HAL open science

Automatic Analysis and Repair of Exception Bugs for Java Programs

Benoit Cornu

To cite this version:

Benoit Cornu. Automatic Analysis and Repair of Exception Bugs for Java Programs. Software Engineering [cs.SE]. Université de Lille, 2015. English. NNT : . tel-01250092

HAL Id: tel-01250092

https://theses.hal.science/tel-01250092

Submitted on 4 Jan 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Département de formation doctorale en informatique **École doctorale SPI Lille**
UFR IEEA

Automatic Analysis and Repair of Exception Bugs for Java Programs

THÈSE

présentée et soutenue publiquement le 26/11/2015

pour l'obtention du

Doctorat de l'Université Lille 1
(spécialité informatique)

par

Benoit Cornu

Composition du jury

Président :	Jean-Christophe Routier, *Université Lille 1*
Rapporteurs :	Pierre-Etienne Moreau, *Université de Lorraine* Christophe Dony, *Université de Montpellier-II*
Examinateurs :	Jean-Christophe Routier, *Université Lille 1* Earl Barr, *University College London*
Directeurs :	Lionel Seinturier, *Université Lille 1* Martin Monperrus, *Université Lille 1*

Centre de Recherche en Informatique, Signal et Automatique de Lille
UMR Lille 1/CNRS 8022 – Inria Lille - Nord Europe

Mis en page avec la classe thloria.

Abstract

The world is day by day more computerized. There is more and more software running everywhere, from personal computers to data servers, and inside most of the new popularized inventions such as connected watches or intelligent washing machines. All of those technologies use software applications to perform the services they are designed for. Unfortunately, the number of software errors grows with the number of software applications.

In isolation, software errors are often annoyances, perhaps costing one person a few hours of work when their accounting application crashes. Multiply this loss across millions of people and consider that even scientific progress is delayed or derailed by software error: in aggregate, these errors are now costly to society as a whole.

There exists two techniques to deal with those errors. **Bug fixing** consists in repairing errors. **Resilience** consists in giving an application the capability to remain functional despite the errors.

This thesis focuses on bug fixing and resilience in the context of exceptions. Exceptions are programming language constructs for handling errors. They are implemented in most mainstream programming languages and widely used in practice.

We specifically target two problems:

Problem #1: There is a lack of debug information for the bugs related to exceptions. This hinders the bug fixing process. To make bug fixing of exceptions easier, we will propose techniques to enrich the debug information. Those techniques are fully automated and provide information about the cause and the handling possibilities of exceptions.

Problem #2: There are unexpected exceptions at runtime for which there is no error-handling code. In other words, the resilience mechanisms against exceptions in the currently existing (and running) applications is insufficient. We propose resilience capabilities which correctly handle exceptions that were never foreseen at specification time neither encountered during development or testing.

In this thesis, we aim at finding solutions to those problems. We present four contributions to address the two presented problems.

In **Contribution #1**, we lies the foundation to address both problems. To improve the available information about exceptions, we present a characterization of the exceptions (expected or not, anticipated or not), and of their corresponding resilience mechanisms. We provide definitions about what is a bug when facing exceptions and what are the already-in-place corresponding resilience mechanisms. We formalize two formal resilience properties: source-independence and pure-resilience as well as an algorithm to verify them. We also present a code transformation that uses this knowledge to enhance the resilience of the application.

Contribution #2 aims at addressing the limitations of **Contribution #1**. The limitations is that there are undecidable cases, for which we lack information to characterize them in the conceptual framework of **Contribution #1**. We focus on the approaches that use the test suite as their main source of information as in the case of **Contribution #1**. In this contribution, we propose a technique to split test cases into small fragments in order to increase the

efficiency of dynamic program analysis. Applied to **Contribution #1**, this solution improves the knowledge acquired by providing more information on more cases. Applied to other dynamic analysis techniques which also use test suites, we show that it improve the quality of the results. For example, one case study presented is the use of this technique on Nopol, an automatic repair tool.

Contribution #1 and **#2** are generic, they target any kind of exceptions. In order to further contribute to bug fixing and resilience, we need to focus on specific types of exceptions. This focus enables us to exploit the knowledge we have about them and further improve bug fixing and resilience. Hence, in the rest of this thesis, we focus on a more specific kind of exception: the null pointer dereference exceptions (NullPointerException in Java).

Contribution #3 focuses on **Problem #1**, it presents an approach to make bug fixing easier by providing information about the origin of null pointer dereferences. We present an approach to automatically provide information about the origin of the null pointer dereferences which happen in production mode (i.e. those for which no efficient resilience mechanisms already exists). The information provided by our approach is evaluated w.r.t. its ability to help the bug fixing process. This contribution is evaluated other 14 real-world null dereference bugs from large-scale open-source projects.

Contribution #4 addresses **Problem #2**, we present a way to tolerate the same kind of errors as **Contribution #3**: null pointer dereference. We first use dynamic analysis to detect harmful null dereferences, skipping the non-problematic ones. Then we propose a set of strategies able to tolerate this error. We define code transformations to 1) detect harmful null dereferences at runtime; 2) allow a runtime ehavior modification to execute strategies; 3) assess the correspondance between the modified behavior and the specifications. This contribution is evaluated other 11 real-world null dereference bugs from large-scale open-source projects.

To sum up, this thesis studies the exceptions, their behaviors, the information one can gathered from them, the problems they may cause and the applicable solutions to those problems.

Contents

1 Introduction **1**

1.1 Context 1

1.2 Thesis Contribution 2

1.3 Outline 3

1.4 Publications 3

2 State of the art **5**

2.1 Definitions 5

2.1.1 Exception 5

2.1.2 Resilience 5

2.1.3 Specifications and Test Suites 6

2.2 Exception Handling 6

2.2.1 Static Analysis 7

2.2.2 Dynamic Analysis 8

2.2.3 Using Fault Injection 8

2.3 Debugging and Understanding 9

2.3.1 General Debugging 9

2.3.2 Exception Debugging 9

2.3.3 Fault Detection and Localization 11

2.4 Bug Fixing and Tolerance 12

2.4.1 Bug Fixing 12

2.4.2 Bug Tolerance 13

2.5 Resilience 14

2.6 Proposed Improvements on Exception Handling 15

2.7 Summary 16

3 Analyzing and Improving Software Resilience against Unanticipated Exceptions 17

3.1 Automatic Analysis and Improved Software Resilience 18
3.1.1 Definition of Two Contracts for Exception Handling 18
3.1.2 The Short-circuit Testing Algorithm 23
3.1.3 Resilience Predicates 24
3.1.4 Improving Software Resilience with Catch Stretching 26
3.2 Empirical Evaluation 28
3.2.1 Dataset 28
3.2.2 Relevance of Contracts 30
3.2.3 Catch Stretching 32
3.2.4 Summary 33
3.3 Discussion 33
3.3.1 Injected Exceptions 33
3.3.2 Relevance of Exception Contracts for Single-Statement Try Blocks . . . 36
3.3.3 Try Scope Decrease 36
3.3.4 Threats to Validity 37
3.3.5 Fault Injection 37
3.4 Conclusion 38

4 Improving Dynamic Analysis with Automatic Test Suite Refactoring 39

4.1 Background and Motivation 41
4.1.1 Pitfall of Repairing Real-World Bugs 41
4.1.2 Automatic Software Repair with Nopol 43
4.1.3 Real-World Example: Apache Commons Math 43
4.2 Test Suite Refactoring 44
4.2.1 Basic Concepts 44
4.2.2 B-Refactoring – A Test Suite Refactoring Approach 47
4.2.3 Implementation 50
4.3 Empirical Study on Test Suite Refactoring 51
4.3.1 Projects 51
4.3.2 Empirical Observation on Test Case Purity 51
4.3.3 Empirical Measurement of Refactoring Quality 53
4.3.4 Mutation-based Validation for Refactored Test Suites 54
4.4 Improving Dynamic Analysis Using Test Suite Refactoring 55
4.4.1 Test Suite Refactoring for Automatically Repairing Three Bugs 55

4.4.2 Test Suite Refactoring for Exception Contract Analysis 59
4.5 Threats to Validity 61
4.6 Discussion 61
4.7 Conclusion 62

5 Casper: Debugging Null Dereferences with Ghosts and Causality Traces 63
5.1 Null Debugging Approach 66
5.1.1 Null Dereference Causality Trace 66
5.1.2 Null Ghosts 68
5.1.3 Casper's Transformations 68
5.1.4 Semantics Preservation 71
5.1.5 Implementation 72
5.2 Empirical Evaluation 73
5.2.1 Dataset 73
5.2.2 Methodology 76
5.2.3 Results 78
5.2.4 Causality Trace and Patch 80
5.2.5 Threats to Validity 80
5.3 Conclusion 81

6 NpeFix: Tolerating Null Dereference at Runtime with Test Suite Validation 83
6.1 Concepts 84
6.1.1 Detecting Null Dereference Bugs 84
6.1.2 Tolerating 86
6.1.3 Viability 89
6.2 Methodology 89
6.2.1 Detection 90
6.2.2 Tolerance 91
6.3 Evaluation 96
6.3.1 Dataset 96
6.3.2 Overall Efficiency 96
6.3.3 Replacement Efficiency 97
6.3.4 Skipping Efficiency 98
6.4 Discussion 99
6.4.1 Limitations 99
6.4.2 False Positive Cases 100
6.4.3 Typing 100

6.4.4 Value Creation or Reuse 100
6.5 Conclusion 100

7 Conclusion **101**
7.1 Summary 101
7.2 Perspectives 102
7.2.1 Study of Exception Errors 102
7.2.2 Repair 103
7.2.3 Resilience 103

Bibliography **105**

Chapter 1

Introduction

1.1 Context

The world is day by day more computerized. There are more and more applications running everywhere, from personal computers to data servers, and inside most of the new popularized inventions such as connected watches or intelligent washing machines. All of those technologies use software applications to perform the services they are designed for. Unfortunately, the number of software errors grows as well as the number of software applications.

