On Oracles for Automated Diagnosis and Repair of Software Bugs

Martin Monperrus

HABILITATION À DIRIGER DES RECHERCHES DE L'UNIVERSITÉ LILLE 1, SCIENCES ET TECHNOLOGIES

Spécialité

Informatique

École doctorale Sciences Pour l'Ingénieur (SPI)

Présentée par

Martin Monperrus

Sujet de l'habilitation :

On Oracles for Automated Diagnosis and Repair of Software Bugs

(version of Thursday 26th May, 2016)

Soutenue le 3 mai 2016

devant le jury composé de :

Jean-Paul	Delahaye	Professeur Émérite à l'Université Lille 1	Président
Yves	Ledru	Professeur à l'Université J. Fourier Grenoble	Rapporteur
Martin	Rinard	Professeur au Massachussets Institute of Technology	Rapporteur
Walter	Tichy	Professeur au Karlsruhe Institute of Technology	Rapporteur
Laurence	Duchien	Professeur à l'Université Lille 1	Examinatrice

Foreword

This document is a manuscript for the the French degree "Habilitation à Diriger des Recherches" (HDR). This degree comes after the PhD, typically a couple of years after the defense. It aims at acknowledging two capabilities: 1) leading a consistent and successful line of research 2) well supervising students, in particular PhD students. In France, this degree is also a pre-requisite for applying to certain professional promotions and positions.

This document focuses on my work on automatic diagnosis and repair done over the past years. Among my past publications, it highlights three contributions, respectively published in ACM Transactions on Software Engineering and Methodology (TOSEM), IEEE Transactions on Software Engineering (TSE) and Elsevier Information & Software Technology (IST).

More than stapling them together, my goal is to show that they share something deep, that they are founded on a unifying concept, which is the one of oracle. The introduction of this manuscript, as well as all chapter introductions, refer to the concept of oracle and explain the relationship between the contribution and this concept.

Acknowledgements

The work presented here is pure academic work, done in total freedom and driven by scientific curiosity above all motivations. I really want to acknowledge the University of Lille for having given me an academic position of great quality with true academic freedom. I thank my collaborators there, in particular Laurence Duchien and Lionel Seinturier, who have always supported my work to the greatest extent. I also thank all my colleagues at the computer science laboratory LIFL/CRISTAL and at Inria, for the key financial and administrative support.

I'd like to thank all my direct collaborators having worked on bugs with me: Alexandre Bartel, Matias Martinez, Benoit Cornu, Vincenzo Musco, Thomas Durieux, Marcel Bruch, Sebastian Marcote, Favio DeMarco, Diego Mendez, Anthony Maia, Maxence Germain de Montauzan, Jifeng Xuan, Nicolas Petitprez, Gérard Paligot, Philippe Preux, Daniel Le Berre, Romain Rouvoy, Raphael Marvie, Earl Barr, Jacques Klein, Yves Le Traon, Mira Mezini. I present my apologies to those I've forgotten.

A special acknowledgement goes to Benoit Baudry, the best discussion partner ever about software correctness and oracles.

I am really grateful to the HDR committee for accepting to assess this work: Jean-Paul Delahaye (University of Lille), Yves Ledru (University of Grenoble), Martin Rinard (Massachussets Institute of Technology), and Walter Tichy (Karlsruhe Institute of Technology). It is a great honor if I am considered as one of their peers.

Contents

Chapter 1

Introduction

In ancient Greece, oracles were persons capable of predicting the future. In software testing, oracles are procedures capable of assessing software correctness. The relation between them is that software oracles can be stated before actual implementation, meaning that they somehow predict the future behavior, as Greek oracles do. The key difference is that software oracles are to be enforced, meaning that software would make it to production only if the actual behavior is corresponding to the expected one. If the actual behavior is not the expected one, we say that there is a bug. Software oracles specify the future.

On the contrary, Greek oracles such as the Delphi's Pythia, were not capable of enforcing that the future indeed meets their prediction. Nobody has ever specified the future! (However, it may be argued that when the future does not correspond to Greek oracles' predictions, it is in fact a bug in the reality).

My interest in software oracles comes from my work in automatic repair and fault localization over the past years. To devise advanced and efficient automatic repair and diagnosis techniques, one needs a close, almost intimate knowledge of the underlying oracles. With hindsight, I think that the concept of "oracle" is indeed the backbone of my recent research. In this manuscript, I revisit three pieces of work in the area of bug detection, automatic repair, and software resilience under this unifying concept. What is interesting is that those three contributions use oracles of different nature: input-output oracles, generic oracles and statistical oracles.

An *input-output oracle* states the expected output for a given input. For instance "assert 6 = factorial(3)" is such an oracle. For those oracles, both the input and output can be much more complex structures than in this example, such as sequences of events or graphs or interconnected objects. A *generic oracle* states properties that hold for many domains. The classical example is "software should not crash". In those two kinds of oracles, somebody (a domain expert, an engineer) states the oracle. On the contrary, oracles can be the result of a collective decision procedure, where the expected behavior reflects "normality", reflects what the majority does. We call them "statistical oracles". For instance, if 99% of web developers use hashing to store user passwords, it can be considered that hashing must be used for this purpose (even without any knowledge of security and cryptography). For those statistical oracles, nobody in isolation is capable of stating the correct behavior, only the aggregation matters.

1.1 Focus of this Manuscript

In Chapter 2, I present a contribution about statistical oracles. In the context of object-oriented software, we have defined a notion of context and normality that is specific to a fault class:

missing method calls [70, 67]. We first mine large amounts of code for regularities, and then detect violations according to those inferred regularities. The violations have two characteristics, they have a very low rate of false positives, and they come with a concrete recommendation of what to do to fix the detected bug. This new statistical oracle is unique with respect to those two properties.

In Chapter 3, oracles are at the heart of an automatic repair process. We consider the problem statement of test suite based repair, where the goal is to generate a patch that makes a failing test case passing without introducing any regression. We have proposed a test-suite based repair system called Nopol, which targets the automatic repair of conditionals statements [23, 24]. This system is based on the idea of refining the knowledge given by the violation of the oracle of the failing test case into finer-grain information. We call this finer-grain information a micro-oracle. In Nopol, a micro-oracle is at the level of a single boolean expression, and is obtained by speculatively executing branches of if/then/else statements. By considering micro-oracles, we are capable of obtaining at the same time a precise fault localization diagnostic and a well-formed input-output specification to be used for program synthesis.

In Chapter 4, our main concern is to reason about resilience against unhandled exceptions. In mainstream programming languages, exception handling suffers from a lack of reasoning – whether static or dynamic – about the expected behavior under unhandled exceptions. We propose two novel contracts that are dedicated to exceptions and give programmers confidence about the system capabilities to handle unknown program states. Those contracts are oracles when associated with a novel verification technique called short-circuit testing. This technique consists of injecting exception at appropriate locations and moments during test suite execution.

My research is epistemologically founded on empiricism. We evaluate the effectiveness of our statistical oracle by finding bugs in 146 845 extracted from 5 large-scale open-source software applications written in Java. We show that our automatic repair systems is capable of repairing 18 real bugs in Java software that consists of up to 79701 lines of code spread over 1158 classes. We analyze the resilience against unhandled exceptions for 214 catch blocks extracted from 9 open source Java projects.

> The related work often takes a narrow perspective on oracles, for instance by only focusing on input-output oracles or formal contracts. The uniqueness of this manuscript is to present three technical contributions that use three different kinds of oracles within a higher-level perspective of software correctness. We will eventually discuss in Section 5 that they are all related to a rather unexplored notion that I would call "behavioral neighborhood".

1.2 Core Definitions

This thesis is founded on the concept of "oracle". While this word is prevalent in the software engineering literature, and can be traced back to the 1970s [43], the first serious survey on this topic appeared in 2015 in IEEE Transactions on Software Engineering [4]. There has also been a notable article on this topic in the proceedings of ICSE 2011 [90]. Let us first try to come up with a definition of "oracle" based on those two papers, a definition that would fit with the work presented here.

According to Staats, Bar and their colleagues [90, 4], an oracle:

- "is a procedure that distinguishes between the correct and incorrect behaviors of the System Under Test (SUT)." [4]

- "is a predicate that determines whether a given test activity sequence is an acceptable behavior of the SUT" [4]
- "is intended to determine, for each test, if the program has violated the specification." [90]
- "determines, for a given program and test if the test passes." [90]
- "determines if the result of executing a program p using a test t is correct." [90]

Those five definitions come from only two papers yet do not perfectly overlap. While they are along the same line, they use different concepts. For instance, only the first and last refer to the notion of correctness. Let us now remix those five interpretations and set up a definition, which is hopefully better and holds for the rest of this manuscript.

> Definition: An oracle determines, for a given program and test, if the actual program behavior is not acceptable or has violated the expected behavior stated in the specification.

In particular, this definitions moves away from the notion of "predicate" and "correctness" which are too binary. I want to accept probabilistic oracles [4] as well as the "acceptability oriented computing" idea of correctness [84], hence the use to "acceptable" in the definition. Also, not using the notion of "execution" in the definition widens the scope to oracles that are not based on runtime, in particular all kinds of static and concolic[1] analysis based oracles.

Let us now give a concrete example. The following code excerpt shows two oracles that assess the expected size and content of a list after creation and addition. The oracles are expressed with a classical keyword "assert" that exist in many programming languages and libraries. This is a paradigmatic example of an input-output based oracle, as used daily by millions of developers when writing test suites.

```
1 l = new List()
2 l.add(42)
3
4 assert l.size() == 1
5 assert l.contains(42)
```

Coming back to terminology, we wholeheartedly accept and use the classical terminology and definitions of fault, error and failure, as very nicely summarized by Avizienis et al. [2]: unacceptable behaviors are failures, due to activated faults that have propagated as errors. In this manuscript, we also use the word "bug" as an umbrella term above fault, error and failure, corresponding to the intuitive, gut-feeling based meaning of "bug" for all developers and software users around the world.

When an oracle is violated, the first task is to diagnose the cause of the problem. The diagnosis concern has been explored in different threads in the literature. The early work refers to "automatic debugging" [46] and "model-based diagnosis" in related fields [39, 14]. Today, the term "fault localization" is very much used for this purpose (e.g. [87]), and in the context of software testing, it often refers to spectrum based fault localization [72]. Diagnosis can be seen as a kind of automatic explanation of why an oracle has been violated. We thus define diagnosis as follows.

[1] a concolic analysis mixes symbolic and concrete execution.

> Definition: Automatic software diagnosis is the discipline of automatically identifying the root cause and the causal chain of events that result in an unacceptable behavior of a software system.

When an oracle is violated, the ultimate goal beyond diagnosis is automatic repair. Loosely defined, automatic repair is the process of fixing bugs automatically. With respect to our definition of oracle, *automatic repair is the transformation of an unacceptable behavior of a program execution into an acceptable one.* The transformation can happen at the level of the program code, this is behavioral repair, or at the level of the program state, this is state repair. We refer to our survey for an overview of this field [65].

> Definition: Automatic software repair is the process of automatically preventing the deleterious consequences of faults or failures in software systems, either by automatically removing the faults, automatically stopping the error propagation, or automatically activating the appropriate recovery mechanisms.

Those definitions should be qualified enough for supporting the unambiguous comprehension of the rest of this manuscript.

1.3 Overview of My Other Work on Automatic Diagnosis and Repair

This manuscript highlights three specific contributions done within a larger research thread on the topic of automatic diagnosis and repair. I briefly present now the big picture of my work in this area.

§1 Foundations of Automatic Repair My ICSE'14 article [64] is an essay on the problem statement and the evaluation of automatic software repair. It highlights dangerous pitfalls in this area, in particular if one does not carefully qualify the class of bugs under repair and the evaluation setup. It proposes a distinction between behavioral repair (changing the program code) and state repair (changing the program state at runtime). This distinction grounds my survey of automatic repair [65]. We have also shown that there are two main families of repair techniques [66]: so-called "plastic" techniques are more generic, randomized and evolutionary, "rigid" techniques are more bug-class specific and symbolic in their reasoning.

§2 Automatic Diagnosis In [67, 70], we have proposed a novel technique to automatically detect a certain class of bugs (missing method calls), which will be presented in Chapter 2. The detection is based on anomaly detection, where an anomaly is a violation of common patterns mined from existing code. The detection comes with a diagnosis: the system recommends the concrete method that is missing, which is considered as the root cause of the detected bug.

The goal of fault localization is to automatically identify the faulty piece of code responsible for an incorrect behavior. In [99], we have shown that automatically transforming test cases can improve spectrum-based fault localization process. In [98], we have used a machine-learning technique called “learning-to-rank” to post-process the fault localization results.

In [16], we have proposed a novel technique to automatically diagnose the root cause of null dereferences (null pointer exceptions in Java). The technique is based on code transformation,

so that the transformed code automatically collects at runtime the causality trace of the null dereference.

The first step to find the root cause of a production bug is to reproduce it. In [100], we have proposed a novel technique to reproduce exceptions that happen in production. The reproduction is based on test case mutations. Our idea is fundamentally different from existing work, it is based on the insight that the behaviors exercised in production are close to the ones exercised during testing.

§3 Automatic Patch Generation A patch is a result of the application of one or more repair operators (aka repair actions). For instance, a repair operator is "change an if condition". In [58], we have shown that not all repair operators are equally important. We have mined from 89993 commits of Java software the frequency of each repair operators and set up a simulation experiment to show that taking into account the frequency speeds up the repair.

In [59], we have conducted an empirical study to validate the assumption that some bugs are repaired by only re-arranging existing code (i.e. without inventing new code). This "redundancy assumption" is behind the famous Genprog system [36] and is a way to dramatically reduce the search space of automatic repair. Our study was the very first to empirically validate this assumption.

In [23, 97], we have proposed a patch generation system for Java called Nopol (which will be presented in Chapter 3). This system is based on speculative execution and code synthesis to generate patches for conditional bugs (wrong if conditions and missing preconditions). The speculative execution technique is called angelic value mining, it is the dynamic counterpart of angelic debugging that is done with symbolic execution. The code synthesis phase is based on component-guided synthesis. We have shown that Nopol is able to repair bugs from large scale Java software [24].

In [56], we have remixed Nopol in a system called Infinitel. Infinitel is the very first system capable of automatically repairing infinite loops by proposing a source code patch. The patch is a new condition to be put in a while loop. Astor is a generic framework for building automatic repair systems for Java [57], we have used it to provide a faithful implementation of Genprog for Java.

In [33], we have described a vision for repairing bugs in mobile applications based on the crowd of users. The vision blends user feedback mining, adaptive monitoring and patch generation for automatically repairing bugs. It perfectly fits in the context of centralized app stores, that would then become self-improving in the pure sense of autonomic computing.

§4 Runtime Resilience In [18], we qualify the resilience of software with respect to unhandled exceptions with two novel contracts, this will be the topic of Chapter 4. We describe a dynamic technique for verifying those contracts based on existing tests, it is called "short-circuit testing". The result of the contract verification phase enables us to propose possible "catch stretching" that is an automatic increase in resilience with respect to unhandled exceptions.

NpeFix [17], is a system built for repairing null dereferences at runtime. It proposes a set of 9 generic strategies that deal with crashing exceptions. It is based on the idea that alternative execution semantics can be provided at runtime in place of abruptly stopping execution.

§5 Software Engineering for Repair Automatic repair is a new concern for software engineering, it comes on top of the classical concerns such as reuse, modularity, etc. In [96], we have shown that test suites are suboptimal with respect to repair. Consequently, we have proposed Banana-refactoring, a technique for refactoring test cases on the fly with the goal of improving the effectiveness of repair. However, the technique is more generic and can be applied to other

dynamic analysis techniques. In [99], we have shown that test suites are suboptimal with respect to fault localization, and proposed a test case manipulation called test case atomization, which improves the fault localization results.

1.4 Research Supervision

As explained in the Foreword, one of the goals of this document is to demonstrate my capability of fruitfully supervising students. The following table summarizes the joint-work with students, incl. Alexandre Bartel, PhD student whose work is not included in this manuscript.

Role	Name	
PhD main advisor	Matias Martinez	Chapter 3
PhD main advisor	Benoit Cornu	Chapter 4
PhD main advisor	Alexandre Bartel	[6, 5]
PhD support	Marcel Bruch	Chapter 2
Postdoc	Jifeng Xuan	Chapter 3
Master's & Bachelor degree advisor	Favio De Marco, Sebastian Marcote, Maxence de Montauzan, Maxime Clément	Chapter 3, 4

Chapter 2

A Statistical Oracle for Bug Detection

This chapter is based on content from:

- Detecting Missing Method Calls as Violations of the Majority Rule (Martin Monperrus, Mira Mezini), In Transactions on Software Engineering and Methodology, ACM, volume 22, pp 1–25, 2013.
- Detecting Missing Method Calls in Object-Oriented Software (Martin Monperrus, Marcel Bruch, Mira Mezini), In Proceedings of the 24th European Conference on Object-Oriented Programming, pp 2–25, Springer, 2010.

Contents

One oracle of correctness is "normalness", meaning that a piece of code is expected to do what every other related piece of code does. For instance, if all functions that write to the console call function "flush", a missing call to flush in the same context can be considered as a bug. Let's call those oracles "statistical". Here, statistical is first taken in a loose manner, it refers to the fact that we need a lot of data to consider a behavior as recurrent and normal. However, in some special settings, it is even possible to define quantitative confidence metrics on the oracle. In the best case, those confidence metrics indeed reflect the classical Type I and Type II errors in statistics. A Type I error (aka a false positive) is when the oracle is wrong, meaning that the code considered as incorrect by the oracle is indeed correct. A Type II error (aka a false negative) is when a bug is missed by the statistical oracle. When such quantitative confidence metrics can be defined and measured, the term "statistical oracle" can be understood literally.

To sum up, a statistical oracle defines, in the context of software, what is "deviant behavior" [25], also referred to as an "anomaly" [40]. It is defined in opposition to what is normal, recurrent, done by the majority. We will briefly discuss in Section 2.1 the most important characterizations of "deviant code" of the literature.

> While the related work has mostly explored statistical oracles for low-level, procedural code (mostly written in C) [25, 53], there is little work on statistical oracles for high level object-oriented programs.

I have contributed to this field with a new statistical oracle for a specific class of bugs: "missing method calls". The definition of this class of bug is almost in its name: in the context of object-oriented software a method is said to be missing if a call should be made at a certain location. For instance, when using the specific graphical user interface library called SWT, calls to method `dispose` have to be made at certain locations, otherwise performance and memory bugs arise.

This new oracle is founded on the concept of *type-usages*. A type-usage is a list of method calls on a variable of a given type occurring somewhere within the context of a particular method body. Our intuition is that a type-usage is likely to host a defect if there are few similar type-usages and a majority of slightly different other ones. We design a new metric called *S-score*, which formalizes this idea. The *S-score* measures the degree of deviance of type-usages: the higher the *S-score*, the higher the type-usage smells.

We evaluate the effectiveness of our statistical oracles by finding bugs in 146 845 type-usages extracted from 5 large-scale open-source software applications written in Java. This evaluation shows that the approach is very precise (few false positives) and uncovers a wide range of development errors.

To summarize, the contributions of this chapter are:

- A comprehensive set of empirical facts on the problems caused by missing method calls. We present +30 examples of real software artifacts affected by missing method calls, a comprehensive study of this problem in the Eclipse Bug Repository, and an extensive analysis of the missing calls that our tool found in the Eclipse code base.
- A new characterization of *deviant code* for missing method calls, based on the concept of "type-usage". We also propose a technique to transform deviance data as concrete recommendations: the system tells the developer what methods seem to be missing at a particular place in code.
- An evaluation of our technique to detect missing method calls done over 146 845 type-usages extracted from 5 large-scale open-source software applications written in Java.

The reminder of this chapter is structured as follows. Section 2.1 briefly discusses the related work on statistical oracles. In Section 2.2, we elaborate on empirical facts about missing method calls. Section 2.3 presents our approach and the underlying algorithm. Section 2.4 describes the evaluation of our technique. Section 2.5 concludes the paper and sketches areas of future work.

2.1 Main Related Work on Bug Detection With Statistical Oracles

This section is a brief presentation of the most notable literature done on bug detection with statistical oracles.

Engler et al. [25] present a generic approach to infer errors in system code as violations of implicit contracts. They introduce six templates based on statistical analysis. For instance, a template checks that a specific lock protects a specific variable, where the pair lock name, variable name is mined in the software code base. They then use the Z statistic for proportions to filter out false positives.

Another interesting approach from the OS research community is PR-Miner [53]. PR-Miner addresses missing procedure calls in system code. PR-Miner uses frequent item set mining to find missing procedure calls .

Wasylkowski et al. [93] searched for *locations in programs that deviate from normal object usage – that is, defect candidates.* Their definition of object usage anomalies is also based on method calls with method call ordering and object states. Nguyen et al. [74] introduced the concept of *groum* to refer to graph-based representation of source code mined from existing code. Their paper contains a proposal of using *groums* to detect anomalies.

Other pattern-specific approaches to detecting deviant code include the one from Williams and Hollingsworth [95], who propose an automatic checking of return values of function calls. Chang et al. [13] target another specific pattern of deviant code: neglected tests in conditional statements.

Ernst et al [28], Hangal and Lam [40], and Csallner et al. [19] mine invariants. Yang et al [101] and Dallmeier et al. [21] mine traces for ordered sequences of functions.

There are also approaches that devise statistical oracles based on historical artifacts of software (e.g. two different versions of the same software package or the full revision control data) [54, 75]. For instance, Livshits et al. [54] extract common change patterns from software revision histories. Hence, an anomaly is detected when 1) a large number of occurrences of the same kind of bug arise and 2) a large number of corrections of these bugs is observed.

Overall, what was really missing at the time of this contribution was a powerful statistical oracles for high level object-oriented programs, which today represent a very large proportion of software applications.

2.2 Fault Class: Missing Method Calls

This section presents qualitative and quantitative empirical facts in order to characterize missing method calls.

2.2.1 Concrete Example

Let us tell a little story that shows that missing method calls are likely and can be the source of real problems. The story is inspired from several real world posts to Internet forums and mailing lists. Sandra is a developer who wants to create a dialog page in Eclipse. She finds a

class corresponding to her needs in the API named `DialogPage`. Using the new-class-wizard of Eclipse, she automatically gets a code snippet containing the methods to override, shown below:

```
public class MyPage extends DialogPage {
 @Override
 public void createControl(Composite parent) {
  // Auto-generated method stub
 }
}
```

Since the API documentation of `DialogPage` does not mention special things to do, Sandra writes the code for creating a control, a `Composite`, containing all the widgets of her own page. Sandra knows that to register a new widget on the UI, one passes the parent as parameter to the Composite constructor.

```
 public void createControl(Composite parent) {
   Composite mycomp = new Composite(parent);
   ....
 }
```

Sandra gets the following error message at the first run of her code (the error log is unfortunately empty)!

```
An error has occurred. See error log for more details.
org.eclipse.core.runtime.AssertionFailedException
null argument:
```

When extending a framework class, there are often some contracts of the form "*call method x when you override method y*", which need to be followed. The Eclipse JFace user-interface framework expects that an application class extending `DialogPage` calls the method `setControl` within the method that overrides the framework method `createControl`. However, the documentation of `DialogPage` does not mention this implicit contract; Sandra thought that registering the new composite with the parent would be sufficient.

The described scenario pops up regularly in the Eclipse newsgroup[1] and shows that one can easily fail to make important method calls. Furthermore, the resulting runtime error that Sandra got is really cryptic and it may take time to understand and solve it.

Sandra had to ask a question on a mailing list to discover that this problem comes from a missing call to `this.setControl`. After the addition of `this.setControl(mycomp)` at the end of her code, Sandra could finally run the code and commit it to the repository; yet, she lost 2 hours in solving this bug related to a missing method call.

2.2.2 An Empirical Study of Bug Reports Related to Missing Calls

Some missing method calls are not detected before leaving the developer's machine. We have searched for bug descriptions related to missing method calls in the Eclipse Bug Repository[2].

Our search process went through the following steps: 1) establish a list of syntactic patterns which could indicate a missing method call, 2) for each pattern of the list created in the previous step, query the bug repository for bug descriptions matching the pattern 3) read the complete

[1] cf. the Google results of "`setcontrol+site:https://www.eclipse.org/forums/`")

[2] `http://bugs.eclipse.org`

Table 2.1: The number of bug reports in the Eclipse Bug Repository per syntactic pattern related to missing method calls. The second column shows the number of occurrences of the pattern, the third one is the number of bug reports that are actually related to missing method calls after manual inspection.

Pattern	Matched	Confirmed
"should call"	49	26 (53%)
"does not call"	39	28 (72%)
"is not called"	36	26 (72%)
"should be called"	34	9 (26%)
"doesn't call"	16	13 (81%)
"do not call"	10	6 (60%)
"are not called"	7	0 (0%)
"must call"	7	4 (57%)
"don't call"	6	2 (33%)
"missing call"	6	2 (33%)
"missing method call"	1	1 (100%)
Total	211	117 (55%)

description of each resulting bug report to assess whether it is really related to missing method calls.

To know that a report is really due to a missing method call or not, we read the whole sentence or paragraph containing the occurrence of the syntactic pattern. This gives a clear hint to assess whether this report is really related to a missing method call. For instance, bug #186962 states that *"setFocus in ViewPart is not called systematically"*: it is validated as related to missing method call; bug #13478 mentions that *"CVS perspective should be called CVS Repository Exploring"*: it is not related to our concern.

Table 2.1 summarizes the results. For illustration consider the numbers in the first row, which tell that 49 bug reports contain the syntactic pattern "should call", and 26 of them are actually related to missing method calls. In the 211 bug reports found by our syntactic patterns, 117 are actually related to missing method calls. This number shows that missing method call survive development time, especially if we consider that the number is probably an underestimation, since we may have missed other syntactic patterns. Indeed, we will also show in the evaluation section that we are able to find other missing method calls in Eclipse.

2.3 A Statistical Oracle for Missing Method Calls

This section presents an approach to qualify missing method calls as violations of the majority rule in object-oriented software.

