GUARD(c.text)
 6:         trueProperty = GET-TRUE-PROPERTY(c.text)
 7:         falseProperty = GET-FALSE-PROPERTY(c.text)
 8:         TRANSLATE(guard, method)
 9:         TRANSLATE(trueProperty, method)
10:         TRANSLATE(falseProperty, method)
11:     else
12:         TRANSLATE(c.text, method)
13:     end if
14: end function

15: function TRANSLATE(text, method)
16:     propositions = EXTRACT-PROPOSITIONS(text)
17:     javaExpression = ‘’
18:     for all proposition p = ⟨subject, predicate⟩ in propositions do
19:         javaSubjects = MATCH(subject, method)
20:         for all javaSubject in javaSubjects do
21:             javaPredicate = MATCH(predicate, method)
22:             MERGE-INTO-EXPRESSION(javaSubject, javaPredicate, javaExpression)
23:         end for
24:     end for
25:     return javaExpression
26: end function

27: function EXTRACT-PROPOSITIONS(text)
28:     propositions = ∅                 ▷ Data structure that considers grammatical conjunctions among propositions.
29:     for all sentences s in t do
30:         semanticGraph = GET-SEMANTIC-GRAPH(s)                         ▷ Relies on the Stanford Parser.
31:         subjectList = IDENTIFY-SUBJECTS(semanticGraph)
32:         for all subjects subj in subjectList do
33:             proposition = ⟨subj, IDENTIFY-PREDICATE(subj, semanticGraph)⟩
34:             propositions.add(proposition)
35:         end for
36:     end for
37:     return propositions
38: end function

39: function MATCH(text, method)
40:     candidates = COLLECT-CANDIDATES(method)
41:     REMOVE-STOP-WORDS(text)
42:     LEMMATIZATION(text)
43:     translation = PATTER-MATCHING(text)
44:     if translation is empty then
45:         translation = LEXICAL-MATCHING(text, candidates)
46:     end if
47:     if translation is empty then
48:         translation = SEMANTIC-MATCHING(text, candidates)
49:     end if
50:     return translation
51: end function
```

specifies the conditions for the properties to hold, either the true or the false property. In the example the guard is "if this graph did not already contain the specified edge".

In the case case of `@return` comments, the comment translator first identifies guards, true and false properties through pattern identification (Algorithm 1, lines 5–7), and then translates the identified elements into Java predicates (Algorithm 1, lines 8–10). In the case of `@param` and `throws` comments, the comment translator directly translates the identified elements into Java predicates (line 12). Toradocu identifies the propositions in the comments (Algorithm 1, line 16) as a first step of the comment translation.

Proposition Extraction

Toradocu understands the semantics of a Javadoc comment, by extracting subject-predicate pairs from the comment. In the NLP community, this activity is called *Information Extraction* (IE). Information Extraction recently evolved into *Open Information Extraction* (Open IE) [2, 3, 28, 33, 84]. While IE identifies specific information in a text, Open IE identifies general propositions or relations from the text. In general, the output of an Open IE technique is a list of tuples with two or more elements where each tuple describes a relation (proposition) in the input text. For example, given the text "Barack Obama served as the 44th President of the United States from 2009 to 2017.", Open IE may extract the proposition `(Barack Obama, served as the 44th President of the United States from 2009 to 2017)`. A proposition consists, at least, of a subject and a predicate. A subject is a noun phrase that the sentence is about, `Barack Obama` in the example, and the predicate is the remainder of the sentence, which says something about the subject, `served as the 44th President of the United States from 2009 to 2017` in the example. The predicate may contain different information, for example it may miss the `from 2009 to 2017` part, depending on the goal of the Open IE analysis. State-of-the-art Open IE techniques precisely extract relations, but do not support well relations interconnected with a grammatical relation such as "and", "or". For instance, given the comment "if foo or bar is null.", an Open IE tool may extract two propositions `(foo, is null)` and `(bar, is null)`, missing the connection between the two propositions. We designed and implemented a custom IE algorithm to identify propositions in Javadoc comments.

Proposition extraction transforms a Javadoc comment text into propositions, invoking function EXTRACT-PROPOSITIONS at line 16 of Algorithm 1. A single natural language sentence may contain multiple connected propositions, which we represent in a linked list. Elements of the list are the propositions, links between elements are the grammatical conjunctions connecting the propositions. A single Javadoc comment may be composed of multiple sentences. Propositions belonging to different sentences are heuristically joined with an "or" conjunction.

Function EXTRACT-PROPOSITIONS (Algorithm 1, line 27) relies on the Stanford Parser [57, 60] to process the natural language Javadoc comment, and identify propositions in the input text. Given the Javadoc comment text, the function first identifies the different sentences composing the whole comment using the *sentence splitting* functionality of the Stanford Parser. It then uses the Stanford Parser to produce a *semantic graph* for each of the sentences (line 30). A semantic graph is a representation of a sentence where the nodes are the words composing the sentence, and the edges are the grammatical relations between the words.[7] Given the semantic graph of a sentence, the condition translator traverses the graph and identifies subjects (line 31) and related predicates (line 33). If the sentence contains multiple subjects, function IDENTIFY-SUBJECTS (line 31) identifies them all. Function IDENTIFY-SUBJECTS also handles the case where the subject

[7] `http://nlp.stanford.edu/software/dependencies_manual.pdf`

is a compound noun, and considers the head noun plus the noun compound modifier as (one) subject of the sentence.

For each identified subject, function IDENTIFY-PREDICATE (line 33) traverses the semantic graph to identify the predicate of a subject, by looking for specific patterns, that is, specific chains of grammatical relations that commonly appear in Javadoc documentation. For example, the parts of the predicate of an active form sentence are *verb* and *complement*. Toradocu identifies as predicate only some words of the sentence. The IE approach in Toradocu is not complete, in the sense that it may not correctly extract the predicate in every possible English sentence, but is effective and adequate for common Javadoc comments.

Toradocu adds each identified proposition to a list of identified propositions (line 34), by considering the specific grammatical relation that joins the current proposition to the propositions already present in the list. Function EXTRACT-PROPOSITIONS returns the proposition list (line 37).

Subject and Predicate Translation

Toradocu translates subjects and predicates of the identified propositions (subject-predicate pairs) into Java expressions, by exploiting the references to the code in the Javadoc comments. Toradocu uses code elements of the documented code, like methods and parameters, as well as general Java constructs to translate subjects and predicates. For example, the Toradocu translator associates the subject "any of the specified vertices" of the `@throws` comment at line 5 in Figure 4.5 to the parameters `sourceVertex` and `targetVertex`, thus translating the single subject into two Java code elements, each of them with its own predicate. In general, a single Java element may be referred to either by name or by type name, as in this example. A single subject may refer to multiple Java elements. For example, the subject "argument" in the comment `@return [...] if an argument is null` refers to each parameter of the method.

The translation of a subject happens at line 19 of Algorithm 1. Toradocu translates the predicate corresponding to each subject (line 21), and merges the translation of subject and predicate into a well-formed Java expression (function MERGE-INTO-EXPRESSION).

Function MATCH translates the input text into Java elements. Function MATCH collects translation candidates from the source code by gathering *identifiers* and *types* from the code (line 40). Function MATCH translates a subject to:

A formal parameter of the documented method This is the most common, and we already exemplified it (Figure 4.5, line 5). Sometimes, Javadoc comments refer to parameters by means of their types. For example, the comment "if the collection is empty" may refer to a parameter of type `java.lang.Collection`. For this reason the algorithm identifies both identifiers and type names, and considers all supertypes: transitive superclasses and implemented interfaces.

A method of the class under test For example, the comment "the capacity of the container" refers to a private field of the class under test named `capacity`. In this case the field is not visible outside the class. Toradocu translates the subject `capacity` to the getter method `getCapacity()`. Toradocu only considers nullary non-void methods of the class under test.

The receiver object For example, in the comment "if the comparator is locked", the subject "the comparator" refers to the instance of the receiver object, instance of the class whose documented method belongs to, which implements the `Comparator` interface.

Toradocu first removes stop words from the input text (line 41), and applies a lemmatization to the input text (line 42), then it selects the best candidates for the translation by exploiting three different strategies applied sequentially: textual pattern, lexical, and semantic matching.

Textual Pattern Matching Toradocu tries to translate the input text using a *textual pattern matching* on a set of predefined patterns (line 43). The textual pattern matching tries to find a match for the input text with a set of predefined patterns that we derived looking into Javadoc comments. Table 4.1 reports an excerpt of the list of patterns currently supported in Toradocu. For instance, the predicate "is positive", which can be applied to numeric types (byte, short, int, long, float, double, and respective wrappers), produces the Java condition `subject>0`. When the subject element's type is a reference type, a non-primitive type, the only relevant pattern is a check whether an object is null: "is null" → `== null`.

Table 4.1. Comment Translation Selected Patterns

Comment Words	**Translation**
is/are positive	> 0
is/are negative	< 0
is/are < 1	< 1
is/are <= 0	<= 0
is/are true	== true
is/are false	== false
is/are null	== null

Lexical Matching If the pattern matching fails, Toradocu selects the Java elements with the smallest edit distance within a given threshold among all the possible candidates. As edit distance, Toradocu uses the Levenshtein distance extended with a new operation, the word removal, where a word deletion is a single edit action (Algorithm 2, lines 45, 54–71).

Semantic Matching If both the pattern and the lexical matching fails, Toradocu proceeds with *semantic matching* (line 48). For example, in the comment `@throws [...] if [...] is not found in the graph`, Toradocu identifies the predicate "is not found in the graph" that should be translated to "target.contains(args[0])". Both pattern and lexical matching fail in translating the predicate, since "is not found in the graph" and "contains" do not have many characters in common. However, their *semantics* is close. In the semantic matching (Algorithm 2, lines 72– 80), Toradocu exploits the semantics of input text and candidates to select the best candidates for the translation.