In isolation, software errors are often annoyances, perhaps costing one person a few hours of work when their accounting application crashes. Multiply this loss across millions of people and consider that even scientific progress is delayed or derailed by software error [1]: in aggregate, these errors are now costly to society as a whole.

Bug fixing consists in repairing those errors and is a large part of the work in software engineering. The goal of bug fixing is to make the behavior of a software identical to the expected behavior. An example of bug fixing is the addition of an if precondition to check whether a variable is not null before accessing it.

Bug fixing is a time-consuming task historically done by software developers. Consequently, the increase of number of bugs produces an increase in the demand of developers for fixing those bugs. For the industry, bug fixing has an economical impact: companies need to pay developers for fixing bugs. In the open-source world, bug fixing represents a loss in the time offered by the volunteers. In any case, the time used to repair a bug in an application can not be used to improve the application (e.g. by adding a new feature).

To decrease the time of bug fixing, techniques and tools have been proposed to help developers in some of the bug fixes activities such as debugging, or fault localization. In all the cases, the first step of bug fixing is to be aware that a bug exists. This awareness may come from the developer's intuition of a problematic case or can be discovered at the first time this problematic case happens.

Resilience consists in the capability of an application to remain functional despite the errors. For example, at Fukushima's power plant, the anticipated maximum tsunami height was 5.6 meters [2]. Considering the power plant as a program and the wave as an error, this arbitrary chosen high corresponds to the errors anticipated at specification and design time in software. Maybe while building the power plant, one builder had to block some cracks in the wall around structure, this corresponds to the errors encountered at development and

testing time in software. On March 11, 2011, the highest waves struck at 15 meters, this was an unanticipated error, it corresponds in software as an error that happens in production mode yet never anticipated nor encountered, as Fukushima's tsunami. In this example, bug fixing consists in finding a solution for the next wave of 15 meters, where resilience would have been to build a structure which support any high of tsunami.

This thesis focuses on bug fixing and resilience in the context of exceptions [3]. Exceptions are programming language constructs for handling errors. They are implemented in most mainstream programming languages [4] and widely used in practice [5].

Problem #1: There is a lack of debug information for the bugs related to exceptions. This hinders the bug fixing process. To make bug fixing of exceptions easier, we will propose techniques to enrich the debug information. Those techniques are fully automated and provide information about the cause and the handling possibilities of exceptions.

Problem #2: There are unexpected exceptions at runtime for which there is no error-handling code. For example, on October 22 2009, a developer working on the Apache Common Math open source project encountered an null pointer exception and reported it as bug #305[1]. Our goal is to propose resilience capabilities which correctly handle exceptions that were never foreseen at specification time neither encountered during development or testing.

To sum up, this thesis studies the exceptions, their behaviors, the information one can gathered from them, the problems they may cause and the applicable solutions to those problems.

1.2 Thesis Contribution

We present four contributions to address the two presented problems.

In **Contribution #1**, we lies the foundation to address both problems. To improve the available information about exceptions, we present a characterization of the exceptions (expected or not, anticipated or not), and of their corresponding resilience mechanisms. We provide definitions about what is a bug when facing exceptions and what are the already-in-place corresponding resilience mechanisms. We also present a code transformation that uses this knowledge to enhance the resilience of the application.

Contribution #2 aims at addressing the limitations of **Contribution #1**. The limitations is that there are undecidable cases, for which we lack information to characterize them in the conceptual framework of **Contribution #1**. We focus on the approaches that use the test suite as their main source of information as in the case of **Contribution #1**. In this contribution, we propose a technique to improve the results of those approaches by splitting test cases into smaller test cases. Applied to the **Contribution #1**, this solution improves the knowledge acquired by providing more information on more cases. Applied to other techniques using test suite, we show that it improve the quality of the results. for example, one case study presented is the use of this technique on Nopol [6], an automatic repair tool.

Contribution #3 focuses on **Problem #1**, it presents an approach for providing information about the origin of errors. We present an approach to automatically provide information about the origin of the errors which happen in production mode (i.e. those for which no efficient resilience mechanisms already exists). This work presents a more specific approach of the exceptions. Indeed, to be able to propose more specific data about the exceptions, we

[1] `https://issues.apache.org/jira/browse/MATH-305`

focus on a more specific kind of exception. The type of exceptions under study is the null pointer dereference exceptions (NullPointerException in Java). The information provided by our approach is evaluated w.r.t. its ability to help the bug fixing process.

Contribution #4 addresses **Problem #2**, we present a way to tolerate a particular kind of errors, the well-known null pointer dereferences. This novel fault-tolerance technique is applied on the errors for which the **Contribution #3** gives information about the origin. Using the knowledge **Contribution #3**, we present a set of strategies to tolerate this kind of errors. We also propose code transformations which allow to execute any of the given strategies.

1.3 Outline

The remained of this thesis is structured as follows. Chapter 2 provides an overview of the previous works that are related to automatic repair and to exception handling. Chapter 3 presents a categorization of the resilience properties in existing applications and a tool to automatically classify them. Chapter 4 presents a technique to improve the results of the approaches which use the test suite as their main source of information. Chapter 5 presents an approach and a tool which automatically provide information about the actions which lead from the origin of the problem to the error itself. Chapter 6 presents a technique to tolerate the null dereference bugs.

1.4 Publications

This work has been supported by several publications:

- corresponding to the Chapter 3: Exception Handling Analysis and Transformation using Fault Injection: Study of Resilience against Unanticipated Exceptions (Elsevier Information and Software Technology, journal, Volume 57, January 2015, Pages 66–76).
- corresponding to the Chapter 5: Casper: Debugging Null Dereferences with Ghosts and Causality Traces, poster format, ACM Student Research Competition, International Conference on Software Engineering 2015.

Parts of this work are still under submission:

- corresponding to the Chapter 4: Dynamic Analysis can be Improved with Automatic Test Suite Refactoring, paper format, submitted at Elsevier Information and Software Technology journal.
- corresponding to the Chapter 5: Casper: Debugging Null Dereferences with Ghosts and Causality Traces, paper format, submitted at Source Code Analysis and Manipulation 2015.

The last part of this work (Chapter 6) is original work for this thesis, that has not yet been submitted for publication.

Chapter 2

State of the art

In this thesis, we aim at studying software exceptions, their behaviors, the information one can gathered from them, the problems they may cause and the solutions to those problems. In Section 2.1 we will present some definitions. In the following sections we present works related to this thesis. In Section 2.2 we present general studies on exception handling. In Section 2.3 we present proposed approaches to debug bugs. In Section 2.4 we present proposed approaches to tolerate bugs. In Section 2.5 we present proposed approaches related to the resilience properties. In Section 2.6 we present proposed improvements on exception handling.

2.1 Definitions

2.1.1 Exception

Exceptions are programming language constructs for handling errors [3]. 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 nominal 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[2].

2.1.2 Resilience

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

Definition Resilience is *“the persistence of service delivery that can justifiably be trusted, when facing changes”* [8]. *“Changes may refer to unexpected failures, attacks or accidents (e.g., disasters)”* [9].

[2]the heap may change if the programming language contains a finalization mechanism (e.g. in Module-2+ [7])

Along with Trivedi et al., we formalize the idea of “unexpected” events with the notion of “design envelope", 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 mis-usages and attacks. According to that, “resilience deals with conditions that are outside the design envelope" [9].

In this thesis, we focus on the resilience in the context of software that uses exceptions.

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

2.1.3 Specifications and Test Suites

A test suite is a collection of test cases where each test case contains a set of assertions [10]. The assertions specify what the software is meant to do (i.e. it defines the design envelope). Hence, in the rest of this thesis, we consider that a test suite is a specification. Conversely, when we use the term “specification”, we refer to the test suite (even if they are an approximation of an idealized specification [11]). 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.

For instance, in regression testing, the role of a test suite is to catch new bugs – the regressions – after changes [12]. Test suites are used in a wide range of dynamic analysis techniques: in fault localization, a test suite is executed for inferring the location of bugs by reasoning on code coverage [13]; in invariant discovery, input points in a test suite are used to infer likely program invariants [14]; in software repair, a test suite is employed to verify the behavior of synthesized patches [15].

2.2 Exception Handling

The first part of our work concerns the study of the behavior of the exceptions. Indeed, fixing a bug require to understand the bug. Because we focus on exception related bugs, we study the behavior of all exceptions. Hence, a part of this work is to detect which exceptions are harmful and which are not.

Sinha [16] analyzed the effect of exception handling constructs (throw and catch) on different static analyses. This study presents different analysis techniques on programs using exceptions. They apply their approach on 18 programs (for a total of 30400 methods). They show their approach may be useful for different techniques such as program slicing or test covering. The same authors described [17] a complete tool chain to help programmers working with exceptions. They focus on the implicit control flow caused by exception handling and build a tool that identifies several inappropriate coding patterns. Implicit control flow causes complex interactions, and can thus complicate software-engineering tasks, such as

debugging. They evaluate their approach on 4 applications, and show some useful results such as the number of ignored exceptions or the coverage of throw-catch pairs.

Robillard & Murphy [18] present the concept of exception-flow information and design a tool that supports the extraction and view of exception flows. This tool helps the developer to detect unused catch clauses and to detect potential cases of improper handling of unexpected exceptions. The tool, called Jex, performs a static analysis and provides a view of the exception flow at a method level. They discuss 3 case studies (2 libraries and one application) where the tool point them to potential cases of improper handling.

Barbosa and Adachi [19] study means to improve the quality of exception handling. This study is meant to help developers to achieve better software robustness against exceptions. They based their studies on the high quantity of what they consider as "poor quality exception handling" (e.g. catch only logging or returning). They presents ideas about a declarative language to specify exception handling and a tool to assess the corresponding between the declaration and the implementation.