The historical rationale behind this approach is that, in our previous work, we came to the point that we should have a radically new viewpoint over code to lower the curse of high false positive rate (noted by Kim and Ernst in [48]). Especially, we assumed that we need a more abstract viewpoint over code compared to related work (for instance abstracting over call ordering and control flow). Our proposal of abstraction over code is called *type-usage* and is presented in Section 2.3.1 below. Section 2.3.2 introduces two relations between type-usages. Then, Section 2.3.3 leverages those relations to define a measure of strangeness for type-usages. Finally, Section 2.3.4 uses all these concepts in an algorithm that predicts missing method calls.

Figure 2.1: Extraction Process of Type-Usages in Object-Oriented Software.

```
1   class A extends Page {
2     Button b;
3
4     Button createButton() {
5       b = new Button();
6       b.setText("hello");
7       b.setColor(GREEN);
8       ...(other code)
9       Text t = new Text();
10      return b;
11    }
12  }
```

```
T(b) = 'Button'
C(b) = 'Page.createButton()'
M(b) = {<init>, setText, setColor}

T(t) = 'Text'
C(t) = 'Page.createButton()'
M(t) = {<init>}
```

2.3.1 The Concept of Type-Usage

Our approach is grounded on the concept of *type-usage*, that can be defined as follows:

Definition 1 *A type-usage is a list of method calls on a variable of a given type occurring in the body of a particular method.*

Figure 2.1 shows a code excerpt to illustrate this definition. In a method `createButton`, there is one type-usage of type `Button`, which contains three method calls `Button.<init>`, `Button.setText()`, `Button.setColor()`. There is exactly one type-usage per variable x in source code, and a type-usage can be completely expressed by the following descriptors:

- $T(x)$ is the type of the variable containing the type-usage. If there are two variables of the same type in the scope of a method, they are represented by two type-usages.
- $C(x)$ is the context of `x`, which we define as the signature of the containing method (i.e. name, and ordered parameter types in Java)
- $M(x)$ is the set of methods invoked on `x` within $C(x)$.

Figure 2.1 illustrates the conversion of Java code to type-usages. A code snippet is shown on the left-hand side of the figure; the corresponding extracted type-usages are shown on the right-hand side of the figure. There are two extracted type-usages, for Button `b` and for Text `t`. The context is the method `createButton` for both. The set of invoked methods on `b` is $M(b) = \{< init >, setText, setColor\}$, `t` is just instantiated ($M(t) = \{< init >\}$).

The main insight behind our definition of type-usage is that it is a strong abstraction over code. In particular, it abstracts over call ordering and control flow as opposed to for example Anmons et al.'s "scenario" [1] or Nguyen et al.'s "groums" [74]. For instance, let us consider that in the majority of cases, one has a call to `setText` before `setColor`. This does not mean that the opposite would be an anomaly, a deviation to correct usage. Our definition removes a sufficient amount of application-specific details and focuses on the core of using an API class: calling methods. That said, we don't claim that all bugs related to method calls can be caught by this abstraction. In particular, bugs related to object protocols are not addressed in our approach.

Figure 2.2: Examples of Exactly-Similar and Almost-Similar Relations. *b* and *aBut* are exactly-similar, *myBut* is almost-similar to *b*.

```
1  class A extends Page {
2    Button createButton() {
3      Button b = new Button();
4      ...(interlaced code)
5      b.setText("hello");
6      ...(interlaced code)
7      b.setColor(GREEN);
8      return b;
9    }
10 }
```

```
1  class B extends Page {
2    Button void createButton() {
3      ... (code before)
4      Button aBut = new Button();
5      ...
6      aBut.setText("great");
7      aBut.setColor(RED);
8      return aBut;
9    }
10 }
```

```
1  class C extends Page {
2    Button myBut;
3    Button void createButton() {
4      myBut = new Button();
5      myBut.setColor(PINK)
6      nyButton.setText("world");
7      myBut.setLink("http://bit.ly");
8      ... (code after)
9      return myBut;
10   }
11 }
```

2.3.2 Relations of Similarities between Type-Usages

Let us now informally define two binary relations between type-usages: *exact-similarity* and *almost-similarity*.

A type-usage is *exactly-similar* to another type-usage if it has the same type, and is used in the method body of a similar method containing the same method calls. For instance, in Figure 2.2 the type-usage in class B (top-right snippet) is exactly-similar to the type-usage of class A (top-left snippet): (a) they both occur in the body of the method `Button createButton()`, i.e. they are used in the same context (the notion of "context" is defined in 2.3.1 as the signature of the containing method), and (b) they both have the same set of method calls. We use the term "similar" to highlight that at a certain level of detail the type-usages related by exact-similarity are different: variables names may be different, interlaced and surrounding code as well.

A type-usage is *almost-similar* to another type-usage if it has the same type, is used in a similar context and contains the same method calls plus another one. In figure 2.2 the type-usage in class C (bottom snippet) is almost-similar to the type-usage of class A (top-left snippet): they are used in the same context, but the type-usage in class C contains all methods of A plus another one: `setLink`. We need the term *almost-similar* to denote that the relationship between two type-usages is more similar than different, i.e., there is some similarity, while being not *exactly-similar*.

Those relations can be formalized as follows:

Definition 2 *The relation* exactly-similar *(noted E) is a relation between two type-usages x and y of object-oriented software if and only if:*

$$\begin{aligned} xEy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) = M(y) \end{aligned}$$

We also define for each type-usage x the set $E(x)$ of all exactly-similar type-usages: $E(x) = \{y | xEy\}$.

Definition 3 *The relation* almost-similar *(noted A) is a relation between two type-usages x and y if and only if:*

$$\begin{aligned} xAy \iff & T(x) = T(y) \\ & \wedge\, C(x) = C(y) \\ & \wedge\, M(x) \subset M(y) \\ & \wedge\, |M(y)| = |M(x)| + 1 \end{aligned}$$

For each type-usage x of the codebase, the set $A(x)$ of almost-similar type-usages is noted:

$$A(x) = \{y | xAy\}$$

It is possible to parameterize the definition of almost-similarity by allowing a bigger amount of difference, i.e. $|M(y)| = |M(x)| + k, k \geq 1$. However, our intuition is that k=1 is the best value because developers are more likely to write and commit code with a small deviation to standard usage. Indeed, committing code with large deviations from correct would produce more visible bugs much faster.

Furthermore, we can see that we obtain no *exactly-similar* or *almost-similar* type-usages if the set of methods are not used in similar contexts. Hence, for our approach to be applicable, for a given type, we need to have several usages of the type in the same context. Consequently, we define a "redundancy" relation between type-usages. Two type-usages are said redundant if they have the same type and the same context. We will see in the evaluation that this redundancy condition is largely met in real software.

Finally, note that computing $E(x)$ and $A(x)$ for a type-usage x with respect to a given codebase Y of n type-usages is done in linear time $O(n)$.

2.3.3 S-score: A Measure of Strangeness for Type-usages

Our approach is grounded on the assumption that the majority rule holds for type-usages too, i.e. that a type-usage is deviant if: 1) it has a small number of other type-usages that are *exactly-similar*. and 2) it has a large number of other type-usages that are *almost-similar*. Informally, a small number of *exactly-similar* means "only few other places do the same thing" and a large number of *almost-similar* means "the majority does something slightly different". Assuming that the majority is right, the type-usage under inspection seems deviant and may reveal an issue in software.

Now, let us express this idea as a measure of strangeness for type-usages, which we call *S-score*. This measure will allow us to order all the type-usages of a codebase so as to identify the strange type-usages that are worth being manually analyzed by a software engineer.

Definition 4 *The S-score is:*

$$\textit{S-score}(x) = 1 - \frac{|E(x)|}{|E(x)| + |A(x)|}$$

This definition correctly handles extreme cases: if there are no exactly-similar type-usages and no almost-similar type-usages for a type-usage a, i.e. $|E(a)| = 1$ ($E(x)$ always contains x by definition) and $|A(a)| = 0$, then S-score(a) is zero, which means that a unique type-usage is not a strange type-usage at all. On the other extreme, consider a type-usage b with $|E(b)| = 1$ (no other similar type-usages) and $|A(b)| = 99$ (99 almost-similar type-usages). Intuitively, a developer expects that this type-usage is very strange, may contain a bug, and should be investigated. The corresponding S-score is 0.99 and supports the intuition.

Figure 2.3: An example computation of the likelihoods of missing method calls

$$
\begin{aligned}
T(x) &= Button \\
M(x) &= \{< init >\} \\
A(x) &= \{a, b, c, d\} \\
M(a) &= \{< init >, setText\} \\
M(b) &= \{< init >, setText\} \\
M(c) &= \{< init >, setText\} \\
M(d) &= \{< init >, setText\} \\
M(e) &= \{< init >, setFont\}
\end{aligned}
\qquad
\begin{aligned}
R(x) &= setText, setFont \\
\phi(setText) &= \frac{4}{5} = 0.80 \\
\phi(setFont) &= \frac{1}{5} = 0.20
\end{aligned}
$$

2.3.4 An Algorithm for Predicting Missing Method Calls

Let us consider a strange type-usage x (i.e. with at least one almost-similar type-usage). Once A(x) is computed, we compute missing method call predictions as follows. First, we collect the set R of all calls that are present in almost-similar type-usages but missing in x. In other terms:

$$R(x) = \{m | m \notin M(x) \wedge m \in \bigcup_{z \in A(x)} M(z)\}$$

For each recommended method in $R(x)$, we compute a likelihood value $\phi(m, x)$. The likelihood is the frequency of the missing method in the set of almost-similar type-usages:

$$\phi(m, x) = \frac{|\{z | z \in A(x) \wedge m \in M(z)\}|}{|A(x)|}$$

Eventually, predicting missing method calls for a type-usage x consists of listing calls that are in R, and have a likelihood value greater than a threshold t.

$$missing(x, t) = \{m | m \in R(x) \wedge \phi(m, x) > t\}$$

For illustration, consider the example in figure 2.3. The type-usage under study is x of type `Button`, it has a unique call to the constructor. There are 5 almost-similar type-usages in the source code (a, b, c, d, e). They contain method calls to `setText` and `setFont`. `setText` is present in 4 almost-similar type-usages out of a total of 5. Hence, its likelihood is $4/5 = 80\%$. For `setFont`, the computed likelihood is 20%. Given a threshold of 75%, the system recommends to the developer `setText` as a missing method call. Finally, we emphasize that t is the only tuning parameter of the overall approach. The sensitivity of this parameter will be studied in the next section.

Note that the prediction of missing method calls is based upon the majority rule (the majority being $A(x)$), hence it is only effective to detect missing calls in the types that are extensively used in the source code. If a method call is missing in a type-usage of a rarely used type or in an uncommon context, our approach is not effective.

Table 2.2: Descriptive Statistics of the Datasets. They all have an order of magnitude of 10^4 type-usages.

Datasets	#types	#contexts	#type-usages	#redundant
eclipse-swt	389	7839	41193	28306 (68%)
eclipse-java.io	54	3300	9765	6820 (70%)
eclipse-java.util	70	14536	40251	21780 (54%)
derby	1848	10218	36732	11164 (30%)
tomcat	1355	3891	18904	8467 (45%)

2.4 Evaluation

This section gives experimental results to validate our approach to detecting missing method calls. It combines different quantitative techniques to validate the approach from different perspectives:

- We show that in real-software (presented in 2.4.1) (a) the S-score is low for most type-usages of real software, i.e. the majority of real type-usages is not strange (see Section 2.4.2), and (b) the S-score is able to catch type-usages with a missing method call, i.e. that the S-score of such type-usages is on average higher than the S-score of normal type-usages (see Section 2.4.3).
- We show that our algorithm is able to predict missing method calls that are actually missing (see Section 2.4.4).

2.4.1 Datasets

We have implemented an extractor of type-usages for Java bytecode using the Soot bytecode analysis toolkit [92]. In all, we mined 5 different large-scale datasets:

eclipse-swt All type-usages of the Eclipse development environment version 3.5 related to SWT types (SWT is the graphical user-interface library underlying Eclipse)

eclipse-java.util All type-usages of the Eclipse development environment version 3.5 related to standard Java types from package `java.util`.

eclipse-java.io All type-usages of the Eclipse development environment version 3.5 related to standard Java types from package `java.io`.

apache-tomcat All type-usages of the Apache Tomcat web application server related to domain types (i.e. `org.apache.tomcat.*`).

apache-derby All type-usages of the Apache Derby database system related to domain types (i.e. `org.apache.derby.*`).

The program pairs were selected to cover different application domains (IDE, web server, database) and libraries (GUI, IO). The dataset is publicly available at `http://www.monperrus.net/martin/dmmc`.

Table 2.2 presents descriptive statistics for the datasets. The first column gives the number of different types in the datasets and the second the number of different contexts (the number of different method signatures, see Section 2.3.1). For instance, in the Eclipse codebase, there are

Table 2.3: Distribution of the S-score on different datasets

Datasets	#type-usages	Median S-score	Mean S-score	S-score<0.1	S-score>0.5	S-score>0.9
eclipse-swt	41193	0	0.04	89%	2.7%	0.1%
eclipse-java.io	9765	0	0.03	92%	0.7%	0%
eclipse-java.util	40251	0	0.02	94%	0.8%	0.003%
derby	36732	0	0.01	98%	0.3%	0%
tomcat	18904	0	0.01	98%	0.1%	0%

389 different types from SWT that are used in 7839 different methods. The third column gives the overall number of type-usages extracted and the last one the number of redundant type-usages (at least one other type-usage with same context and same type, see Section 2.3.2) associated with the ratio of redundant type-usages. Those statistics support the following interpretations:

First, the shape of each dataset is quite different, both in terms of types and in terms of contexts. For instance, the number of types of eclipse-java.io is only 54 because this library is used for input input/output, whose logic can be encapsulated in a few types (e.g. `File` and `PrintStream`). On the contrary, the whole application logic of the Derby database system is spread over 1848 different types. Hence, the more complex the logic, the more classes are used to express it.

Second, we note that the distributions of type-usages per type and type-usages per context follow a power-law distribution, which means that a few types and a few contexts trust a large number of type-usages. For instance, for eclipse-swt, the top-20 most popular types (out of 389) cover 62% of the type-usages. This goes along the same line as the results of Baxter et al. [7].

Third, since our approach requires equality of type-usage contexts (the signature of the enclosing method) to build the set of almost-similar type-usages, it is applicable only when one can observe at least two type-usages for a given context (this is the key part of the definition of redundant type-usages). Since the number of redundant type-usages is high (up to 68% as shown by the last column of table 2.2), the overall applicability of our approach is validated.

2.4.2 The Correctness of the Distribution of the S-score

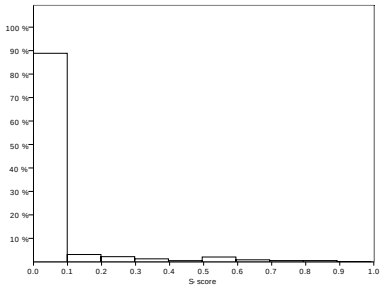
For each dataset and for each type-usage, we have computed the sets of exactly and almost-similar type-usages ($E(x)$ and $A(x)$) and the corresponding S-score. Since all datasets are extracted from widely used and mature software, we assume that most of the type-usages have a low degree of strangeness, i.e. a low S-score.

Table 2.3 validates the assumption: the median S-score is 0 for all datasets (i.e. more than 50% of all type-usages have a S-Score of 0). Furthermore, there is always a very small proportion of strange and very strange type-usages (S-score>0.5 and S-score>0.9). Figure 2.4 shows the complete distribution of the S-score for dataset *eclipse-swt*. This figure graphically shows that a large majority of type-usages has a low S-score and that the distribution is exponential. This kind of distribution shape holds for all datasets.

2.4.3 The Ability of S-score to Catch Degraded Code

Now, we show that a faulty type-usage would be caught by the S-score (faulty in the sense of suffering from a missing method call). For this to be true, a type-usage with a missing method

Figure 2.4: Distribution of the S-score based on the type-usages of type SWT.* in the Eclipse codebase. Most type-usages have a low S-score, i.e. are not strange.




```
1  $L = {} // the set of degraded type−usages
2  foreach type−usage $t
3    check that $t has redundant type−usages
4    foreach method−call $m of $t
5      $o = clone($t)
6      remove $m from $o
7      add $o to $L
8    end
9  end
10 return $L
```

Listing 2.1: An Algorithm to Simulate Missing Method Calls

call should have a higher S-score than a normal type-usage.

To assess the validity of this assumption, our key insight is to simulate missing method calls. Given a type-usage from real software with n method calls, our idea is to create n degraded type-usages by removing one by one each method call. This strategy to create validation input data has several advantages: (a) there is no need to manually assess whether a type-usage is faulty, we know it by construction, (b) it is based on real data (the type-usages come from real software) and (c) it yields a large-scale evaluation with a large number of evaluation cases.

Listing 2.1 presents this algorithm as pseudo-code. Two for-loops traverse the whole dataset for each type-usage and for each method calls. Line 3 filters the type-usages that have no redundant type-usages. The rationale of this filtering is that if a type-usage is already the only one of a given type in a given context, i.e. is already an outlier in the dataset, it does not make sense to degrade it further (because by construction, all resulting degraded type-usages would have no almost-similar type-usages).

Once we have a set of artificial degraded type-usages, we can analyze the distribution of their S-score and compare it with the distribution of real type-usages computed in 2.4.2. Although the S-score has been designed to reveal missing method calls, it may be inefficient due to the peculiarities of real data. The comparison enables us to assess that the S-score is actually well-designed.

We have conducted this evaluation for all datasets. Table 2.4 gives the results. For all datasets, the median S-score of degraded type-usages is 1 (recall that a S-score of 1 means that there is no exactly-similar type-usages) and the mean S-score always greater than 0.60.

Table 2.4: Distribution of the S-score of Degraded Type-usages resulting from Algorithm 2.1. The S-score is able to capture faulty type-usages.

Datasets	#simulated missing calls	Median S-score	Mean S-score	S-score<0.1	S-score>0.5	S-score>0.9
eclipse-swt	42845	1	0.78	18%	79%	73%
eclipse-java.io	9698	1	0.69	30%	68%	67%
eclipse-java.util	34049	1	0.76	22%	76%	74%
derby	16254	1	0.61	38%	61 %	60%
tomcat	10589	1	0.78	22%	78 %	78%

A significant proportion of degraded type-usages has a S-score greater than 0.9 (from 60% for *derby* to 78% for *tomcat*). Also, there is always some degraded data that cannot be recognized as problematic (S-score lesser than 0.1), up to 38% for *derby*.

Finally, let us discuss the number of simulated missing method calls (the first column in table 2.4). Since there is on average two method calls per type-usage, the number of simulated missing calls should be of the same order of magnitude as two-times the number of redundant type-usages. The comparison of table 2.2 and table 2.4 validates this assumption. For instance, there are 22673 redundant type-usages in *eclipse-swt* and 42845 degraded type-usages resulting from algorithm 2.1.

2.4.4 The Performance of Missing Method Calls Prediction

The third evaluation of our system measures its ability to guess missing method calls. The assumption underlying this evaluation is that our approach to detect missing method calls (presented in 2.3.4) should be able to predict calls that were artificially removed.

For this, we have used the same setup used for evaluating the characteristics of the S-score (see Section 2.4.3) but instead of looking at the distribution of the S-score of degraded data, we have tried to guess the method call that was artificially removed.

For instance, given a real type-usage of the codebase representing a `Button` and containing `<init>` and `setText`, we test the system with two different queries: 1) `<init>` only and 2) `setText` only. The system may predict several missing method calls, but a perfect prediction would be `setText` as missing method call for the first query and `<init>` for the second query.

Hence, for each dataset, the system is evaluated with the same number of queries as in 2.4.3. Then, we collect the following indicators for each query: $correct_i$ is true if the removed method call is contained in the set of predicted missing method calls; $answered_i$ is true if the system predicts at least one missing method call (we recall that the system predicts a missing method call if there is at least one almost-similar type-usage); $sizeanswer_i$ is the number of predicted missing method calls; $prefect_i$ is true if the system predicts only one missing method call and it's the one that was removed, i.e. $perfect_i \implies correct_i \wedge (answered_i = 1)$.

For N queries, we measure the overall performance of the system using the following metrics:

- **ANSWERED** is the percentage of answered queries. A query is considered as answered if the system outputs at least one missing method call.

$$ANSWERED = \frac{|\{i|answered_i\}|}{N}$$

- **CORRECT** is the percentage of correctly answered queries.

Table 2.5: Performance metrics the DMMC system for different datasets for threshold $t = 0.9$.

Dataset	N_{query}	ANSWERED	CORRECT	FALSE	PRECISION	RECALL
eclipse-swt	42845	76%	89%	11%	77%	68%
eclipse-java.io	9698	66%	87%	13%	83%	58%
eclipse-java.util	34049	75%	98%	12%	81%	66%
derby	16254	54%	76%	24%	72%	41 %
tomcat	10589	67%	73%	27%	59%	49%

- **FALSE** is the percentage of incorrectly answered queries, i.e. queries without the removed method call in the predicted missing ones.
- **PRECISION** is the classical information retrieval precision measure. Since the precision is not computable for empty recommendations (i.e. unanswered queries),

$$PRECISION = \frac{\sum_{i|correct_i} 1/sizeanswer_i}{|\{i|answered_i\}|}$$

- **RECALL** is the classical information retrieval recall measure:

$$RECALL = \frac{|\{i|correct_i\}|}{N}$$

For each dataset, we evaluated the DMMC system based on the evaluation process and performance metrics presented above. Table 2.5 presents the results. The system is able to answer from 54% to 76 % of the queries depending on the dataset, and when it answers, it's usually correct (CORRECT varies from 73% to 89%). Furthermore, the correct missing method call is not lost among a large number of wrong predictions, since the PRECISION value remains high, from 59% to 83%.

The precision and recall values presented here are lower than those presented in the conference paper [67] because we have changed the evaluation strategy. In the previous version of the evaluation algorithm which simulates missing method calls, we discarded type-usages that have only one call to be the seed of degraded type-usages (meaning we discarded all artificial queries with no calls). We discovered that those queries are actually the most difficult to answer, yielding to lower precision and recall. Since it's often the case that real type-usages contain no call (objects simply passed as parameter to other methods), our new evaluation strategy is more realistic, but also decreases the measured performance.

2.4.5 Case Studies

The evaluation results presented above suggest that a software engineer should seriously consider analyzing a type-usage if it has a high S-score. However, it may be the case that our process of artificially creating missing method calls does not reflect real missing method calls that occur in real software.

As a counter-measure to this threat of validity, we used the DMMC system in a real-world setting. We searched for missing method calls in the Eclipse, Apache Tomcat and Apache Derby software packages. We analyzed approximately 30 very strange type-usages - 30 corresponding to approximately 3 full days of analysis, because we were totally unfamiliar with the Eclipse

```
1  --- src/org/eclipse/equinox/.../NewNodeDialog.java 18 Apr 2008
2  +++ src/org/eclipse/equinox/.../NewNodeDialog.java 29 Sep 2010
3  @@ -47,8 +47,7 @@
4     }
5
6     protected Control createDialogArea(Composite parent) {
7  -  Composite compositeTop = (Composite) super.createDialogArea(parent);
8  -  Composite composite = new Composite(compositeTop, SWT.NONE);
9  +  Composite composite = (Composite) super.createDialogArea(parent);
10
11       setMessage(SecUIMessages.newNodeMsg);
```

Listing 2.2: Patch submitted to the Eclipse Bug Repository to Solve a Strange Type-Usage (issue number #326504)

Table 2.6: Issues reported based on the predictions of our system.

Issue identifier	Outcome
Eclipse issue #296552	validated, patch accepted
Eclipse issue #297840	no answer
Eclipse issue #296554	code no longer maintained
Eclipse issue #296586	wrong analysis
Eclipse issue #296581	validated, patch accepted
Eclipse issue #296781	validated, patch accepted
Eclipse issue #296782	validated, patch accepted
Eclipse issue #296578	no answer
Eclipse issue #296784	validated, patch accepted
Eclipse issue #296481	validated, patch accepted
Eclipse issue #296483	validated, patch accepted
Eclipse issue #296568	validated
Eclipse issue #275891	validated
Eclipse issue #296560	code no longer maintained
Eclipse issue #326504	validated, patch accepted
Tomcat issue #50023	wrong analysis
Derby issue #DERBY-4822	validated, patch accepted

code base. For each of them, we analyzed the source code in order to understand what the high S-score means. Then, we tried to formulate the problem as an issue report and to write a patch that solves the issue.

For example, let us consider issue #326504[3] in Eclipse. The system reports a strange type-usage (S-score: 0.93, #almost-similar: 109) in NewNodeDialog.createDialogArea related to a composite. By reading the source code, it turns out that this method does not comply with an API best practice. Then we reported the issue in natural language (*"There is a muddle of new, super and returned Composite in NewNodeDialog.createDialogArea"*) and we attached a patch presented in listing 2.2.

2.4.5.1 Real issues in mature software

Table 2.6 presents our issue reports that are based on the predictions of our system. In all, we reported 17 issues and got positive feedback for 11 of our reports, including 9 accepted

[3] https://bugs.eclipse.org/bugs/show_bug.cgi?id=326504

```
1 Control composite= super.createDialogArea(parent);
2 // this check is not relevant anymore
3 if (!(composite instanceof Composite)) {
4         composite.dispose();
5         composite= new Composite(parent, SWT.NONE);
6 }
```

Listing 2.3: Software Aging: Unnecessary Code Related to a Past Version of the API

patches that are now in the HEAD version of the corresponding revision control system. This shows that the S-score is able to reveal issues in mature software. To really appreciate these numbers, consider this question: what is the probability of submitting a valid patch on a very large unknown codebase (between 10^5 and 10^6 lines of code)? We believe that the answer is "very low". However, with tool support, we were able to do so. Our system pointed us directly to very strange pieces of code, and gave us sufficient information (what method calls the majority does) to understand the issue and submit a patch.