Toradocu relies on *word embedding* where words are mapped to numbers and embedded into a vector space. In the vector space, words with similar semantics are close together. In particular, Toradocu uses GloVe [75] as word vectors, to capture the semantic relation between pairs of single words. Since often predicates are composed of multiple words, we augmented GioVe with two algorithms that compute the semantic distance considering multiple word at once: *Vector Sum* [66] and *Word Mover's Distance* [52] (WMD). Vector Sum exploits linear algebra operations on vectors to solve word analogy tasks involving multiple words. The classic example is that “king” - “man” + “woman” is semantically equivalent (has no distance) to “queen”. Toradocu exploit Vector Sum computing the sum of all the term vectors to come up with a single concept, and applies it to both the input text to translate and the candidates. Then Toradocu measures the distance between the unique “concept” representing the input text and the “concept” representing a candidate. Toradocu uses this approach when the number of words in the text to translate is less than four. For texts with four or more words, Toradocu uses WMD, which computes the distance between two lists of words as the cumulative distance that words from a list has to travel in the vector space to exactly match words in the other list. Both Vector Sum and WMD reduce spurious translations, that would jeopardize the precision of the overall approach, with thresholds.

If Toradocu fails to find a match for a subject or predicate, the specific proposition has an empty translation.

After translating all propositions, Toradocu assembles the Java condition (Algorithm 1, line 22). To compose conditions, the algorithm decides whether the condition evaluates to `true` or `false`, by analysing the predicate: if the predicate contains the string “not” or “n’t” the condition is evaluated to `false` (`aList.isEmpty() == false`), otherwise to `true` (`aList.isEmpty()`).

Toradocu instantiates the common approach that we outlined in this section to translate both subjects and predicates with few simple differences. The most notable difference is the collection of candidates. While translating the predicate, Toradocu already knows the subject and exploits that information to collect candidates that are not collected during the subject translation. For example, when the subject is of a non-primitive type, for the predicate translation Toradocu also collects as candidates methods belonging to subject’s class.

4.2.3 Oracle Generator

The comment translator produces procedure specifications in the form of method pre- and post-conditions that are tuples of the form $\langle m, pre, post, excpost\rangle$ where:

- m is a method of the class under test;
- pre are preconditions of m: Java Boolean conditions that should hold before m is invoked.
- $post = \{\langle g, tp, fp\rangle\}$ are postconditions of m: when the Java Boolean condition g (guard) holds, the Java Boolean condition tp (true property) must hold. Otherwise, the Java Boolean condition fp (false property) must hold.
- $excpost = \{\langle e, c\rangle\}$ are postconditions of m: where e is the type of the expected exception and c is the Java Boolean condition that holds when the exception is expected.

The conditions pre, $post$, and $excpost$ may be empty.

The oracle generator processes these tuples to produce the oracles that it deploys into test cases, given as input to Toradocu. Input test cases can be either manually defined by developers or automatically generated with automatic test generators like EvoSuite [35] and Randoop [70].

To automatically create and inject test oracles into test cases, the oracle generator exploits aspect-oriented programming (AOP) [51] as implemented in AspectJ.[8] The oracle generator creates an AspectJ aspect working as oracle for each method m, for which Toradocu generates

[8] `http://www.eclipse.org/aspectj`

Algorithm 2 Comment Translator (Matching)

```
52: function PATTERN-MATCHING(text)
        // Translates input text using patter matching.
53: end function

54: function LEXICAL-MATCHING(text, candidates)
55:     matches = $\emptyset$
56:     threshold = 2
57:     minimumDistance = $\infty$
58:     for all candidate in candidates do
59:         distance = EDIT-DISTANCE(text, candidate)
60:         if distance <= threshold then
61:             if distance < minimumDistance then
62:                 minimumDistance = distance
63:                 matches = $\emptyset$
64:                 matches.add(candidate)
65:             else if distance == minimumDistance then
66:                 matches.add(candidate)
67:             end if
68:         end if
69:     end for
70:     return matches
71: end function

72: function SEMANTIC-MATCHING(text, candidates)
73:     matches = $\emptyset$
74:     if text.words.length > 3 then
75:         matches = WMD(text, candidates)        ▷ Relies on wmd4j library to compute Word Mover's Distance.
76:     else
77:         matches = VECTOR-SUM(text, candidates)                                   ▷ Relies on GloVe library.
78:     end if
79:     return matches
80: end function
```

at least a procedure specification, and uses the aspects to inject oracles into the input test suite. At runtime the aspects behave as oracles checking all the available specifications.

Given the aspect corresponding to the tuple $\langle m, pre, post, excpost \rangle$ and a test case that invokes method m, the generated oracle works as follows:

1. **Preconditions**: right before (each) invocation of m in the test case, the oracle checks whether all the preconditions in pre hold or not. If all the preconditions hold, the test execution continues. Otherwise, the execution terminates. In this case the oracle does not notify a failure, rather it signals a malformed test input.

2. **Postconditions**: before the invocation of m, the oracle checks all the guards g in postconditions $\langle g, tp, fp \rangle$. If g does not hold, the oracle marks the false property fp that is checked after the execution of m.

3. **Exceptional Postconditions**: before the invocation of m, the oracle checks if m may raise an exception, by evaluating the condition c of each exceptional postcondition $\langle e, c \rangle$ in $excpost$. If c holds, the throw of an exception of type e is an acceptable behavior of m. Every exception type e that is expected is collected to be used later on.

4. **Execution**: method m is executed.

5. **Postconditions Check**: after the execution of m, the oracle checks if m threw an exception. If no exception was thrown, the oracle verifies the postconditions that should be valid, that is, the true/false properties identified in step 2.

6. **Exceptional Postconditions Check**: if m threw an exception, the oracle checks if the type of the raised exception is among the expected ones. If not, the oracle signals a failure.

Figure 4.7 shows the template of the aspect generated from a tuple $\langle m, pre = \{p\}, post = \{\langle g, tp, fp \rangle\}, excpost = \{\langle e, c \rangle\}\rangle$. When executing a test case that invokes m, the aspect is invoked right before the invocation of m (Figure 4.7, line 1) and is executed *instead of* method m, thus changing the original control flow of the test case. The oracle checks the validity of the preconditions (line 6), and if one precondition does not hold, the oracle signals a failure, and interrups the execution of the (invalid) test case. The oracle collects the postconditions to be checked *after* the execution of the method under test (line 7) and the expected exceptions (line 8). In this way, conditions are evaluated *before* the execution of method m (lines 41 and 53). Then method m is executed normally (line 11).

If m terminates normally (without an exception), then the oracle verifies if any exception was expected (line 12). If so, the oracle lets the test fail because an exception was expected but no exception was raised (line 13). If no exception was expected and raised, the oracle checks the validity of the postconditions (line 16). If a postcondition does not hold, the oracle signals the failure and the test case execution terminates. When every postcondition holds, the test execution continues (line 20).

If m terminates with an exception, the oracle check whether the exception was among the expected ones (line 12). If so, the test case execution terminates successfully (line 23). Otherwise, the oracle reports the failure terminating the execution of the test case (line 25).

By relying on AspectJ aspects, Toradocu does not modify the source code of the input test cases, rather, it *modifies the bytecode* of the input test cases, injecting oracles where needed.

```
@Around("call(m)")
public Object advice(ProceedingJoinPoint jp) throws Exception {
  Object receiver = jp.getTarget();
  Object[] args = jp.getArgs();

  checkPreconditions(pre);
  List<Postcondition> postconditions = getPostconditions(receiver, args);
  List<Class> expectedExceptions = getExpectedExceptions(receiver, args);

  try {
    Object returnValue = jp.proceed(args); // Method m is invoked here.
    if (!expectedExceptions.isEmpty()) {
      fail("Expected exception not thrown");
    }
    for (Postcondition p : postconditions) {
      if (!p) { // Postconditions are checked here.
        fail("Postcondition does not hold");
      }
    }
    return returnValue;
  } catch (Throwable e) {
    if (expectedExcepts.contains(e.getClass()) {
      pass("Raised exception was expected");
    } else {
      fail("Unexpected exception thrown");
    }
  }
}

private void checkPreconditions(Object receiver, Object[] args) {
  for (Precondition p : pre) {
    if (!p) { // Precondition is checked here.
      pass("Precondition does not hold")
    }
  }
}

private List<Postcondition> getPostconditions(Object receiver, Object[] args) {
  List<Postcondition> postconditions = new ArrayList<>();
  for (Postcondition p : post) {
    if (g) { // Guard g is checked here.
      p.checkTrueProperty(); // Indicates that (only) tp has to be checked.
    } else {
      p.checkFalseProperty(); // Indicates that (only) fp has to be checked.
    }
    postconditions.add(p);
  }
}

private List<Class> getExpectedExceptions(Object receiver, Object[] args) {
  List<Class> expectedExcepts = new ArrayList<>();
  for (<e, c> : excpost) {
    if (c) { // Condition c is checked here.
      expectedExcepts.add(Class.forName(e));
    }
  }
  return expectedExcepts;
}
```

Figure 4.7. Model of an Aspect Generated by the Oracle Generator

4.3 Toradocu Evaluation

Toradocu translates Javadoc comments into actionable exceptional postconditions that can be used to automatically generate test oracles. When used in combination with automatic test input generators, Toradocu enables the automatic unit testing of software systems.

We conducted several experiments, each of them investigating a different research question. We first studied how well exceptional behaviors are tested by manually written test suites. This is not a direct evaluation of Toradocu, rather it gives a feeling on the impact Toradocu could have. We then experimentally validated the accuracy of Toradocu in translating comments into procedure specification. In particular, we measure the translation accuracy of Toradocu in terms of precision and recall. We compared the accuracy of Toradocu with a state-of-the-art technique. We also measured how Toradocu affects the effectiveness of automatically generated test cases. Specifically, we evaluated if and how much Toradocu-generated oracles reduce the number of false alarms and increase the number of true alarms produced by automatically generated test cases. Our evaluation aims to address the following research question:

RQ1 To what extent *is Toradocu accurate* in translating Javadoc comments into procedure specifications? What is the impact of the semantic translator engine on the accuracy of Toradocu? How does Toradocu accuracy compare with a state-of-the-art technique, namely @tComment?