Cacho et al. [20] present a study about the impact of exception handling mechanism on software robustness. They claim that developers trade robustness for maintainability, for example some uncaught exception flows are introduced due to changes in a catch block. They perform their analysis on 119 versions of 16 C# programs. They provide statistics on the changes included by exception handling mechanisms on those programs. For example, they highlight that in most of the cases when the nominal code is changed, the exceptional code is also modified.

Entwisle et al. [21] present a model-driven framework to support the reliability of software systems against exceptions. Their approach helps the productivity and the maintainability of applications by simplifying the modeling of domain specific exception types. They present a UML-based modeling syntax, in which the developer models its exception types, and a tool to generate the corresponding C# code. They demonstrate the usability of their approach on one case study including 7 classes and where they generate 2 exception types.

There still exists challenges for improving exception handling in specific contexts. For example, some studies focus on the problem of exception handling in component-based systems [22] or with asynchronous objects [23].

2.2.1 Static Analysis

Several works concern the static analysis of exception constructs.

Karaorman et al. [24] proposed a tool called jContractor to manage a set of contracts (precondition and postconditions) in Java code. For example, one exception contract of jContractor says that if a method A throws an exception the method B will be executed (the method B is a kind of catch outside the method body). The developer manually adds a contract to a piece of code, then jContractor asserts that the contract is respected for each call.

Fu and Ryder [25] presented a static analysis for revealing the exception chains (exception encapsulated in one another). Those chains may be used for several different goals such as showing the error handling architecture of a system or support better testing of error recovery code. They perform a compile-time exception-chain analysis, that examines reachable catch clauses to identify catch clauses that rethrow exceptions then they build a graph of the possible chain of exceptions. They give numbers acquired from 5 well-used applications (such as Tomcat) then discuss them to detect patterns such as the number of exception-chain starting from a rethrow.

Mercadal [26] presented an approach to manage error-handling in a specific domain (pervasive computing). They present an ADL (Architecture Description Language) extended with error-handling declaration using new keywords and syntax. This ADL is then transformed into abstract class which respect both the logic of the ADL declaration and the error handling principles as defined in their improved ADL. This methodology facilitates the separation between error handling and the application logic, and asserts that the error handling complies with its specifications. They have used this framework to develop 3 applications which are a part of a project containing 3000 LOC.

2.2.2 Dynamic Analysis

Ohe et al. [27] provide runtime information about exceptions, such as handling and propagation traces of thrown exceptions in real-time. To obtain those information, they provide an exception monitoring system. This tool allows programmers to examine exception handling process and should help them to handle exceptions more effectively (i.e. knowing more information about it). They use Barat [28] to instrument the code of the application by adding loggers at some chosen points (such as when an exception is instantiated, when it is caught, etc.). After that, during the application execution, the developer accesses those information about the exception behavior. They apply their approach on 5 Java benchmark programs and show that they acquire results such as the number of exception thrown, caught or propagated.

Other papers manage to extend the knowledge they can acquire about exceptions. Zhang and Elbaum [29] have presented an approach that amplifies test to validate exception handling. Their technique allows developers to increase the test coverage of the exception handling, and hence to detect bugs which were not detected before. To do so, they automatically add tests which expose a program exception handling construct to new behavior so that it returns normally or throws an exception following a predefined pattern. They apply their approach on 5 applications, increasing (in average) 5 times the coverage of exception handling and highlighting 116 anomalies. Those anomalies contains 65% of the reported bugs on the applications and 77% of the detected anomalies are reported.

2.2.3 Using Fault Injection

One way of studying the exceptional behavior of applications is to inject errors into the application.

Segal et al. [30, 31] invented Fiat, an validation system based on fault injection. By injecting faults during design and predeployment prototyping they help developers to be aware of the capability of their program to detect such failures and to handle them. Their fault model simulates hardware fault by changing bit values in the memory used by the program. To do so they inject fault on two programs for a total of 5 scenarii. Then, they look at the response of the program (does it crash? return a wrong result? etc.) w.r.t. the injected fault.

Kao et al. [32] have described "Fine", a fault injection system for Unix kernels. The difference with Fiat is that Fine also simulates 4 kind of software faults, by changing the initialization of data structure for example. They inject 250 of those faults in a Unix kernel: SunOS 4.1.2 then observe the behavior of the system. Using those results, they propose different kinds of fault propagation models, each one corresponding to one kind of fault.

2.3 Debugging and Understanding

The first step for debugging an error is to find the cause of this error. For the debugging part, our work focuses on a particular type of exception: those which comes from null dereferences in the recent high-level languages (e.g. "NullPointerException" in Java) .

Harman et al. [33] provided a comprehensive survey of oracles in software testing. They present the different types of currently used oracles, the specified oracles (written in a test suite), the derived oracles (extracted from the program execution such as regression oracles), the implicit oracles (such as: the program does not crash), and finally, how to handle the lack of oracles. Finally, they present how they might combine metamorphic oracles and regression testing in the future.

2.3.1 General Debugging

Chilimbi et al. [34] presents HOLMES, a statistical debugging tool using a new kind of profiling model to perform bug detection. Using other kind of profiles improves the efficiency of localizing bugs by reducing the search space. They use path profiling (counting the number of time an arc in the execution path is executing instead of counting the number of time a line is executed). They show that for a given percentage of code under examination, path profiling is able to find more bugs than branch or predicate profiling.

Zeller [35] provides cause-effect chains about C bugs. This helps the developer to create a patch by knowing the root cause of the error. Their approach compares the memory graph and the execution of two versions of a same program (one faulty and one not faulty) to extract the instructions and the memory values which had lead to the error. They evaluate their work on one real case study where their approach is able to provide the 4 steps which lead to the crash.

Zamfir et al [36] presents DCop, a system collecting runtime information about deadlocks. DCop helps the developer to fill the gap between " the information available at the time of a software failure and the information actually shipped to developers in the corresponding bug report". In practice, DCop provides a breadcrumb corresponding to the call stack of the multiple threads leading to the error. They acquire this information by modifying a thread library (FreeBSD's *libthr*). They evaluate the overhead of their modification, which corresponds to less than 0.17% of the execution time.

Fetzer et al. [37] introduce the failure atomicity problem and present a system to address it. They propose techniques to detect non-atomic exception handling situations and ways to prevent it. They define a method as non-atomic if it does not preserve state consistency, i.e. the state of the program must be the same before the error and after the exception is handled. Their system injects both declared and undeclared exceptions at runtime and performs source code and byte code modifications to detect such atomicity. They evaluate their approach on 16 real world applications and found that the non-atomic methods average 20%.

2.3.2 Exception Debugging

Tracey et al. [38] present an approach to generate test-data to test the exceptions. This approach find test-data which cause exceptions (and cover the exceptional behavior of an application) and may help proving exception freeness for safety-critical software systems. They

use genetic programming to generate test cases and iterate through those generations. They present a two part evaluation. First, they evaluate the effectiveness of the technique on 7 small ADA programs where their approach allows to execute 23 out of the 24 possible exception conditions. Second, they present a case study on a commercial application where their approach partially proves the exception freeness of the application.

Bond et al. [39] present an origin tracking technique for null and undefined value errors in the Java virtual machine and a memory-checking tool. They give the line number of the first assignment of a null value responsible of a NPE. It allows the developer to know the origin of the dereference. To do it, they modify the Java VM to store null assignment lines instead of just null in null values. They evaluate their approach on 12 case studies, for all of them it gives the good origin. However, in our work we provide the complete causality trace of the error.

2.3.2.1 Null Dereference Debugging

Romano et al. [40] give information of the possible locations of null dereferences and provide a test case which executes those null dereferences. This helps the developer to both locate, understand and reproduce possible NPEs. They extract a Control Flow Graph of the application under study, then they search for paths where a null value may be dereferenced. After that, their approach uses a genetic algorithm to generate the test case, which must uses the designated path. They present an empirical evidence (27 cases from 6 Java open source systems) of the capability of the proposed approach to detect artificially injected NPEs.

Hovemeyer et al. [41] use byte-code analysis to provide possible locations where null dereference may happen, included in a common bug-finding tool. This allows the developer to be aware of the possibility of unexpected null dereference in its code. They develop it as a part of FindBugs (ref needed), they perform dataflow analysis on control-flow graph representations of the Java methods. This allow them to find where a value is null, may be null or is not null. In addition, they perform several operations to detect and eliminate infeasible NPEs (e.g. they detect the locations of defensive null checks). They evaluate their approach on more than 3 thousand student versions of projects, and on one version of Eclipse for which they find 73 possible NPEs and manually said that 16 of them are false positive.

Sinha et al. [42] use source code path finding to find the locations where a bug may happen. They develop a tool which uses reverse path finding to provide additional contextual information about those possible bugs, such as the root cause of the null dereference. This works may help developers to understand and patch the possible future bugs in their application. They evaluate their Null Pointer Analysis over 13 real bugs and said that each of them has been detected by their approach as possible NPE.

Nanda and Sinha [43] presents an approach for analyzing possible null dereferences in Java. Their approach eliminates some of the impossible highlighted null dereference by tools such as XYLEM, FindBugs and Jlint. For each possible null dereference, their analysis looks at the paths that the program has to take to reach the null dereference, then it eliminates the highlighted null dereference for which there is no possible way to execute it. They evaluate their approach on those 3 tools on 6 real world applications. Their tool eliminates an average of 5% of the reported bugs by those tools.

Ayewah and Pugh [44] discusses about the null dereference analysis techniques for Java. They give reasons to not prevent every possible null dereference. They study a null dereference analysis tool, XYLEM [43], on 5 large real world applications (including Eclipse). The

tool highlight 189 possible null dereferences. Their first finding is that less than the half of the null dereference warnings provided by tools such as XYLEM are plausible (88 out of 189), numerous of them are implausible (32 out of 189) or even impossible (70 out of 189). In addition, they claim that in some cases (such as resource freeing), keeping a possible null dereference is better than fixing it.