2.4.5.2 Missing method calls and software aging

Missing method calls may reveal problems related to software aging. Let us explain this finding with two concrete cases.

According to the system, Eclipse's `ExpressionInputDialog` contains strange code related to closing and disposing the widgets of the dialog. We reported this issue as issue #296552 [4] and our patch was quickly accepted. Interestingly, by mining the history of this class, we found a set of commits and a discussion around another issue report [5]. Even if this other issue report was closed as *solved*, the code was never cleaned and the measures and counter-measures taken during the discussion degraded the code quality and increased its strangeness.

The other example is in `ChangeEncodingAction` which contains defensive code related to a very old version of the API (see listing 2.3). This check is completely unnecessary with the current version. Our approach finds that having calls to `super.createDialogArea`, `dispose` and `new Composite` is really strange for a type-usage of type `Composite`. Following our remark about this class to the Eclipse developers, the code has been actualized. In this case, software aging comes from changes of the API that were not reflected in client code.

2.4.5.3 False-positive

The expression *tyranny of the majority* is sometimes used to refer to political and decision systems in which minorities are not allowed to express their differences. The concept applies very well to our system: if a type-usage has a small number of exactly-similar other type-usages and a high number of almost-similar, the type-usage is part of a minority (the exactly-similar type-usages) with respect to a large majority (the almost-similar type-usages). By construction, the formula of the S-score always yields high values for minorities, even if they are actually correct.

For instance, there is a couple of cases in Eclipse where an empty `Label` and an empty `Composite` are used to create filler and spacers. Usually and conceptually, labels are used to contain a text (and host a call to `setText`) and composite are used to contain widgets organized with a layout strategy (and host a call to `setLayout`). As a result, an empty `Label` and an empty `Composite` always trigger high S-scores to the corresponding type-usages. To a certain extent, using an empty label or composite is more a hack than a good practice. A better solution

[4] see https://bugs.eclipse.org/bugs/show_bug.cgi?id=296552
[5] see https://bugs.eclipse.org/bugs/show_bug.cgi?id=80068

would be, for instance, to configure the margins of the layout, or to introduce a `Filler` class in the SWT library.

In those cases, there is a conjunction of three factors:

- the lack in the considered API of a class dedicated to a particular concern (e.g. fillers in SWT).
- the developer choice to use a lightweight hack rather than a heavyweight strategy (e.g. empty labels versus complex layout objects in SWT).
- the nature of the S-score that is sensible to the tyranny of the majority by construction.

To sum up, the high S-Scores that are due the tyranny of the majority still give us insights on the analyzed software.

2.5 Conclusion

In this chapter, we have presented a new statistical oracle for detecting bugs. The class of bugs addressed by this oracle is missing method calls. The evaluation of the system showed that: 1) the system gives a majority of correct results; 2) the high confidence warnings produced by the system are related to real missing method calls in mature software; 3) missing method calls often reveal issues that are larger in scope including software aging, cloning, and violations of API best practices.

The concept of *almost-similarity* is not specific to method calls and could be applied to other software artifacts. For instance, searching for *almost-similar* traces could yield major improvements in the area of runtime defect detections. Also, searching for *almost-similar* conditional statements could improve the resilience of software with respect to incorrect input.

We have further experimented with other kinds of statistical correctness oracle. We realized that it is really difficult to design a good one. The reason is that the core assumption "what is not standard is a bug" rarely holds in software. In many cases, there is a long tail of diverse usages and behaviors [62], where each of them are indeed rare, but correct.

With retrospect, this statistical oracle on missing method calls works well (and indeed it does), especially because it has a strong definition of the context (overriding a specific method). To generalize, we would say that if the context imposes strong restrictions on usage, a statistical oracle is meaningful, otherwise, it is doomed by the curse of diversity.

Chapter 3

Micro-Oracles for Automatic Repair

This chapter is based on content from:

- Nopol: Automatic Repair of Conditional Statement Bugs in Large-Scale Object-Oriented Programs (Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clément, Sebastian Marcote, Daniel Le Berre, Martin Monperrus), IEEE Transactions on Software Engineering, 2016.
- Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT (Favio DeMarco, Jifeng Xuan, Daniel Le Berre, Martin Monperrus), In Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014), 2014.

Contents

Automatic repair is always realized with respect to a concrete oracle (see Chapter 1.2). For instance, in test-suite based repair, a system takes a buggy program as well as a test suite as input and generates a patch as output. This test suite must contain at least one violated oracle (a test case assertion) that highlights the bug to be repaired.

However, the violated oracle tells nothing about which executed statement should be repaired (fault localization) and how to repair it (patch synthesis). There are thus two combined search spaces, the fault localization and the patch synthesis ones. While the former is bounded by the number of executed statements (typically in the hundreds), the latter is much bigger. However, it may be possible to analyze the oracle violation so as to identify points in those search spaces that are likely to be valid patches.

Our key insight is to automatically transform the global, single oracle into a set of multiple micro-oracles at the level of program expressions. The global oracle is in the failing test case assertion, the micro-oracles are at the level of conditional expressions. For instance, if 10 test cases execute an if-statement x at a particular location, a tuple $(b_1, \ldots, b_{10})$ of boolean values is the micro-oracle of the condition of x. By doing so, we can transform the violation of the global oracle into a violation of the micro oracle. This "micro violation" has two major advantages. First, it localizes the bug much more precisely than the global oracle and hence reduces the search space. Second, it enables us to transform the repair problem into a synthesis problem: to repair the bug, one has to synthesize an if condition that satisfies the micro-oracle. We have realized this idea into an automatic repair system for Java called Nopol[1].

> Automatic patch generation is a recent research field where the biggest challenge is the enormous size of the search space. Our key insight of efficiently transforming the global test case oracle into a localized micro oracle at the level of conditional expressions addresses the search space problem, and enables us to be the first to repair real bugs in large scale Java software.

Nopol repairs two classes of faults: buggy if-conditions and missing preconditions. Nopol analyzes all statements executed in the failing test case. For each statement, the process of generating the patch consists of three major phases. First, Nopol detects whether there exists a fix location for a potential patch in this statement with a new and scalable technique called "angelic fix localization" (Section 3.2.2), this is where the micro-oracle is collected.

Second, Nopol collects runtime traces from test suite execution through code instrumentation (Section 3.2.4). These traces basically contain local program states at all candidate fix locations. The collected trace consists of both primitive data types (e.g., integers and Booleans) and object-oriented data (e.g., nullness or object states obtained from method calls).

Third, given the runtime traces, the problem of synthesizing a new conditional expression that matches the angelic value is translated into a Satisfiability Modulo Theory (SMT) problem. Our encoding extends the technique by Jha et al. [45] by handling rich object-oriented conditionals. We use our own implementation of the encoding together with an off-the-shelf SMT solver (Z3 [71]) to check whether there exists a solution.

[1] Nopol is an abbreviation for "no polillas" in Spanish, which literally means "no moth anymore".

```
1 - if (u * v == 0) {
2 + if (u == 0 || v == 0) {
3       return (Math.abs(u) + Math.abs(
            v));
4   }
```

Figure 3.1: Patch example of Bug CM5: a bug related to a buggy IF condition. The original condition with a comparison operator == is replaced by disjunction between two comparisons.

```
1 + if (sb != null) {
2       sb.append(": "); //sb is a string
            builder in Java
3 + }
```

Figure 3.2: Patch example of Bug PM2: a precondition is added to avoid a reference of `null`. The patch is specific to the object-orientation of Java.

If such a solution exists, NOPOL translates it back to source code, i.e., generates a patch. We re-run the whole test suite to validate whether this patch is able to make all test cases pass and indeed repairs the bug under consideration.

To evaluate and analyze our repair approach NOPOL, we collect a dataset of 22 bugs (16 bugs with buggy IF conditions and 6 bugs with missing preconditions) from real-world projects. Our result shows that 17 out of 22 bugs can be fixed by NOPOL. We manually check the repair result and show the details of these bugs as well as their patches. Four case studies are conducted to present the benefits of generating patches via NOPOL and five bugs are employed to explain the limitations.

The main contributions of this chapter are as follows.

- The design of a repair approach for fixing conditional statement bugs of the form of buggy IF conditions and missing preconditions.
- Two algorithms of *angelic fix localization* for identifying potential fix locations and repair oracles at once.
- An extension of the SMT encoding in [45] for handling nullness and certain method calls of object-oriented programs.
- An evaluation on a dataset of 22 bugs in real-world programs with an average of 25K executable lines of code for each bug.
- A publicly-available automatic software repair for supporting further replication and research [76].

The remainder of this chapter is organized as follows. Section 3.1 provides the background of test-suite based repair. Section 3.2 presents our approach for repairing bugs with buggy IF conditions and missing preconditions. Section 3.3 details the evaluation on 22 real-world bugs. Section 3.4 presents potential issues. Section 3.5 concludes this chapter.

3.1 Background

We present the background on test-suite based repair, its application to real-world bugs, and two kinds of bugs targeted in this contribution.

3.1.1 Test-Suite Based Repair

Test-suite based repair consists in repairing programs according to a test suite, which contains both passing test cases as a specification of the expected behavior of the program and at least one failing test case as a specification of the bug to be repaired. Failing test cases can either identify a regression bug or reveal a new bug that has just been discovered. Then, repair algorithms search for patches that make all the test cases pass. If such a patch is found, the patch fixes the bug and at the same time, does not degrade the existing functionalities.

The *core assumption* of test-suite based repair is that the test suite is good enough to thoroughly model the program to repair [64]. This is a case when the development process ensures a very strong programming discipline. For example, most commits of Apache projects (e.g., Apache Commons Lang) contain a test case specifying the change. If a commit is a bug fix, the commit contains a test case that highlights the bug and was failing for the bug before the fix.

Test-suite based repair has been evaluated in different experiments. In 2012, Le Goues et al. [51] conducted a systematic study of fixing 55 out of 105 bugs via the pioneering repair method, GenProg [36]. In 2013, Weimer et al. [94] reported that an adaptive repair method based on program equivalence, called AE, can fix 54 out of the same 105 bugs while evaluating fewer test cases. Then in 2014, Qi et al. [81] compared the search strategy in GenProg with random search and proposed RSRepair, which can efficiently fix 24 bugs (these 24 bugs are derived from a subset of 55 fixed bugs by GenProg). However, recent work by Qi et al. [82] has manually examined all the generated patches by the above three methods, GenProg [36], AE [94], and RSRepair [81]. Their results show that for all the considered 105 bugs, only 2 bugs by GenProg, 3 by AE, and 2 by RSRepair are correctly fixed via patch generation. All the reported patches for the other bugs are incorrect due to either an experimental error or because of the weaknesses of the test suite under consideration [82].

3.1.2 Fault Class: Buggy IF Condition Bugs

Conditional statements (e.g., `if (condition){}` in Java), are widely-used in programming languages. Pan et al. [77] show that among seven studied Java projects, up to 18.6% of bug fixes have changed a buggy condition in IF statements. A buggy IF condition is defined as a bug in the condition of an `if-then-else` statement. Given a program with a bug, a buggy IF condition may lead to a different branch. Our work on NOPOL addresses the problem of automatically fixing conditional bugs.

The bug in Fig. 3.1 is a real example of a buggy IF condition in Apache Commons Math. This bug is a code snippet of a method, which calculates the greatest common divisor between two integers. The condition in that method is to check whether either of two parameters `u` and `v` is equal to 0. In the buggy version, the developer compares the product of the two integers to zero. However, this may lead to an arithmetic overflow. A safe way to proceed is to compare each parameter to zero. This bug was fixed by NOPOL (see Bug CM5 in Table 3.2).

3.1.3 Fault Class: Missing Precondition Bugs

Another class of common bugs related to IF statements is missing preconditions. A precondition aims to check the state of certain variables before the execution of a statement. Examples of

common preconditions include detecting a null pointer or an invalid index in an array. In software repositories, we can find commits that add preconditions (i.e., which were previously missing).

The bug in Fig. 3.2 is a missing precondition with the absence of null pointer detection. The buggy version without the precondition will throw an exception signaling a null pointer at runtime. NOPOL fixed this bug by adding the precondition (see Bug PM2 in Table 3.2).

3.2 Nopol: Automatic Repair for Java

This section presents our approach to automatically repairing buggy IF conditions and missing preconditions. Our approach is implemented in a tool called NOPOL that repairs Java code.

3.2.1 Overview

NOPOL is a test-suite based repair approach dedicated to bugs with buggy IF conditions and missing preconditions. As input, NOPOL requires a test suite which represents the expected program functionality with at least one failing test case that exposes the bug to be fixed. Given a buggy program and its test suite, NOPOL returns the final patch as output. Fig. 3.1 and Fig. 3.2 are two examples of output patches for buggy IF conditions and missing preconditions by NOPOL, respectively.

How to use Nopol. From a user perspective, given a buggy program with a test suite, including failing test cases (written by a developer or maintained in a continuous integration server), the user would run NOPOL and obtain a patch, if any. Before applying NOPOL to the buggy program, the user does not need to know whether the bug relates to conditions. Instead, the user runs NOPOL for any buggy program. The output of NOPOL will be an automatically-generated patch or no patch after the exhaustively exploring the search space. If a patch is available, then the user would manually inspect and validate the patch for integration in the code base. As further dicussion in Section 3.3.4, we can also add a pre-defined timeout, e.g., 90 seconds as suggested in experiments or a longer timeout like five hours. In that case, the output will be a patch or no patch due to the timeout.

Fig. 3.3 shows the overview of NOPOL. NOPOL employs a fault localization technique to rank statements according to their suspiciousness of containing bugs. For each statement in the ranking, NOPOL considers it as a buggy IF condition candidate if the statement is an IF statement; or NOPOL considers it as a missing precondition candidate if the statement is any other non-loop or non-branch statement (e.g., an assignment or a method call). NOPOL processes candidate statements one by one with three major phases.

First, in the phase of **angelic fix localization**, NOPOL arbitrarily tunes a conditional value (`true` or `false`) of an IF statement to pass a failing test case. If such a conditional value exists, the statement is identified as a fix location and the arbitrary value is viewed as the expected behavior of the patch. In NOPOL, there are two kinds of fix locations, IF statements for repairing buggy conditions and arbitrary statements for repairing missing preconditions.

Second, in the phase of **runtime trace collection**, NOPOL runs the whole test suite in order to collect the execution context of each fix location. The context includes both variables of primitive types (e.g., Booleans or integers) and a subset of object-oriented features (nullness and certain method calls); then such runtime collection will be used in synthesizing the patch in the next phase.

Third, in the phase of **patch synthesis**, the collected trace is converted into a Satisfiability Modulo Theory (SMT) formula. The satisfiability of SMT implies that there exists a program expression that preserves the program behavior and fixes the considered bug. That is, the expression makes all test cases pass. If the SMT formula is satisfiable, the solution to the SMT

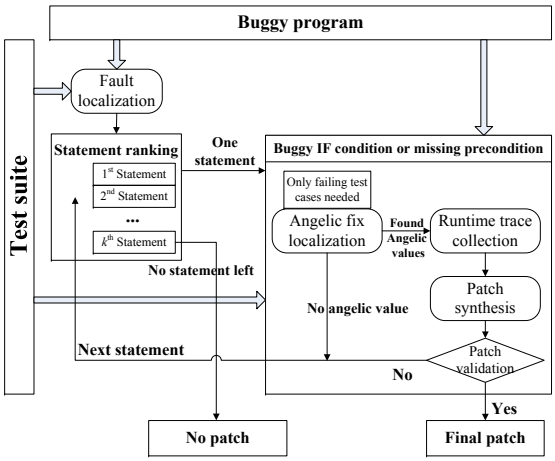


Figure 3.3: Overview of our approach to automatically repairing buggy IF conditions and missing preconditions.

is translated as a source code patch; if unsatisfiable, NOPOL will go to next statement in the statement ranking, until all statements are processed.

After the above three phases, the whole test suite is re-executed to validate that the patch is correct. This validation could be skipped if the SMT encoding is proven to be correct. Indeed, theoretically, if the SMT solver says "satisfiable", it means that a patch exists. However, there could be an implementation bug in the trace collection, in the SMT problem generation, in the off-the-shelf SMT solver, or in the patch synthesis. Consequently, we do make the final validation by re-executing the test suite.

3.2.2 Micro-oracles for conditional statements

In NOPOL, we propose to use value replacement [44] to detect potential fix locations. Value replacement [44] comes from fault localization research. It consists in replacing at runtime one value by another one. More generally, the idea is to explore speculative program states for localizing faults. There are a couple of papers that explore this idea. For instance, Zhang et al. [103] use the term "predicate switching" and Chandra et al. [12] use the term "angelic debugging". The former is a dynamic approach, the latter a symbolic one, but both are based on speculative analysis.

NOPOL replaces conditional values in IF statements. We refer to conditional values that make test cases pass as *angelic values*.

Definition (Angelic Value) An angelic value is an arbitrarily-set value during test execution by an omniscient angel, which enables a failing test case to pass.

To facilitate the description of our approach, we follow existing work [38] to introduce the concept of locations. A *location* is an integer value, which identifies the position of organizing source code.

Definition (Angelic Tuple) An angelic tuple is a triplet $(loc, val, test)$, where the statement at a location loc is evaluated to a value val to make a failing test case $test$ pass.

Here, we refer to the technique of modifying the program state to find the values for angelic tuples $(loc, val, test)$ as *angelic fix localization*. If an angelic tuple $(loc, val, test)$ is found, there

Input :
stmt, a candidate IF statement;
T_f, a set of failing test cases.
Output:
R, a set of angelic tuples.

$R \leftarrow \emptyset$;
Initialize two sets $T_{true} \leftarrow \emptyset$ and $T_{false} \leftarrow \emptyset$;
Let *cond* be the condition in *stmt* and let *loc* be the location of *cond*;

Force *cond* to `true` and execute test cases in T_f;
foreach *failing test case* $t_i \in T_f$ **do**
  if t_i *passes* **then**
    $T_{true} \leftarrow T_{true} \cup \{t_i\}$;
  end
end
Force *cond* to `false` and execute test cases in T_f;
foreach *failing test case* $t_i \in T_f$ **do**
  if t_i *passes* **then**
    $T_{false} \leftarrow T_{false} \cup \{t_i\}$;
  end
end

`// All test cases in` T_f `are passed`
if $(T_f \setminus T_{true}) \cap (T_f \setminus T_{false}) = \emptyset$ **then**
  foreach $t_i \in T_{true}$ **do**
    $R \leftarrow R \cup \{(loc, \texttt{true}, t_i)\}$;
  end
  foreach $t_i \in T_{false}$ **do**
    $R \leftarrow R \cup \{(loc, \texttt{false}, t_i)\}$;
  end
end

Algorithm 1: Angelic Fix Localization Algorithm for Buggy IF Conditions

may exist a patch in the location *loc* in source code. *An angelic value can be seen as a micro-oracle at the level of single statements*

A single test case *test* may evaluate the statement at the location *loc* several times. Consequently, according to our definition, the value *val* is fixed across all evaluations of a given statement for one test case. This point is key for having a tractable search space (will be discussed in Section 3.2.3). On the other hand, one angelic value is specific to a test case: for a given location *loc*, different failing test cases may have different angelic values.

3.2.2.1 For Buggy IF Conditions

For buggy IF conditions, angelic fix localization works as follows. For each IF condition that is evaluated during test suite execution, an angel forces the IF condition to be `true` or `false` in a failing test case, which embodies the bug to be fixed. An angelic tuple $(loc, val, test)$, i.e., (IF condition location, Boolean value, failing test case), indicates that a fix modifying this IF condition may exist (if the subsequent phase of patch synthesis succeeds).

Algorithm 1 is the pseudo-code of angelic fix localization for buggy IF conditions. For a given

Input :
stmt, a candidate non-IF statement;
T_f, a set of failing test cases.
Output:
R, a set of angelic tuples.

$R \leftarrow \emptyset$;
Initialize a test case set $T_{pre} \leftarrow \emptyset$;
Let *loc* be the location of a potential precondition of *stmt*;

Force *stmt* to be skipped and execute T_f;
foreach *failing test case* $t_i \in T_f$ **do**
  if t_i *passes* **then**
    $T_{pre} \leftarrow T_{pre} \cup \{t_i\}$;
  end
end

`// All test cases in` T_f `are passed`
if $T_{pre} = T_f$ **then**
  foreach $t_i \in T_{pre}$ **do**
    $R \leftarrow R \cup \{(loc, \texttt{false}, t_i)\}$;
  end
end

Algorithm 2: Angelic Fix Localization Algorithm for Missing Preconditions

IF statement *stmt* and its condition *cond*, both `true` and `false` are set to pass originally failing test cases at runtime. If all failing test cases are passed, angelic tuples are collected for further patch synthesis; otherwise, there exists no angelic value for the test case and the location under consideration. To sum up, Algorithm 1 extracts micro-oracles for if conditions.

3.2.2.2 For Missing Preconditions

Angelic fix localization for missing preconditions is slightly different from that for IF conditions. For each non-branch and non-loop statement that is evaluated during test suite execution, an angel forces to skip it. If a failing test case now passes, it means that a potential fix location has been found. The oracle for repair is then "false"; that is, the added precondition must be `false`, i.e., the statement should be skipped. Then, an angelic tuple $(loc, val, test)$ is (precondition location, `false`, failing test case).

Algorithm 2 is the pseudo-code of this algorithm. Given a non-IF statement *stmt*, we skip this statement to check whether all failing test cases are passed. If yes, the location *loc* of the precondition as well as its angelic value `false` is collected. If skipping the statement does not pass all the failing test cases, no angelic values will be returned. This technique also works for missing preconditions for entire blocks since blocks are just specific statements in Java. In our implementation, we only consider adding missing preconditions for single statements rather than blocks. Manual examination on the dataset in Section 3.3.4 will show that our dataset does not contain missing preconditions for blocks. To sum up, Algorithm 1 extracts micro-oracles for statement preconditions.

3.2.3 Characterization of the Search Space

We now characterize the search space of angelic values. If an IF condition is executed more than once in a failing test case, there may exist a sequence of multiple different angelic values resulting in a passing test case. For example, a buggy IF condition that is executed three times by one failing test case may require a sequence of three different angelic values to pass the test case.

§1 Search space for buggy IF conditions. In general, if one failing test case executes a buggy IF condition for n_e times, the search space of all sequences of angelic values is 2^{n_e}. To avoid the problem of combinatorial explosion, NOPOL assumes that, for a given failing test case, the angelic value is the same during the multiple executions on one statement. The search space size becomes 2 for one failing test case instead of 2^{n_e}. Under this assumption, the search space is shown as follows.

For buggy IF condition, the search space is $2 \times n_c$ where n_c is the number of executed IF statements for a given failing test case.

§2 Search space for missing preconditions. Similarly to angelic fix localization for buggy IF conditions, if a statement is executed several times by the same failing test case, angelic fix localization directly adds a precondition (with a `false` value) and completely skips the statement for a given test case.

For missing precondition bugs, the search space size is n_s where n_s is the number of executed statements by test cases. (It is not $2 \times n_s$ because we only add a precondition and check whether the `false` value will pass the failing test case).

NOPOL does not decide a priority between updating existing conditions or adding new preconditions. A user can try either strategy, or both. There is no analytical reason to prefer one or the other; our evaluation does not give a definitive answer to this question. In our experiment, we perform both strategies for statements one by one (see Section 3.2.1).

§3 On the Absence of Angelic Values. If no angelic tuple is found for a given location, there are two potential reasons. First, it is impossible to fix the bug by changing the particular condition (resp. adding a precondition before the statement). Second, only a sequence of different angelic values, rather than a single angelic value, would enable the failing test case to pass.

Hence, NOPOL is incomplete: there might be a way to fix an IF condition by alternating the way of finding angelic values, but this is left to future work.

3.2.4 Runtime Trace Collection for Repair

Once an angelic tuple is found, NOPOL collects the values that are accessible at this location in the program execution. Those value are used to synthesize a correct patch. In our work, we collect different kinds of data to generate the patch, presented in the following.

3.2.4.1 Expected Outcome Data Collection

As mentioned in Section 3.2.2, an angelic value indicates that this value enables a failing test case to pass. To generate a patch, NOPOL collects the expected outcomes of conditional values to pass the whole test suite: angelic values for failing test cases as well as actual execution values for the passing ones.

Let O be a set of expected outcomes in order to pass all test cases. An expected outcome $O_{loc,m,n} \in O$ is an actual value (e.g. "1" for an integer variable), it refers to the value at location

loc during the m-th execution in order to pass the n-th test case. NOPOL collects $O_{loc,m,n}$ for all executions of location *loc*.

For buggy IF conditions. $O_{loc,m,n}$ is the expected outcome of the condition expression at *loc*. For a failing test case, the expected outcome is the angelic value; for a passing test case, the expected outcome is the runtime value $eval(loc)$, i.e., the result of the evaluation during the actual execution of the IF condition expression.

$$O_{loc,m,n} = \begin{cases} eval(loc), & \text{for passing test cases} \\ \text{angelic value} & \text{for failing test cases} \end{cases}$$

For missing preconditions. $O_{loc,m,n}$ is the expected value of the precondition of the statement at *loc*, i.e., `true` for passing test cases and `false` for failing test cases. The latter comes from angelic fix localization: if the precondition returns `false` for a failing test case, the buggy statement is skipped and the test case passes.

$$O_{loc,m,n} = \begin{cases} true & \text{for passing test cases} \\ false & \text{for failing test cases} \end{cases}$$

Note that not all the bugs with missing preconditions can be fixed with the above definition. Section 3.3.6.3 will present the limitation of this definition with a real example.

3.2.4.2 Primitive Type Data Collection

At the location of an angelic tuple, NOPOL collects the values of all local variables, method parameters, and fields that are typed with a basic primitive type (Booleans, integers, floats, and doubles). Such collection aims to provide reusable code in patches, which is based on the code redundancy assumption of program repair [59].