RQ2 To what extent do Toradocu test oracles *reduce the number of false alarms* reported by automatically generated test cases? Automatically generated test cases rely on simplistic oracles and heuristics that often misclassify test executions, producing many false alarms, that is, a test execution fails when the software under test is correct, that developers have to manually inspect and discard. We measure if and to what extent Toradocu oracles can reduce the number of false alarms, thus reducing the overall costs of automatic testing.

RQ3 To what extent do Toradocu test oracles *increase the number of discovered faults (bugs)* in the SUT? Automatically generated test cases miss many faults that may leak into production code. We measure if and to what extent Toradocu oracles can complement automatically created test cases to identify erroneous exceptional behaviors, thus improving the effectiveness of automatic testing more effective.

4.3.1 RQ1: Translation Accuracy

This research question investigates the accuracy of Toradocu in translating Javadoc comments into procedure specifications. We measure accuracy in terms of *precision* and *recall*, two standard metrics for information retrieval tasks. Precision measures the portion of the output that is correct. Recall measures the portion of the desired output that is actually produced. The output of a Javadoc translation technique that produces specifications like Toradocu can be:

correct (C) when the produced specification exactly matches the expected one;

missing (M) when the technique does not produce any specification, and a specification was expected;

wrong (W1) when the technique produces a specification, but no specification was expected;

wrong (W2) when the actual translation does not match the expected one.

We define precision as the ratio between the number of correct outputs (C) and the total number of outputs (C + W1 + W2):

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

We define recall as the ration between the number of correct outputs (C) and the total number of expected outputs (C + W2 + M):

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

To answer RQ1, we manually derived procedure specifications from a set of Javadoc comments. Such manually derived specifications constitute the ground truth. We then translated the same Javadoc comments using @tComment and Toradocu (with and without the semantic matching enabled), and we measured precision and recall. In the experiment we proceeded as follows:

- We selected 7 popular open-source Java projects, and, for each of them, we *randomly* selected 5 classes with more than 4 Javadoc block tags comments introduced by `@param`, `@return`, and `@throws`. To select classes with meaningful and documented behavior, we ignored methods inherited from `java.lang.Object`, getters (methods whose name starts with "get"), and setters (methods whose name starts with "set"). In total we randomly selected 35 classes as subjects of the experiment.
- We manually derived procedure specification from the Javadoc comments of methods in each class. (When we could not derive any specification, we discarded the class and select another one within the same project.)
- We ran Toradocu (with and without the semantic matching) and @tComment on the subject classes, recording the produced specifications. We used a reimplementation of @tComment that produces an output compatible with the one produced by Toradocu and, therefore, easy to compare.
- We measured precision and recall of the three configurations. We conservatively define a wrong output specification as a specification that does not completely match the expected one. Thus, we considered as wrong partially correct translations. For example, given the comment `@throws [...]  if x is negative or y is null`, the specification "x < 0" is considered wrong (and in particular of W2 type).

Table 4.2 shows the characteristics of experimental subjects. For each system, the table reports the total number of classes in the system, the number of randomly selected classes, the total number of methods in the selected classes, and the number of methods that are documented with a Javadoc comment (column *Doc'd*). Table 4.2 also shows the number of manually derived preconditions (column *Pre*), normal postconditions (column *Post*), and exceptional postconditions (column *Ex. Post*). Overall, the ground truth is composed of 154 specifications.

[9] `https://commons.apache.org/collections`
[10] `https://commons.apache.org/math`
[11] `http://www.freecol.org`
[12] `http://graphstream-project.org`
[13] `http://github.com/google/guava`
[14] `http://jgrapht.org`
[15] `http://mernst.github.io/plume-lib`

Table 4.2. Toradocu Accuracy Evaluation Subjects

	Classes		**Methods**		**Specifications**		
System	**Total**	**Selected**	**Total**	**Doc'd**	**Pre**	**Post**	**Ex. Post**
Commons Collections 4.1[9]	320	5	31	16	17	2	11
Commons Math 3.6.1[10]	990	5	41	9	0	2	9
FreeCol 0.11.6[11]	678	5	334	14	0	13	1
GraphStream 1.3[12]	233	5	115	10	1	8	1
Guava 19[13]	469	5	91	16	2	0	16
JGraphT 0.9.2[14]	205	5	48	7	0	1	8
Plume-lib 1.1[15]	50	5	241	67	4	37	21
Total	**2945**	**35**	**901**	**139**	**24**	**63**	**67**

Table 4.3. Toradocu Accuracy Evaluation Results

	`@param`		`@return`		`@throws`		**Overall**		
	Pre	**Rec**	**Pre**	**Rec**	**Pre**	**Rec**	**Pre**	**Rec**	**F**
@tComment	1.00	0.74	n.a.	0.00	0.60	0.18	0.78	0.13	0.23
Toradocu	0.84	0.91	0.89	0.76	0.94	0.90	0.90	0.84	0.87
Toradocu-Sem	0.84	0.91	0.75	0.78	0.94	0.90	0.84	0.85	0.85

Table 4.3 shows the results of our evaluation with @tComment and Toradocu without (row *Toradocu*) and with the semantic matching (row *Toradocu-Sem*). For each comment type, Table 4.3 reports precision (column *Pre*), and recall (column *Rec*). Column *Overall* reports the precision, recall, and F-measure (column *F*) considering all the specification kinds.

In deriving preconditions and exceptional postconditions, *Toradocu consistently outperforms @tComment*. @tComment obtains a higher precision than Toradocu *only* in the translation of `@param` comments. This is because @tComment is tailored to translate null-related comments matching specific patterns. Everything else is ignored. This leads @tComment to have good precision and poor recall, since @tComment misses many opportunities to derive specifications. Another important limitation of @tComment, when compared with Toradocu, is that @tComment does not support comments composed of many clauses connected with grammatical conjunctions. For instance, the comment `@throws [...] if x and y are positive or z is negative` is correctly supported by Toradocu, while cannot be translated correctly with @tComment. Overall, *Toradocu outperforms @tComment, obtaining better precision (90% vs. 78%) and better recall (84% vs. 13%)*.

Toradocu, with and without semantic matching, translates better `@param` and `@throws` comments than `@return` comments. `@return` comments may express conditions that, in themselves, are difficult to check. For example, the comment `@return [...] if no overflow occurs` should generate a guard condition checking whether an overflow occurs during the execution of the documented method. Also, consider comments like `@return the sorted array` documenting a sorting procedure. The comment translates into a postcondition checking that the procedure output is actually sorted. This is a complex check involving iterations on the entire output array. Such comments are not supported by the current version of Toradocu.

In translating `@param` and `@throws`, the semantic matching has no sensible impact on precision and recall. The semantic matching does have an impact on the precision and recall in the case of `@return`. In particular, the semantic matching improves the recall (78% vs. 76%) at the cost of a worse precision (75% vs. 89%). Overall, *with the semantic matching enabled Toradocu achieves better recall (85% vs. 84%) and a worse precision (84% vs. 90%)*.

We replicated the initial experiment with a large set of subject classes, following the same experimental setup, with a different selection of subjects. In this new experiment, we selected classes by manually identifying classes looking at the amount of Javadoc comments, and at our familiarity with the functionality provided by the class, to reduces the time required to derive the ground truth. The larger number of comments of the selected classes, reduces the bias that may derive from a non random selection and produces interesting results.

Table 4.4 reports the subjects we selected for this experiment. Columns in Table 4.4 have the same semantics of the columns in Table 4.2. The manually derived ground truth is composed of 755 specifications.

Table 4.5 shows the results obtained with @tComment and Toradocu with (column *Toradocu-Sem*) and without the semantic matching (column *Toradocu*). The results confirms the results obtained with fewer and randomly-selected classes in the previous experiments: Toradocu is more accurate @tComment in translating `@param` and `@throws` comments. Overall, Toradocu and @tComment obtains a comparable precision, while Toradocu recall is greatly better than @tCommetn recall: 81% vs. 23%. Also in this experiment, the semantic matching does not impact in an appreciable way the translation of `@param` and `@throws` comments. On the other hand, the semantic matching increases recall on `@return` comments while decreasing precision.

Table 4.4. Toradocu Accuracy Evaluation Subjects (Non-random)

	Classes		Methods		Specifications		
System	**Total**	**Selected**	**Total**	**Doc'd**	**Pre**	**Post**	**Ex. Post**
Commons Collections 4.1	320	13	224	158	141	24	157
Commons Math 3.6.1	990	30	568	154	47	16	165
FreeCol 0.11.6	678	3	456	24	0	16	4
GraphStream 1.3	233	2	6	4	2	0	0
Guava 19	469	17	296	56	30	12	37
JGraphT 0.9.2	205	10	81	23	0	6	26
Plume-lib 1.1	50	11	404	78	7	38	27
Total	**2945**	**86**	**2035**	**497**	**227**	**112**	**416**

Table 4.5. Toradocu Accuracy Evaluation Results (Non-random)

	`@param`		`@return`		`@throws`		Overall		
	Pre	**Rec**	**Pre**	**Rec**	**Pre**	**Rec**	**Pre**	**Rec**	**F**
@tComment	0.96	0.62	n.a.	0.00	0.78	0.15	0.90	0.23	0.36
Toradocu	0.95	0.96	0.65	0.62	0.95	0.76	0.90	0.80	0.85
Toradocu-Sem	0.94	0.96	0.57	0.67	0.95	0.77	0.87	0.81	0.84

In summary, Toradocu is accurate in translating Javadoc comments into procedure specifications, and is more accurate than @tComment, a state-of-the-art technique. In Toradocu, the semantic matching has a double effect, increasing the recall and worsening the precision. Thus, the choice of exploiting the semantic matching or not depends on whether the Toradocu user values more the soundness or the completeness of generated specifications.