2.3.3 Fault Detection and Localization

Jones et al. [45] present Tarantula, a tool using test information to improve fault localization. This tool helps the developer to locate a bug by coloring the lines of code with respect to their likelihood of being responsible for the fault. To do it, Tarantula looks at the execution of the test suite, then it looks at which test case executes which line of code, and finally it colors the statement according to an equation on how much test executes this line and passes and how much tests execute this line and fail. They evaluate Tarantula on *Space* (array definition language interpreter written in C), on the 33 versions of the program, with a total of 33 bugs and more than 13K test cases. They show that in more than 60% of the cases, Tarantula highlight at least one of the faulty statements.

Bruntink et al. [46] studies the return-code idom as exceptional handling. They demonstrates that this way of dealing with errors is error-prone. They present a tool, called SMELL, which statically detects the locations in the code where the logged or returned error code is different that the one received from the first error. They evaluate their approach on 5 small ASML components, they show that SMELL is able to find 236 possible errors on the error handling, they manually evaluate that 141 of them are valid possible errors.

Perkins et al. [47] presents a tool, named Clearview, which uses runtime monitors to flag erroneous executions and then identifies invariant violations characterizing the failure, generates candidate patches that change program state or control flow accordingly, and deploys and observes those candidates on several program variants to select the best patch for continued deployment.

Smirnov et al. [48, 49] propose an approach to dynamically detect and log overflows attacks. Their tool automatically inject code in C programs to detect and log information about those attacks. Then it tries to create a patch to correct this bug. They evaluate their approach on 7 networks with known vulnerabilities and say that it takes less than 4 iterations of the attack to their approach to create a patch.

Thummalapenta and Xie [50] present CAR-Miner, an approach which mines exception-handling rules in the form of sequence association rules. The goals of this approach are to extract exception handling rules from the nominal and exceptional behavior of the applications and then to compare those rules with the behavior of the exceptional cases of other application to try to detect defects. They perform a static analysis on the usage of the SQL database with Java (package java.sql). Their experiments on 5 large Java applications (more than 7 million LOC in total) shows that their approach is able to extract an average of 100 rules per application, sadly, a manual verification shows that only 55% of them are real ones, 3% corresponds to usage patterns and 42% are false positive. They detect more than 1600 violations of those numerous rules, by focusing on the ones which breaks the first ten rules, they obtain 264 violations, 160 of them are identified as real defects.

2.4 Bug Fixing and Tolerance

A part of this thesis concerns the tolerance of null dereference errors. This section presents first the patch finding techniques, then tolerance techniques. We first present patch finding techniques because numerous tolerance techniques are patch finding techniques modified to be applied at runtime, with all the constraints that it represents (e.g. avoiding overhead).

2.4.1 Bug Fixing

Automatic software repair aims to generate patches to fix software bugs. Software repair employs a given set of test cases to validate the correctness of generated patches.

Kim et al. [51] propose PAR, a tool which generates patches based on human-written patches. The repair patterns in their work are used to avoid nonsensical patches due to the randomness of some mutation operators. They manually inspect more than 60k human-written patches (from Eclise JDT) and extract fix-patterns from it. They present 8 fix patterns covering more than 30% of the cases. Those patterns are used to generate human-likely patches. They evaluate PAR on 119 bugs, and successfully patch 27 out of the 119 bugs. In addition, according to 69 developers and 17 students, PAR generates patches which are more acceptable than the previous technique.

Nguyen et al. [52] propose SemFix, a semantic-analysis based approach, also for C bugs. Semfix narrows down the search space of repair expressions. This approach combines symbolic execution, constraint solving, and program synthesis. It derives repair constraints from tests and solve them using a SMT solver. They evaluate Semfix using 90 buggy programs from Software-artifact Infrastructure Repository (SIR). Semfix is able to generate a fix for 48 of them. In addition, Semfix generates patches for 4 out of 9 bugs of Coreutils.

Ackling et al. [53] describes pyEDB, a method for automatic defects repair on python. Considering added changes as small patches allows the genetic programming approach not to diverge too much from the original program. pyEDB uses genetic programming to modify small parts of the program instead of modifying the whole program. They evaluate the capability of pyEDB to fix bugs on 2 small benchmarks (13 and 61 LOC) where they seed faults (12 faults in total). The evaluation shows that pyEDB fixes those faults.

Vo et al. [54] present Xept a tool which “can be used to add to object code the ability to detect, mask, recover and propagate exceptions from library functions” for C applications. This tool allows developers to fix bugs when the source code of the error is not available, i.e. when the error comes from a library which is no longer supported or for which the source code is not available. They provide a specification language for intercepting and handling code, and tools to implements those specifications and to recover from errors on those functions, by changing the function calls to one written by the developer. They evaluate Xept by providing artificial sample cases of interceptions and of modifications performed by Xept.

Weimer et al. [55] propose GenProg, a genetic-programming based approach to fixing C bugs. Their goal is to have an automatic method for locating and repairing C bugs. GenProg first locate the bug, giving multiple possible locations for the root cause. Then, GenProg uses a fraction of source code as an AST and updates ASTs by inserting and replacing known AST nodes at those locations. They show that GenProg is able to repair a bug (i.e. a failing test case) on 10 real life cases. LeGoues [56] has made another evaluation of GenProg, the approach fixed 55 out of 105 bugs.

DeMarco et al. [6] presents NOPOL, a test-suite based approach to automatically repair

buggy if conditions and missing preconditions. NOPOL is able to generate human readable patches for those kind of bugs. NOPOL uses the program execution to learn data about the bug, then it generates constraints and use a SMT solver to solve those constraints, if the solver find a solution, it is a functional patch. They gives three case studies (including a real bug from common math) to demonstrate the feasibility.

Qi et al. [57] present RSRepair, an approach that repairs programs with the same operators as GenProg. The main difference is that RSRepair uses random search to guide the process of repair synthesis where GenProg uses genetic programming. They evaluate RSRepair by comparing it to GenProg, they show that RSRepair is more efficient. Indeed, it requires less iterations on the patch candidates to find a working solution and it requires less test case executions.

Dallmeier et al. [58] present a tool, named Pachika, for automatically repairing program executions. Pachika examines the differences between passing and failing runs and generates fixes that modify the behavior of the failing run to correspond to the behavior of the passing one. Pachika automatically builds behavioral models for the passing and the failing test cases. Then, it modifies the transitions (adding new ones or delete existing ones) to change the behavior of the failing model. Pachika was able to fix 3 out of 18 bugs from an open source project.

Koo et al. [59] consider the problem of bringing a distributed system to a consistent state after transient failures. They present an algorithm which allows the application to only have a minimal number of processes which are forced to take checkpoints when a process have to rollback. They provide empirical proof of their method.

Wei et al. [60] present AutoFix-E, an tool which automatically repair bugs using software contracts (preconditions, postconditions, etc). AutoFix-E integrates the contracts provided by the developer as well as dynamically collected information. They evaluate AutoFix-E on 42 bugs from Eiffel libraries, it is able to fix 16 out of those 42.

Weimer et al. [61] presents a tool, named AE, which is an extension of GenProg. This tool improves GenProg performances by decreasing the time cost. To do it, AE analyses the given programs and reduces the test case executions. They evaluate their approach on the same bugs as GenProg and show that overall, AE is three times cheaper than GenProg in terms of time needed. Arcuri and Yao [62] present an automatic patch generation technique using genetic programming. The main difference with other approaches such as GenProg or PAR is that their approach needs formal specifications where the other approaches uses test cases. Another difference is that they use co-evolutionary techniques to generate or select test cases. They evaluate their approach on 8 artificial bugs, where they are able to fix 5 of them.

Candea et al. [63] present a tool, named JAGR, for repairing web services using micro reboots. They reduce the recovery time by selecting sub-components to reboot instead of rebooting the whole application. Those components are chosen based on a learning based on the paths executed which had lead to the error. They show on a case study using fault injection that JAGR allows the application to take into account twice more requests.

2.4.2 Bug Tolerance

Long et al. [64] presents RCV, a system which allows programs to survive divide-by-zero and null dereference errors. This allows programs to avoid undesirable crashes. RCV skips the faulty statements, then it monitors the program to assess that the modified behavior

does not impact other processes. They evaluate their approach on 18 errors from 7 large real world applications (such as LibreOffice and GIMP), in 17 out of the 18 cases, RCV allows the program to continue its execution, and on 11 out of those 18 cases, the solution used by RCV is similar to the patch deployed to fix the error.

Dobolyi [65] uses source code modifications to avoid the throwing of NullPointerException. They want to increase robustness by replacing code that raises null pointer exceptions with error-handling code, allowing the program to continue execution. They locate potential NPEs and add preconditions before the faulty line to execute something else before or instead of the line which will throw a NPE. For example, one policy is to inject a new well-typed object inside the faulty null value, another is to skip the faulty line. They use three case studies to show that the approach is feasible.

Wang et al. [66, 67] present a method for performing checkpoint and rollback on multi-threaded applications. Their idea is to start by a small rollback (number of operations/method-call to be restored), to try the execution, if the execution still produces errors, they increase the number of operations to be restored and so on. They evaluate their work on an artificial case and shows that checkpointing (and logging messages passed between services) costs around 10% of execution time. They do not clearly evaluate the gain of making such progressive rollbacks.

Sidiroglou et al. [68] propose ASSURE, a tool which enhance the performances of checkpoint and rollback strategies. Their goal is to be able to deal with deterministic bugs, those for which any number of checkpoints and rollbacks will lead to the same result. In practice, when an error occurs, ASSURE examines the available checkpoints in the execution flow, selects one of those checkpoints and instead of letting the program continue its execution, which will lead to the same error, modifies the behavior of the application at the location of the rollbacked checkpoint (for example, by returning an error code). The behavioral modification is considered correct as long as the patched program still passes the previously given tests. They evaluate the overall time cost of this approach, which is from 1 to 9% of execution time. They sadly do not evaluate the correctness of their patched versions of the programs.