Let $C_{loc,m,n}$ be the set of collected values at location *loc* during the m-th execution of the n-th test case. In order to synthesize conditions that use literals (e.g., `if (x > 0)`), $C_{loc,m,n}$ is enriched with constants for further patch synthesis. First, NOPOL collects static values that are present in the program. Second, we add three standard values {0, -1, 1}, which are present in many bug fixes in the wild (for instance for well-known off-by-one errors).[2] Based on these standard values, other values can be formed via wiring other building blocks. For example, a value 2 in a patch will be formed as $1 + 1$, if 2 is not collected during runtime trace collection.

3.2.4.3 Object-Oriented Data Collection

NOPOL aims to support automatic repair for object-oriented programs. In particular, we would like to support nullness checks and some particular method calls. For instance, NOPOL is able to synthesize the following patch containing a method call.

```
1  + if (obj.size() > 0) {
2        compute(obj);
3  + }
```

To this end, in addition to collecting all values of primitive types, NOPOL collects two kinds of object-oriented features. First, it collects the nullness of all variables of the current scope. Second, NOPOL collects the output of "state query methods", defined as the methods that inspect

[2] Besides collecting static fields in a class, we have also tried to collect other class fields of the class under repair, but the resulting patches are worse in readability than those without collecting class fields. Hence, no class fields other than static fields are involved in data collection.

the state of objects and are side-effect free. A state query method is an argumentless method with a primitive return type. For instance, methods `size()` and `isEmpty()` of `Collection` are state query methods. The concept of state query methods is derived from "argument-less Boolean queries" of "object states" by Pei et al. [79].

NOPOL is manually fed with a list of such methods. The list is set with domain-specific knowledge. For instance, in Java, it is easy for developers to identify such side-effect free state query methods on core library classes such as `String`, `File` and `Collection`. For each object-oriented class T, those predefined state query methods are denoted as $sqm(T)$.

NOPOL collects the nullness of all visible variables and the evaluation of state query methods for all objects in the scope (local variables, method parameters, and fields) of a location where an angelic tuple exists. Note that this incorporates inheritance; the data are collected based on the polymorphism in Java. For instance, when the value of `obj.size()` is collected, it may be for one implementation of `size()` based on array lists and for another implementation of `size()` based on linked lists. This means that a patch synthesized by NOPOL can contain polymorphic calls.

3.2.5 Main Repair Equation

The patch synthesis of buggy IF conditions and missing preconditions consists of finding an expression exp such that

$$\forall_{loc,m,n} \quad exp(C_{loc,m,n}) = O_{loc,m,n} \tag{3.1}$$

Equation 3.1 transforms the repair problem as a synthesis problem. In particular, it transforms a global specification (the test suite) into a micro-specification at the level of a conditional statement. This micro specification contains micro-oracles, which are the expected value of the evaluation of the boolean condition in a specific context. For the failing test cases, the micro-oracles are obtained with speculative execution. For the passing ones, the micro-oracles are obtained by runtime monitoring.

With the micro specification at the level of conditional expressions, one can use any synthesis technique. We have experimented with several ones. The default alternative in Nopol is based on our own implementation of oracle-guided component-based program synthesis [45]. This consists of transforming the micro-specification into a Satisfiability Modulo Theory (SMT) problem. If there is a feasible solution to the SMT problem (i.e. the problem is satisfiable) it means that a patch has been found. NOPOL translates this solution into a source code patch. Since NOPOL repairs bugs with buggy IF conditions and missing preconditions, the patch after translation is a conditional expression, which returns a boolean value.

3.3 Evaluation

We now evaluate our repair approach, NOPOL, on a dataset of 22 real-world bugs. First, we describe our evaluation methodology in Section 3.3.1; second, we introduce the setup of our dataset in Section 3.3.2 and the implementation details in Section 3.3.3; third, we present the general description of the synthesized repairs in Section 3.3.4; fourth, four bugs are employed as case studies in Section 3.3.5 and five bugs are used to illustrate the limitations in Section 3.3.6.

3.3.1 Evaluation Methodology

Our evaluation methodology is based on the following principles.

P1. We evaluate our tool, NOPOL, on real-world buggy programs (Section 3.3.4).

Table 3.1: Dataset of 22 bugs with buggy IF conditions and missing preconditions in two open-source projects.

Bug type	Project	Bug description			Executable	#Classes	#Methods	#Unit	Buggy method	
		Index	Commit ID†	Bug ID‡	LoC			tests	Description	Complexity
Buggy IF condition	Math	CM1	141003	-	4611	153	947	363	Returns a specified percentile from an array of real numbers	7
		CM2	141217	-	5539	212	1390	543	Returns an exact representation of the Binomial Coefficient	8
		CM3	141473	-	6633	191	1504	654	Returns the natural logarithm of the factorial for a given value	3
		CM4	159727	-	7442	206	1640	704	Computes a polynomial spline function	5
		CM5	735178	Math-238	25034	468	3684	1502	Gets the greatest common divisor of two numbers	12
		CM6	791766	Math-280	37918	632	5123	1982	Finds two real values for a given univariate function	11
		CM7	831510	Math-309	38841	650	5275	2105	Returns a random value from an Exponential distribution	3
		CM8	1368253	Math-836	64709	975	7374	3982	Converts a double value into a fraction	11
		CM9	1413916	Math-904	70317	1037	7978	4263	Computes a power function of two real numbers	38
		CM10	1453271	Math-939	79701	1158	9074	4827	Checks whether a matrix has sufficient data to calculate covariance	3
	Lang	CL1	137231	-	10367	156	2132	781	Replaces a string with another one inside	4
		CL2	137371	-	11035	169	2240	793	Removes a line break at the end of a string	4
		CL3	137552	-	12852	173	2579	994	Gets a sub-string from the middle of a string from a given index	5
		CL4	230921	-	15818	215	3516	1437	Finds the first matched string from the given index	10
		CL5	825420	Lang-535	17376	86	3744	1678	Extracts the package name from a string	6
		CL6	1075673	Lang-428	18884	211	3918	1934	Checks whether the char sequence only contains unicode letters	4
Missing precondition	Math	PM1	620221	Math-191	16575	396	2803	1168	Checks the status for calculating the sum of logarithms	1
		PM2	1035009	-	44347	745	5536	2236	Builds a message string from patterns and arguments	3
	Lang	PL1	504351	Lang-315	17286	233	4056	1710	Stops a process of timing	3
		PL2	643953	Lang-421	17780	240	4285	1829	Erases a string with the Java style from the character stream	19
		PL3	655246	Lang-419	18533	257	4443	1899	Abbreviates a given string	9
		PL4	1142389	Lang-710	18974	218	3887	1877	Counts and translates the code point from an XML numeric entity	19
Average					25480.6	399.1	3960.4	1784.6		8.6
Median					17585.0	225.5	3818.5	1694.0		5.5

† A commit ID is an identifier that indicates the commit of the patch, in both SVN and the FishEye system. According to this commit, we can manually check relevant patched code and test cases. For instance, the commit of Bug CM1 can be found at https://fisheye6.atlassian.com/changelog/commons?cs=141003.

‡ For some bugs, bug IDs are not obviously identified in the bug tracking system. These bugs can be found in the version control system. For example, Apache projects previously used Bugzilla as a bug tracking system before moving to Jira. The Bugzilla system is not available anymore.

P2. For bugs that NOPOL can fix, we examine the automatically generated patches, compare them with human produced patches, and study four representative bugs in details (Section 3.3.5).

P3. For bugs that NOPOL cannot correctly fix, we check the details of these bugs and highlight the reasons behind the unrepairability (Section 3.3.6). When the root cause is an incorrect test case (i.e., an incomplete specification), we modify the test case and re-run NOPOL.

P4. We deliberately do not compute a percentage of repaired bugs because this is a potentially unsound measure. According to our previous investigation [64], this measure is sound if and only if 1) the dataset is only composed of bugs of the same kind and 2) the distribution of complexity within the dataset reflects the distribution of all in-the-field bugs within this defect class. In our opinion, the second point is impossible to achieve.

We have not quantitatively compared our approach with existing repair approaches on the same dataset because 1) either existing approaches are inapplicable on this dataset (e.g., GenProg [36] and SemFix [73] are designed for C programs); 2) or these approaches are not publicly available (e.g., PAR [47] and mutation-based repair [22]).

3.3.2 Dataset of Real-World Bugs

NOPOL focuses on repairing conditional bugs, i.e., bugs in IF conditions and preconditions. Hence, we build a dataset of real-world bugs of buggy IF conditions and missing preconditions. Since our prototype implementation of NOPOL repairs Java code, these 22 bugs are selected from two open-source Java projects, Apache Commons Math[3] and Apache Commons Lang[4] (*Math* and *Lang* for short, respectively).

[3]Apache Commons Math, http://commons.apache.org/math/.
[4]Apache Commons Lang, http://commons.apache.org/lang/.

Both Math and Lang manage source code using Apache Subversion[5] (SVN for short) and manage bug reports using Jira.[6] Jira stores the links between bugs and related source code commits. In addition, these projects use the FishEye browser to inspect source code and commits.[7]

In our work, we employ the following four steps to collect bugs for the evaluation. First, we automatically extract small commits that modify or add an IF condition using Abstract Syntax Tree (AST) analysis [29]. We define a *small commit* as a commit that modifies at most 5 files, each of which introduces at most 10 AST changes (as computed by the analytical method, GumTree [29]). In Math, this step results in 161 commits that update IF conditions and 104 commits that add preconditions; in Lang, the commits are 165 and 91, respectively. The lists of commits are available at NOPOL project [23]. Second, for each extracted commit, we collect its related code revision, i.e., the source program corresponding to this commit. We manually check changes between the code revision and its previous one; we only accept changes that contain an IF condition or a missing precondition and do not affect other statements. Those commits could also contain other changes that relate to neither a bug nor a patch, such as a variable renaming or the addition of a logging statement. In this case, changes of the patch are separated from irrelevant changes. Third, we extract the test suite at the time of the patch commit, including failing test cases.[8] Fourth, we manually configure programs and test suites to examine whether bugs can be reproduced. Note that the reproducibility rate is very low due to the complexity of the projects Math and Lang.

Table 3.1 summarizes the 22 bugs in two categories, i.e., bug types of buggy IF conditions and missing preconditions. We index these bugs according to their types and projects. A *bug index* (Column 3) is named based on the following rule. Letters C and P indicate bugs with buggy IF conditions and missing preconditions, respectively; M and L are bugs from Math and Lang, respectively. For instance, CM1 refers to a bug with a buggy IF condition in the project Math. We also record the number of executable Lines of Code (LoC, i.e., the number of lines that exclude empty lines and comment lines) for each source program (Column 6). Moreover, we show the number of classes, the number of methods in the buggy program, and the number of unit test cases (Columns 7-9). For each method that contains the buggy code, we describe the functionality of this method and record its Cyclomatic Complexity (Columns 10 and 11). The *Cyclomatic Complexity* [60] is the number of linearly independent paths through the source code of a method. This complexity indicates the testability of a method and the difficulty of understanding code by developers.

As shown in Table 3.1, the dataset contains 16 bugs with buggy IF conditions and 6 bugs with missing preconditions. Among these bugs, 12 bugs are from Math and 10 bugs are from Lang. In average, a buggy program consists of 25.48K executable lines of code. The average complexity is 8.6; that is, a buggy method consists of 8.6 independent paths in average. Note that the method complexity of Bug PM1 is 1 since its buggy method contains only one `throw` statement (which misses a precondition); the method complexity of Bug CM9 is 38 and its buggy method contains 30 IF statements.

3.3.3 Implementation Details

Our approach, NOPOL, is implemented with Java 1.7 on top of Spoon 3.1.0.[9] Spoon [78] is a library for transforming and analyzing Java source code. It is used for angelic value min-

[5] Apache Subversion, http://subversion.apache.org/.
[6] Jira for Apache, http://issues.apache.org/jira/.
[7] FishEye for Apache, http://fisheye6.atlassian.com/.
[8] In considered commits, bug fixes are always committed together with originally failing test cases (which are passed after fixing the bugs). This is a rule in Apache development process [30].
[9] Spoon 3.1.0, http://spoon.gforge.inria.fr/.

Table 3.2: Buggy code, manually-written patches, and generated patches for bugs in the dataset.

Bug type	Bug index	Buggy code	Patch written by developers	Patch generated by NOPOL	Correctness	Test case modification
Buggy IF condition	CM1	`pos > n`	`pos >= n`	`length <= fpos`	Correct	A ‡
	CM2	`n <= 0`	`n < 0`	`if (n < 0) { ... }` †	Correct	T
	CM3	`n <= 0`	`n < 0`	`if (MathUtils.ZS != n) { ...`	Correct	A T
	CM4	`v < knots[0] || v >= knots[n]`	`v < knots[0] || v > knots[n]`	`v < knots[0] || knots[n] < v`	Correct	A
	CM5	`u * v == 0`	`u == 0 || v == 0`	`v == 0 || MathUtils.ZB == u`	Correct	A T
	CM6	`fa * fb >= 0.0`	`fa * fb > 0.0`	`if (-1 == b) { ...`	Incorrect	T D
	CM7	`mean < 0`	`mean <= 0`	`mean <= 0.0`	Correct	T D
	CM8	`p2 > overflow || q2 > overflow`	`FastMath.abs(p2) > overflow || FastMath.abs(q2) > overflow`	- (Timeout in SMT)	-	-
	CM9	`y >= TWO_POWER_52 || y <= -TWO_POWER_52`	`y >= TWO_POWER_53 || y <= -TWO_POWER_53`	- (Timeout in test execution)	-	-
	CM10	`nRows < 2 || nCols < 2`	`nRows < 2 || nCols < 1`	`if (1 != nCols) { ...`	Correct	-
	CL1	`text == null`	`text == null || repl == null || with == null || repl.length() == 0`	`with.length() != 0`	Incorrect	D
	CL2	`lastIdx == 0`	`lastIdx <= 0`	`lastIdx < blanks.length()`	Correct	T
	CL3	`pos > str.length()`	`len < 0 || pos > str.length()`	`len <= -1 || str.length() < pos`	Correct	T
	CL4	`startIndex >= size`	`substr == null || startIndex >= size`	`(startIndex >= size || substr == null) && size != -1`	Correct	T D
	CL5	`className == null`	`className == null || className.length() == 0`	`className.length() == 0`	Incorrect	-
	CL6	`cs == null`	`cs == null || cs.length() == 0`	- (Timeout due to a `null` pointer)	-	-
Missing precondition	PM1	`throw new IllegalStateException("")`	`if (getN() > 0) { ...`	- (No angelic value found)	-	-
	PM2	`sb.append(": ")`	`if (specific != null) { ...`	`if (specific != null) { ...`	Correct	-
	PL1	`stopTime ← System.currentTimeMillis()`	`if (this.runningState == STATE_RUNNING) { ...`	`if (StopWatch.STATE_RUNNING == runningState) {...`	Correct	-
	PL2	`out.write('\\')`	`if (escapeForwardSlash) { ...`	`if (escapeSingleQuote) { ...`	Incorrect	-
	PL3	`lower ← str.length()`	`if (lower > str.length()) { ...`	- (Timeout in SMT)	-	T
	PL4	`return 0`	`if (start == seqEnd) { ...`	`if (start == seqEnd) { ...`	Correct	T

† An added precondition is in a form of "`if( ) { ...`" to distinguish with an updated condition.
‡ For test case modification, "A" stands for added test cases, "T" for transformed test cases, and "D" for deleted test cases.

ing, instrumentation, and final patch synthesis in our work. Fault localization is implemented with GZoltar 1.1.0.[10] GZoltar [11] is a fault localization library for ranking faulty statements. The SMT solver inside NOPOL is Z3 4.3.2.[11] We generate SMT-LIB[12] files using jSMTLIB.[13] jSMTLIB [15] is a library for checking, manipulating, and translating SMT-LIB formatted problems. The test driver is JUnit 4.11. For future replication of the evaluation, the code of NOPOL is available on GitHub [76].

All experiments are run on a PC with an Intel Core i7 2.70 GHz CPU and an Ubuntu 12.04 operating system. The maximum heap size of Java virtual machine was set to 2.50 GB.

3.3.4 General Results

We present the general evaluation of NOPOL on the dataset via answering six Research Questions (RQs).

> RQ1: Can NOPOL fix real bugs in large-scale Java software?

In test-suite based repair, a bug is *fixed* if the patched program passes the whole test suite [36]. Table 3.2 presents the evaluation of patches on 22 bugs. Column 3 shows the buggy code (the condition for each bug with a buggy IF condition and the statement for each bug with a missing precondition). Column 4 shows the patches that were manually-written by developers as found in the version control system: the updated condition for each bug with a buggy IF condition and the added precondition for each bug with a missing precondition. Column 5 presents the generated patches by NOPOL. Column 6 is the result of our manual analysis of the correctness

[10] GZoltar 1.1.0, http://gzoltar.com/.
[11] Z3, http://z3.codeplex.com/.
[12] SMT-LIB, http://smt-lib.org/.
[13] jSMTLIB, http://sourceforge.net/projects/jsmtlib/.

of the patches (will be explained in RQ2). Finally, column 7 shows whether we had to modify existing test cases: "A" stands for added test cases, "T" for transformed test cases, "D" for deleted test cases. The purpose of test case modification is to yield a correct repair (will be explained in RQ3).

As shown in Table 3.2, among 22 bugs, NOPOL can fix 17 bugs: 13 out of 16 bugs with buggy IF conditions and 4 out of 6 bugs with missing preconditions. Meanwhile, four out of five unfixed bugs are related to timeout. In our work, the execution time of NOPOL is limited to up to five hours. We will empirically analyze the fixed bugs in Section 3.3.5 and explore the limitations of our approach as given by the five unfixed bugs in Section 3.3.6.

Table 3.2 also shows that patches generated by NOPOL consist of both primitive values and object-oriented features. For the object-oriented features, two major types can be found in the generated patches: nullness checking (patches of Bugs CL4 and PM2) and the `length()` method of strings (patches of Bugs CL1, CL2, CL3, and CL5).

Note that four bugs (Bugs CM2, CM3, CM6, and CM10) with buggy IF conditions are fixed by adding preconditions rather than updating conditions. One major reason is that a non-IF statement is ranked above the buggy IF statement during the fault localization; then NOPOL adds a patch, i.e., a precondition to this non-IF statement. Hence, the condition inside the buggy IF statement will not be updated. This shows that those two kinds of patches are intrinsically related to each other. To further understand this phenomenon, we have performed repair only in the mode of "condition" in NOPOL: the four bugs could also be fixed via only updating IF conditions.

RQ2: Are the synthesized patches as correct as the manual patch written by the developer?

In practice, a patch should be more than making the test suite pass since test cases may not be enough for specifying program behaviors [64], [82]. We consider a generated patch as *correct* if and only if the patch is functionally equivalent to the manually-written patch by developers.

For each synthesized patch, we have followed Qi et al. [82] to perform a manual analysis of its correctness. The manual analysis consists of understanding the domain (e.g., the mathematical function under test for a bug in Apache Commons Math), understanding the role of the patch in the computation, and understanding the meaning of the test case as well as its assertions.

As shown in Table 3.2, 13 out of 17 synthesized patches are correct. Among these 13 correct patches, four patches (for CM1, CM3, CM4, and CM5) are generated based on not only the original test suite but also the added test cases. The reason is that the original test suite is too weak to drive the synthesis of a correct patch; then we had to manually write additional test cases (all additional test cases are publicly-available on the companion website [23][14]). This will be discussed in next research question.

For four bugs (CM6, CL1, CL5, and PL2), we are not able to synthesize a correct patch, even if we can write additional test cases.

RQ3: What is the root cause of test case modification?

As shown in Table 3.2, some bugs are correctly repaired only after the test case modification (including test case addition, transformation, and deletion). The most important modification is test case addition. Four bugs (Bugs CM1, CM3, CM4, and CM5) with added test cases correspond to too weak specifications. We manually added test cases for these bugs to improve the coverage of buggy statements. Without the added test case, the synthesized patch is degenerated. A case study of Bug CM1 (Section 3.3.5.4) will further illustrate how added test cases help to synthesize patches. Note that all added test cases appear in the bugs, which are reported in the

[14] Additional test cases, http://sachaproject.gforge.inria.fr/nopol/dataset/data/projects/math/.

Table 3.3: Analysis for patched statements in 17 fixed bugs.

Bug type	Bug index	Patch location	#Test cases e_f†	#Test cases e_p†	Patched statement rank	#Suspicious statements ‡	Execution time (seconds)	SMT level
Buggy IF condition	CM1	Same as dev.	1	7	3	28	10	2
	CM2	Different	1	1	171	584	17	2
	CM3	Different	1	1	2	12	7	2
	CM4	Same as dev.	4	2	71	116	23	3
	CM5	Same as dev.	1	2	1	2	50	3
	CM6	Different	1	1	36	189	57	2
	CM7	Same as dev.	3	10	36	74	64	2
	CM10	Different	2	0	1	104	18	1
	CL1	Same as dev.	2	4	2	9	33	2
	CL2	Same as dev.	1	8	2	4	9	2
	CL3	Same as dev.	2	10	4	5	13	3
	CL4	Same as dev.	2	23	1	20	12	3
	CL5	Same as dev.	1	37	1	2	39	1
Missing precondition	PM2	Same as dev.	1	1	1	12	11	1
	PL1	Same as dev.	1	14	6	22	88	2
	PL2	Same as dev.	1	1	1	18	8	1
	PL4	Same as dev.	1	2	1	26	8	2
Median			1	2	2	20	17	2
Average			1.5	7.3	20.0	72.2	27.5	2

† e_f and e_p denote the number of failing and passing test cases that execute the patched statement.
‡ #Suspicious statements denotes the number of statements whose suspiciousness scores by fault localization are over zero.

Table 3.4: Analysis for buggy statements in 5 non-fixed bugs.

Bug type	Bug index	#Test cases e_f†	#Test cases e_p†	Buggy statement rank	#Suspicious statements ‡	Execution time (seconds)
Buggy IF condition	CM8	1	51	1206	1296	-
	CM9	2	73	625	705	-
	CL6	1	10	4	4	-
Missing precondition	PM1	4	0	3	132	41
	PL3	18	3	1	16	-
Median		2	10	4	132	-
Average		5.2	27.4	367.8	430.6	-

† e_f and e_p denote the number of failing and passing test cases that execute the buggy statement.
‡ #Suspicious statements denotes the number of statements whose suspiciousness scores by fault localization are over zero.

early stage of the project Math. One possible reason for such test case addition is that test cases are not well-designed when Math is inmature.

In test case transformation (CM2, CM3, CM5, CM6, CM7, CL2, CL3, CL4, PL3, and PL4), we simply break existing test cases into smaller ones, in order to have one assertion per test case. This is important for facilitating angelic value mining since our implementation of NOPOL has a limitation, which collects only one runtime trace for a test case (Section 3.2.2). We note that such test case transformation can even be automated [96].

The reason behind the four bugs with deleted test cases (CM6, CM7, CL1, and CL4) is accidental and not directly related to automatic repair: these deleted test cases are no longer compatible with the Java version and external libraries, which are used in NOPOL.

RQ4: How are the bugs of the dataset specified by test cases?

To further understand the repair process of bugs by NOPOL, Tables 3.3 and 3.4 show the detailed analysis for patched statements in 17 fixed bugs and buggy statements in five non-fixed bugs, respectively. Tables 3.3 gives the following information: whether a synthesized patch is at the same location as the one written by the developer, the number of failing (e_f) and passing (e_p) test cases executing the patched statements, the fault localization metrics (the rank of the

patched statement and the total number of suspicious statements), the overall execution time of NOPOL, and the synthesis level. In Table 3.4, e_f and e_p denote the number of failing and passing test cases executing the buggy statements while the rank of the buggy statement is listed.

Table 3.3 and Table 3.4 show the numbers of failing (e_f) and passing (e_p) test cases that execute the patched or buggy statements. Such numbers reflect to which extent the statement under repair is specified by test cases. As shown in Table 3.3, the average of e_f and e_p are 1.5 and 7.3 for 17 bugs with synthesized patches. In Table 3.4, the average of e_f and e_p are 5.2 and 27.4 for five bugs without patches.

For all 22 bugs under evaluation, only one bug has a large number of failing test cases ($e_f \geq 10$): Bug PL3 with $e_f = 18$. For this bug, although the buggy statement is ranked at the first place, NOPOL fails in synthesizing the patch. This is caused by the large number of failing test cases which results in a complex SMT problem. Section 3.3.6.3 will explain the reason behind this failure.

RQ5: Where are the synthesized patches localized? How long is spent synthesizing patches?

A patch could be localized in a different location from the patch which is manually-written by developers. We present the details for patch locations for all the 17 patched bugs in Table 3.3. For 13 out of 17 fixed bugs, the patched statements (i.e., the location of the fix), are exactly the same as the buggy ones. For the other four bugs, i.e., Bugs CM2, CM3, CM6, and CM10, NOPOL generates patches by adding new preconditions rather than updating existing conditions, as mentioned in Table 3.2.

For 17 fixed bugs in Table 3.3, the average execution time of repairing one bug is 27.5 seconds while for five non-fixed bugs in Table 3.4, four bugs are run out of time and the other one spends 41 seconds. The execution time of all the 22 bugs ranges from 7 to 88 seconds. We consider that such execution time, i.e., fixing one bug within 90 seconds, is acceptable.

In practice, if applying NOPOL to a buggy program, we can directly set a timeout, e.g., 90 seconds (over 88 seconds as shown in Table 3.3) or a longer timeout like five hours in our experiment. Then for any kind of buggy program (without knowing whether the bug is with a buggy condition or a missing precondition), NOPOL will synthesize a patch, if it finds any. Then a human developer can check whether this patch is correct from the user perspective.

RQ6: How effective is fault localization for the bugs in the dataset?

Fault localization is an important step in our repair approach. As shown in Table 3.3, for patched statements in 17 fixed bugs, the average fault localization rank is 20.0. In four out of 17 bugs (CM2, CM4, CM6, and CM7), patched statements are ranked over 30. This fact indicates that there is room for improving the fault localization techniques. Among the five unfixed bugs, the buggy statements of Bugs CM8 and CM9 are over 600.