4.3.2 RQ2: False Alarms Reduction

Toradocu translates Javadoc comments into procedure specifications that can be used to generate test oracles. Such test oracles can be combined with automatically generated test inputs to make them more accurate and more effective in finding faults. In this research question, we evaluate the effectiveness of Toradocu in generating test oracles, by reducing the number of false alarms reported by automatically generated test cases. To answer RQ2, we proceeded as follows:

- We selected nine subject classes from the open-source Java project Google Guava. We selected subject classes from packages `base`, `collections`, and `primitives` because of our familiarity with those packages that reduces the time required to manually investigate the results of the experiment. Table 4.6 lists the selected subject classes with the number of `@throws` Javadoc comments present in each class (column *@throws*).

- We ran Toradocu on each class to generate test oracles from `@throws` Javadoc comments. Table 4.6 reports the number of generated oracles for each subject class (column *Toradocu Oracles*). With the word *oracles* we actually denote *aspects*. At runtime each oracle (aspect) is executed multiple times, therefore, column *Toradocu Oracles* does not report the number of assertions deployed into the test suite. We manually inspected generated oracles and we classified them into *Correct* and *Partial*. Correct oracles are oracles that perform a check that is precisely what is described by the `@throws` comment from which the oracle is derived. Partial oracles are those oracles that perform an incomplete, although correct, check. Partial oracles stem from incomplete translations of `@throws` comments, where Toradocu can understand the meaning of the comment only partially. Column *Missing* reports the amount of comments that Toradocu cannot translate into an oracle. Toradocu did not generate wrong aspect in our evaluation.

- For each subject class, we created a test suite with EvoSuite. We configured EvoSuite to avoid missed alarms (false negatives), that is, we configured EvoSuite not to treat exceptions (both checked and unchecked) as expected, legal behavior. This is because there is no generally valid rule that states when a thrown exception is *the* expected behavior. We enabled all EvoSuite' search goals and we gave EvoSuite a search budget of 60 seconds.

- For each subject class, we ran EvoSuite-generated test suites with and without Toradocu oracles.

- We manually inspected each failing test case to check whether it was a true or false alarm.

Table 4.7 shows the results of the experiment. Table 4.7 reports the number of partial and correct oracles (aspects) generated by Toradocu for each subject class (column *O*), and the results of the execution of EvoSuite-generated test suite with and without Toradocu oracles. Table 4.7 shows the number of successful Evosuite test case executions (column *Pass*), the number of Evosuite test cases failing because of a bug (column *True Alarms*), and the number

Table 4.6. Toradocu False Alarms Evaluation Subjects and Toradocu-generated Oracles

		Toradocu Oracles		
Subject Class	**`@throws`**	**Correct**	**Partial**	**Missing**
ArrayListMultimap	1	1	0	0
AtomicDoubleArray	1	1	0	0
ConcurrentHashMultiset	12	8	1	3
Doubles	4	3	1	0
Floats	4	3	1	0
MoreObjects	1	1	0	0
Shorts	6	3	1	2
Strings	1	1	0	0
Verify	4	4	0	0
Total	**34**	**25**	**4**	**5**

of Evosuite test cases that fail while they should not, that is, they terminates because of a thrown exception that is exactly the intended behavior of the SUT (column *False Alarms*). For EvoSuite+Toradocu executions, we distinguish between false alarms caused by the generated test input being illegal, and Toradocu limitations that prevents a correct oracle generation of a specific `@throws` comment.

Table 4.7. Toradocu False Alarms Evaluation Results

		EvoSuite			EvoSuite + Toradocu			
Subject Class	**O**	**Pass**	**True Alarms**	**False Alarms**	**Pass**	**True Alarms**	**False Alarms-T**	**False Alarms-I**
ArrayListMultimap	1	3	0	6	4	0	0	5
AtomicDoubleArray	1	20	0	10	21	0	0	9
ConcurrentHashMultiset	9	32	1	15	40	1	1	6
Doubles	4	42	0	12	47	0	2	5
Floats	4	43	0	18	47	0	2	12
MoreObjects	1	17	0	4	18	0	0	3
Shorts	4	22	0	22	30	0	3	11
Strings	1	6	0	9	9	0	0	6
Verify	4	4	4	1	5	4	0	0
Total	**29**	**189**	**5**	**97**	**221**	**5**	**8**	**57**

Overall, we generated 290 test cases with EvoSuite. Of these test cases, 101 (35%) fail because the generated test input triggers an exception. For some of the subject, EvoSuite generated more failing than passing test cases.

With Toradocu oracles embedded in EvoSuite tests, the overall number of failing tests decreases from 101 to 65 (22%), and Toradocu reduces the percentage of failing test cases from 35% to 22%. We manually inspected every failure reported by test cases augmented with Toradocu oracles, and we classified each failing execution as: 1. true alarm—a failing test cases execution that reveals a fault in the SUT (column *True Alarms*), 2. false alarm due to current limitations of Toradocu (column *False Alarms-T*), and 3. false alarm due to illegal test input generated by EvoSuite (column *False Alarms-I*).

In the experiment, automatic testing discovers five bugs in total. Notice that those four failures are revealed by EvoSuite-generated test input triggering exceptions at runtime. Toradocu oracles confirm those failures and do not mask raised exceptions. Thus, Toradocu oracles correctly report the failures to developers.

Out of five true alarms, four are due to missing `@throws` comments. In other words, there are methods throwing an undocumented exception. (In all four cases the undocumented exception is unchecked.[16]) An example is method `ConcurrentHashMultiset#removeExactly(Object element, int occurrences)` that throws `IllegalArgumentException` if the argument `occurrences` is less than 0. The missing Javadoc comment is `@throws IllegalArgumentException if {@code occurrences} is negative`. To infer whether the missing `@throws` comment was a developer choice or rather a mistake, we inspected similar methods in the same code base and their documentation. We then judged missing comments as a documentation bug likely caused by a developer oversight. We reported the issues to the developers, creating pull requests.[17] Developers accepted our pull requests, confirming that our classification of the alarms as true, fault-revealing alarms was correct. Toradocu oracles can help developers discover undocumented exceptions, even though is the automatically generated test input that triggers the exception.

The other true alarm is due to a wrong exception type raised at runtime by the method `Verify#verifyNotNull(T, String, Object...errorMessageArgs)`. According to its documentation, the method is supposed to throw a `VerifyException` when invoked as `verifyNotNull(null, "", null)`. Instead, the method threw a `NullPointerException`. The exception stems from a dereference of the third argument `errorMessagArgs`, which is marked as `@Nullable`. However, the arguably misleading annotation actually means that single elements in the input array can be `null` and not the array itself. This is a known issue, of which we were not aware, independently reported to the developers who confirmed the issues but refused to fix it cause they do not think the problem is relevant in common usage scenarios of the method.[18] Even in this case the exception is triggered by the test input generated by EvoSuite. Toradocu oracles help developers understand the failure because they clearly signal that the raised exception type is not one of the documented, expected exception types. Therefore, developers have a precise idea of the reason of the failure rather than a generic stack trace without any meaningful message.

Although Toradocu oracles greatly reduce the number of false alarms reported by EvoSuite tests (-33%, from 97 to 65), there are still several false alarms. We analyzed all the false alarms and classified them into: 1. false alarms caused by limitations of Toradocu, 2. false alarms generated by illegal input (produced by EvoSuite).

Out of 65 failures, 8 are caused by limitations of Toradocu, that is, developers correctly documented the raised exception, but Toradocu was not able to correctly parse the comment to generate the corresponding specification. For instance, method `ConcurrentHashMultiset#toAr-`

[16]Unchecked exceptions are of class `RuntimeException`, `Error`, and their subclasses. Unchecked exceptions do not require to be explicitly handled with `try-catch` blocks.

[17]`https://github.com/google/guava/pull/2099`, `https://github.com/google/guava/pull/2106`

[18]`https://github.com/google/guava/issues/1701`

`ray(T[])` is expected to throw an exception when "the runtime type of the specified array is not a supertype of the runtime type of every element in this collection". Toradocu fails to correctly translate this comment into an oracle. As another example, Toradocu fails to correctly parse 4 comments regarding arrays and collections of the kind "if collection or any of its elements is null". Such kind of comments are present in classes `Floats` and `Doubles`. Toradocu is only able to understand the first part of the comment, "if collection ... is null" while missing the second part "or any of its elements is null". The translation as a whole is incorrect. In all cases, the method under test correctly raises an exception that is documented. However, Toradocu is not able to derive an oracle that can interpret the exception as the correct expected behavior of the SUT.

The other 57 failures are caused by illegal inputs generated by EvoSuite. In those cases the method under test raises an exception that is not explicitly documented. A common reason for this is the violation of a precondition. For example, developers can describe a precondition of a method with the comment `@param x must not be null`. The comment states that the argument `x` cannot be null, but does not specify what happens when the precondition is violated. For instance, in the experiment EvoSuite generated tests with the method invocation `Doubles.tryParse(null)` which threw `NullPointerException`. As another example, an EvoSuite-generated test passed a very large value to the `ArrayListMultimap` static constructor `create(int, int)`, then invoked the method `createCollection()`, which threw `OutOfMemoryException`.

The empirical results indicate that Toradocu is effective in reducing the number of false alarms reported by automatically generated test cases. Thus, Toradocu makes the application of automatic testing more efficient for developers.

4.3.3 RQ3: True Alarms Increment

With this research question we investigate the effectiveness of Toradocu oracles in *discovering bugs* in the SUT. To answer the question we applied automatic testing and Toradocu to check the exceptional behavior of several implementations of four classes. Such implementations were created during an empirical evaluation conducted by Rojas and colleagues to investigate how developers use automated unit test generation during the implementation of a new code [82]. In the study, developers were asked to (re-)implement four classes of Apache Commons,[19] using the Javadoc comments as specification. We proceeded as follows:

- We considered all 4 Java classes that were the subjects of the experiment conducted by Rojas and colleagues, and all the implementation snapshots available for each class.
- For each implementation snapshot, we generated a test suite with EvoSuite, with branch coverage as search criterion and 30 seconds as search budget. We set EvoSuite to treat any exception as an erroneous behavior (crash oracle) and avoid any regression assertion.