Avizienis et al. [69] as well as Rohr [70] describe STAR (Self-Testing And Repairing) Computer. This computer allows users to execute their programs in a more secure environment. The STAR Computer proposes a hardware fault-tolerance system based on execution redundancy asserted by oracles values at some points of the execution. In addition, the STAR Computer proposes different strategies for software fault tolerance, such as the Test And Repair Processor (TARP) which is responsible to assert the validity of the operation results and to modify them if needed.

2.5 Resilience

Wang et al. [71] explores the effects of changing the executed branch of a if condition or of skipping loop iterations in a program execution. This short-circuit modifications of programs allows to test the resilience of the program. To test this property, they modify the execution of a thousand conditions in the SPECINT 2000 Benchmark suite (12 large benchmarks such as gzip or gcc). They show in their evaluation that approximately 40% of all dynamic conditional branches may be changed without leading to an error. Note that they do not study the correctness of the results, only the correctness of the application state.

Keromytis [72] studies self healing software systems and describes the design of such a

resilient system. His work studies the main characteristics of such approaches, extracts self-healing patterns and gives possible future directions for this field. He observes several self healing techniques and presents the Observe Orient Decide Act (OODA) loop as the main idea of those techniques.

Zhao et al. [73] presents an approach which injects memory leaks to accelerate software aging. This approach allows to save a lot of time before the failure happens. To do it, they modify the TPC-W implementation to inject memory leaks, then they observe the memory consumption of the program under study. They perform this experiment on their own prototype of a web application (containing a web server, a database server and a set of emulated clients). They extract from those information what is the optimal time to rejuvenation.

2.6 Proposed Improvements on Exception Handling

Souchon et al. [74] propose an exception handling system for multi-agent systems. Its goal is to improve exception handling for being applicable to the agent paradigm. This system allows different levels for the handlers (i.e. handlers associated with a service, an agent or a role). It supports concurrency and manages exception flows between the agents. They present a prototype of this model, tested in a case study.

Munkby and Schupp [75] propose three contracts concerning the safety of the exception handling. Those contracts concern the state of a program when an exception is thrown, i.e. when the execution is interrupted. Those contracts assert that the invariants of the program are preserved and no resources leak (Basic guarantee), that in addition the operation provides rollback semantics in the event of an exception (Strong guarantee) or that the operation does not throw an exception (No-throw guarantee). They present a prototype able to statically assess those contracts on small toy programs, but do not provide real results on those contracts.

Rinard [76] proposes the Acceptability-oriented computing. It promotes the idea that certain regions of a program can be neglected without adversely affecting the overall availability of the system. Instead of trying to have a perfectly working and failure-free application, he proposes to define sections of the software which must not fail (called acceptability envelope). He proposes different theoretical ways such as data structure repair based on precondition and postcondition contracts.

Ogasawara et al. [77] present a technique to improve the performances of exception-intensive programs. Their approach is to change the nominal behavior of exceptions, when an exception is thrown, instead of going up in the stack until founding a corresponding try-catch block, the try-catch are registered and the application can directly "jump" to the corresponding try-catch block. They implement this approach in a Just In Time compiler which creates those "jumps" when an exception path is used multiple times. They evaluate the performance gain on the seven applications of the SPEC jvm98 benchmark suite. The performance improvements averages 4%, from a gain of 14% to a loss of 6%.

Choi et al. [78] present a modeling which allows to improve static analysis. They present a modeling of Control Flow Graphs which include Possible Exception-throwing Instruction. This add allows their own CFG to have bigger blocks: instead of having a block ending for each PEI, their blocks include PEI with the notion that the program may exit at this point. Having bigger blocks allows more effective analysis on the CFG because it leads to more opportunities for local optimization. They evaluate their approach on 14 java programs for

the use of an optimizing compiler. The use of their CFG instead of the basic one averagely reduces of 70% the number of required basic blocks for a high-level intermediate representations for the compiler.

Cacho et al. [79] present a study on the difference between explicit and implicit exception flows. The explicit exception flows are represented using EJFlow [80], which allows to explicitly define where an exception must be caught. Their evaluation shows that using their approach gives almost the same results (the exceptional behaviors correspond with and without the approach) but with a time gain from 3 to 20 %.

2.7 Summary

In this chapter we studied state of the art related from domains related to this thesis. Those domains concern most of the phases of program repair, from bug detection to bug tolerance, including bug localization and patch creation. Moreover, we have detect their limitations and opportunities to extend and improve them.

First, we observe that, to the best of our knowledge, most of the studies about the exceptions concern the exception themselves and not the corresponding try-catch blocks properties. According to that and to have a better idea about the existing practices to handle exceptions, we propose a first contribution (see Chapter 3) which is a categorization of the resilience properties in the existing softwares and a tool to automatically classify them.

Second, we focus on the limitations of the previous work. Hence, we propose a technique (see Chapter 4) to improve the results of the approaches which use the test suite as their main source of information. This technique is used to improve the results of our first contribution.

Third, we focus on the second phase of the repair process (i.e. understanding the bug). We observe that the existing approaches to automatically acquire information about a bug does not provide the complete causality of the errors. Indeed, most of them only provide information about the origin of a bug or about the moment where the bug happens. According to that, we propose an approach (see Chapter 5) and a tool which automatically provide information about the actions which lead from the origin of the problem to the error itself.

Finally, we propose a technique (see Chapter 6) to tolerate the bugs seen in the previous contribution. This tolerance is an improved technique of the existing approaches presented in this state of the art. Indeed, in addition of tolerating a bug, we propose a technique to automatically assess the corresponding between the proposed tolerance strategy and the program specifications.

Chapter 3

Analyzing and Improving Software Resilience against Unanticipated Exceptions

The contributions of this chapter are:

- A definition and formalization of two contracts on try-catch blocks.
- An algorithm and four predicates to verify whether a try-catch satisfies those contracts.
- A source code transformation to improve the resilience against exceptions.
- An empirical evaluation on 9 open-source software applications showing that there exists resilient try-catch blocks in practice.

Related publications:

- Exception Handling Analysis and Transformation using Fault Injection: Study of Resilience against Unanticipated Exceptions (Elsevier Information and Software Technology, journal, Volume 57, January 2015, Pages 66–76).

The first step of bug fixing is to detect that a bug happens. We define a bug as a difference between the actual behavior and the specifications. My thesis focuses on exceptions. According to that, an “exception bug" is when an exception is thrown but does not comply with the specification. This implies to characterize and define when an exception is specified. In this thesis, we use the test suites as a pragmatic, executable and accessible version of the specification. Hence, we want to know whether the behavior of exceptions is specified by test suites. In this first chapter, we introduce contracts for the try-catch blocks in use during the test suite execution. Those contracts both capture resilience properties of the application and specify the behavior of the exceptions in the application. This work reveals the difference

between anticipated (we know it happens) and expected (we want it to happen) exceptions. This goal of this chapter is to discover which throwable exceptions are not anticipated.

The key difficulty behind this goal is the notion of "unanticipated": how to reason on what one does not know, on what one has never seen? To answer this question, we start by proposing one definition of "anticipated exception". First, we consider well-tested software (i.e. those with a good automated test suite). Second, we define an "anticipated exception" as an exception that is triggered during the test suite execution. To this extent, those exceptions and the associated behavior (error detection, error-recovery) are specified by the test suite (which is a pragmatic approximation of an idealized specification [11]).

From a analytical reasoning on the nature of resilience and exceptions, we define two contracts on the programming language construct "try-catch" that capture two facets of software resilience against unanticipated exceptions.

To assess those contracts, we simulate "unanticipated exceptions" by injecting exceptions at appropriate places during test suite execution. This fault injection technique, called "short-circuit testing", 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, a large number of test cases still passes. When this happens, it means that the software under study is able to resist to certain unanticipated exceptions hence can be said "resilient".

Finally, we use the knowledge on resilience obtained with short-circuit testing to replace the caught type of a catch block by one of its super-type. This source code transformation, called "catch stretching" is considered correct if the test suite continues to pass. By enabling catch blocks to correctly handle more types of exception (w.r.t. the specification), the code is more capable of handling unanticipated exceptions.

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.

3.1 Automatic Analysis and Improved Software Resilience

We define in Section 3.1.1 two exceptions contracts applicable to try-catch blocks. We then describe an algorithm (see Section 3.1.2) and formal predicates (see Section 3.1.3) 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 3.1.4). The insight behind our approach is that we can use test suites as an oracle for the resilience capabilities against unanticipated errors.

3.1.1 Definition of Two Contracts for Exception Handling

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" contract[3] specify a property on a pair of methods. This definition is

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

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

Listing 3.1: AN EXAMPLE OF SOURCE-INDEPENDENT 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 3.2: AN EXAMPLE OF SOURCE-DEPENDENT TRY-CATCH BLOCK.

broader in scope than Meyer's "contracts" [81] 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 chapter, we ignore the concept of "finally" block [4] which is more language specific and much less used in practice [5].

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

Listing 3.3: AN EXAMPLE OF PURELY-RESILIENT TRY-CATCH BLOCK.

3.1.1.1 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 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. Section 3.1.3 gives a formal definition.

For example, Listing 3.1 shows a try-catch that *satisfies* the source-independence contract. If a value is missing in the application, and 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 3.2 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 (throws an exception), 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.

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.

We will present a formal definition of this contract and an algorithm to verify it in Section 3.1.3. We will show that both source-independent and source-dependent catch blocks exist in practice in Section 3.2.

3.1.1.2 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 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 3.3 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.

Usages There are different use cases of purely resilient try-catch blocks. We have presented in Listing 3.3 the use case of caching for pure resilience. 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 [82] in mainstream programming languages.

We will present a formal definition of this contract and an algorithm to verify it in Section 3.1.3.

Although the "pure resilience" contract is strong, we will show in Section 3.2 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.

We presented the source independence and pure resilience contracts. Note that these 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.