Note that in Tables 3.3 and 3.4, Bug CM10 and Bug PM1 have no passing test cases. Bug PM1 cannot be fixed by our approach while Bug CM10 can be still fixed because the two failing test cases give a non-trivial input-output specification. The reason behind the unfixed Bug PM1 is not $e_p = 0$ and will be discussed in Section 3.3.6.1.

3.3.5 Case Studies

We take four case studies to show how NOPOL fixes bugs with buggy IF conditions and missing preconditions. These bugs are selected because they highlight different facets of the repair process. The patch of Bug PL4 (Section 3.3.5.1) is syntactically the same as the manually-written one; the patch of Bug CL4 (Section 3.3.5.2) is correct (beyond passing the test suite)

Table 3.5: Three test cases for Bug PL4

Input	Output, `translate(input)`		Test
`input`	Expected	Observed	result
"Test &#x"	"Test &#x"	"Test &#x"	Pass
"Test &#X"	"Test &#X"	"Test &#X"	Pass
"Test &#x30 not test"	"Test \u0030 not test"	"Test &#x30 not test"	Fail

Table 3.6: Three of test cases for Bug CL4

Input			Output, `indexOf(substr, startIndex)`		Test
`parent`	`substr`	`startIndex`	Expected	Observed	result
abab	z	2	-1	-1	Pass
abab	(String) null	0	-1	`NullPointerException`	Fail
xyzabc	(String) null	2	-1	`NullPointerException`	Fail

but different from the manually-written one; the patch of Bug CM2 (Section 3.3.5.3) is correct by adding a precondition rather than updating the buggy condition, as written by developers; and the patches of Bug CM1 (Section 3.3.5.4) requires additional test cases.

3.3.5.1 Case Study 1, Bug PL4

NOPOL can generate the same patches for four out of 22 bugs as the manually-written ones. We take Bug PL4 as an example to show how the same patch is generated. This bug is fixed by adding a precondition. Fig. 3.4 shows a method `translate()` and the buggy method `translateInner()` of Bug PL4. The method `translate()` is expected to translate a term in the form of `&#[xX]?\d+;?` into codepoints, e.g., translating the term "&#x30" into "\u0030".

To convert from `input` to codepoints, the characters in `input` are traversed one by one. Note that for a string ending with "&#x", no codepoint is returned. Lines 15 to 20 in Fig. 3.4 implement this functionality. However, the implementation at Lines 17 to 19 ignores a term in a feasible form of "&#[xX]\d+", e.g., a string like "&#x30". A precondition should be added to detect this feasible form, i.e., the comment at Line 18 of `start == seqEnd`.

The buggy code at Line 19 is executed by two passing test cases and one failing test case. Table 3.5 shows these three test cases. For the two passing test cases, the behavior of the method is expected not to change variable `input` while for the failing test case, the `input` is expected to be converted. For the passing test cases, the value of the precondition of the statement at Line 19 is expected to be `true`, i.e., both `start` and `seqEnd` equals to `8`, while for the failing test case, the condition is expected to be `false`, i.e., `start` and `seqEnd` are `8` and `19`, respectively. The `false` value is the angelic value for a missing precondition.

According to those expected precondition values for test cases, NOPOL generates a patch via adding a precondition, i.e., `start == seqEnd`, which is exactly the same as the manually-written patch by developers. Besides the patch of Bug PL4, patches of Bug CM4, CM7, and PM2 are also syntactically the same as the patch written by developers, among 22 bugs in our dataset.

3.3.5.2 Case Study 2, Bug CL4

For several bugs, NOPOL generates literally different patches from the manually-written patches, but these generated patches are correct. In this section, we present a case study where NOPOL synthesizes a correct patch for a bug with a buggy IF condition. Bug CL4 in Lang fails to find the index of a matched string in a string builder. Fig. 3.5 presents the buggy method of Bug CL4: to return the first index of `substr` in a parent string builder from a given basic index `startIndex`. The condition at Line 4 contains a mistake of `startIndex >= size`, which omits

```
1  String translate(CharSequence
       input) {
2    int consumed = translateInner(
         input);
3    if(consumed == 0)
4      ... // Return the original
           input value
5    else
6      ... // Translate code points
7  }
8
9  int translateInner(CharSequence
       input) {
10   int seqEnd = input.length();
11   ...
12   int start = 2;
13   boolean isHex = false;
14   char firstChar = input.charAt(
         start);
15   if(firstChar == 'x' || firstChar
         == 'X') {
16     start++;
17     isHex = true;
18 // FIX: if(start == seqEnd)
19     return 0;
20   }
21   int end = start;
22   //Traverse the input and parse
         into codepoints
23   while(end < seqEnd && ... )
24   ...
25 }
```

Figure 3.4: Code snippet of Bug PL4. The manually-written patch is shown in the `FIX` comment at Line 18. Note that the original method `translate` consists of three overloaded methods; for the sake of simplification, we use two methods `translate` and `translateInner` instead.

checking whether `substr == null`. A variable `size` is defined as the length of the parent string builder. The manually-written fix is shown at Line 3.

The buggy code at Line 4 in Fig. 3.5 is executed by 23 passing test cases and two failing test cases. One of the passing test cases and two failing test cases are shown in Table 3.6. For the passing test case, a value `-1` is expected because no matched string is found. For the two failing test cases, each input `substr` is a `null` value, which is also expected to return a non-found index `-1`. This requires the checking of `null` to avoid `NullPointerException`, i.e., the condition at Line 3.

For the passing test case in Table 3.6, the condition at Line 4 is `false`. For the two failing test cases, NOPOL extracts the angelic value `true` to make both failing test cases pass. According to these condition values, a patch of `(startIndex >= size || substr == null) && size != -1` can

```
1  int indexOf(String substr, int
       startIndex) {
2    startIndex = (startIndex
         < 0 ? 0 : startIndex);
3  // FIX: if (substr == null ||
       startIndex >= size) {
4    if (startIndex >= size) {
5      return -1;
6    }
7    int strLen = substr.length();
8    if (strLen > 0 && strLen <=
         size) {
9      if (strLen == 1)
10       return indexOf(substr.
             charAt(0), startIndex);
11     char[] thisBuf = buffer;
12     outer:
13     for (int i = startIndex; i <
           thisBuf.length
14               - strLen; i++) {
15       for (int j = 0; j < strLen; j
             ++) {
16         if (substr.charAt(j) !=
               thisBuf[i + j])
17           continue outer;
18       }
19       return i;
20     }
21   } else if (strLen == 0) {
22     return 0;
23   }
24   return -1;
25 }
```

Figure 3.5: Code snippet of Bug CL4. The manually-written patch is shown in the `FIX` comment at Line 3, which updates the buggy IF condition at Line 4.

be synthesized. This patch is different from the manually-written one at Line 3. The difference is the additional `size != -1`. However, from the context in the method, the length of the parent string builder is no less than zero, i.e., `size >= 0` holds. The value of `size != -1` is always `true`. Then the synthesized patch is equivalent to `startIndex >= size || substr == null`, which is correct. The reason for the unnecessary expression in the patch (i.e., `&& size != -1`) is that a constant value `-1` is collected as either runtime data or standard primitive data by NOPOL (see Section 3.2.4.2). This value is then encoded in the SMT instance. The resulted expression based on the solution to the SMT will not weaken the repairability of synthesized patches since it does not result in more failing test cases. A recent method for finding simplified patches, proposed by Mechtaev et al. [61], could be used to avoid such redundant expression.

```
1  long binomialCoefficient(int n,
       int k) {
2    if (n < k)
3      throw new
           IllegalArgumentException
           (...);
4    if (n <= 0) // FIX: if (n < 0)
5      throw new
           IllegalArgumentException
           (...);
6    if ((n == k) || (k == 0))
7      return 1;
8    if ((k == 1) || (k == n - 1))
9      return n;
10   long result = Math.round(
11       binomialCoefficientDouble(n
             , k));
12   if (result == Long.
         MAX_VALUE)
13     throw new
           ArithmeticException(...);
14   return result;
15 }
```

Figure 3.6: Code snippet of Bug CM2. The manually-written patch is shown in the `FIX` comment at Line 4.

3.3.5.3 Case Study 3, Bug CM2

In this section, we present Bug CM2, a correctly patched bug via adding a precondition, rather than updating the existing condition, as written by developers. The buggy method in Bug CM2 is to calculate the value of Binomial Coefficient by choosing `k`-element subsets from an `n`-element set. Fig. 3.6 presents the buggy method. The input number of elements `n` should be no less than zero. But the condition at Line 4 reads `n <= 0` instead of `n < 0`. The manually-written patch by developers is in the `FIX` comment at Line 4.

The buggy code at Line 4 in Fig. 3.6 is executed by one passing test case and one failing test case. Table 3.7 shows these two test cases. For the passing test case, an exact value is calculated. For the failing test case, an `IllegalArgumentException` is thrown rather than the exact value.

To fix this bug, NOPOL generates a patch via adding a missing precondition `n < 0` to the statement in Line 5. Then this statement owns two embeded preconditions, i.e., `n <= 0` and `n < 0`. Hence, the generated patch is equivalent to the manually-written patch, i.e., updating the condition in Line 4 from `n <= 0` to `n < 0`. The reason of adding a precondition instead of updating the original condition that the statement in Line 5 is ranked prior to the statement in Line 4. This has been explained in Section 3.3.4. Consequently, the generated patch of Bug CM2 is correct and equivalent to the manually-written patch.

Table 3.7: Two test cases for Bug CM2

Input		Output, `binomialCoefficient(n,k)`		Test
`n`	`k`	Expected	Observed	result
-1	-1	Exception	Exception	pass
0	0	1	Exception	fail

Table 3.8: Two of original test cases and two added test case for Bug CM1

Input		Output, `evaluate(values,p)`		Test
`values`	`p`	Expected	Observed	result
Two original test cases				
{0,1}	25	0.0	0.0	pass
{1,2,3}	75	3.0	Exception	fail
Two added test case				
{1,2,10}	75	10.0	Exception	fail
{1,2,10,15,75}	100	75.0	75.0	pass

Table 3.9: Summary of the Limitation Analysis.

Bug index	Root cause	Result of repair	Reason for the unfixed bug
PM1	Angelic fix localization	No angelic value found. Termination before runtime trace collection and patch synthesis	One failing test case executes the missing precondition for more than once.
CM9	Angelic fix localization	Timeout during test suite execution	An infinite loop is introduced during the trial of angelic values.
PL3	Runtime trace collection	Timeout during SMT solving	The expected value of a precondition is incorrectly identified.
CM8	Patch synthesis	Timeout during SMT solving	A method call with parameters is not handled by SMT.
CL6	Patch synthesis	Timeout during SMT solving	A method of a `null` object yields an undefined value for SMT.

3.3.5.4 Case Study 4, Bug CM1

Insufficient test cases lead to trivial patch generation. We present Bug CM1 in Math, with a buggy IF condition. This bug cannot be correctly patched with the original test suite due to the lack of test cases. In our work, we add two test cases to support the patch generation. Fig. 3.7 presents the buggy source code in method `evaluate()` of Bug CM1. This method returns an estimate of the percentile `p` of the values stored in the array `values`.

According to the API document, the algorithm of `evaluate()` is implemented as follows. Let `n` be the length of the (sorted) array. The algorithm computes the estimated percentile position `pos = p * (n + 1) / 100` and the difference `dif` between `pos` and `floor(pos)`. If `pos >= n`, then the algorithm returns the largest element in the array; otherwise the algorithm returns the final calculation of percentile. Thus, Line 15 in Fig. 3.7 contains a bug, which should be corrected as `if(pos >= n)`.

As shown in Table 3.3, this bug is executed by seven passing test cases and one failing test case. Table 3.8 shows one of the seven passing test cases and the failing test case. In the failing test case, an `ArrayIndexOutOfBounds` exception is thrown at Line 16. For the passing test case, the value of the condition at Line 15 is equal to the value of the existing condition `pos > n`, i.e., `true`; for the failing test case, setting the condition to be `true` makes the failing test case pass; that is, the angelic value for the failing test case is also `true`. Thus, according to these two test cases, the generated patch should make the condition be `true` to pass both test cases.

With the original test suite, NOPOL generates a patch as `length == intPos`, which passes all test cases. This patch is incorrect. To obtain a correct patch (the one shown in Table 3.2), we add two test cases of `values` ← `{1,2,10}`, `p` ← `75` and `values` ← `{1,2,10,15,75}`, `p` ← `100`, as shown in Table 3.8. Then the expected values of `evaluate()` are `10.0` and `75.0`, respectively.

```
1  double evaluate(double[] values,
       double p) {
2    ...
3    int length = values.length;
4    double n = length;
5    ...
6    double pos = p * (n + 1)
         / 100;
7    double fpos = Math.floor(pos);
8    int intPos = (int) fpos;
9    double dif = pos - fpos;
10   double[] sorted = new double[
         n];
11   System.arraycopy(values, 0,
         sorted, 0, n);
12   Arrays.sort(sorted);
13   if (pos < 1)
14     return sorted[0];
15   if (pos > n) // FIX: if (pos
         >= n)
16     return sorted[n - 1];
17   double lower = sorted[intPos
         - 1];
18   double upper = sorted[intPos];
19   return lower + dif * (upper -
         lower);
20 }
```

Figure 3.7: Code snippet of Bug CM1. The manually-written patch is shown in the `FIX` comment at Line 15.

After running NOPOL, a patch of `length <= fpos`, which is different from the manually-written one (`pos >= n`). However, from the source code at Line 7, `fpos` is the `floor()` value of `pos`, i.e., `fpos` is the largest integer that no more than `pos`. That is, `fpos <= pos`. Meanwhile, `n == length` holds according to Line 4. As a result, the generated patch `length <= fpos` implies the manually-written one, i.e., `pos >= n`. We can conclude that NOPOL can generate a correct patch for this bug by adding two test case.

3.3.6 Limitations

As shown in Section 3.3.4, we have collected five bugs that reveal five different limitations of NOPOL. Table 3.9 lists these five bugs in details. We analyze the related limitations in this section.

3.3.6.1 No Angelic Value Found

In our work, for a buggy IF condition, we use angelic fix localization to flip the Boolean value of the condition for failing test cases. For Bug PM1, no angelic value is found as shown in Table

Table 3.10: Two test cases for Bug PL3

Input			Output, abbreviate(str,lower,upper)		Test
str	lower	upper	Expected	Observed	result
"0123456789"	0	-1	"0123456789"	"0123456789"	pass
"012 3456789"	0	5	"012"	"012 3456789"	fail

3.2. The reason is that both `then` and `else` branches of the IF condition are executed by one failing test case. Hence, no single angelic value (`true` or `false`) can enable the test case to pass. As discussed in Section 3.2.3, the search space of a sequence of angelic values is exponential and hence discarded in our implementation of NOPOL.

To mitigate this limitation, a straightforward solution is to discard the failing test case, which leads to no angelic values (keeping the remaining failing ones). However, this may decrease the quality of the generated patch due to the missing test data and oracles. Another potential solution is to refactor test cases into small snippets, each of which covers only `then` or `else` branches [96]. A recent proposed technique SPR [55] could help NOPOL to enhance its processing of sequential angelic values.

3.3.6.2 Performance Bugs Caused by Angelic Values

NOPOL identifies angelic values as the input of patch synthesis. In the process of angelic fix localization, all failing test cases are executed to detect conditional values that make failing test cases pass (see Section 3.2.2). However, sometimes the trial of angelic fix localization (forcing to `true` or `false`) may result in a performance bug. In our work, Bug CM9 cannot be fixed due to this reason, i.e., an infinite loop caused by the angelic value.

A potential solution to this issue is to set a maximum execution time to avoid the influence of performance bugs. But a maximum execution time of test cases may be hard to be determined according to different test cases. For instance, the average execution time of test cases in Math 3.0 is much longer than that in Math 2.0. We leave the setting of maximum execution time as one piece of future work.

3.3.6.3 Incorrectly Identified Output of a Precondition

As mentioned in Section 3.2.4.1, the expected output of a missing precondition is set to be `true` for a passing test case and is set to be `false` for a failing one. The underlying assumption for a passing test case is that the `true` value keeps the existing program behavior. However, it is possible that given a statement, both `true` and `false` values can make a test case pass. In these cases, synthesis may not work for bugs with missing preconditions.

This is what happens to Bug PL3. Fig. 3.8 shows the code snippet of Bug PL3. The manually-written patch is a precondition in Line 3; Table 3.10 shows one passing test case and one failing test case. Based on the angelic fix localization in Algorithm 2, the expected precondition values of all passing test cases are set to be `true`. However, in the manually-written patch, the precondition value by the passing test case in Table 3.10 is `false`, i.e., `lower > str.length()` where `lower` is `0` and `str.length()` is `10`. Thus, it is impossible to generate a patch like the manually-written one, due to a conflict in the input-output specification. Consequently, in the phase of patch synthesis, the SMT solver executes with timeout.

A potential solution to this problem is to discard the test case that can be passed by both `true` and `false` values for a precondition. But this may lead to the same problem of lack of test data and oracles as mentioned in Section 3.3.6.1.

```
1  String abbreviate(String str, int
       lower, int upper) {
2    ...
3    // FIX: if (lower > str.length())
4    lower = str.length();
5
6    if (upper == -1 || upper > str.
       length())
7      upper = str.length();
8    if (upper < lower)
9      upper = lower;
10   StringBuffer result = new
       StringBuffer();
11   int index = StringUtils.indexOf
       (str, " ", lower);
12   if (index == -1)
13     result.append(str.substring(0,
         upper));
14   else ...
15   return result.toString();
16 }
```

Figure 3.8: Code snippet of Bug PL3. The manually-written patch is shown in the `FIX` comment at Line 3.

3.3.6.4 Complex Patches using Method Calls with Parameters

In our work, we support the synthesis of conditions that call unary methods (without parameters). However, our approach cannot generate a patch if a method with parameters has to appear in a condition. For example, for Bug CM8, the patch that is written by developers contains a method `abs(x)` for computing the absolute value. Our approach cannot provide such kinds of patches because methods with parameters cannot be directly encoded in SMT. Then the lack of information of method calls will lead to the timeout of an SMT solver.

A workaround would generate a runtime variable to collect existing side-effect free method calls with all possible parameters. For example, one could introduce a new variable `double tempVar = abs(x)` and generate a patch with the introduced variable `tempVar`. However, this workaround suffers from the problem of combinatorial explosion.

3.3.6.5 Unavailable Method Values for a Null Object

Our repair approach can generate a patch with objected-oriented features. For example, a patch can contain state query methods on Java core library classes, such as `String.length()`, `File.exists()` and `Collection.size()`. We map these methods to their return values during the SMT encoding. However, such methods require that the object is not `null`; otherwise a `null` pointer exception in Java will be thrown.

Let us consider Bug CL6, whose manually-written patch is `cs == null || cs.length() == 0`. For this bug, one passing test case detects whether the object `cs` is `null`. For this test case, the value of `cs.length()` is undefined and not given to SMT. Thus, it is impossible to generate a patch, which contains `cs.length()` if `cs` is `null` by at least one test case. Consequently, the SMT

solver shows timeout because it tries to find a complex patch that satisfies the constraints.

A possible solution is to encode the undefined values in the SMT. Constraints should be added to ensure that the unavailable values will not be involved in the patch. This needs important changes in the design of the encoding, which is left to future work.

3.4 Threats to Validity

We discuss the threats to the validity of our results along four dimensions.

3.4.1 External Validity

In this work, we evaluate our approach on 22 real-world bugs with buggy IF conditions and missing preconditions. One threat to our work is that the number of bugs is not large enough to represent the actual effectiveness of our technique which is a threat to the external validity. While the number of bugs in our work is fewer than that in previous work [36, 73, 47], the main strength of our evaluation is twofold. On the one hand, our work focuses on two specific types of bugs, i.e., buggy IF conditions and missing preconditions (as opposed to general types of bugs in [47]); on the other hand, our work is evaluated on real-world bugs in large-scale object-oriented programs (as opposed to bugs in small-scale programs in [73] and bugs without object-oriented features in [36]). We note that it is possible to collect more real-world bugs, to the price of more human labor. As mentioned in Section 3.3.2, reproducing a specific bug is complex and time-consuming.

3.4.2 Single Point of Repair

As all the previous work in test-suite based repair, our approach can only deal with a program that contains only one buggy statement, which is the single point of repair. In the current implementation of NOPOL, we do not target programs with multiple faults, or bugs which require patches at multiple locations.

3.4.3 Test Case Modification

In our work, we aim to repair bugs with buggy IF conditions and missing preconditions. Test cases are employed to validate the generated patch. In our experiment, several test cases are modified to facilitate repair. As mentioned in Section 3.3.4, such test case modification consists of test case addition, transformation, and deletion. The answer to RQ3 analyzes the root causes of test case modification. All test case modifications are listed in our project website [23].

3.4.4 Dataset Construction

We describe how to construct our dataset in Section 3.3.2. The manually-written patches of conditional statements are extracted from commits in the version control system. However, it is common that a commit contains more code than the patch in buggy IF conditions and missing preconditions. In our work, we manually separate these patch fragments. In particular, the fixing commit of Bug PM2 contains two nested preconditions within a complex code snippet. We manually separate the patch of this bug according to the code context and keep only one precondition. There is a potential bias with this method.

3.5 Conclusion

In this chapter, we have presented NOPOL, a novel test-suite based approach to automatic repair. The key insight of Nopol is to transform the global specification and oracles that are in the test suite into micro-oracles at the level of boolean expression. Indeed, Nopol extracts micro-oracles for two kinds of bugs: buggy IF conditions and missing preconditions.

The process to obtain those micro-oracles introduces a novel kind of speculative execution called angelic value mining. It also uses fine-grained monitoring to extract a micro regression oracle. Once the micro-oracles are known, off-the-shelf synthesis techniques can be used to synthesize a new and correct boolean expression. The evaluation on 22 real-world bugs from large scale open source software applications shows that the approach works well. We are indeed capable to obtain micro-specifications and micro-oracles at the level of if conditions, and those micro-oracles are rich and complete enough to drive the synthesis of a new expression. Section 5 will discuss the perspectives of this work.

Chapter 4

A Generic Oracle for Software Resilience

This chapter is based on content from:

- Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions (Benoit Cornu, Lionel Seinturier, Martin Monperrus), In Information and Software Technology, Elsevier, volume 57, pp. 66–76, 2015.

Contents

There are two different kinds of oracles: the application-specific ones and the generic ones. For instance, the statistical oracles presented in Chapter 2 are domain-specific: "a call to method `dispose` on SWT widget objects should be made" is valid only in the context of using the SWT user-interface toolkit. Similarly, the test suite assertions used for repair by Nopol (Chapter 3) are specific to the project under repair.

A generic oracle is an expected behavior that holds across different software applications and domains. For instance, it is widely expected that a program shall not crash unexpectedly. This is an example of generic oracle. Other examples of generic oracles are for instance the required absence of deadlocks in concurrent programs [26], or of memory leaks [41].

There are different levels of genericity. Some generic oracles span multiple applications in a single domain, or only apply to a single programming paradigm. For instance, a generic oracle is that there shall not be broken links [83], which is generic to all web applications, yet specific to web applications only.

In this chapter, we present a contribution on generic oracles in the context of software exceptions. Every programmer knows that "no unhandled exception should crash the program", and experiences it the hard way, since this happens daily in production. However, there is little work on advanced contracts and verification mechanisms with respect to unhandled exceptions.

> Surprisingly, little work has addressed the huge practical problem of unhandled exceptions. We present a contribution that is unique in providing both a conceptual oracle framework for reasoning about unhandled exceptions and an actionable algorithm to verify a resilience property with respect to them.

We consider the exception system of the Java programming language, which resembles the ones of most mainstream modern programming languages. Java exception's system is based on try blocks, catch blocks (aka handlers) and throw instructions (aka signals). We define resilience against unhandled exceptions as the ability to correctly handle exceptions that were encountered neither during development nor during testing. This chapter presents two contracts on exceptions handling related to this notion of resilience.

To verify those contracts, our key insight is to simulate unanticipated exceptions by injecting exceptions at appropriate places during test suite execution. This novel fault injection technique is called "short-circuit testing": it consists of throwing exceptions at the beginning of try-blocks, simulating the worst error case when the complete try-block is skipped due to the occurrence of a severe error. Despite the injected exceptions, we observe that a number of test cases still pass. When this happens, it means that the software under study is able to resist to a certain class of unanticipated exceptions. It can be said "resilient" according to our definition of resilience against unhandled exceptions.

Finally, we use the error-handling characterization obtained with the two exception oracles to improve the resilience of the application under study. We propose a novel code transformation, called "catch stretching" that consists of replacing the caught type of a catch block by one of its super-type. By enabling catch blocks to correctly handle more types of exception, the code is more capable of handling unanticipated exceptions. We conduct a deep analysis to show the meaningfulness of catch stretching.

We evaluate our approach by analyzing the resilience of 9 well-tested open-source applications written in Java. In this dataset, we analyze the resilience capabilities of 241 try-catch blocks and show that 92 of them satisfy at least one resilience contract and 24 try-catch blocks violate a resilience property.

Our technique is fundamentally novel. There are techniques to provide information about the test suite with respect to exceptions ([9, 31, 32, 102]). On the contrary, our contribution is on analyzing and improving the application code itself (and not the test suite which is just a means). Also, other papers make static analysis of exception handling ([88, 89]), as opposed to this work which is a dynamic technique.

To sum up, our contributions are:

- A definition and formalization of two oracles on try-catch blocks,
- An algorithm and four predicates to verify whether a try-catch exposes the expected behavior.
- A code transformation to improve the resilience against exceptions,
- An empirical evaluation on 9 open sources applications with one test suite each showing that there exists resilient try-catch blocks in practice.

4.1 Background

In our work, we use Avizienis, Laprie and Randell's definitions [2] of faults, errors and failures, given in Section 1.2. However, we also consider the common usage, which consists of "fault-injection" and "error-handling" whereas it might sometimes be more appropriate to say "error-injection" or "fault-handling". In our paper, for sake of understandability, we prefer the common usage and use well-known expressions such as "fault-injection" or "fault model".