- For each implementation snapshot, we generated tests with Randoop. We ran Randoop with 20 seconds of time budget and with a test generation limit of 150. We also set Randoop to treat any exception as an erroneous behavior (crash oracle) and avoid any regression assertion. For each target implementation, Randoop generates a regression and an error test suite: regression test suite collects tests that pass, while error test suite collects all the tests that fail.

[19]`https://commons.apache.org`

- We ran Toradocu on each of the four subject classes, obtaining test oracles for the exceptional behavior of each class. We manually analyzed specifications produced by Toradocu, and we discarded all the wrong translations. We deemed a specification as wrong if it was not a correct translation of the corresponding `@throws` comment.

- We ran test suites generated with EvoSuite and Randoop with and without (correct) Toradocu oracles. For each test we recorded the outcome: pass, fail because of a failing assertion, fail because of a runtime exception.

Table 4.8. Toradocu True Alarms Evaluation Subjects

Class	Executable Members	Implementation Snapshots	`@throws`	Toradocu Corr	Toradocu Miss
collections.map.FilterIterator	10	401	4	2	2
collections.comparators.FixedOrderComparator	9	391	9	5	4
math.genetics.ListPopulation	13	269	10	2	8
collections.map.PredicatedMap	7	283	4	2	2
Total	**39**	**1344**	**27**	**11**	**16**

Table 4.8 summarizes the selected subjects. For each class, the table reports the number of executable members (constructors and methods) and the number of implementation snapshots that are the subjects of our experiment. The table reports the number of `@throws` Javadoc comments present in each class and the number of `@throws` comments that Toradocu correctly translates into oracles (column *Corr*) that are encoded into AspectJ aspects. Column *Miss* is the number of `@throws` comments that Toradocu is not able to translate into a specification. In all these cases Toradocu does not produce any output. All the eleven translations provided by Toradocu are correct.

Table 4.9 presents the results of the experiment. Columns *EvoSuite* and *EvoSuite+T* report the results obtained with EvoSuite-generated test suites without and with Toradocu oracles, respectively. Columns *Randoop-E* and *Randoop-E+T* report the results obtained with Randoop-generated *error* test suites without and with Toradocu oracles, respectively. Columns *Randoop-R* and *Randoop-R+T* report the results obtained with Randoop-generated *regression* test suites without and with Toradocu oracles, respectively. For each category, we distinguish tests that pass (column *Pass*), fail because of an assertion (column *FA*), and fail because of a runtime exception (column *RE*).

EvoSuite test cases report many failures due to runtime exceptions (column EvoSuite, RE). Toradocu oracles produce a twofold effect. First, the number of reported failures consistently decrease for each subject class. Second, Toradocu oracles report several failures that were undetected by the implicit EvoSuite oracle (column *EvoSuite+T, FA*). The same effect can be appreciated with Randoop error test suites (columns *Randoop-E* and *Randoop-E+T*). When combined with Randoop regression test suites (column *Randoop-R*), Toradocu oracles detect faults that are not detected otherwise. In summary, Toradocu oracles can both detect semantic failures of the SUT, and reduce the number of false alarms reported by automatically generated test cases.

Table 4.9. Toradocu True Alarms Evaluation Results

	EvoSuite			EvoSuite + T			Randoop-E			Randoop-E + T			Randoop-R			Randoop-R + T		
Subject Class	**Pass**	**FA**	**RE**	**Pass**	**FA**	**RE**	**Pass**	**FA**	**RE**	**Pass**	**FA**	**RE**	**Pass**	**FA**	**RE**	**Pass**	**FA**	**RE**
FilterIterator	2205	0	546	1940	423	388	0	0	12453	1423	3032	7998	17440	0	0	8927	8513	0
FixedOrderComparator	2885	0	1160	3221	65	759	0	0	23670	26	262	23382	12910	0	0	12670	240	0
ListPopulation	2208	0	1330	2310	115	1113	0	0	12693	97	162	12434	11614	0	28	11543	78	21
PredicateMap	1520	0	452	1332	420	220	0	0	24257	136	37	24084	6186	0	0	6076	110	0

We manually analyzed a sample of failures reported by Toradocu oracles. We randomly selected 20 failure reports for EvoSuite+T, Randoop-E+T, and Randoop-R+T. Of the 60 selected failures none of them is a false alarm, that is, Toradocu oracles reported only true alarms corresponding to faults in the implementation.

The experiments indicate that Toradocu oracles are effective in identifying faults, and that Toradocu oracles well complement automatically generated test inputs, improving their fault-finding effectiveness.

4.4 Limitations and Threats to Validity

Toradocu translates Javadoc comments into procedure specifications that can be used to create effective test oracles. Such test oracles have a twofold effect: They identify violations of method preconditions, *and* they identify violations of method postconditions. In particular, violations of the preconditions occur when a method is invoked with a context (e.g., state of the receiver object, method arguments) that does not fulfill method preconditions described in the method Javadoc documentation. A test case whose execution leads to a method precondition violation should be deemed as invalid. (Notice that the preconditions described in the Javadoc documentation might be incorrect.) Violations of the postconditions occur when a method produces a result that is different from the result described in its Javadoc documentation. Such kind of violations highlights a mismatch between *documented* and *actual* behavior of a method. A mismatch represents an error, either in the implementation of the method or in its the Javadoc documentation (or both). Either way, the error has to be fixed. Toradocu oracles are effective in identifying the aforementioned issues. Every other problem is outside the scope of Toradocu oracles. For example, non-functional properties and properties not described in the Javadoc documentation are not checked by Toradocu oracles.

The empirical evaluation we performed shows the accuracy of Toradocu and the effectiveness of produced oracles. Nonetheless, the results we obtained have limitations impacting both their internal and external validity. Threats to the internal validity may derive from errors in the implementation of Toradocu[20] that, in turn, may lead to imprecise results. To limit the impact of such threat, we carefully inspected and tested the implementation, addressing all the problems arisen during testing. Threats to the external validity stem from the process we used to select the evaluation subjects. For the evaluation, we selected classes from open-source Java projects that are intended to be used by others. Their collaborative nature and their reusability may impact the way Javadoc comments are written. Therefore, the classes we used in our evaluation may not be representative of the systems internally developed in a company, not intended to be publicly released and used by others. In closed-source systems, Javadoc documentation could be less present and accurate (or the other way around, more present and accurate). The limited number of subject classes used in the evaluation could affect the generalizability of the results. However, the classes belong to systems in different domains. For example, Commons Math classes are generally dedicated to perform mathematical operations, while Commons Collections classes represent general purpose data structure. Different domains imply different Javadoc documentation (e.g., different kind of preconditions). The heterogeneity of the comments we considered in the evaluation should favor the generalizability of the results. For the accuracy evaluation (Section 4.3.1) we manually defined the ground truth against to we measured precision/recall values of different techniques. Errors in the ground truth can affect

[20]Toradocu is open-source and publicly available: `https://github.com/albertogoffi/toradocu`.

precision/recall values. To mitigate such risk, “golden” specifications were cross-checked by at least two people (i.e., for each “golden” specification the author of this thesis was either an author or a reviewer of the specification).

Chapter 5

Conclusions

Automatic generation of test oracles is a widely investigated problem in both academic and industrial research laboratories, aiming to complement automatic input generation to produce a fully automated testing process. The approaches proposed so far produce oracles that are either inexpensive but with limited fault-finding effectiveness, or effective but expensive to generate. In this thesis, we propose approaches to generate cost-effective oracles, that is, oracle that can be generated inexpensively from code and existing artifacts, and effective, that is, can identify a relevant set of failures with a reduced amount of false alarms. We use information generally available as byproduct of standard development practices to produce test oracles without incurring high generation cost. We use information that is specific to the system under test (SUT), and that encodes the semantics of the SUT. This dissertation introduces two kinds of automatic oracles: cross-checking oracles (CCOracles) and Toradocu oracles.

CCOracles exploit intrinsic software redundancy to generate test oracles. Modern software systems are intrinsically redundant, that is, they provide the same functionality through different executions. We encode such redundant functionalities as equivalent sequences of method calls, and design a technique that exploits equivalent sequences to automatically generate test oracles. We demonstrate the effectiveness of CCOracles in identifying synthetic faults when combined with both automatically-generated and manually-written test inputs. We also show that CCOracles can detect real faults in the SUTs. The main cost factor of CCOracles lies in the definition of equivalent sequences. We briefly discuss a search-based technique called SBES that automatically discovers intra-class equivalent sequences. We show that SBES is effective and accurate. SBES reduces the cost of manually identifying equivalent sequences. (SBES is part of the contributions of Andrea Mattavelli's Ph.D. thesis [61].)

Toradocu automatically translates natural language procedure documentation into Java procedure specifications that it uses to generate test oracles. Both the specification generation and the oracle generation are completely automated. We show that Toradocu accurately translates procedure documentation into procedure specifications and that oracles derived from such specifications are effective in identifying faults in the SUT.

Contributions

This thesis contributes to the state-of-the-art by defining two cost-effective techniques to generate test oracles from existing information. CCOracles automatically generate oracles from the intrinsic software redundancy, while Toradocu generates test oracles form procedure documentation written in natural language. We now summarize the different aspects of this thesis' contributions.

CCOracles. We propose a technique to automatically derive test oracles from redundant method call sequences, that is, sequences of method calls that are different but produce equivalent outcomes. A CCOracle executes pairs of expected equivalent sequences by controlling the interferences between the two executions, and semantically compares the outcomes, using an hybrid approach that combines the notion of equivalence provided by the developers of the SUT and an observational equivalence that probes the outcomes through their public interface.

Empirical evaluation of CCOracles effectiveness. We empirically evaluate the effectiveness of CCOracles with both manually-written and automatically-generated test cases. CCOracles improve the fault-finding effectiveness of automatically generated test cases detecting 6.5 times more faults than the simple implicit oracles usually that characterize automatically generated test oracles. CCOracles are less effective than manually-written assertions, since they identify about 2/3 of the faults detected by hand written assertions, but can identify faults that go undetected with developers-written assertions.