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.

```
1  begin
2    try_catch_list ← static_analysis(A)    ▷ retrieve all the try-catch of the application
3    standard_behavior ← standard_run(TS)   ▷ get test colors and try-catch behaviors
4    for t ∈ try_catch_list                 ▷ For each try-catch t in the application
5    do
6      prepare_injection(t, c)              ▷ prepare the try-catch t by setting an injector
7                                           ▷ which will throw an exception of the type caught by c
8                                           ▷ at the beginning of each execution of the try t
9      as ← standard_behavior : get_test_using(t)   ▷ retrieve all tests in the test suite
       TS
10                                          ▷ using the try of the try-catch t
11     for a ∈ as                           ▷ For all test a which use the current try
12     do
13       pass ← run_test_with_injection(a)  ▷ get the result of the test under
         injection
14       M[c, t] = pass                     ▷ store the result of the catch c for the test t under injection
15   return M
```

Figure 3.1: The Short-Circuit Testing Algorithm. Exception injection is used to collect data about the behavior of catch blocks.

3.1.2 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 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. Consequently, this technique allows us to study the resilience of try-catch blocks in unanticipated scenarii. We call this technique "short-circuit testing" because it resembles electrical short-circuits: when an exception is injected, the code of the try block is somehow short-circuited. The name 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.

3.1.2.1 Algorithm

Our algorithm for short-circuit testing is given in Figure 3.1. 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 3.1.1). 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 3.1.3.1).

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 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[5].

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] see `https://github.com/Netflix/Hystrix`

[5] Throwing another type of Exception is out of the scope of this chapter but discussed in Section 3.3.1

3.1.2.2 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 3.1.3. Short-circuit testing is performed with source code transformations. Monitoring and fault injection code is added to the application under analysis. Listing 3.4 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.

3.1.3 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.

3.1.3.1 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 usage as follows:
1/ “pink usage": No exception is thrown during the execution of the try block and the test case passes;
2/ “white usage": An exception is thrown during the execution of the try block and this exception is caught by the catch and the test case passes;
3/ “blue usage": An exception is thrown during the execution of the try block but this exception is not caught by the catch (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 to verify the contracts (see Section 3.1.3).

3.1.3.2 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:

$$\begin{aligned} source_independent[x] = \forall a \in A_x | \forall u_a \in usages_in(x, a) | \\ (is_white_usage[u_a] \implies pass_with_injection[a, x]) \end{aligned} \tag{3.1}$$

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:

$$\begin{aligned} source_dependent[x] = \exists a \in A_x | \forall u_a \in usages_in(x, a) | \\ (is_white_usage[u_a] \wedge \neg pass_with_injection[a, x]) \end{aligned} \tag{3.2}$$

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.

3.1.3.3 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 execute 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:

$$\begin{aligned} resilient[x] = (\forall a \in A_x | pass_with_injection[a, x]) \\ \wedge (\exists a \in A_x | \exists u_a \in usages_in(x, a) | is_pink_usage[u_a]) \end{aligned} \tag{3.3}$$

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

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.

```
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 3.4: 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.

This predicate reads as:

$$not_resilient[x] = \exists a \in A_c | \neg pass_with_injection[a, x] \quad (3.4)$$

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.

3.1.4 Improving Software Resilience with Catch Stretching

We know that some error-handling is specified in test suites. We have defined two formal criteria of software resilience and an algorithm to verify them (Section 3.1.3). 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 3.1.3). The automated refactoring of the test suite is outside the scope of this chapter but we will discuss in Section 3.2 how developers can manually refactor their test suites to improve the automated reasoning on the resilience.

3.1.4.1 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 set in the introduction, this means improving the capability of

the software under analysis to correctly handle unanticipated exceptions. For this, a solution is to transform the catch 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). This transformation may look naive, but there are strong arguments behind it. Let us now examine them.

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 3.1.1). The reasoning is as follows.

3.1.4.2 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.

3.1.4.3 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 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 is a blue test case (the test case 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 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 the resilience.

3.1.4.4 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 3.2.3 that most (91%) of source-independent try-catch blocks can be safely stretched according to the specification.

3.2 Empirical Evaluation

We have presented two exception contracts: pure resilience and source independence (Section 3.1.1.2). 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 those exception contracts.

3.2.1 Dataset

In this chapter, we analyze the specification of error-handling in the test suites of 9 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[6]. 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%.

[6]See `http://nemo.sonarsource.org`

	# 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 3.1: 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.

3.2.2 Relevance of Contracts

The experimental protocol is as follows. We run the short-circuit testing algorithm described in Section 3.1.2 on the 9 reference test suites described in Section 3.2. As seen in Section 3.1.2, short-circuit testing runs n times the test suite, 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 3.1 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.

3.2.2.1 Source Independence

Our approach is able to demonstrate that 92 try-catch blocks (sum of the fourth column of Table 3.1) 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 theoretical: they cover a reality of error-handling code.

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 that the beginning of the try has always been executed when the exception occurs*. 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 thesis 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 3.5. 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*). 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 3.1), there is not enough tests to decide whether the source-independence contract holds. This also increases the developer awareness. 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, the refactoring consists of splitting the test case so that the try/catch block under investigation

```
public class MemoryDatabaseColector extends AbstractDatabaseColector {
  public void start(){
    try{
      super.start(); // code below
    }catch (DatabaseException ex) {
      if (started==false) // this is the source−dependence
        throw ex;
      setEntityManagerFactory();
    }}}

 public void start(){
   ...
   // depending on the execution of the following statement
   // the catch block of the caller has a different behavior
   started = true;
   ...}}
```

Listing 3.5: A Source-Dependent Try-Catch Found in Sonar-core using Short Circuit Testing.

is executed only once.

3.2.2.2 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 3.1.

Listing 3.6 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 into 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 exceptions. Consequently, it is normal to observe much less try-catch blocks verifying 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-

```java
// task of try−catch:
// given a prototype object
Class clazz = prototype.getClass();
// return a Collection class that has an accessible constructor
// which is compatible with the prototype's class
try {
  // plan A: returns the prototype's class if a constructor exists
  prototype.getDeclaredConstructor();
  return clazz;
} catch (NoSuchMethodException e) {
  // plan B: returns a known instantiable collection
  // which is compatible with the prototype's class
  if (LinkedList.class.isAssignableFrom(clazz)) {
    return LinkedList.class;
  } else if (List.class.isAssignableFrom(clazz)) {
    return ArrayList.class;
  } else if (SortedSet.class.isAssignableFrom(clazz)) {
    return TreeSet.class;
  } else {
    return LinkedHashSet.class;
  }
}
```

Listing 3.6: A Purely-Resilient Try-Catch Found in spojo-core (see SpojoUtils.java)

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.

3.2.3 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 3.1.4, i.e. for which the caught exception can be set to “Exception”). We use source code transformation and the algorithm described in Section 3.1.4.

The last column of Table 3.1 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 3.1 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 3.1.4, this corresponds to the case where the stretching results is 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 3.1.4. For instance in joda-time, all four source-independent try-catch blocks present are never traversed by an exception – case A of Section 3.1.4.3. (for instance the one at line 560 of class ZoneInfoCompiler). We have shown that analytically, they can safely 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 upper in the stack. Although the stretching modifies the test case execution, the test suite passes, this means that the stretching is correct with respect to the test suite.

3.2.4 Summary

To sum up, this empirical evaluation has shown that the short-testing approach of exception contracts enables to increase the knowledge one has on a piece of software. 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 to 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.

3.3 Discussion

3.3.1 Injected Exceptions

3.3.1.1 Injection Location

In short-circuit testing we inject worst-case exceptions, recall that “worst-case” means that we inject exceptions at the beginning of the try block, i.e. the location which skips as much code as possible in the try block. By doing that, the injected exception discards the whole code of the try block.

Another possibility would be to inject exceptions at any location in the try block (for instance, before line 1 of the try block, after line 1, after line 2, etc.). Instead of executing each test once with worst-case injection, this algorithm would execute each test for each possible injection location. Let us call this algorithm “fine-grain injection". If the satisfaction or