4.1.1 Background on Exceptions

Exceptions are programming language constructs for handling errors [34]. Exceptions can be thrown and caught. When one throws an exception, this means that something has gone wrong and this cancels the nominal behavior of the application: the program will not follow the normal control-flow and will not execute the next instructions. Instead, the program will "jump" to the nearest matching catch block. In the worst case, there is no matching catch block in the stack and the exception reaches the main entry point of the program and consequently, stops its execution (i.e. crashes the program). When an exception is thrown then caught, it is equivalent to a direct jump from the throw location to the catch location: in the execution state, only the call stack has changed, but not the heap[1]. For a practical presentation of exceptions in mainstream programming languages, we refer to any introductory textbook, e.g. [86]. Avizienis et al. [2] do not mention exceptions. We consider exceptions as errors and we use the term error in the paper as much as possible.

4.1.2 Definition of Resilience

We embrace the definition of "software resilience" by Laprie as interpreted by Trivedi et al.:

Definition 5 *Resilience is* "the persistence of service delivery that can justifiably be trusted, when facing changes" *[50]*. "Changes may refer to unexpected failures, attacks or accidents (e.g., disasters)" *[91]*.

[1] the heap may change if the programming language contains a finalization mechanism (e.g. in Module-2+ [85])

Along with Trivedi et al., we interpret the idea of "unexpected" events with the notion of "design envelope" [91], a known term in safety critical system design. The design envelope defines all the anticipated states of a software system. It defines the boundary between anticipated and unanticipated runtime states. The design envelope contains both correct states and incorrect states, the latter resulting from the anticipation of misusages and attacks. According to that, "resilience deals with conditions that are outside the design envelope" [91]. Along this line, we consider that the main difference between software resilience and software robustness is that software robustness deals with anticipated kinds of errors (i.e. inside the "design envelope"). In this chapter, we focus on the resilience in the context of software that uses exceptions. We interpret and refine this general definition in the context of mainstream exception handling.

Definition 6 *Resilience against exceptions is the software system's ability to reenter a correct state when an unanticipated exception occurs.*

4.1.3 Specifications and Test Suites

A test suite is a collection of test cases where each test case contains a set of assertions [8]. The assertions specify what the software is meant to do (i.e. it defines the design envelope). Hence, in the rest of this chapter, we consider the test suites as specifications[2]. For instance, "assert(3, division(15,5))" specifies that the result of the division of 15 by 5 should be 3.

A test suite may also encode what a software package does outside standard usage (error defined in the design envelope). For instance, one may specify that "division(15,0)" should throw an exception "Division by zero not possible". Hence, the exceptions that are thrown during test suite execution are the anticipated errors. If an exception is triggered by the test and caught later on in the application code, the assertions specify that the exception-handling code has worked as expected.

Our definition of resilience relates to exceptions that are not specified, that are outside the design envelope. We consider test suites as approximation of the design envelope. Consequently, the considered resilience consists in handling unanticipated exceptions where we define unanticipated exceptions as exceptions that are not triggered during test suite execution.

4.2 Two Generic Oracles for Unhandled Exceptions

We define in Section 4.2.1 two exception contracts applicable to try-catch blocks. We then describe an algorithm (see Section 4.2.2) and formal predicates (see Section 4.2.3) to verify those contracts according to a test suite. Finally we present the concept of catch stretching, a technique to improve the resilience of software applications against exceptions (see Section 4.2.4). The insight behind our approach is that we can use test suites as an oracle for the resilience capabilities against unanticipated errors.

4.2.1 Definition

We now present two novel contracts for exception-handling programming constructs. We use the term "contract" in its generic acceptation: a property of a piece of code that contributes to reuse, maintainability, correctness or another quality attribute. For instance, the "hashCode/equals"

[2] Conversely, when we use the term "specification", we refer to the test suite (even if they are an approximation of an idealized specification [90])

Listing 4.1: An Example of Source-Independent Try-Catch Block.

```
try{
 String arg = getArgument();
 String key = format(arg);
 return getProperty(key ,
        isCacheActivated);
}catch(MissingPropertyException
      e){
  return "missing property";
}
```

Listing 4.2: An Example of Source-Dependent Try-Catch Block.

```
boolean isCacheActivated = false;
try{
  isCacheActivated =
        getCacheAvailability();
  return getProperty(key ,
        isCacheActivated);
}catch(MissingPropertyException
      e){
  if( isCacheActivated ){
    return "missing property";
  }else{
    throw new
          CacheDisableException();
} }
```

Listing 4.3: An Example of Purely-Resilient Try-Catch Block.

```
try{
  return getPropertyFromCache(
        key);
}catch(MissingPropertyException
      e){
  return getPropertyFromFile(key)
        ;
}
```

contract[3] is a property on a pair of methods. Our definition is broader in scope than Meyer's "contracts" [63] which refer to preconditions, postconditions and invariants contracts.

We focus on contracts on the programming language construct try and catch blocks, which we refer to as "try-catch". A try-catch is composed of one try block and one catch block. Note that a try with multiple catch blocks is considered as a set of pairs consisting of the try block and one of its catch blocks. This means that a try with n catch blocks is considered as n try-catch blocks. This concept is generalized in most mainstream languages, sometimes using different names (for instance, a catch block is called an "except" block in Python). In this work, we ignore the concept of "finally" block [35] which is more language specific and much less used in practice [10].

4.2.1.1 Source Independence Contract

Motivation When a harmful exception occurs during testing or production, a developer has two possibilities. One way is to avoid the exception to be thrown by fixing its root cause (e.g. by inserting a not null check to avoid a null pointer exception). The other way is to write a try block surrounding the code that throws the exception. The catch block ending the try block defines the recovery mechanism to be applied when this exception occurs. The catch block responsibility is to recover from the particular encountered exception. By construction, the same recovery would be applied if another exception of the same type occurs within the scope of the try block at a different location.

This motivates the source-independence contract: the normal recovery behavior of the catch block must work for the foreseen exceptions; but beyond that, it should also work for exceptions that have not been encountered but may arise in a near future.

We define a novel exception contract that we called "source-independence" as follows:

Definition 7 *A try-catch is source-independent if the catch block proceeds equivalently, whatever the source of the caught exception is in the try block.*

For now, we loosely define "proceeds equivalently": if the system is still in error, it means that the error kind is the same; if the system has recovered, it means that the available functionalities are the same.

[3] http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

For example, Listing 4.1 shows a try-catch that *satisfies* the source-independence contract. If a value is missing in the application, an exception is thrown and the method returns a default value “missing property". The code of the catch block (only one return statement) clearly does not depend on the application state. The exception can be thrown by any of the 3 statements in the try, and the result will still be the same.

On the contrary, Listing 4.2 shows a try-catch that *violates* the source-independence contract. Indeed, the result of the catch process depends on the value of *isCacheActivated*. If the first statement fails, the variable *isCacheActivated* is false, then an exception is thrown. If the first statement passes but the second one fails, *isCacheActivated* can be *true*, then the value *missing property* is returned. The result of the execution of the catch depends on the state of the program when the catch begins (here it depends on the value of the *isCacheActivated* variable). In case of failure, a developer cannot know if she will have to work with a default return value or with an exception. This catch is indeed source-dependent.

We will present a formal definition of this contract and an algorithm to verify it in Section 4.2.3. We will show that both source-independent and source-dependent catch blocks exist in practice in Section 4.3.

Discussion How can it happen that developers write source-dependent catch blocks? Developers discover some exception risks at the first run-time occurrence of an exception at a particular location. In this case, the developer adds a try-catch block and puts the exception raising code in the try body. Often, the try body contains more code than the problematic statement in order to avoid variable scope and initialization problems. However, while implementing the catch block, the developer still assumes that the exception can only be thrown by the problematic statement, and refers to variables that were set in previous statements in the try block. In other words, the catch block is dependent on the application state at the problematic statement. If the exception comes from the problematic statement, the catch block works, if not, it fails to provide the expected recovery behavior.

The source independence contract shows that if an unanticipated exception happens the catch block would still be able to recover the application state. This means that the try-catch is able to handle unanticipated exceptions, hence it is resilient w.r.t. the chosen definition of resilience (see Section 4.1.2).

4.2.1.2 Pure Resilience Contract

Motivation In general, when an error occurs, it is more desirable to recover from this error than to stop or crash. A good recovery consists in returning the expected result despite the error and in continuing the program execution.

One way to obtain the expected result under error is to be able to do the same task in a way that, for the same input, does not lead to an error but to the expected result. Such an alternative is sometimes called “plan B”. In terms of exception, recovering from an exception with a plan B means that the corresponding catch contains the code of this plan B. The plan B performed by the catch is an alternative to the “plan A" which is implemented in the try block. Hence, the contract of the try-catch block (and not only the catch or only the try) is to correctly perform a task T under consideration whether or not an exception occurs. We refer to this contract as the “pure resilience" contract.

A pure resilience contract applies to try-catch blocks. We define it as follows:

Definition 8 *A try-catch is purely resilient if the system state is equivalent at the end of the try-catch execution whether or not an exception occurs in the try block.*

By system state equivalence, we mean that the effects of the plan A on the system are similar

to those of plan B from a given observation perspective. If the observation perspective is a returned value, the value from plan A is semantically equivalent to the value of plan B (e.g. satisfies an "equals" predicate method in Java).

For example, Listing 4.3 shows a purely resilient try-catch where a value is required, the program tries to access this value in the cache. If the program does not find this value, it retrieves it from a file. We will present a formal definition of this contract and an algorithm to verify it in Section 4.2.3.

Usage There are different use cases of purely resilient try-catch blocks. We have presented the use case of caching for pure resilience in Listing 4.3. One can use purely resilient try-catch blocks for performance reasons: a catch block can be a functionally equivalent yet slower alternative. The efficient and more risky implementation of the try block is tried first, and in case of an exception, the catch block takes over to produce a correct result. Optionality is another reason for pure resilience. Whether or not an exception occurs during the execution of an optional feature in a try block, the program state is valid and allows the execution to proceed normally after the execution of the try-block.

Discussion The difference between source-independence and pure resilience is as follows. Source-independence means that *under error* the try-catch has always the same observable behavior. In contrast, pure resilience means that *in nominal mode and under error* the try-catch block has always the same observable behavior. This shows that pure resilience subsumes source-independence: by construction, purely resilient catch blocks are source-independent. The pure resilience contract is a loose translation of the concept of recovery block [42] in mainstream programming languages. A purely resilient try-catch is able to handle unanticipated exceptions in any situation while still performing the expected operation. Hence it is resilient w.r.t. the chosen definition of resilience (see Section 4.1.2).

Although the "pure resilience" contract is strong, we will show in Section 4.3 that we observe purely resilient try-catch blocks in reality, without any dedicated search: the dataset under consideration has been set up independently of this concern. The source independence and pure resilience contracts are not meant to be mandatory. The try-catch blocks can satisfy one, both, or none. We only argue that satisfying them is better from the viewpoint of resilience, according to our technical definition of resilience given in Section 4.1.2.

4.2.2 The Short-circuit Testing Algorithm

We now present a technique, called "short-circuit testing", which allows one to find source-independent and purely-resilient try-catch blocks.

Definition 9 *Short-circuit testing consists of dynamically injecting exceptions during the test suite execution in order to analyze the resilience of try-catch blocks.*

The system under test is instrumented so that the effects of injected exceptions are logged. This data is next analyzed to verify whether a try-catch block satisfies or violates the two contracts aforementioned. The injected exceptions represent unanticipated situations.

According to our definition of resilience and test suites as specification, everything that is in the test suite is anticipated. To identify unanticipated exceptions, we need to artificially create runtime cases that are not in the test suite. This is exactly the key point of short-circuit testing: it complements the test suite with unanticipated scenarios. Short-circuit testing allows us to study the resilience of try-catch blocks in unanticipated scenarios.

We call this technique "short-circuit testing" because it ressembles electrical short-circuits: when an exception is injected, the code of the try block is somehow short-circuited. The name

Input: An Application A, a test suite TS specifying the behavior of A.
Output: a matrix M (try-catch *times* test cases, the cells represent test success or failure.
begin
  $try\_catch\_list \leftarrow static\_analysis(A)$ ▷ retrieve all the try-catch of the application
  $standard\_behavior \leftarrow standard\_run(TS)$ ▷ get test colors and try-catch behaviors
  for $t \in try\_catch\_list$ ▷ For each try-catch tc in the application
  do
    $prepare\_injection(tc)$ ▷ prepare the try-catch tc by setting an injector
    ▷ which will throw an exception of the type caught by tc
    ▷ at the beginning of each execution of the try tc
    $as \leftarrow get\_test\_using(tc, standard\_behavior)$ ▷ retrieve all tests in the test suite TS
    ▷ using the try of the try-catch tc
    for $a \in as$ ▷ For all test a which use the current try
    do
      $pass \leftarrow run\_test\_with\_injection(a)$ ▷ get the result of the test under injection
      $M[tc, a] = pass$ ▷ store the result of the test a under injection in tc
    end
  end
  return M
end

Algorithm 3: The Short-Circuit Testing Algorithm. Exception injection is used to collect data about the behavior of catch blocks.

of software short-circuit is also used in the Hystrix resilience library[4].

What can we say about a try-catch when a test passes while injecting an exception in it? We use the test suite as an oracle of execution correctness: if a test case passes under injection, the new behavior triggered by the injected exception is in accordance with the specification. Otherwise, if the test case fails, the new behavior is detected as incorrect by the test suite.

4.2.2.1 Algorithm

Our algorithm for short-circuit testing is given in Figure 3. Our algorithm needs an application A and its test suite TS.

First, a static analysis extracts the list of existing try-catch blocks. For instance, the system extracts that method foo() contains one try with two associated catch blocks: they form two try-catch blocks (see Section 4.2.1). In addition, we also need to know which test cases specify which try-catch blocks, i.e. the correspondence between test cases and try-catch blocks: a test case is said to specify a try-catch if it uses it. To perform this, the algorithm collects data about the standard run of the test suite under consideration. For instance, the system learns that the try block in method foo() is executed on the execution of test #1 and #5. The standard run of short-circuit testing also collects fine-grain data about the occurrences of exceptions in try-catch blocks during the test suite execution (see Section 4.2.3.1).

Then, the algorithm loops over the try-catch pairs (recall that a try with n catch blocks is split into n conceptual pairs of try/catch). For each try-catch pair, the set of test cases using t, called as, is extracted in the monitoring data of the standard run. The algorithm executes each one of these tests while injecting an exception at the beginning of the try under analysis. This simulates the worst-case exception, worst-case in the sense that it discards the whole code

[4]see https://github.com/Netflix/Hystrix

of the try block. The type of the injected exception is the statically declared type of the catch block. For instance, if the catch block catches "ArrayIndexOutOfBoundsException", an instance of "ArrayIndexOutOfBoundsException" is thrown.

Consequently, if the number of catch blocks corresponding to the executed try block is N, there is one static analysis, one full run of the test suite and N runs of *as*. In our example, the system runs its analysis, and executes the full test suite once. Then it runs tests #1 and #5 with fault injection twice. The first time the injected exception goes in the first catch block, the second time, it goes, thanks to typing, in the second catch block.

4.2.2.2 Output of Short-circuit Testing

The output of our short-circuit testing algorithm is a matrix M which represents the result of each test case under injection (for each try-catch). M is a matrix of boolean values where each row represents a try-catch block, and each column represents a test case. A cell in the matrix indicates whether the test case passes with exception injection in the corresponding try-catch. This matrix is used to evaluate the exception contract predicates described next in Section 4.2.3.

Short-circuit testing is performed with source code transformations. Monitoring and fault injection code is added to the application under analysis. Listing 4.4 illustrates how this is implemented. The injected code is able to throw an exception in a context dependent manner. The injector is driven by an exception injection controller at runtime.

4.2.3 Resilience Predicates

We now describe four predicates that are evaluated on each row of the matrix to assess whether: the try-catch is source-independent (contract satisfaction), the try-catch is source-dependent (contract violation), the try-catch is purely-resilient (contract satisfaction), the try-catch is not purely-resilient (contract violation).

As hinted here, there is no one single predicate p for which $contract[x] = p[x]$ and $\neg contract[x] = \neg p[x]$. For both contracts, there are some cases where the short-circuit testing procedure yields not enough data to decide whether the contract is satisfied or violated. The principle of the excluded third (*principium tertii exclusi*) does not apply in our case.

4.2.3.1 Definition of Try-catch Usages

Our exception contracts are defined on top of the notion of "try-catch usages". A try-catch usage refers to the execution behavior of try-catch blocks with respect to exceptions. We define three kinds of try-catch usages as follows: (1) No exception is thrown during the execution of the try block (called "pink usage") and the test case passes; (2) An exception is thrown during the execution of the try block and this exception is caught by the catch (called "white usage") and the test case passes; (3) An exception is thrown during the execution of the try block but this exception is not caught by the catch (called "blue usage" – this exception may be caught later or expected by the test case if it specifies an error case).

In these usages, the principle of the excluded third (*principium tertii exclusi*) applies: a try-catch usage is either pink or white or blue. Note that a single try-catch can be executed multiple times, with different try-catch usages, even in one single test. This information is used later in this Section to verify the contracts.

4.2.3.2 Source Independence Predicate

The decision problem is formulated as: given a try-catch and a test suite, does the source-independence contract hold? The decision procedure relies on two predicates.

Predicate #1 ($source\_independent[x]$): Satisfaction of the source independence contract: A try-catch x is source independent if and only if for all test cases that execute the corresponding catch block ($white\_usage$), it still passes when one throws an exception at the worst-case location in the corresponding try block.

Formally, this reads as:

$$\begin{aligned} source\_independent[x] = \forall a \in A_x | \forall u_a \in usages\_in(x,a)| \\ (is\_white\_usage[u_a] \implies pass\_with\_injection[a,x]) \end{aligned} \tag{4.1}$$

In this formula, x refers to a try-catch (a try and its corresponding catch block), A_x is the set of all tests executing x (passing in the try block), u is a try-catch usage, i.e. a particular execution of a given try-catch block, $usages\_in(x,a)$ returns the runtime usages of try-catch x in the test case a, $is\_white\_usage[u]$ evaluates to true if and only if an exception is thrown in the try block and the catch intercepts it, $pass\_with\_injection$ evaluates to true if and only if the test case t passes with exception injection in try-catch x.

Predicate #2 ($source\_dependent[x]$):Violation of the source independence contract: A try-catch x is not source independent if there exists a test case that executes the catch block ($white\_usage$) which fails when one throws an exception at a particular location in the try block.

This is translated as:

$$\begin{aligned} source\_dependent[x] = \exists a \in A_x | \forall u_a \in usages\_in(x,a)| \\ (is\_white\_usage[u_a] \wedge \neg pass\_with\_injection[a,x]) \end{aligned} \tag{4.2}$$

Pathological cases: By construction $source\_dependent[x]$ and $source\_independent[x]$ cannot be evaluated to true at the same time (the decision procedure is sound). If $source\_dependent[x]$ and $source\_independent[x]$ are both evaluated to false, it means that the procedure yields not enough data to decide whether the contract is satisfied or violated.

4.2.3.3 Pure Resilience Predicate

The decision problem is formulated as: given a try-catch and a test suite, does the pure-resilience contract hold? The decision procedure relies on two predicates.

Predicate #3 ($resilient[x]$): Satisfaction of the pure resilience contract A try-catch x is purely resilient if it is covered by at least one pink usage and all test cases that executes the try block pass when one throws an exception at the worst-case location in the corresponding try block. In other words, this predicate holds when all tests pass even if one completely discards the execution of the try block.

Loosely speaking, a purely resilient catch block is a “perfect plan B".

This is translated as:

$$\begin{aligned} resilient[x] = (\forall a \in A_x | pass\_with\_injection[a,x]) \\ \wedge (\exists a \in A_x | \exists u_a \in usages\_in(x,a) | is\_pink\_usage[u_a]) \end{aligned} \tag{4.3}$$

where is_pink_usage[u] evaluates to true if and only if no exception is thrown in the try block.

```
try{
    // injected code
    if(Controller.isCurrentTryCatchWithInjection())
      if(Controller.currentInjectedExceptionType() == Type01Exception.class ){
          throw new Type01Exception();
      }else if(Controller.currentInjectedExceptionType() == Type02Exception.class ){
          throw new Type02Exception();
      }

    ... //normal try body
...
} catch (Type01Exception t1e) {
... //normal catch body
} catch (Type02Exception t2e) {
... //normal catch body
}
```

Listing 4.4: Short-circuit testing is performed with source code injection. The injected code is able to throw an exception in a context dependent manner. The injector can be driven at runtime.

Predicate #4 ($not\_resilient[x]$): Violation of the pure resilience contract A try-catch x is not purely resilient if there exists a failing test case when one throws the exception at a particular location in the corresponding try block.

This predicate reads as:

$$not\_resilient[x] = \exists a \in A_c | \neg pass\_with\_injection[a, x] \tag{4.4}$$

Pathological cases By construction $resilient[x]$ and $not\_resilient[x]$ cannot be evaluated to true at the same time (the decision procedure is sound). Once again, if they are both evaluated to false, it means that the procedure yields not enough data to decide whether the contract is satisfied or violated.

4.2.4 Improving Software Resilience with Catch Stretching

We have defined two formal criteria of software resilience and an algorithm to verify them (Section 4.2.3). How to put this knowledge in action?

For both contracts, one can improve the test suite itself. As discussed above, some catch blocks are never executed and others are not sufficiently executed to be able to infer their resilience properties (the pathological cases of Section 4.2.3). The automated refactoring of the test suite is outside the scope of this work.

4.2.4.1 Definition of Catch Stretching

We now aim at improving the resilience against unanticipated exceptions, those exceptions that are not specified in the test suite and even not foreseen by the developers. According to our definition of resilience, this means improving the capability of the software under analysis to correctly handle unanticipated exceptions.

One solution to force the contract satisfaction is to reduce the size of try blocks, so that they only contain statements involving exceptions during test suite execution. This is a trivial

	# executed try-catch	# purely resilient try-catch	# source-independent try-catch	# source-dependent try-catch	Unknown w.r.t. resilience	Unknown w.r.t. source independence	# Stretchable try-catch
commons-lang	49	3/49	18/49	5/49	1/49	26/49	16/18
commons-codec	14	0/14	12/14	0/14	0/14	2/14	12/12
joda time	18	0/18	4/18	0/18	2/18	14/18	4/4
spojo core	1	1/1	1/1	0/1	0/1	0/1	1/1
sonar core	10	0/10	9/10	1/10	1/10	0/10	7/9
sonar plugin	6	0/6	3/6	0/6	0/6	3/6	3/3
jbehave core	42	2/42	7/42	2/42	9/42	33/42	7/7
shindig-java-gadgets	80	2/80	30/80	12/80	21/80	38/80	26/30
shindig-common	21	1/21	8/21	4/21	4/21	9/21	8/8
total	241	9	92	24	38	125	84/92

Table 4.1: The number of source independent , purely resilient and stretchable catch blocks found with short-circuit testing. Our approach provides developers with new insights on the resilience of their software.

solution, it does not improve resilience. It does actually the opposite: fewer exceptions can be caught.

The other solution is to transform the catch blocks so that they catch more exceptions than before. This is what we call "catch stretching": replacing the type of the caught exceptions. For instance, replacing `catch(FileNotFoundException e)` by `catch(IOException e)`. The extreme of catch stretching is to parametrize the catch with the most generic type of exceptions (e.g. `Throwable` in Java, `Exception` in .NET).

In other words, in situations where unanticipated exceptions appear, the control flow would be broken, and the program would likely crash. This is what we consider as "incorrect". After catch stretching, the program would not crash, and this is what we consider as "better".

Let us now examine why this simple transformation catch is meaningful with respect to unanticipated exceptions.

We claim that *all source-independent catch blocks are candidates to be stretched.* This encompasses purely-resilient try-catch blocks as well since by construction they are also source-independent (see Section 4.2.1). The reasoning is as follows.

4.2.4.2 Catch Stretching Under Short-Circuit Testing

By stretching source independent catch-blocks, the result is equivalent under short-circuit testing. In the original case, injected exceptions are of type X and caught by `catch(X e)`. In the stretched case, injected exceptions are of generic type *Exception* and caught by `catch(Exception e)`. In both cases, the input state of the try block is the same (as set by the test case), and the input state of the catch block is the same (since no code has been executed in the try block due to fault injection). Consequently, the output state of the try-catch is exactly the same. Under short-circuit testing, catch stretching yields strictly equivalent results.

4.2.4.3 Catch Stretching and Test Suite Specification

Let us now consider a standard run of the test suite and a source-independent try-catch. In standard mode, with the original code, there are two cases: either all the exceptions thrown in the try block under consideration are caught by the catch (case A), or at least one exception traverses the try block without being caught because it is of an uncaught type (case B). In both cases, we refer to exceptions normally triggered by the test suite, not injected ones.

In the first case, catch stretching does not change the behavior of the application under test: all exceptions that were caught in this catch block in the original version are still caught in the stretched catch block. In other words, the stretched catch is still correct according to the specification. And it is able to catch many more unanticipated exceptions: it corresponds to our definition of resilience. On those source-independent try-catch of case (A), *catch stretching improves the resilience of the application.*

We now study the second case (case B): there is at least one test case in which the try-catch x under analysis is traversed by an uncaught exception. There are again two possibilities: this uncaught exception bubbles to the test case, which expects the exception (it specifies that an exception must be thrown and asserts that it is actually thrown). If this happens, we don't apply catch stretching. Indeed, it is specified that the exception must bubble, and to respect the specifications we must not modify the try-catch behavior. The other possibility is that the uncaught exception in try-catch x is caught by another try-catch block y later in the stack. When stretching try-catch x, one replaces the recovery code executed by try-catch y by executing the recovery code of try-catch x. However, it may happen that the recovery code of x is different from the recovery code of y, and that consequently, the test case that was passing with the execution of the catch of y (the original mode) fails with the execution of the catch x.