Toradocu. We propose an approach to derive procedure specifications from semi-structured natural language (English) comments, and transform the derived procedure specifications into test oracles. Toradocu employs natural language processing techniques to derive subject-predicate pairs from input comments, and then translates subject-predicate pairs matching subjects and predicates to Java code elements (mainly existing in the SUT).

Empirical Evaluation of Toradocu accuracy and effectiveness. We conduct two experimental evaluations, one with comments from randomly selected classes and one with a larger set of comments from non-randomly selected classes. In both cases Toradocu accurately translates comments into procedure specifications with a precision of about 90% and a recall of about 80%. Toradocu-generated oracles both effectively reduce the number of false positives and detect more implementation faults than implicit oracles generally employed by automatically generated tests.

Discussion

In this thesis we propose two ways to use information present in software systems to generate effective test oracles with a reasonable cost. In Section 2.4 we introduce a conceptual landscape in which oracles are classified according to their cost-effectiveness. In particular, with the term effectiveness we mean the capacity of an oracle to detected faults. The extent to which an oracle is able to detect faults is called *completeness*. Along with completeness, an oracle is defined by its *soundness* that is the extent to which an oracle reports *only* true alarms.

Fig. 5.1 shows where CCOracles and Toradocu oracles stand in the landscape: both of them are more effective than implicit oracles normally employed by automatic test case generators, as shown by the results of our empirical evaluation (Sections 3.2.2 and 4.3.3). Compared to implicit oracles, CCOracles and Toradocu oracles detect more faults and trigger less false alarms, i.e., they are *more complete and more sound* than implicit oracles. Regarding the cost, Toradocu oracles come roughly for free when Java constructors and methods in the SUT are commented

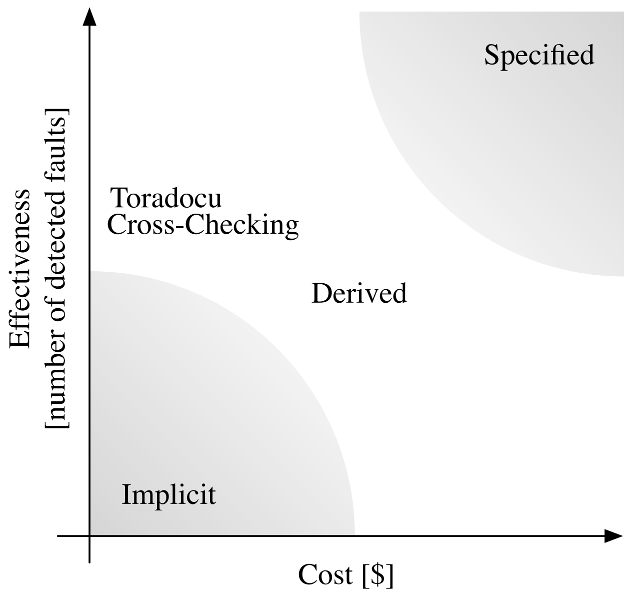


Figure 5.1. Cost-effectiveness of cross-checking and Toradocu oracles

with Javadoc comments. Instead, CCOracles require the mining and encoding of the intrinsic redundancy in the SUT as redundant sequences. This activity can be assisted and made less expensive by automatic tools like SBES (Section 3.1.1). In summary, Toradocu and CCOracles are more effective than implicit oracles with a marginally higher cost. On the other hand, both Toradocu and CCOracles are less costly than specified oracles since no formal specifications have to be manually produced.

Specified oracles are sort of *perfect* oracles, in the sense that formal specifications enable the generation of sound and complete oracles (as long as the implementation of the SUT perfectly matches its formal specification). Being sound and complete, specified oracles do not need, in general, to be complemented with other kind of oracles. However, formal specifications are rarely available in practice and so are specified oracles.

When we move from sound and complete specified oracles to incomplete oracles, the complementarity of the oracles becomes important. To thoroughly check the SUT we need several, and crucially *different*, oracles that can be used together to combine their fault finding abilities. Implicit oracles are by nature limited and they have to be complemented with other oracles to achieve a good fault-finding effectiveness. In our experiments we showed how both Toradocu and CCOracles nicely complement implicit oracles to get a more thorough test of the SUT. Not only Toradocu and CCOracles can be independently combined to implicit oracles, but they can also be applied together to get a more complete testing. Toradocu and CCOracles are different because they are generated from different information sources, that can be available in different amount and with different timing along the development cycle of the SUT. Toradocu and CCOracles are different also because they focus on different classes of faults. Toradocu oracles can detect are inconsistencies between the behavior of a method described in its documentation and the actual behavior of the method. Instead, CCOracles detect violations of the redundancy relations among constructors and methods. Errors revealed by CCOracles are not code-comments inconsistencies. Instead, they are true faulty behaviors due to faults in the implementation of the SUT. Thus, CCOracles reveal implementation errors, while Toradocu oracles detect code-comments inconsistencies.

As an example of the difference, consider the case of procedure preconditions. Toradocu is able to derive preconditions from the Javadoc documentation and can detect when a procedure invocation violates such preconditions. Precondition violations are useful to identify invalid tests. CCOracles do not check preconditions and are not helpful to identify invalid tests. In addition to preconditions, CCOracles ignore exceptional behaviors. This is because *redundant* sequences of method invocations often show a different exceptional behavior. This also implies that CCOracles are not able to detect errors in the exceptional behavior of methods, while Toradocu can spot inconsistencies between the documented and the actual behavior of a method.

Javadoc comments are not intended to be complete. This means that Toradocu oracles, that are derived from Javadoc comments, are not good in completely checking the results of a documented procedure. That is exactly what CCOracles are good at. CCOracles aim to precisely find errors in the results of procedure invocations.

To summarize, specified oracles are complete, but when formal specifications are not available, they become not cost-effective. Differently from specified oracles derived oracles are not complete, they are partial. We need to combine multiple partial oracles, in addition to simple implicit oracles, to thoroughly check the behavior of a SUT. Toradocu and CCOracles are a sort of derived oracle, they nicely complement implicit oracles and they identify different classes of faults. Thus, Toradocu and CCOracles can be used together (and in addition to other partial oracles) to make testing more effective.

Open Research Directions

The contributions of this thesis open new research directions towards the automatic generation of effective test oracles.

Efficient CCOracle execution and equivalence check mechanisms. CCOracles execute equivalent sequences in isolation by relying on a cloning mechanism that does not guarantee a completely safe and sound isolation, mitigate the issue with consistency checks before executing two equivalent sequences, and check equivalence by combining the Java `equals` method with observational equivalence. The results open the problems of defining safe and sound mechanisms to guarantee the isolation of executing equivalence sequences, and of finding effective ways to check for the equivalence of redundant method call sequences.

Possible research directions for a safe and sound execution in isolation involve the study of static/dynamic analysis to record the values that are read and written by the sequences, and to check that values written in a sequence are not read in the other sequence. Possible research directions towards precise and effective equivalence checks involve the exploitation of information about the differences in the objects that represent the outcome of the two equivalent sequences.

Another relevant research direction involves the study of the performance of CCOracles. Both the execution order of redundant sequences and the equivalence check affect the overall performance of a CCOracle. Possible research directions involve the study of performance issues to identify performance bottlenecks and propose efficient approaches. Concurrent executions of equivalences and equivalence checks could largely improve efficiency.

Generalizing CCOracles beyond object oriented systems. This thesis explores the use of CCOracles to test object-oriented (Java) systems. The core idea can be used to address different domains, by devising suitable extensions of cross-check execution, equivalence check, and oracle deployment mechanism. Possible research directions can consider the test of Web applications where the outcome of specific system functions can be a Web page or a JSON file. Studying how to execute sequences in isolation from the same initial state and how to compare their outcomes are future work.

Generalizing Toradocu to non-structured natural language documentation. Toradocu generates test oracles from natural language semi-structured Java procedure documentation. Only an arguably small portion of the documentation is semi-structured. A lot of documentation is written as unstructured natural language, like the description comment fragment in Javadoc comments. Parsing and understanding the semantics of unstructured text is more complicated than semi-structured text. Unstructured text presents new complex challenges that are also investigated in research communities dealing with the automatic comprehension of natural language. How to exploit unstructured natural language documentation to generate test oracle is yet another important an open research issue.

Generating test oracles by exploiting other sources of information. This thesis proposes to exploit information available in the code and related artifacts to generate test oracles, and focuses on code redundancy and Javadoc comments. This opens many research directions towards the exploitation of the many other artifacts that are available in software systems. Requirements documentation, models, messages exchanged between developers are example of information sources that can be studied and from which interesting information can be mined. In principle, every document somehow encoding or describing the expected behavior of the SUT is valuable and can be considered for automatically generating test oracles.

Bibliography

[1] ai Sun, C., Wang, G., Mu, B., Liu, H., Wang, Z. and Chen, T. Y. [2011]. Metamorphic testing for web services: Framework and a case study, *Proceedings of the IEEE International Conference on Web Services*, ICWS '11, IEEE Computer Society, pp. 283–290.

[2] Angeli, G., Premkumar, M. J. and Manning, C. D. [2015]. Leveraging linguistic structure for open domain information extraction, *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, ACL '15, Association for Computational Linguistics.

[3] Banko, M., Cafarella, M. J., Soderland, S., Broadhead, M. and Etzioni, O. [2007]. Open information extraction from the web, *Proceedings of the International Joint Conference on Artifical Intelligence*, IJCAI '07, pp. 2670–2676.

[4] Baresi, L. and Young, M. [2001]. Test oracles, *Technical Report CIS-TR-01-02*, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A. `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`.

[5] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S. [2015]. The oracle problem in software testing: A survey, *IEEE Transactions on Software Engineering* **41**(5): 507–525.

[6] Beizer, B. [1990]. *Software Testing Techniques*, 2 edn, Van Nostrand Reinhold Co., New York, NY, USA.

[7] Beschastnikh, I., Brun, Y., Schneider, S., Sloan, M. and Ernst, M. D. [2011]. Leveraging existing instrumentation to automatically infer invariant-constrained models, *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '11, ACM, pp. 267–277.