```
boolean a = true;
try {
   // injection location #1
   a = foo();//returns false
   // injection location #2
   a = bar();//returns true
   // injection location #3
   return myMethod();//exception thrown
} catch (Exception e) {
   if (a){
      return 0;
   }else{
      return −1;
   }
}
```

Listing 3.7: An example of try-catch block categorized differently by worst-case exception injection (short-circuit testing) and fine-grain exception injection

violation of the contracts is undecidable with short-circuit testing, it would still be undecidable with fine-grain injection because the verifiability of contracts depends on the test suite coverage of try-catch only (see Section 3.1.3.2). In addition, this algorithm could not infirm the source dependence of catch blocks (if a test case fails with short-circuit, it would still fail with fine-grain injection). Finally, this algorithm could show that some catch blocks that are characterized as source independent by short-circuit testing are actually source dependent. We will show this is unlikely but theoretically possible. Let us now explain in which case this can happen.

Let us consider the code shown in Figure 3.7 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 assignment 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* and `a` is still equals to `true` when entering the catch 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 additional 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*), and the catch block works under injection at this location. In the second run an exception is injected at *injection location #2*, thus `a = false` because the last assignment 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 one test fails under injection the catch block under test is proved source dependent (as said by Predicate #2 in Section 3.1.3.2).

This example is very artificial, over the course of our experiments we never found any similar cases.

Furthermore, fine-grain algorithm does not cover all the possible cases. For example, let us consider the case where the 2 statements (`a = foo(); a = bar();`) are in another method `getValue`, those statements are replaced in the try block by `a=getValue();`. The fine-grain algorithm can still inject exceptions at *injection location #1 and #3* but the *injection location #2* is now inaccessible. By testing the two remaining locations, fine-grain algorithm says that the catch is source independent. But one can says that if an exception occurs during the execution of `bar()` (during the call `a=getValue();`), the catch does not work as expected. In this case, one can say that the catch is source dependent but even the fine-grain algorithm is not fine enough to find the case where it fails. The main idea beyond this example is that if one wants to know if a catch block is totally source independent, she should use an algorithm which is able to inject exceptions at each possible location in the method and recursively in the method calls.

The cost of executing such of this “finest-grain" algorithm is far too expensive. For characterizing one try-catch block each test case must be executed once for each statement contained in the try and for each statement contained in the method calls contained in the try recursively.

Note that we do not take in consideration that an exception may happen in different location inside a JDK call (and inside a native call), which is far worse and more expensive to simulate.

Worst-case injection is sound for violations and can be mostly trusted for satisfaction. Knowing that and given the cost of executing finest-grain algorithm, we think that worst-case injection is the most valuable in our context.

3.3.1.2 Checked and Unchecked

In java, Exception can be divided in two main categories : checked and unchecked exceptions [7]. The main difference between them is that an unchecked exception may occur anywhere, where a checked exception has to be declared. So, one knows where checked Exceptions can occur. In practice, any method can throw any unchecked exception at any time. In the opposite, if a method wants to throw a checked Exception `A`, it has to declare than it can `throws A` in its signature. In addition, when such a `throws A` is declared, the developer has to take it in consideration, she can either put the corresponding call in a try-catch, or says that her method can `throws A` too.

The previously defined contracts take in consideration the fact that an exception which is different that the tested one may occur in another location. Knowing that an unchecked exception may occur anywhere, it is obvious that the contracts are valuable for unchecked exceptions. What about checked Exceptions?

In any case, short-circuit testing is independent of programming language. It allows us to verify contract at catch level. Once the catch is defined, the fact that the caught exception has the exact caught type or a subtype does not change the behavior of the catch. That's why catch Stretching shows that the type is not important wrt resilience. An exception which cannot be caught will be catchable under catch-stretching so uncatchable exceptions are out of the scope of this chapter.

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`

[7] because the behavior of Throwable is the same as checked Exception and the behavior of Error the same as unchecked ones, we respectively consider them as checked and unchecked Exception.

`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.

3.3.2 Relevance of Exception Contracts for Single-Statement Try Blocks

Let us now consider the special case of try blocks containing one and only one statement. Listing 3.5 shows an example of this kind of try block, which contains only `super.start();`.

The pure resilience contract asserts that if the try block throws an exception, the code of the catch will perform the expected operation. This notion of plan B is independent of the size of the corresponding try block. Even a single statement can crash, and in this case having a plan B is as valuable as if there are multiple statements.

For the source independence contract, there are two possibilities. Either, the statement contains a function call or it does not.

If the statement contains a function call, the thrown exception can come from different locations inside the called method. From the point of view of short circuit testing, there is no theoretical difference between skipping the first statements inside the try before the expected exception is thrown and skipping the first statements of the body of the encapsulated method before the expected exception is thrown. The short-circuit testing algorithm finds source dependent try-catch blocks even if there is one single statement in the try. For example, Listing 3.5, found with short-circuit testing, is source dependent.

If the statement does not contain a function call, it may still contain multiple expressions and not be “side-effect free". For example, let us consider an integer array a and an integer b. The statement `return a[0] / b++ ;` is not side-effect free. Indeed, if an exception is thrown during the evaluation of `a[0]/b` (“Division By Zero" for example) the evaluation of `b++` would still be executed. In this case, if *b=0* before the execution, after the exception is thrown, *b=1*. In the point of view of short circuit testing, skipping the whole expression is valuable because it simulates the case where another exception is thrown before the Division By Zero. For example, this skipping simulates the case where an “ArrayIndexOutOfBoundsException" is thrown before the evaluation of `b++` (if the array *a* has no element). In this case, it stills *b=0* after the try block instead of *b=1*, if the code of the catch uses the value of b, the catch is source dependent.

Note that if the statement contains only one operation with no side-effects, the try will be source independent by construction. Indeed, because there is only one operation, there is only one possible location of exception throwing. In this case, the only operation inside the try is aborted when the exception is thrown.

3.3.3 Try Scope Decrease

Pure resilience is a strong contract, it is meant to characterize a small number of resilience cases. Since source-independence has a larger scope, let us now discuss how to transform a source-dependent try-catch (i.e. violating the source-independence contract according to the test data) into a source-independent try-catch (i.e. satisfying the source-independence

contract). Transforming a try-catch means either modifying the try block or modifying the catch block.

When a try-catch block is proven to violate the source-independence contract, our approach gives a precise counter-example. The counter example is of the form: if the system is in the state specified by test case X and an catchable exception occurs at the beginning of the try block, the system fails to recover from it. We have explained that the root cause is that the code of the catch block depends on state changes occurring in the try block. A solution is to decrease the scope of the try block so that the catch block only depends on the state at the beginning of the try block only. In order to keep all test cases passing, the decrease stops at the statement for which the specified exception occurs (and not the inject ones).

Decreasing a try scope is not a solution to increase the resilience. Before the refactoring, all exceptions of a certain type X were caught. After the refactoring, all specified exceptions are still caught. But if an unanticipated of type X occurs in the statements that were moved outside the try block, there is no exception-handling code to recover from it. This highlights again our key motivation: resilience is about unanticipated errors. (In addition, automatically moving code outside a try block poses a number of compilation challenges in terms of checked-exceptions, variable visibility and control-flow soundness).

3.3.4 Threats to Validity

The main threat to the construct validity lies in bugs in our implementation of short-circuit testing. To mitigate this threat, over the course of our experiments, we have regularly thoroughly analyzed try-catch blocks for which satisfaction or violation of one of the contracts was identified. The analysis consists of reading the try-catch, the surrounding code and the test cases that execute it. In the end, this gives us good confidence that there is no serious bug in the experimental code.

The internal validity of our experiment is threaten if the test case behavior we observe is not causally connected to the exception that are thrown. Since we use mature deterministic software (a Java Virtual Machine) and a deterministic dataset, this is actually unlikely that spurious exceptions mix up our results.

Finally, there remains 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).

3.3.5 Fault Injection

Other works also use fault injection for different goals. Some of those works are presented below.

Bieman et al. [83] added assertions in software that can be handled with an "assertion violation" injector. The test driver enumerates different state changes that violate the assertion. By doing so, they are able to improve branch coverage, especially on error recovery code. This is different from our work since: we do not manually add any information in the system under study (tests or application). Fu et al. [84] described a fault injector for exceptions similar to ours in order to improve catch coverage. In comparison to both [83] and

[84], we do not aim at improving the coverage but to identify the try-catch blocks satisfying exception contracts.

Candea et al. [85] used exception injection to capture the error-related dependencies between artifacts of an application. They inject checked exceptions as well as 6 runtime, unchecked exceptions. We also use exception injection but for a different goal: verifying the exception contracts.

Ghosh and Kelly [86] did a special kind of mutation testing for improving test suites. Their fault model comprises "abend" faults: abnormal ending of catch blocks. It is similar to short-circuiting. We use the term "short-circuit" since it is a precise metaphor of what happens. In comparison, the term "abend" encompasses many more kinds of faults. In this chapter, we claim that the new observed behavior resulting from short-circuit testing should not be considered as mutants to be killed. Actually we claim the opposite: short-circuiting should remain undetected for sake of source independence and pure resilience.

3.4 Conclusion

In this chapter, we have explored the concept of software resilience against exceptions. We have contributed with different results that, to our knowledge, are not discussed in the literature. First, we have shown to what extent test suites specify exception-handling. Second, we have formalized two formal resilience properties: source-independence and pure-resilience as well as an algorithm 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. Sadly, we also discover that numerous cases are undecidable due to the behavior of the test cases. According to that, in the next chapter, we present a method to improve the possibility of assessing of those contracts.

Chapter 4

Improving Dynamic Analysis with Automatic Test Suite Refactoring

The contributions of this chapter are:

- A formulation of the problem of automatic test case refactoring for dynamic analysis.
- The concept of pure and impure test cases.
- B-Refactoring, an approach for automatically refactoring test suites according to a specific criterion.
- An evaluation of B-Refactoring to assist two existing dynamic analysis tasks from the literature: automatic repair of `if`-condition bugs and automatic analysis of exception contracts.

This work is currently under submission at the journal Information and Software Technology.

In Chapter 3, we presented ways to detect bugs. However, we show that we miss information to classify more than half of them (125 out of 241 try-catch). In this chapter, we focus on improving the existing approaches using the test suites. We do not modify the approaches themselves, we modify the test suite to provide more information to the approach. Many dynamic analysis techniques are based on the exploitation of execution traces obtained by each test case [12, 14, 87]. Hence, in this work, we focus on improving those specifications by providing a more precise information w.r.t. the required type of information.

Indeed, Xuan et al. [88] show that traces by an original test suite are suboptimal with respect to *fault localization*. The original test suite is updated to enhance the usage of *assertions* in fault localization. In the current chapter, we propose a generalized way of test suite refactoring, which optimizes the usage of the whole *test suite* according to a *given dynamic analysis technique*.

In this chapter, we propose a new technique of test suite refactoring, called B-Refactoring.

The idea behind B-Refactoring is to split a test case into small test fragments, which cover a simpler part of the control flow to provide better support for dynamic analysis. For example, in the previous chapter a try-catch cannot be categorized as resilient if the test suite executes it in different ways in a single test case. The goal of this work is to split this test case in multiple smaller test cases, each one containing only the required information. The test suite after refactoring consists of smaller test cases that do not reduce the potential of bug detection. This approach is evaluated over two existing analysis, the try-catch purity presented in the previous chapter and Nopol [6], an automatic `if` conditions repair tool.

Different types of dynamic analysis techniques require different types of traces. The accuracy of dynamic analysis depends on the structure of those traces, such as length, diversity, redundancy, etc. For example, several traces that cover the same paths with different input values are very useful for discovering program invariants [14]; fault localization benefits from traces that cover different execution paths [87] and that are triggered by assertions in different test cases [88]. However, in practice, one manually-written test case results in only one trace during the test suite execution; on the other hand, test suite execution traces can be optimal with respect to test suite comprehension (from the human viewpoint by authors of the test suite) but might be suboptimal with respect to other criteria (from the viewpoint of dynamic analysis techniques).

In this chapter, instead of having a single test suite used for many analysis tasks, our hypothesis is that a system can automatically optimize the design of a test suite with respect to the requirements of a given dynamic analysis technique. For instance, given an original test suite, developers can have an optimized version with respect to fault localization as well as another optimized version with respect to automatic software repair. This optimization can be made on demand for a specific type of dynamic analysis.

Our approach to test suite refactoring, called B-Refactoring,[8] detects and splits impure test cases. In our work, an *impure test case* is a test case, which executes unprocessable path in one dynamic analysis technique. The idea behind B-Refactoring is to split a test case into small "test fragments", where *each fragment is a completely valid test case and covers a simple part of the control flow*; test fragments after splitting provide better support for dynamic software analysis. The purified test suite is semantically equivalent to the original one: it triggers exactly the same set of behaviors as the original test suite and detects exactly the same bugs. However, it produces a different set of execution traces. This set of traces suits better for the targeted dynamic program analysis. Note that our definition of purity is specific to test cases and is completely different from the one used in the programming language literature (e.g., pure and impure functional programming in [89, 90, 91]).

To evaluate our approach, we consider two dynamic analysis techniques, one in the domain of automatic software repair [6] and the other in the context of dynamic verification of exception contracts [92]. We briefly present the case of software repair here and will present in details the dynamic verification of exception contracts in Section 4.4.2. For software repair, we consider Nopol [6], an automatic repair system for bugs in `if` conditions. Nopol employs a dynamic analysis technique that is sensitive to the design of test suites. The efficiency of Nopol depends on whether the same test case executes both `then` and `else` branches of an `if`. This forms a purification criterion that is given as input to our test suite refactoring technique. In our dataset, we show that purification yields 66.34% increase in the number of

[8] B-Refactoring is short for Banana-Refactoring. We name our approach with *Banana* because we split a test case as splitting a banana in the ice cream named Banana Split.

purely tested `if`s (3,746 instead of 2,252) and *unlocks new bugs which are able to be fixed by the purified test suite*.

The remainder of this chapter is organized as follows. In Section 4.1, we introduce the background and motivation of test suite refactoring. In Section 4.2, we define the problem of refactoring test suites and propose our approach B-Refactoring. In Section 4.3, we evaluate our approach on five open-source projects; in Section 4.4, we apply the approach to automatic repair and exception contract analysis. Section 4.5 discusses the threats to validity. Section 4.6 lists the related work and Section 4.7 concludes our work.

4.1 Background and Motivation

Test suite refactoring can be used in different tasks of dynamic analysis. In this section, we present one application scenario, i.e., `if`-condition bug repair. However, test suite refactoring is general and goes beyond software repair. Another application scenario of exception handling can be found in Section 4.4.2.

4.1.1 Pitfall of Repairing Real-World Bugs

In test suite based repair [55], [52], [6], a repair method generates a patch for potentially buggy statements and then validates the patch with a given test suite. For example, a well-known test suite based method, GenProg [55], employs genetic programming to generate patch code via updating Abstract Syntax Tree (AST) nodes in C programs. The generated patch is to pass the whole test suite.

Research community of test suite based repair has developed fruitful results, such as GenProg by Le Goues et al. [15], Par by Kim et al. [51], and SemFix [52]. However, applying automatic repair to real-world bugs is unexpectedly difficult.

In 2012, a case study by Le Goues et al. [93] showed that in their dataset, 55 out of 105 bugs can be fixed by GenProg [15]. This work has set a milestone for the real-world application of test suite based repair techniques. Two years later, Qi et al. [57] empirically explored the search strategy (genetic programming) inside GenProg and showed that this strategy does not perform better than random search. Their proposed approach based on random search, RSRepair, worked more efficiently than GenProg. Recent work by Qi et al. [94] has examined the "plausible" results in the experimental configuration of GenProg and RSRepair. Their result pointed out that only 2 out of 55 patches that are reported in GenProg [93] are actually correct; 2 of the 24 patches that are reported in RSRepair [57] are correct. All the other reportedly fixed bugs suffer from problematic experimental issues and unmeaningful patches.

Repairing real-world bugs is not easy. Test suite based repair is able to generate a patch, which passes the whole test suite. But this patch may not repairs the same functionality as the real-world patches. In other words, a patch by a test suite based repair technique could be semantically incorrect, comparing with a manually-written patch by developers.