To overcome this issue, we propose to again use the test suite as the correctness oracle. For source-independent try-catch blocks of case B, one stretches the catch to "Exception", one then runs the test suite, and if all tests still pass, we keep the stretched version. As for case A, the stretching enables software to handle more unanticipated exceptions while remaining correct with respect to the specification. Stretching source-independent try-catch blocks of both case A and case B improves resilience.

4.2.4.4 Summary

To sum up, improving software resilience with catch stretching consists of: First, stretching all source-independent try-catch blocks of case A. Second, for each source-independent try-catch blocks of case B, running the test suite after stretching to check that the transformation has produced correct code according to the specification. Third, running the test suite with all stretched catch blocks to check whether there is no strange interplay between all exceptions.

We will show in Section 4.3.3 that most (91%) of source-independent try-catch blocks can be safely stretched according to the specification.

4.3 Evaluation

We have presented two exception contracts: pure resilience and source independence (Section 4.2.1). We now evaluate those contracts from an empirical point of view. Can we find real world try-catch blocks for which the corresponding test suite enables us to prove their source independence? Their pure resilience capability? Or to prove that they violate of these exception contracts?

4.3.1 Dataset

We analyze the specification of error-handling in the test suites of 9 Java open-source projects: Apache commons-lang, Apache commons-code, joda-time, Spojo core, Sonar core, Sonar Plugin, JBehave Core, Shindig Java Gadgets and Shinding Common. The selection criteria are as follows. First, the test suite has to be in the top 50 of most tested exceptions according to the SonarSource Nemo ranking[5]. SonarSource is the organization behind the software quality assessment platform "Sonar". The Nemo platform is their show case, where open-source software is continuously tested and analyzed. Second, the test suite has to be runnable within low overhead in terms of dependencies and execution requirements.

The line coverage of the test suites under study has a median of 81%, a minimum of 50% and a maximum of 94%. This dataset contains a total of 767 catch blocks.

4.3.2 Relevance of Contracts

The experimental protocol is as follows. We run the short-circuit testing algorithm described in Section 4.2.2 on the 9 reference test suites described before. As seen in Section 4.2.2, short-circuit testing runs the test suite n times, where n is the number of executed catch blocks in the application. In total, we have thus 241 executions over the 9 test suites of our dataset.

Table 4.1 presents the results of this experiment. For each project of the dataset and its associated test suite, it gives the number of executed catch blocks during the test suite execution, purely resilient try-catch blocks, source-independent try-catch blocks, and the number of try-catch blocks for which runtime information is not sufficient to assess the truthfulness of our two exception contracts.

4.3.2.1 Source Independence

Our approach is able to demonstrate that 92 try-catch blocks (sum of the fourth column of Table 4.1) are source-independent (to the extent of the testing data). This is worth noticing that with no explicit ways for specifying them and no tool support for verifying them, some developers still write catch blocks satisfying this contract. This shows that our contracts are not purely theoretical: they reflect properties of error-handling code that can be found in real software.

Beyond this, the developers not only write some source-independent catch blocks, they also write test suites that provide enough information to decide with short-circuit testing whether the catch is source-independent or not.

Our approach also identifies 24 try-catch blocks that are source-dependent, i.e. that violate the source-independence predicate. Our approach makes the developers aware that some catch blocks are not independent of the source of the exception: *the catch block implicitly assumes a state resulting from the execution of several statements at the beginning of the try.* Within the development process, this is a warning. The developers can then fix the try or the catch block if they think that this catch block should be source independent or choose to keep them source-dependent, in total awareness. It is out of the scope of this work to automatically refactor source-dependent try-catch blocks as source-independent.

For instance, a source-dependent catch block of the test suite of sonar-core is shown in Listing 4.5. Here the "key" statement is the *if (started == false)* (line 6). Indeed, if the call to *super.start()* throws an exception before the variable *started* is set to true (*started = true* line 15), an exception is thrown (line 7). On the contrary, if the same DatabaseException occurs after line 15, the catch block applies some recovery by setting default value (*setEntityManagerFactory*).

[5]See http://nemo.sonarsource.org

```
1  public class MemoryDatabaseColector extends AbstractDatabaseColector {
2     public void start(){
3        try{
4           super.start(); // code below
5        }catch (DatabaseException ex) {
6           if (started==false) // this is the source−dependence
7              throw ex;
8           setEntityManagerFactory();
9        }}}
10
11   public void start(){
12       ...
13       // depending on the execution of the following statement
14       // the catch block of the caller has a different behavior
15       started = true;
16       ...}}
```

Listing 4.5: A Source-Dependent Try-Catch Found in Sonar-core using Short Circuit Testing.

Often, source-dependent catch blocks contain if/then constructs. To sum-up, short-circuit testing catches assumptions made by the developers, and uncover causality effects between the code executed within the try block and the code of the catch block.

Finally, our approach highlights that, for 24 catch blocks (fifth column of Table 4.1), there are not enough tests to decide whether the source-independence contract holds. This signals to the developers that the test suite is not good enough with respect to assessing this contract. This knowledge is directly actionable: for assessing the contracts, the developer has to write new tests or refactor existing ones. In particular, as discussed above, if the same test case executes several times the same catch block, this may introduce noise to validate the contract or to prove its violation. In this case, the refactoring consists of splitting the test case so that the try/catch block under investigation is executed only once.

4.3.2.2 Pure Resilience

We now examine the pure-resilience contracts. In our experiment, we have found 9 purely resilient try-catch blocks in our dataset. The distribution by application is shown in the third column of Table 4.1.

Listing 4.6 shows a purely resilient try-catch block found in project spojo-core using short-circuit testing. The code has been slightly modified for sake of readability. The task of the try-catch block is to return an instantiable Collection class which is compatible with the class of a prototype object. The plan A consists of checking that the class of the prototype object has an accessible constructor (simply by calling `getDeclaredConstructor`). If there is no such constructor, the method call throws an exception. In this case, the catch block comes to the rescue and chooses from a list of known instantiable collection classes one that is compatible with the type of the prototype object. According to the test suite, the try-catch is purely resilient: always executing plan B yields passing test cases.

The pure resilience is much stronger than the source independence contract. While the former states that the catch has the same behavior wherever the exception comes from, the latter states that the correctness as specified by the test suite is not impacted in presence of unanticipated

```
1  // task of try−catch:
2  // given a prototype object
3  Class clazz = prototype.getClass();
4  // return a Collection class that has an accessible constructor
5  // which is compatible with the prototype's class
6  try {
7      // plan A: returns the prototype's class if a constructor exists
8      prototype.getDeclaredConstructor();
9      return clazz;
10 } catch (NoSuchMethodException e) {
11     // plan B: returns a known instantiable collection
12     // which is compatible with the prototype's class
13     if (LinkedList.class.isAssignableFrom(clazz)) {
14         return LinkedList.class;
15     } else if (List.class.isAssignableFrom(clazz)) {
16         return ArrayList.class;
17     } else if (SortedSet.class.isAssignableFrom(clazz)) {
18         return TreeSet.class;
19     } else {
20         return LinkedHashSet.class;
21     }
22 }
```

Listing 4.6: A Purely-Resilient Try-Catch Found in spojo-core (see SpojoUtils.java)

exceptions. Consequently, it is normal to observe far fewer try-catch blocks satisfying the pure resilience contract compared to the source-independent contract. Despite the strength of the contract, this contract also covers a reality: perfect alternatives, ideal plans B exist in real code.

One also sees that there are some try-catch blocks for which there is not enough execution data to assess whether they are purely resilient or not. This happens when a try-catch is only executed in white try-catch usages and in no pink try-catch usage. By short-circuiting the white try-catch usages (those with internally caught exceptions), one proves it source-independence, but we also need to short-circuit a nominal pink usage of this try-catch to assess that plan B (of the catch block) works instead of plan A (of the try block). This fact is surprising: this shows that some try-catch blocks are only specified in error mode (where exceptions are thrown) and not in nominal mode (with the try completing with no thrown exception). This also increases the awareness of the developers: for those catch blocks, test cases should be written to specify the nominal usage.

4.3.3 Catch Stretching

We look at whether, among the 92 source-independent try-catch blocks of our dataset, we can find stretchable ones (stretchable in the sense of Section 4.2.4, i.e. for which the caught exception can be set to "Exception"). We use source code transformation and the algorithm described in Section 4.2.4.

The last column of Table 4.1 gives the number of stretchable try-catch blocks out of the number of source-independent try-catch blocks. For instance, in commons-lang, we have found 18 candidates source-independent try-catch blocks. Sixteen (16/18) of them can be safely stretched:

all test cases pass after stretching.

Table 4.1 indicates two results. First, most (91%) of the source-independent try-catch blocks can be stretched to catch all exceptions. In this case, the resulting transformed code is able to catch more unanticipated exceptions while remaining correct with respect to the specification.

Second, there are also try-catch blocks for which catch stretching does not work. As explained in Section 4.2.4, this corresponds to the the case where the stretching results in hiding correct recovery code (w.r.t. to the specification), with new one (the code of the stretched catch) that proves unable to recover from a traversing exception.

In our dataset, we encounter all cases discussed in Section 4.2.4. For instance in joda-time, all four source-independent try-catch blocks represent are never traversed by an exception – case A of Section 4.2.4.3. (for instance the one at line 560 of class ZoneInfoCompiler). We have shown that analytically, they can safely be stretched. We have run the test suite after stretching, all tests pass.

We have observed the two variations of case B (try-catch blocks traversed by exceptions in the original code). For instance, in sonar-core, by stretching a NonUniqueResultException catch to the most generic exception type, an IllegalStateException is caught. However, this is an incorrect transformation that results in one failing test case.

Finally, we discuss the last and most interesting case. In commons-lang, the try-catch at line 826 of class ClassUtils can only catch a ClassNotFoundException but is traversed by a NullPointerException during the execution of the test ClassUtilsTest.testGetClassInvalidArguments. By stretching ClassNotFoundException to the most generic exception type, the NullPointerException is caught: the catch block execution replaces another catch block further up in the stack. Although the stretching modifies the test case execution, the test suite passes, meaning that the stretching is correct with respect to the test suite.

4.3.4 Summary

To sum up, this empirical evaluation has shown that the short-circuit testing approach of exception contracts enables developers to increase their knowledge on their software"s behavior. First, it indicates source-independent and purely resilient try-catch blocks. This knowledge is actionable: those catch blocks can be safely stretched to catch any type of exceptions. Second, it indicates source-dependent try-catch blocks. This knowledge is actionable: it says that the error-handling should be refactored so as to resist unanticipated errors. Third, it indicates "unknown" try-catch blocks. This knowledge is actionable: it says that the test suite should be extended and/or refactored to support automated analysis of exception-handling.

4.4 Threats to Validity

4.4.1 Injection Location

In short-circuit testing we inject worst-case exceptions (see Section 4.2.2), where "worst-case" means that we inject exceptions at the beginning of the try block. Thus, the injected exception discards the whole code of the try block.

Another possibility would be to inject exceptions at different locations in the try block (for instance, before line 1 of the try block, after line 1, after line 2, etc.). Let us call this algorithm "fine-grain injection". It would have the following property. First, it would take longer; instead of executing each test once with worst-case injection, this algorithm would execute each test for each possible injection location. Second, if the satisfaction or violation of the contracts is undecidable with short-circuit testing, it would still be undecidable with fine-grain injection

```
1   boolean a = true;
2   try {
3       // injection location #1
4       a = foo();//returns false
5       // injection location #2
6       a = bar();//returns true
7       // injection location #3
8       return myMethod();//exception thrown
9   } catch (Exception e) {
10      if (a){
11          return 0;
12      }else{
13          return -1;
14      }
15  }
```

Listing 4.7: An example of try-catch block categorized differently by worst-case exception injection (short-circuit testing) and fine-grain exception injection

because the verifiability of contracts depends on the test suite coverage of try-catch only and not on the location (see Section 4.2.3). Also, this algorithm could not invalidate the source dependence of catch blocks (if a test case fails with short-circuit, it would still fail with fine-grain injection which encompasses the worst case). The most interesting property of fine-grain injection is that it could show that some catch blocks that are characterized as source independent by short-circuit testing are actually source dependent. Let us show that it is theoretically possible yet unlikely.

Let us consider the code in Listing 4.7 and a test case which asserts that this method returns 0 in a specific error scenario. In this error scenario, the exception comes from the call to `myMethod()`. In this case, when entering the catch block, `a=true` because the last assigment to `a` is `a=bar()` and `bar` returns `true`. Consequently the catch block returns `null` and the test passes. With short-circuit testing an exception is thrown at *injection location #1* when `a` is still equals to `true` (assigned just before the try block). As a result the catch block still returns `null` as expected by the test, which passes. The behavior of the catch block is the same as the expected one, so short-circuit testing assesses that this catch block is source independent. The fine-grain injection algorithm would identify 2 additionnal injection locations *injection location #2 and #3*. It would execute the test 3 times, each time with injection at a different location. In the first run, it works just as short-circuit testing (injection in *injection location #1*). In the second run an exception is injected at *injection location #2*, thus `a = false` because the last assigment to `a` is `a=foo()` and `foo` returns `false`. Consequently the catch block returns -1 instead of 0. The behavior of the catch is different under injection, the test case fails (-1 returned instead of 0). Because of a failing test case under injection, the catch block under test would be proven source dependent (as said by Predicate #2 in Section 4.2.3.2).

This example shows that there may be a sequence of program states that are either recoverable or unrecoverable. The state switching occurs not only between the statements of the try block but also within the code executed in the called methods. Consequently, a meaningful fine-grain injection would not only happen at each statement of the try under analysis but also between the statements of the called methods, in a recursive manner. In other words, for worst-case injection, there is only one injection required for reasoning on the resilience, in fine grain injection, there is a

myriad of injected exceptions for the same try block (it is common to have one thousand or more recursively executed statements within the same try block execution[6]). Hence, we recommend worst-case injection, which is sound for the detection of contract violations and has a predictable and affordable computational cost (one test suite run by analyzed try-catch).

The same reasoning applies to single-statement try blocks (a try block with only one line of code). It is not equivalent to receive a thrown exception from this single statement and to inject a worst-case exception at the beginning of the try block. Within the recursively called methods and executed statements, there may still be a sequence of recoverable and unrecoverable states. Short-circuit testing of single-statement try blocks ensures that the catch block makes no assumption on what is internally done during the execution of the statement. Indeed, the real-world code of Listing 4.5 shows an example of a single-statement try block, which is identified as source-dependent thanks to short circuit testing.

4.4.2 Internal Validity

The internal validity of our experiment is threatened if the test case behavior we observe is not causally connected to the exception that are thrown. The worst case would be that exceptions are randomly thrown by the virtual machine with no causal connection with the software execution under study. Since we use mature deterministic software (a Java Virtual Machine) and a deterministic dataset, this is unlikely that spurious exceptions mix up our results.

4.4.3 External Validity

Finally, concerning the external validity, we have shown that in open-source Java software, there exists source-dependent, source-independent and purely-resilient try-catch blocks. This may be due to the programming language or the dataset. We believe that it is unlikely since our contracts relate to language and domain-independent concepts (contracts, application state, recovery). Future work on this topic will strengthen those findings.

4.5 Conclusion

In this chapter, we have explored the concept of software resilience against unanticipated exceptions. We have formalized two resilience oracles: source-independence and pure-resilience. We have devised an algorithm, called short-circuit testing, to verify them. Finally, we have proposed a source code transformation called "catch stretching" that improves the ability of the application under analysis to handle unanticipated exceptions. Our empirical evaluation on 9 open-source applications shows that those contracts characterize the exceptional behavior of real code: there exists try-catch blocks that satisfy and violate the contracts.

[6]Note that we do not take in consideration that an exception may happen within a call to the standard library. In theory, all intermediate states during a library call (and inside a native call) are also candidate to exception injection, which is really expensive to compute.

Chapter 5

Conclusion and Perspectives

Contents

5.1 Beyond Three Contributions

In this thesis, we have presented three contributions made over the last years around the concept of oracle.

In Chapter 2, we have presented a novel statistical oracle designed for detected missing method calls. The oracle is based on regularities observed over large amounts of code. The oracle embeds a very precise definition of programming context in order to achieve a low rate of false positives. In Chapter 3, the considered oracles are classical test suites, used in the context of automatic program repair. We have described a system for automatic repair of Java programs called Nopol. Nopol targets the class of faults related to buggy conditional statements (buggy if conditions, missing pre-conditions). The key insight behind Nopol is to transform the test suite oracles (aka assertions), into micro-oracles that specify the expected behavior of conditional statements in a surgical manner, for a single condition expression. In Chapter 4, we have presented two novel generic oracles for exception handling. Those oracles assess a kind of resilience against unhandled exceptions. They are generic in the sense that they are applicable for any application domain, in any implementation technology. They are in the line of classical generic oracles such as the expected absence of dirty crashes.

Besides those three contributions highlighted in this document, I have done other work related to oracles. In particular, I have worked in the field of fault localization. An oracle tells that there is a bug after the execution of a piece of software, but does not tell where the bug is. Fault localization consists of extracting more detailed diagnosis information from the short violation statement issued by the oracle. For instance, if the incorrect behavior is detected after the execution of 100 000 lines of code spread over 2000 methods, fault localization consists of ranking

those lines or methods according to their likelihood of being the root cause of the bug. To this extent, fault localization can be stated as a a recommendation system problem: it transforms an oracle violation in a ranked list of suspicious programming elements. In this context, we have addressed two important problems. First, in mainstream automated testing environments (such as Junit in Java), oracle violations are implemented as exceptions. This means that when an oracle is violated, the rest of the test case, coming after the violated assertion, is discarded. This is a waste of potentially useful diagnosis information. We have addressed this problem in [99], where we propose a technique, called test case purification, that blends code transformation and slicing in order to extract more diagnosis information from test cases. Also, we have presented a statistical machine learning technique for combining different spectrum-based fault localization metrics in order to boost the effectiveness of fault localization [98].

The second work related to oracles is in the context of software engineering for mobile applications [5, 6]. In mobile applications, and especially on the Android platform, there is a permission system that provides end-users with some kind of protection with respect to security and privacy. For instance, when users install new applications on their mobile devices, those new applications cannot send data over the internet by default. This prevents spyware to send all your private information to a remote server. Applications have to explicitly ask for permission before using any sensitive resource such as the Internet. However, the permission system never checks that the application does not ask for too many permissions with respect to what it does. We have proposed a novel oracle for this particular problem [5, 6]. This oracle is based on static analysis. We have devised two static analyses, one for the Android framework and one for the end user application code, to see whether there is a mismatch between the required permissions and the used ones.

Finally, we have also started to work on crash reproduction. A crash is the violation of the generic oracle stating that the system should stay up and running. Crash reproduction is the discipline of off-line reproducing crashes that have happened in the field, in production. The motivation behind crash reproduction is that it is an essential step before understanding the chain of events that leads to the crash, and consequently, a step before taking the appropriate correction action such as writing a code patch. Stated this way, crash reproduction is close to fault localization presented above: both aim at transforming an oracle violation into a more complete diagnostic. In [100], we have proposed to mutate existing test cases in order to reproduce production crashes where the reproducibility criterion is having the same stack trace. The insight behind this work is that test suites encode standard behaviors that are expected in production. Since many production crashes are located in the behavioral neighborhood of production usages, it is meaningful to explore the behavioral neighborhood of test cases in order to find the crash there. The test case mutations are indeed a kind of search in the behavioral neighborhood of normal specified usages.

This concept of "behavioral neighborhood" is indeed a cross-cutting concepts of my work. We can even revisit this thesis under its light. In Chapter 2 on missing method calls, the similarity relations defined over type-usages indeed defines a behavioral neighborhood: type-usages that are similar reflect close behaviors because they involve the same method calls. A violation of the statistical oracle happens when a type-usage is in the neighborhood of many other points, while not being close enough. Seen in this perspective, it means that the statistical oracle expects a specific structure in the behavioral space under consideration. Points in this space are expected to be highly clustered, where each cluster is far away from each other. It loosely resembles to galaxies that are clusters of stars, separated by enormous distances of interstellar void.

In Chapter 3 on micro-oracles and Nopol, angelic value mining is a technique consisting of slightly changing the execution of programs by forcefully changing the execution of conditionals. It can also be seen as a technique for exploring the behavioral neighborhood of programs, because

a execution trace after angelic modification is often close to the original trace. This is very effective for repair, because program repair is indeed founded on a neighborhood assumption: the patch is expected to be small compared to the original program, meaning that the patched program has to be in the behavioral neighborhood of the existing buggy program.

In Chapter 4, our exception contracts and short circuit testing algorithm can be reframed in the following way: for whatever state in the neighborhood of buggy states that are specified by the test suite, the error-handling code that is in the catch block should be effective. Again, we reason on the expected behavior (the error should be handled) based on a behavioral neighborhood (a close error state).

Now, if we step back, the key to the success of those approaches based on behavioral neighborhood is twofold. First, it requires one to define in a observation space (aka as feature space) that reflects the correctness properties considered by the oracle. Second, within that space, it requires a domain-specific notion of distance that is sound and effective. This is what has been done in all three contributions presented in this manuscript.

5.2 Perspectives

We now present the perspectives of this work. It is the research agenda I will follow in the mid-term (say 3-5 years). It is a natural follow-up of my work on automatic diagnosis and repair, with a focus on repair in production against unanticipated failures.

5.2.1 Recovery against Unanticipated Failures

First, I want to revisit error-recovery, I want to make three disruptive innovations in that field: a) apply specification mining to error-recovery in order to learn recovery post-conditions; b) consider error-recovery as a special case of program synthesis, in order to automatically synthesize optimal error-recovery code; c) state error-recovery as a search-based, trial and error process that converges to successful recovery, esp. in dynamic and changing environments.

The mainstream way to communicate and handle failures is exception handling. In software design books, entire chapters are dedicated to discussing good and bad practices on exception handling [20]. For instance, according to a preliminary study, in the Hadoop system, the error handling according to exceptions is done through 9888 catch blocks. Briefly, classical exception handling works as follows:

Failure detection:

```
1 // failure condition (1)
2 if (file==null) {
3   throw new InputException();
4 }
5 ...
```

Failure handling:

```
1 // recovery condition (2)
2 catch (InputException) {
3   // recovery strategy (3)
4   defaultText = "foo bar";
5 }
```

One sees the three main components of failure handling using exceptions: 1) the failure condition (here `file==null`) encodes a predicate on the system state whose value encodes the presence of a failure; 2) the recovery condition states when a failure can be handled (here when an exception of type `InputException` arrives at this location); 3) the recovery code repairs the system state according to the content of the catch block (here `defaultText = "foo bar"`).

Today, all those three components are manually written and hard-coded. This is a fundamental limitation. In today's open-ended systems, the kinds of failures and recovery are open-ended. For instance, the Eclipse development environment runs on at least 5 million different machines.

Can the Eclipse developers foresee all possible failures? Can they hard-code all possible failure conditions, recovery conditions and recovery strategies? No.

To overcome this problem, my vision is as follows: first failure recovery strategies should be reused even in unanticipated cases (as opposed to only triggered in manually specified cases); second failure detection recovery has to be automatically adapted: when recovery fails, alternative solutions in a recovery space are explored, for instance by taking into account new portions of the system state (new variables); third, recovery should be considered from a declarative angle, using recovery post-conditions and oracles. Such an approach will result in software that is much more dependable, much more resilient to unanticipated problems in production.

> My future research will completely revisit the notion of error handling in particular by considering error-recovery as a search-based, trial and error process that converges to successful recovery. This will pave the way to useful recovery even in dynamic and unpredictable environments. This requires powerful oracles at runtime, esp. recovery oracles, which, once present, allow for recovery code to be synthesized.

5.2.1.1 Automatic learning of recovery post-conditions

There is a need for automatically mining "recovery post-conditions". A recovery post-condition is an oracle on the system state correctness after recovery. The following example illustrates this idea with a classical try-catch example enriched with a recovery post-condition.

```
1 try {
2   ...
3 } catch (failureRecoverable(exception,
        context)) {
4   recover(defaultText,output);
5 }
6 assertTrue(recoveryPostCondition());
```

The recovery post-condition is necessary for differentiating successful from failed recovery. Having recovery post-conditions is of primary importance for recovery synthesis, an idea that will be presented in Section 5.2.1.2, and for search-based recovery, that will be discussed in Section 5.2.2.

This vision can only be realized by rooting this work in the literature of specification mining whether static or dynamic [27, 1]. I plan to design a theoretical and algorithmic framework for mining and learning recovery post-conditions, which would provide oracles on the validity of the system state after recovery.

5.2.1.2 Synthesis of failure-recovery code

Now, I want to question the classical, manually coded recovery code (e.g. the catch blocks). I claim that it is sub-optimal because of the missing big picture of error-handling that the developer does not have. This claim is backed up by our preliminary work in this domain [18].

Let us now imagine that we have good recovery post-conditions. We can then start considering recovery as a synthesis problem. The current research on code synthesis has shown that synthesis is doable only if a careful qualification of the synthesis space is done beforehand [37]. In our current work on automatic repair, we use one such kind of well-delimited synthesis problem and a

boolean input-output based specification [23]. For failure-recovery code to become automatically adaptable, we also need to qualify recovery in order to enable synthesis of failure handling code.

As shown in our preliminary work [69], one kind of failure-recovery code is to return an appropriate best-effort value. We plan to encode this as a synthesis problem as follows. The returned object is the result of an expression *exp* such that

$$\forall_{l,m,n} \quad exp(C_{l,n}) = R_{l,n} \tag{5.1}$$

where $C_{l,n}$ is the set of variable values at location l during the n-th previous execution of the catch block and $R_{l,n}$ is the expected returned value that enables the software to recover from the n-th previous failure at this place. Subsequently, *exp* can be synthesized using a Satisfiability Modulo Theory (SMT) constraint solver [23].

To experiment with this idea, my insight is that software test suites already contain scenarios that simulate a failure and assess whether the recovery handlers correctly handle it (in our preliminary experiments, this accounts up to 25% for Apache Commons Lang). We envision a system that learns the common patterns between those specified pieces of recovery code. To validate the synthesis techniques, I propose a large-scale experimental protocol as follows: first, one removes the existing failure-handling code and second, one synthesizes the missing recovery code.