[8] Billes, M., Møller, A. and Pradel, M. [2017]. Systematic black-box analysis of collaborative web applications, *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '17, ACM, pp. 171–184.

[9] Bourque, P. and Fairley, R. E. (eds) [2014]. *Guide to the Software Engineering Body of Knowledge, Version 3.0*, IEEE Computer Society.

[10] Bowring, J. F., Rehg, J. M. and Harrold, M. J. [2004]. Active learning for automatic classification of software behavior, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '04, ACM, pp. 195–205.

[11] Börger, E., Cavarra, A. and Riccobene, E. [2000]. Modeling the dynamics of uml state machines, *in* Y. Gurevich, P. W. Kutter, M. Odersky and L. Thiele (eds), *Proceedings of the*

International Workshop on Abstract State Machines - Theory and Applications, ASM ’00, Springer Berlin Heidelberg, pp. 223–241.

[12] Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A. and Pezzè, M. [2014]. Cross-checking oracles from intrinsic software redundancy, *Proceedings of the International Conference on Software Engineering*, ICSE ’14, ACM, pp. 931–942.

[13] Carzaniga, A., Gorla, A., Mattavelli, A., Pezzè, M. and Perino, N. [2013]. Automatic recovery from runtime failures, *Proceedings of the International Conference on Software Engineering*, ICSE ’13, IEEE Computer Society, pp. 782–791.

[14] Carzaniga, A., Gorla, A., Perino, N. and Pezzè, M. [2015]. Automatic workarounds: Exploiting the intrinsic redundancy of web applications, *ACM Transactions on Software Engineering and Methodologies* **24**(3): 16.

[15] Carzaniga, A., Gorla, A., Perino, N. and Pezzè, M. [2010]. Automatic workarounds for web applications, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE ’10, ACM, pp. 237–246.

[16] Carzaniga, A., Mattavelli, A. and Pezzè, M. [2015]. Measuring software redundancy, *Proceedings of the 37th International Conference on Software Engineering*, ICSE ’15, IEEE Computer Society, pp. 156–166.

[17] Chen, H. Y., Tse, T. H., Chan, F. T. and Chen, T. Y. [1998]. In black and white: an integrated approach to class-level testing of object-oriented programs, *ACM Transactions on Software Engineering and Methodology* **7**(3): 250–295.

[18] Chen, H. Y., Tse, T. H. and Chen, T. Y. [2001]. Taccle: A methodology for object-oriented software testing at the class and cluster levels, *ACM Transactions on Software Engineering and Methodology* **10**(1): 56–109.

[19] Chen, T. Y., Cheung, S.-C. and Yiu, S. M. [1998]. Metamorphic testing: a new approach for generating next test cases, *Technical report*, Department of Computer Science, Hong Kong University of Science and Technology.

[20] Chen, T. Y., Kuo, F.-C., Tse, T. H. and Zhou, Z. Q. [2003]. Metamorphic testing and beyond, *International Workshop on Software Technology and Engineering Practice*, STEP ’03, IEEE Computer Society, pp. 94–100.

[21] Cheon, Y. and Leavens, G. T. [2002]. A simple and practical approach to unit testing: The JML and JUnit way, *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP ’02, pp. 231–255.

[22] Csallner, C. and Smaragdakis, Y. [2004]. JCrasher: An automatic robustness tester for java, *Software: Practice and Experience* **34**(11): 1025–1050.

[23] Csallner, C. and Smaragdakis, Y. [2005]. Check ’n’ Crash: Combining static checking and testing, *Proceedings of the International Conference on Software Engineering*, ICSE ’05, IEEE Computer Society, pp. 422–431.

[24] Csallner, C. and Smaragdakis, Y. [2006]. DSD-Crasher: A hybrid analysis tool for bug finding, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA ’06, ACM, pp. 245–254.

[25] Davis, M. D. and Weyuker, E. J. [1981]. Pseudo-oracles for non-testable programs, *Proceedings of the ACM '81 Conference*, ACM '81, ACM, pp. 254–257.

[26] De Moura, L. and Bjørner, N. [2008]. Z3: An efficient SMT solver, *Proceedings of the International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS/ETAPS '08, Springer, pp. 337–340.

[27] De Nicola, R. and Hennessy, M. [1984]. Testing equivalences for processes, *Theoretical Computer Science* **34**(1-2): 83–133.

[28] Del Corro, L. and Gemulla, R. [2013]. Clausie: Clause-based open information extraction, *Proceedings of the International Conference on World Wide Web*, WWW '13, ACM, pp. 355–366.

[29] DeMillo, R. A., Lipton, R. J. and Sayward, F. G. [1978]. Hints on test data selection: Help for the practicing programmer, *Computer* **11**(4): 34–41.

[30] Doong, R.-K. and Frankl, P. G. [1994]. The ASTOOT approach to testing object-oriented programs, *ACM Transactions on Software Engineering and Methodology* **3**(2): 101–130.

[31] Ernst, M. D., Cockrell, J., Griswold, W. G. and Notkin, D. [2001]. Dynamically discovering likely program invariants to support program evolution, *IEEE Transactions on Software Engineering* **27**(2): 99–123.

[32] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S. and Xiao, C. [2007]. The Daikon system for dynamic detection of likely invariants, *Science of Computer Programming* **69**(1–3): 35–45.

[33] Fader, A., Soderland, S. and Etzioni, O. [2011]. Identifying relations for open information extraction, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, Association for Computational Linguistics, pp. 1535–1545.

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

[35] Fraser, G. and Arcuri, A. [2013]. Whole test suite generation, *IEEE Transactions on Software Engineering* **39**(2): 276–291.

[36] Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M. and Tonella, P. [2014]. Search-based synthesis of equivalent method sequences, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, ACM, pp. 366–376.

[37] Gorla, A. [2011]. *Automatic Workarounds: Exploiting the Intrinsic Redundancy of Software Systems*, PhD thesis, USI Università della Svizzera italiana.

[38] Gotlieb, A. [2003]. Exploiting symmetries to test programs, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '03, IEEE Computer Society, pp. 365–374.

[39] Hailpern, B. and Santhanam, P. [2002]. Software debugging, testing, and verification, *IBM Systems Journal* **41**(1): 4–12.

[40] Hartman, A. [2002]. Is issta research relevant to industry?, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '02, ACM, pp. 205–206.

[41] Holzmann, G. J. [1997]. The model checker spin, *IEEE Transactions on Software Engineering* **23**(5): 279–295.

[42] Jackson, D. [2002]. Alloy: a lightweight object modelling notation, *ACM Transactions on Software Engineering and Methodology* **11**(2): 256–290.

[43] Juergens, E., Deissenboeck, F., Hummel, B. and Wagner, S. [2009]. Do code clones matter?, *Proceedings of the International Conference on Software Engineering*, ICSE '09, IEEE Computer Society, pp. 485–495.

[44] Just, R. [2014]. The Major mutation framework: Efficient and scalable mutation analysis for Java, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '14, pp. 433–436.

[45] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R. and Fraser, G. [2014]. Are mutants a valid substitute for real faults in software testing?, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, ACM, pp. 654–665.

[46] Just, R., Schweiggert, F. and Kapfhammer, G. M. [2011]. Major: An efficient and extensible tool for mutation analysis in a Java compiler, *Proceedings of the International Conference on Automated Software Engineering*, ASE '11, IEEE Computer Society, pp. 612–615.

[47] Kanewala, U. [2014]. Techniques for automatic detection of metamorphic relations, *Proceedings of the International Conference on Software Testing, Verification and Validation Workshop*, ICSTW '14, IEEE Computer Society, pp. 237–238.

[48] Kanewala, U. and Bieman, J. M. [2013]. Using machine learning techniques to detect metamorphic relations for programs without test oracles, *ISSRE*, ISSRE '13, IEEE Computer Society, pp. 1–10.

[49] Kawachiya, K., Ogata, K., Silva, D., Onodera, T., Komatsu, H. and Nakatani, T. [2007]. Cloneable JVM: a new approach to start isolated Java applications faster, *Proceedings of the International Conference on Virtual Execution Environments*, VEE '07, pp. 1–11.

[50] Khamis, N., Witte, R. and Rilling, J. [2010]. Automatic quality assessment of source code comments: The JavadocMiner, *Proceedings of the International Conference on Applications of Natural Language to Information Systems*, NLDB '10, Springer, pp. 68–79.

[51] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. [1997]. Aspect-oriented programming, *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '97, Springer-Verlag, pp. 220–242.

[52] Kusner, M. J., Sun, Y., Kolkin, N. I. and Weinberger, K. Q. [2015]. From word embeddings to document distances, *Proceedings of the International Conference on International Conference on Machine Learning*, ICML '15, pp. 957–966.

[53] Lano, K. and Haughton, H. [1996]. *Specification in B: An Introduction Using the B Toolkit*, World Scientific Publishing Co., Inc.

[54] Lee, D. and Yannakakis, M. [1996]. Principles and methods of testing finite state machines—a survey, *Proceedings of the IEEE* **84**(8): 1090–1123.

[55] Liu, H., Liu, X. and Chen, T. Y. [2012]. A new method for constructing metamorphic relations, *Proceedings of the International Conference on Quality Software*, QSIC '12, IEEE Computer Society, pp. 59–68.

[56] Ma, L., Artho, C., Zhang, C., Sato, H., Gmeiner, J. and Ramler, R. [2015]. GRT: Program-analysis-guided random testing, *Proceedings of the International Conference on Automated Software Engineering*, ASE '15, ACM, pp. 212–223.

[57] Manning, C. D., Surdeanu, M., Bauer, J., Finkel, J. R., Bethard, S. J. and McClosky, D. [2014]. The Stanford CoreNLP natural language processing toolkit, *Proceedings of the Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, ACL '14, Association for Computational Linguistics, pp. 55–60.

[58] Mariani, L. and Pastore, F. [2008]. Automated identification of failure causes in system logs, *Proceedings of the International Symposium on Software Reliability Engineering*, ISSRE '08, IEEE Computer Society, pp. 117–126.

[59] Mariani, L., Pezzè, M., Riganelli, O. and Santoro, M. [2012]. Autoblacktest: Automatic black-box testing of interactive applications, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, IEEE Computer Society, pp. 81–90.

[60] Marneffe, M.-C., MacCartney, B. and Manning, C. [2006]. Generating typed dependency parses from phrase structure parses, *Proceedings of the International Conference on Language Resources and Evaluation*, LREC '06, European Language Resources Association (ELRA), pp. 449–454.

[61] Mattavelli, A. [2016]. *Software Redundancy: What, Where, How*, PhD thesis, Università della Svizzera italiana (USI).

[62] Mattavelli, A., Goffi, A. and Gorla, A. [2015]. Synthesis of equivalent method calls in Guava, *Proceedings of the 7th International Symposium on Search-Based Software Engineering*, SSBSE '15, Springer, pp. 248–254.

[63] McBurney, P. W. and McMillan, C. [2014]. Automatic documentation generation via source code summarization of method context, *Proceedings of the International Conference on Program Comprehension*, ICPC '14, ACM, pp. 279–290.

[64] Meyer, B. [1988]. Eiffel: A language and environment for software engineering, *Journal of Systems and Software* **8**(3): 199–246.

[65] Meyer, B. [1992]. Applying "design by contract", *IEEE Computer* **25**(10): 40–51.

[66] Mikolov, T., Sutskever, I., Chen, K., Corrado, G. and Dean, J. [2013]. Distributed representations of words and phrases and their compositionality, *Proceedings of the International Conference on Neural Information Processing Systems*, NIPS '13, pp. 3111–3119.

[67] Miller, B. P., Fredriksen, L. and So, B. [1990]. An empirical study of the reliability of unix utilities, *Communications of the ACM* **33**(12): 32–44.

[68] Murphy, C., Shen, K. and Kaiser, G. [2009]. Using jml runtime assertion checking to automate metamorphic testing in applications without test oracles, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '09, IEEE Computer Society, pp. 436–445.

[69] Nguyen, C. D., Marchetto, A. and Tonella, P. [2013]. Automated oracles: An empirical study on cost and effectiveness, *Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE '13, ACM, pp. 136–146.

[70] Pacheco, C., Lahiri, S. K., Ernst, M. D. and Ball, T. [2007]. Feedback-directed random test generation, *Proceedings of the International Conference on Software Engineering*, ICSE '07, ACM, pp. 75–84.

[71] Pandita, R., Xiao, X., Zhong, H., Xie, T., Oney, S. and Paradkar, A. [2012]. Inferring method specifications from natural language api descriptions, *Proceedings of the International Conference on Software Engineering*, ICSE '12, IEEE Computer Society, pp. 815–825.

[72] Parnas, D. L., Madey, J. and Iglewski, M. [1994]. Precise documentation of well-structured programs, *IEEE Transactions on Software Engineering* **20**(12): 948–976.

[73] Pate, J. R., Tairas, R. and Kraft, N. A. [2013]. Clone evolution: a systematic review, *Journal of Software: Evolution and Process* **25**: 261–283.

[74] Pawelka, T. and Jürgens, E. [2015]. Is this code written in English? A study of the natural language of comments and identifiers in practice, *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, ICSME '15, IEEE Computer Society, pp. 401–410.

[75] Pennington, J., Socher, R. and Manning, C. D. [2014]. GloVe: Global vectors for word representation, *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '14, pp. 1532–1543.

[76] Perino, N. [2014]. *A Self-Healing Framework for General Software Systems*, PhD thesis, USI Università della Svizzera italiana.

[77] Peters, D. K. and Parnas, D. L. [1998]. Using test oracles generated from program documentation, *IEEE Transactions on Software Engineering* **24**(3): 161–173.

[78] Peters, D. and Parnas, D. L. [1994]. Generating a test oracle from program documentation: Work in progress, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '94, ACM, pp. 58–65.

[79] Pezzè, M. and Young, M. [2007]. *Software Testing and Analysis: Process, Principles and Techniques*, Wiley.

[80] Pezzè, M. and Zhang, C. [2015]. Automated test oracles: A survey, *Advances in Computers*, Vol. 95, Elsevier, pp. 1–48.

[81] Rahman, F., Bird, C. and Devanbu, P. [2012]. Clones: What is that smell?, *Empirical Software Engineering* **17**(4-5): 503–530.

[82] Rojas, J. M., Fraser, G. and Arcuri, A. [2015]. Automated unit test generation during software development: A controlled experiment and think-aloud observations, *Proceedings of the International Symposium on Software Testing and Analysis*, ISSTA '15, ACM, pp. 338–349.

[83] Rubio-González, C. and Liblit, B. [2010]. Expect the unexpected: Error code mismatches between documentation and the real world, *Proceedings of the Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, ACM, pp. 73–80.

[84] Schmitz, M., Bart, R., Soderland, S., Etzioni, O. et al. [2012]. Open language learning for information extraction, *Proceedings of the Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, Association for Computational Linguistics, pp. 523–534.

[85] Segura, S., Fraser, G., Sanchez, A. B. and Ruiz-Cortés, A. [2016]. A survey on metamorphic testing, *IEEE Transactions on Software Engineering* **42**(9): 805–824.

[86] Shavit, N. and Touitou, D. [1997]. Software transactional memory, *Distributed Computing* **10**(2): 99–116.

[87] Shrestha, K. and Rutherford, M. J. [2011]. An empirical evaluation of assertions as oracles, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '11, IEEE Computer Society, pp. 110–119.

[88] Spivey, J. M. [1989]. *The Z Notation: A Reference Manual*, Prentice Hall International Series in Computer Science, Prentice Hall.

[89] Sridhara, G., Pollock, L. and Vijay-Shanker, K. [2011]. Automatically detecting and describing high level actions within methods, *Proceedings of the International Conference on Software Engineering*, ICSE '11, IEEE Computer Society, pp. 101–110.

[90] Steidl, D., Hummel, B. and Juergens, E. [2013]. Quality analysis of source code comments, *Proceedings of the International Conference on Program Comprehension*, ICPC '13, IEEE Computer Society, pp. 83–92.

[91] Tan, L., Yuan, D., Krishna, G. and Zhou, Y. [2007]. /* icomment: Bugs or bad comments? */, *Proceedings of the Symposium on Operating Systems Principles*, SOSP '07, ACM, pp. 145–158.

[92] Tan, L., Yuan, D. and Zhou, Y. [2007]. Hotcomments: How to make program comments more useful?, *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems*, HOTOS '07, USENIX Association, pp. 19:1–19:6.

[93] Tan, L., Zhou, Y. and Padioleau, Y. [2011]. aComment: Mining annotations from comments and code to detect interrupt related concurrency bugs, *Proceedings of the International Conference on Software Engineering*, ICSE '11, pp. 11–20.

[94] Tan, S. H., Marinov, D., Tan, L. and Leavens, G. T. [2012]. @tComment: Testing Javadoc comments to detect comment-code inconsistencies, *Proceedings of the International Conference on Software Testing, Verification and Validation*, ICST '12, IEEE Computer Society, pp. 260–269.

[95] Tretmans, J. [1996]. Test generation with inputs, outputs and repetitive quiescence, *Technical Report TR-CTIT-96-26*, University of Twente, Centre for Telematics and Information Technology (CTIT).

[96] Warmer, J. B. and Kleppe, A. G. [1998]. *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley.

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

[98] Wu, Q., Wu, L., Liang, G., Wang, Q., Xie, T. and Mei, H. [2013]. Inferring dependency constraints on parameters for web services, *Proceedings of the International Conference on World Wide Web*, WWW '13, ACM, pp. 1421–1432.

[99] Xiao, X., Paradkar, A., Thummalapenta, S. and Xie, T. [2012]. Automated extraction of security policies from natural-language software documents, *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '12, ACM.

[100] Ye, M., Feng, B., Zhu, L. and Lin, Y. [2006]. Automated test oracle based on neural networks, *Proceedings of the International Conference on Cognitive Informatics*, ICCI '06, IEEE Computer Society, pp. 517–522.

[101] Zhai, J., Huang, J., Ma, S., Zhang, X., Tan, L., Zhao, J. and Qin, F. [2016]. Automatic model generation from documentation for Java API functions, *Proceedings of the International Conference on Software Engineering*, ICSE '16, IEEE Computer Society, pp. 380–391.

[102] Zhang, J., Chen, J., Hao, D., Xiong, Y., Xie, B., Zhang, L. and Mei, H. [2014]. Search-based inference of polynomial metamorphic relations, *Proceedings of the International Conference on Automated Software Engineering*, ASE '14, ACM, pp. 701–712.

[103] Zhong, H. and Su, Z. [2013]. Detecting API documentation errors, *Proceedings of the Conference on Object-Oriented Programming Systems and Applications*, OOPSLA '13, ACM, pp. 803–816.

[104] Zhong, H. and Su, Z. [2015]. An empirical study on real bug fixes, *Proceedings of the International Conference on Software Engineering*, ICSE '15, IEEE Computer Society, pp. 913–923.

[105] Zhong, H., Zhang, L., Xie, T. and Mei, H. [2009]. Inferring resource specifications from natural language API documentation, *Proceedings of the International Conference on Automated Software Engineering*, ASE '09, IEEE Computer Society, pp. 307–318.

[106] Zhou, Y., Gu, R., Chen, T., Huang, Z., Panichella, S. and Gall, H. [2017]. Analyzing APIs documentation and code to detect directive defects, *Proceedings of the International Conference on Software Engineering*, ICSE '17, IEEE Computer Society, pp. 27–37.

[107] Zhou, Z. Q., Zhang, S., Hagenbuchner, M., Tse, T. H., Kuo, F.-C. and Chen, T. Y. [2012]. Automated functional testing of online search services, *Software Testing, Verification and Reliability* **22**(4): 221–243.