To sum up, this research will result in a synthesis framework dedicated to and optimized for synthesizing recovery code. This framework would learn the patterns in existing recovery code and enables the replacement of the manually-written exception handlers by synthesized exception handlers.

5.2.2 Automatic exploration of the recovery space

With explicit recovery post-conditions and oracles, it becomes possible to start automatically exploring the recovery space of a software system. The idea is that recovery becomes a search-based, trial-and-error process, where recovery attempts are performed one after the other until one valid recovery is found. The recovery post-conditions and oracles can either be explicit or be mined, resulting from the techniques envisioned in Section 5.2.1.1.

Basically, each time a failure is detected in production, the following high-level algorithm would be executed:

1. update the failure detection conditions that are further up in the stack in order to detect the failure earlier next time (automatic improvement)
2. select one satisfied failure recovery condition, if any; if multiple recovery exist, then select the least executed one (advantage for novelty)
3. synthesize recovery code (exploration of the recovery space)
4. assess successful failure recovery based on mined recovery post-condition
5. if failure persists, go to 2 (automatic improvement)

Step #1 refers to the reified function `failureDetected()` of our above example. The most important part of this algorithm is the exploration of the recovery space that happens in steps #2 and #3. In step #2, when the system identifies several applicable recovery strategies, it takes care of applying them according to the execution history. If two failure recovery conditions are evaluated to true, it means that two recovery strategies are available. If the first one has already been executed and not the second one, the system selects the latter. This maximizes novelty discovery and knowledge exploration (whether the recovery is satisfying or not). The envisioned

system always maintains a record of all tried recovery in order to always apply the novel ones, the least executed ones, so as to keep exploring the recovery space. In step #3, recovery code is synthesized. The synthesis takes care of synthesizing a solution that satisfies both the pre- and post-condition, but also that is novel and different from the previously synthesized and executed recovery strategies. Also, the mined recovery post-conditions steer the re-execution of step #3 if appropriate.

To sum up, due to the variety of production conditions, not all failure cases can be foreseen by the designer and not all applied recovery strategies succeed. I want to introduce support for automatically switching from one recovery to another at runtime, together with the exploration of different recovery strategies in a recovery space. This is the solution to constantly maintain a rich pool of possible recoveries to be executed upon unanticipated failures. We will consider "novelty search" algorithms [52] for succeeding in this task.

5.2.3 Constant Assessment of Recovery Capabilities with Injected Perturbations

The second area of future research is the injection of perturbations in production. The motivation is to go beyond automatic repair: automatic repair is reactive, because one waits for bugs to happen. On the contrary, production perturbation is proactive, one triggers erroneous states and inputs to study in advance their impact.

Let us consider a concrete example. According to our statistics on the Internet, the most common failures in Java software are null dereferences ("null pointer exceptions"). Null dereferences cause desktop, server and mobile applications to crash on a daily basis. Now, let's consider a software system that embeds a monitoring system, as well as search-based recovery for null dereferences. As such, the system has already overcome 15 unhandled null dereferences. Now, let us imagine that the system is augmented with a module that selectively injects null values in memory. This system would inject x (say 3) null values per day so as to 1) assess that the system does not crash upon null dereferences, and 2) pro-actively synthesize and validate new null-dereference recovery.

On the hardware side of production systems, this idea is already applied. For instance, datacenters are regularly subject to power cut so as to assess whether the alternative sources of power are up and running. I will explore this disruptive idea on the software side, to constantly assess whether the software recovery code well handles software failures.

This can be seen as an application of the scientific method to failure recovery. The scientific method states that all hypotheses must be experimentally validated using falsification experiments. The same idea has recently been formulated in the Silicon Valley as "chaos engineering"[1].

A recovery capability is also a hypothesis: if an event of type x happens, the system is able to survive. By injecting an event x and assessing successful failure-recovery, one ensures the truthfulness of the recovery hypothesis in production. In biology, "hormesis" refers to the positive response of biological systems (e.g. a cell) in response to a stressor. This idea of fault injection in production can be seen as the exploration of the notion of hormesis in the domain of software systems.

This is different and complementary from performing failure analysis in testing phase. Failure analysis during testing enables to validate recovery against specific, well formed, small failures. However, production systems are too big and too interconnected to be reproduced in a controlled testing environment. This results in unanticipatable situations and behaviors. This idea aims at tackling those unanticipatable, untestable failures that necessarily happen in production.

[1] http://principlesofchaos.org/

5.2.3.1 Characterization of failures that can be injected in production

By injecting failures, there are potential losses due to the propagation of failures to critical components. Examples of loss include loss of money, loss of reputation, loss of lives. In all cases, the losses have to be qualified and a trade-off has to be found between the losses due to the injected failure and the knowledge gained from observing the injected controlled failure in a real setting, where knowledge consists for example of validated recovery, new mined recovery conditions, trigger of better monitoring. There is a need for creating a taxonomy of failures that can be injected in production and the associated potential losses.

To characterize the failures that can be injected in production, one would follow an empirical methodology, as we have successfully done before to characterize documentation and bug fixes [68, 58]. We will apply the grounded theory method, which consists of systematically building a theory of some phenomena based on the analysis of carefully sampled empirical data. This approach has two main phases. The first step consists of collecting a pool of production failures, the second step consists of systematically analyzing them. For each failure, we will answer three questions: 1) what is the loss? 2) is it due to the environment, the execution platform, an incorrect piece of code? 3) can the failure be meaningfully injected so as to perturb the system?

To get started on this research topic, I would first focus on Java software systems (a technological domain where we have a lot of experience). For gathering project-independent knowledge, we would analyze the production logs of production systems from my institution (e.g. Nuxeo and Alfresco respectively installed at the University of Lille and Inria). We will follow the best methodological guidelines for software engineering empirical studies [49].

5.2.3.2 Design of an in-vivo perturbation system

To explore the disruptive idea of perturbing production systems with injected failures, one has to create "in-vivo perturbation systems", where "in-vivo" means "in production". They are responsible for injecting appropriate failures at the relevant places, according to the benefit and loss characterization resulting from the work envisioned in 5.2.3.1. The perturbation would have to be done in a very controlled way, in order to mitigate the potential deleterious propagation of injected failures. Let us consider again the example of null dereferences and analyze the following listing. It is a short example of how this system would work:

```
1  // in the HTTP server of the Hadoop control system
2  void processRequest() {
3
4    // failure activate predicate
5    if (perturbationModule.triggersNullFailure(localcontext)) {
6      perturbationModule.saveDatabaseSession();
7      databaseSession = null; // perturbation injection
8    }
9
10   ... // actual code of processRequest(), using databaseSession
11   perturbationModule.restoreDatabaseSession(); // failure mitigation code
12
13 }
```

When a request arrives, a call is made to the perturbation module to know whether a perturbation is appropriate at this place and this point in time. If the perturbation module agrees, the environment is prepared for the injection (failure injection is always done in a controlled

manner). Then the variable `databaseSession` is set to `null`. This results in a null dereference failure later during the execution of `processRequest`. In this context, the benefit of the injected failure is to assess that the system is capable of resisting to nullified database session. The cost of this assessment is making the current request fail. At the end of the execution of `processRequest`, the database session is restored in order to mitigate the failure. Our key idea is to inject the perturbation and failure mitigation code using metaprogramming. In addition to monitoring and recovery points, the envisioned instrumentation layer will automatically inject perturbation points in the code of the system under study.

The injection of perturbation and failure mitigation code can be achieved with the state-of-the-art metaprogramming techniques. Failure propagation mitigation will be based on the latest advances on isolation and containment [80] as well as checkpoint, rollback and replay mechanisms [3]. To sum up, by combining metaprogramming, isolation, restart and replay, we will design the very first prototype of a perturbation library for production systems.

5.3 Last Words

Writing this manuscript was the opportunity to look back over what I have explored over the last years. If this were to be done again, I would do it, because it was a real excitement to explore those unknown lands, and I feel like having performed science in the noblest sense.

Finally, we are dwarfs standing on the shoulders of giants, our results are grains of sand on top of the immense mountain of knowledge. To that extent, I would like to thank all the authors of the excellent papers I have read, for their ideas that could be either fun, creative or useful, for the new light they shed, for the motivation they provide. All that is necessary for conducting good research and at some point in a career, to put together such a manuscript. In Ancient Greece, there was a concept for this fundamental source of inspiration and energy: it is called “Muse”.

Bibliography

[1] G. Ammons, R. Bodik, and J. Larus. "Mining specifications". In: *Proceedings of the 29th Symposium on Principles of Programming Languages (POPL)*. 2002.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. "Basic Concepts and Taxonomy of Dependable and Secure Computing". In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (2004), pp. 11–33.

[3] E. T. Barr and M. Marron. "Tardis: Affordable Time-travel Debugging in Managed Runtimes". In: *SIGPLAN Not.* 49.10 (Oct. 2014), pp. 67–82.

[4] E. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. "The Oracle Problem in Software Testing: a Survey". In: *IEEE Transactions on Software Engineering* 41.5 (2015), pp. 507–525.

[5] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. "Automatically Securing Permission-Based Software by Reducing the Attack Surface: An Application to Android". In: *Proceedings of the 27th IEEE/ACM International Conference On Automated Software Engineering*. 2012, pp. 274–277.

[6] A. Bartel, J. Klein, M. Monperrus, and Y. Le Traon. "Static Analysis for Extracting Permission Checks of a Large Scale Framework: The Challenges And Solutions for Analyzing Android". In: *IEEE Transactions on Software Engineering* (2014).

[7] G. Baxter, M. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. Tempero. "Understanding the shape of Java software". In: *Proceedings of Object-oriented Programming Systems Languages and Applications (OOPSLA)*. ACM, 2006.

[8] B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[9] J. M. Bieman, D. Dreilinger, and L. Lin. "Using fault injection to increase software test coverage". In: *Seventh International Symposium on Software Reliability Engineering*. IEEE. 1996, pp. 166–174.

[10] B. Cabral and P. Marques. "Exception handling: A field study in Java and .Net". In: *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 2007, pp. 151–175.

[11] J. Campos, A. Riboira, A. Perez, and R. Abreu. "GZoltar: an eclipse plug-in for testing and debugging". In: *Proceedings of Automated Software Engineering*. 2012.

[12] S. Chandra, E. Torlak, S. Barman, and R. Bodik. "Angelic Debugging". In: *Proceeding of the International Conference on Software Engineering*. 2011, pp. 121–130.

[13] R.-Y. Chang, A. Podgurski, and J. Yang. "Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software". In: *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2007.

[14] J. Chen and R. J. Patton. *Robust model-based fault diagnosis for dynamic systems*. Vol. 3. Springer Science & Business Media, 2012.

[15] D. R. Cok. "jSMTLIB: Tutorial, validation and adapter tools for SMT-LIBv2". In: *NASA Formal Methods*. Springer, 2011, pp. 480–486.

[16] B. Cornu, E. T. Barr, L. Seinturier, and M. Monperrus. *Casper: Debugging Null Dereferences with Dynamic Causality Traces*. Tech. rep. hal-01113988. Inria Lille, 2015.

[17] B. Cornu, T. Durieux, L. Seinturier, and M. Monperrus. *NPEFix: Automatic Runtime Repair of Null Pointer Exceptions in Java*. Technical Report 1512.07423. Arxiv, 2015.

[18] B. Cornu, L. Seinturier, and M. Monperrus. "Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions". In: *Information and Software Technology* 57 (2015), pp. 66–76.

[19] C. Csallner, Y. Smaragdakis, and T. Xie. "DSD-Crasher: A hybrid analysis tool for bug finding". In: *ACM Trans. Softw. Eng. Methodol.* 17.2 (2008), pp. 1–37.

[20] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reuseable .NET Libraries*. Addison-Wesley, 2008.

[21] V. Dallmeier, C. Lindig, and A. Zeller. "Lightweight Bug Localization with AMPLE". In: *Proceedings of the Sixth sixth international symposium on Automated analysis-driven debugging* (2005), pp. 99–104.

[22] V. Debroy and W. Wong. "Using Mutation to Automatically Suggest Fixes for Faulty Programs". In: *Proceedings of the International Conference on Software Testing, Verification and Validation*. 2010, pp. 65–74.

[23] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. "Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT". In: *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis (CSTVA 2014)*. 2014.

[24] T. Durieux, M. Martinez, M. Monperrus, R. Sommerard, and J. Xuan. *Automatic Repair of Real Bugs: An Experience Report on the Defects4J Dataset*. Tech. rep. 1505.07002. Arxiv, 2015.

[25] D. Engler, D. Chen, S. Hallem, A. Chou, and B. Chelf. "Bugs as deviant behavior: A general approach to inferring errors in systems code". In: *Proceedings of Symposium on Operating Systems Principles (SOSP)*. ACM, 2001.

[26] D. Engler and K. Ashcraft. "RacerX: effective, static detection of race conditions and deadlocks". In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 237–252.

[27] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. "Dynamically Discovering Likely Program Invariants to Support Program Evolution". In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[28] M. D. Ernst, J. Cockrell, W. Griswold, and D. Notkin. "Dynamically discovering likely program invariants to support program evolution". In: *IEEE Transactions on Software Engineering* 27.2 (2001), pp. 99–123.

[29] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. "Fine-grained and Accurate Source Code Differencing". In: *Proceedings of the International Conference on Automated Software Engineering*. 2014, pp. 313–324.

[30] T. A. Foundation. *On Contributing Patches*. http://commons.apache.org/patches.html. Accessed: 2014-12-01.

[31] C. Fu, R. P. Martin, K. Nagaraja, T. D. Nguyen, B. G. Ryder, and D. Wonnacott. "Compiler-directed program-fault coverage for highly available Java internet services". In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2003.

[32] S. Ghosh and J. L. Kelly. "Bytecode fault injection for Java software". In: *Journal of Systems and Software* 81.11 (2008), pp. 2034–2043.

[33] M. Gomez, M. Martinez, M. Monperrus, and R. Rouvoy. "When App Stores Listen to the Crowd to Fight Bugs in the Wild". In: *Proceedings of the 37th International Conference on Software Engineering (ICSE), track on New Ideas and Emerging Results*. 2015.

[34] J. B. Goodenough. "Exception Handling: Issues and a Proposed Notation". In: *Commun. ACM* 18.12 (1975), pp. 683–696.

[35] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. 3rd. Addison-Wesley, 2005.

[36] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. "GenProg: a Generic Method for Automatic Software Repair". In: *IEEE Transactions on Software Engineering* 38 (2012), pp. 54–72.

[37] S. Gulwani. "Dimensions in Program Synthesis". In: *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming*. 2010, pp. 13–24.

[38] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. "Synthesis of loop-free programs". In: *ACM SIGPLAN Notices* 46.6 (2011), pp. 62–73.

[39] W. Hamscher et al. "Readings in model-based diagnosis". In: (1992).

[40] S. Hangal and M. S. Lam. "Tracking down software bugs using automatic anomaly detection". In: *Proceedings of International Conference on Software Engineering (ICSE)*. 2002, pp. 291–301.

[41] D. L. Heine and M. S. Lam. "A practical flow-sensitive and context-sensitive C and C++ memory leak detector". In: *ACM SIGPLAN Notices* 38.5 (2003), pp. 168–181.

[42] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. *A program structure for error detection and recovery*. Springer, 1974.

[43] W. Howden. "Theoretical and Empirical Studies of Program Testing". In: *IEEE Transactions on Software Engineering* SE-4.4 (1978), pp. 293–298.

[44] D. Jeffrey, N. Gupta, and R. Gupta. "Fault Localization Using Value Replacement". In: *Proceedings of the International Symposium on Software Testing and Analysis*. 2008, pp. 167–178.

[45] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. "Oracle-guided Component-based Program Synthesis". In: *Proceedings of the International Conference on Software Engineering*. Vol. 1. 2010, pp. 215–224.

[46] S. Katz and Z. Manna. "Towards Automatic Debugging of Programs". In: *Proceedings of the International Conference on Reliable Software*. 1975, pp. 143–155.

[47] D. Kim, J. Nam, J. Song, and S. Kim. "Automatic Patch Generation Learned From Human-Written Patches". In: *Proceedings of ICSE*. 2013.

[48] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. "Predicting Faults From Cached History". In: *Proceedings of the 29th International Conference on Software Engineering* (2007), pp. 489–498.

[49] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. "Preliminary guidelines for empirical research in software engineering". In: *IEEE Transactions on Software Engineering* 28.8 (2002), pp. 721–734.

[50] J.-C. Laprie. "From Dependability to Resilience". In: *Proceedings of the International Conference on Dependable Systems and Networks*. 2008.

[51] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. "A Systematic Study of Automated Program Repair: Fixing 55 Out of 105 Bugs for $8 Each". In: *Proceedings of the International Conference on Software Engineering*. 2012, pp. 3–13.

[52] J. Lehman and K. O. Stanley. "Exploiting Open-Endedness to Solve Problems Through the Search for Novelty." In: *ALIFE*. 2008, pp. 329–336.

[53] Z. Li and Y. Zhou. "PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Code". In: *Proc. of the 10th European Software Engineering Conference Held Jointly with 13th International Symposium on Foundations of Software Engineering*. 2005, pp. 306–315.

[54] B. Livshits and T. Zimmermann. "DynaMine: Finding Common Error Patterns by Mining Software Revision Histories". In: *Proceedings of the European Software Engineering Conference Held Jointly with International Symposium on Foundations of Software Engineering*. 2005.

[55] F. Long, S. Sidiroglou-Douskos, and M. C. Rinard. "Automatic Runtime Error Repair and Containment via Recovery Shepherding". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation,* 2014.

[56] S. L. Marcote and M. Monperrus. *Automatic Repair of Infinite Loops*. Tech. rep. 1504.05078. Arxiv, 2015.

[57] M. Martinez and M. Monperrus. *ASTOR: Evolutionary Automatic Software Repair for Java*. Technical Report 1410.6651. Arxiv, 2014.

[58] M. Martinez and M. Monperrus. "Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing". In: *Empirical Software Engineering* 20.1 (Sept. 2013), pp. 176–205.

[59] M. Martinez, W. Weimer, and M. Monperrus. "Do the Fix Ingredients Already Exist? An Empirical Inquiry into the Redundancy Assumptions of Program Repair Approaches". In: *Proceedings of the International Conference on Software Engineering, NIER track*. 2014.

[60] T. J. McCabe. "A complexity measure". In: *IEEE Transactions on Software Engineering* 4 (1976), pp. 308–320.

[61] S. Mechtaev, J. Yi, and A. Roychoudhury. "DirectFix: Looking for Simple Program Repairs". In: *Proceedings of the 37th International Conference on Software Engineering*. 2015.

[62] D. Mendez, B. Baudry, and M. Monperrus. "Empirical Evidence of Large-Scale Diversity in API Usage of Object-Oriented Software". In: *Proceedings of the IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2013.

[63] B. Meyer. "Applying design by contract". In: *Computer* 25.10 (1992), pp. 40–51.

[64] M. Monperrus. "A Critical Review of "Automatic Patch Generation Learned from Human-Written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair". In: *Proceedings of the International Conference on Software Engineering*. 2014, pp. 234–242.

[65] M. Monperrus. *Automatic Software Repair: a Bibliography*. Tech. rep. hal-01206501. University of Lille, 2015.

[66] M. Monperrus and B. Baudry. *Two Flavors in Automated Software Repair: Rigid Repair and Plastic Repair*. Dagstuhl Preprint, Seminar n°13061. 2013.

[67] M. Monperrus, M. Bruch, and M. Mezini. "Detecting Missing Method Calls in Object-Oriented Software". In: *Proceedings of the 24th European Conference on Object-Oriented Programming*. Springer, 2010.

[68] M. Monperrus, M. Eichberg, E. Tekes, and M. Mezini. "What Should Developers Be Aware Of? An Empirical Study on the Directives of API Documentation". In: *Empirical Software Engineering* 17.6 (2012), pp. 703–737.

[69] M. Monperrus, M. Germain De Montauzan, B. Cornu, R. Marvie, and R. Rouvoy. *Challenging Analytical Knowledge On Exception-Handling: An Empirical Study of 32 Java Software Packages*. Tech. rep. hal-01093908. Laboratoire d'Informatique Fondamentale de Lille, 2014.

[70] M. Monperrus and M. Mezini. "Detecting Missing Method Calls as Violations of the Majority Rule". In: *ACM Transactions on Software Engineering and Methodology* 22.1 (June 2013), pp. 1–25.

[71] L. de Moura and N. Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of Tools and Algorithms for the Construction and Analysis of Systems*. 2008, pp. 337–340.

[72] L. Naish, H. J. Lee, and K. Ramamohanarao. "A model for spectra-based software diagnosis". In: *ACM Transactions on software engineering and methodology (TOSEM)* 20.3 (2011), p. 11.

[73] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. "SemFix: Program Repair via Semantic Analysis". In: *Proceedings of the International Conference on Software Engineering*. 2013.

[74] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. "Graph-based mining of multiple object usage patterns". In: *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 2009, pp. 383–392.

[75] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. "Recurring bug fixes in object-oriented programs". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 2010, pp. 315–324.

[76] *Nopol GitHub Project*. http://github.com/SpoonLabs/nopol/. Accessed: 2015-06-01.

[77] K. Pan, S. Kim, and E. J. Whitehead Jr. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14.3 (2009), pp. 286–315.

[78] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier. "Spoon: A Library for Implementing Analyses and Transformations of Java Source Code". In: *Software: Practice and Experience* (2015), na.

[79] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. "Automated Fixing of Programs with Contracts". In: *IEEE Transactions on Software Engineering* 40.5 (2014), pp. 427–449.

[80] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. "Rethinking the library OS from the top down". In: *ACM SIGPLAN Notices* 46.3 (2011), pp. 291–304.

[81] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. "The Strength of Random Search on Automated Program Repair". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014, pp. 254–265.

[82] Z. Qi, F. Long, S. Achour, and M. Rinard. "An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems". In: *Proceedings of ISSTA*. 2015.

[83] F. Ricca and P. Tonella. "Detecting anomaly and failure in web applications". In: *IEEE MultiMedia* 2 (2006), pp. 44–51.

[84] M. Rinard. "Acceptability-oriented Computing". In: *ACM SIGPLAN Notices* 38.12 (2003), pp. 57–75.

[85] P. Rovner. "Extending Modula-2 to build large, integrated systems". In: *Software, IEEE* 3.6 (1986), pp. 46–57.

[86] W. Savitch. *JAVA An Introduction to Problem Solving and Programming*. Pearson, 2012.

[87] A. Sethi et al. "A Survey of Fault Localization Techniques in Computer Networks". In: *Science of Computer Programming* 53.2 (2004), pp. 165–194.

[88] S. Sinha and M. J. Harrold. "Analysis and testing of programs with exception handling constructs". In: *IEEE Transactions on Software Engineering* 26.9 (2000), pp. 849–871.

[89] S. Sinha, A. Orso, and M. J. Harrold. "Automated support for development, maintenance, and testing in the presence of implicit flow control". In: *Proceedings of the 26th International Conference on Software Engineering*. IEEE. 2004, pp. 336–345.

[90] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. "Programs, Tests, and Oracles: the Foundations of Testing Revisited". In: *Proceedings of the International Conference on Software Engineering*. 2011, pp. 391–400.

[91] K. S. Trivedi, D. S. Kim, and R. Ghosh. "Resilience in Computer Systems and Networks". In: *Proceedings of the 2009 International Conference on Computer-Aided Design*. ICCAD '09. San Jose, California: ACM, 2009, pp. 74–77.

[92] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot-a Java bytecode optimization framework". In: *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. IBM Press, 1999, p. 13.

[93] A. Wasylkowski, A. Zeller, and C. Lindig. "Detecting object usage anomalies". In: *Proceedings of European Software Engineering Conference / Foundations of Software Engineering (ESEC-FSE)*. Dubrovnik, Croatia: ACM, 2007, pp. 35–44.

[94] W. Weimer, Z. P. Fry, and S. Forrest. "Leveraging program equivalence for adaptive program repair: Models and first results". In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*. 2013, pp. 356–366.

[95] C. C. Williams and J. K. Hollingsworth. "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques". In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 466–480.

[96] J. Xuan, B. Cornu, M. Martinez, B. Baudry, L. Seinturier, and M. Monperrus. *Dynamic Analysis can be Improved with Automatic Test Suite Refactoring*. Tech. rep. 1506.01883. Arxiv, 2015.

[97] J. Xuan, M. Matias, F. DeMarco, L. Sebastian, D. Thomas, D. Le Berre, and M. Monperrus. *Nopol: Automatic Repair of Conditional Statement Bugs in Large-Scale Object-Oriented Programs*. Underreview at IEEE Transactions on Software Engineering.

[98] J. Xuan and M. Monperrus. "Learning to Combine Multiple Ranking Metrics for Fault Localization". In: *Proceedings of the 30th International Conference on Software Maintenance and Evolution*. 2014.

[99] J. Xuan and M. Monperrus. "Test Case Purification for Improving Fault Localization". In: *Proceedings of the International Symposium on the Foundations of Software Engineering*. 2014.

[100] J. Xuan, X. Xie, and M. Monperrus. "Crash Reproduction via Test Case Mutation: Let Existing Test Cases Help". In: *Proceedings of ESEC/FSE, NIER track*. 2015.

[101] J. Yang, D. Evans, D. Bhardwaj, T. Bhat, and M. Das. "Perracotta: mining temporal API rules from imperfect traces". In: *Proceedings of International Conference on Software Engineering (ICSE)*. Shanghai, China: ACM, 2006.

[102] P. Zhang and S. Elbaum. "Amplifying tests to validate exception handling code". In: *Proceedings of the International Conference on Software Engineering*. IEEE Press. 2012, pp. 595–605.

[103] X. Zhang, N. Gupta, and R. Gupta. "Locating Faults Through Automated Predicate Switching". In: *Proceedings of the 28th International Conference on Software Engineering*. 2006, pp. 272–281.