```
1 public double factorialDouble(final int n) {
2     if (n < 0) {
3         throw new IllegalArgumentException(
4             "must␣have␣n␣>=␣0␣for␣n!");
5     }
6     return Math.floor(Math.exp( factorialLog (n)) + 0.5);
7 }
8
9 public double factorialLog(final int n) {
10    // PATCH: if (n < 0) {
11    if (n <= 0) {
12        throw new IllegalArgumentException(
13            "must␣have␣n␣>␣0␣for␣n!");
14    }
15    double logSum = 0;
16    for (int i = 2; i <= n; i++) {
17        logSum += Math.log((double) i);
18    }
19    return logSum;
20 }
```

(a) Buggy program

```
1 public void testFactorial() { //Passing test case
2     ...
3     try {
4         double x = MathUtils.factorialDouble(-1);
5         fail("expecting␣IllegalArgumentException");
6     } catch (IllegalArgumentException ex) {
7         ;
8     }
9     try {
10        double x = MathUtils.factorialLog(-1);
11        fail("expecting␣IllegalArgumentException");
12    } catch (IllegalArgumentException ex) {
13        ;
14    }
15    assertTrue("expecting␣infinite␣factorial␣value",
16        Double.isInfinite(MathUtils.factorialDouble(171)));
17 }
18 public void testFactorialFail() { //Failing test case
19    ...
20    assertEquals("0", 0.0d, MathUtils.factorialLog(0), 1E-14);
21 }
```

(b) Two original test cases

```
1 // The first fragment must execute the setUp code
2 @TestFragment(origin=testFactorial, order=1)
3 void testFactorial_fragment_1 () {
4     setUp();
5   //Lines from 2 to 14 in Fig. 1b executing then branch
6 }
7
8 // Split between Line 14 and Line 15 in Fig. 1b
9
10 // The last fragment must execute the tearDown code
11 @TestFragment(origin=testFactorial, order=2)
12 void testtestFactorialFail_fragment_2 () {
13  //Lines from 15 to 16 in Fig. 1b executing else branch
14    tearDown();
15 }
16
17 // Already pure test case
18 @Test
19 public void testFactorialFail() {
20    // Executes the then branch
21 }
```

(c) Three test cases after purification

Figure 4.1: Example of test case purification. The buggy program and test cases are extracted from Apache Commons Math. The buggy `if` is at Line 11 of Fig. 4.1.a. A test case `testFactorial` in Fig. 4.1.b executes both `then` (at Line 10 of Fig. 4.1.b) and `else` (at Line 15 of Fig. 4.1.b) branches of the `if` (at Line 11 of Fig. 4.1.a). Fig. 4.1.c shows the test cases after the splitting (between Lines 14 and 15) according to the execution on branches.

4.1.2 Automatic Software Repair with Nopol

Test suites in repairing real-world bugs are worth investigation. A test suite plays a key role in validating whether a generated patch fixes a bug and behaves correctly in test suite based repair. The quality of test suites impacts the patch generation in automatic repair. The test suite refactoring technique, addressed in this chapter, is to enhance the given test suite to assist automatic repair (as well as other dynamic analysis techniques as reported in Section 4.4.2).

To motivate our test suite refactoring in the context of software repair, we introduce an existing repair approach, Nopol [6]. Nopol focuses on fixing bugs in `if` conditions. To generate a patch for an `if` condition, Nopol *requires at least one failing test case and one passing test case*. To avoid Nopol to generate a trivial fix (e.g., `if (true)`), *test cases have to cover both the `then` branch and the `else` branch*.

However in practice, one test case may cover both `then` and `else` branches together. This results in an ambiguous behavior for the repair approach, Nopol. In the best case, the repair approach discards this test case and continues the repair process; in the worst case, the repair approach cannot fix the bug because discarding the test case leads to a lack of test cases. In this chapter, the test suite refactoring technique that we will present enables a repair approach to fix previously-unfixed bugs.

We choose Nopol as the automatic repair approach in our experiment for the following reasons. First, Nopol is developed by our group [6] and is open-source available.[9] Second, the target of Nopol is only to fix `if`-condition bugs; such a target will narrow down the bugs under study and reduce the impact by different kinds of bugs [95].

4.1.3 Real-World Example: Apache Commons Math

We use a real-world bug in Apache Commons Math to illustrate the motivation of our work. *Apache Commons Math* is a Java library of self-contained mathematics and statistics components.[10] Fig. 4.1 shows code snippets from the Apache Commons Math project. It consists of real-world code of a program with a bug in an `if` and two related test cases.[11] The program in Fig. 4.1.a is designed to calculate the factorial, including two methods: `factorialDouble` for the factorial of a real number and `factorialLog` for calculating the natural logarithm of the factorial. The bug, shown in the `if` condition `n<=0` at Line 11, should actually be `n<0`.

Fig. 4.1.b displays two test cases that execute the buggy `if` condition: a passing one and a failing one. The failing test case detects that a bug exists in the program while the passing test case validates the correct behavior. In Fig. 4.1.b, we can observe that test code before Line 14 in the test case `testFactorial` executes the `then` branch while test code after Line 15 executes the `else` branch. Consequently, Nopol fails to repair this bug because it cannot distinguish the executed branch (the `then` branch or the `else` one).

Is there any way to split this test case into two parts according to the execution on branches? Fig. 4.1.c[12] shows two test cases after splitting the test case `testFactorial` between Lines 14 and 15. Based on the test cases after splitting, Nopol works well and is

[9] Nopol Project, `https://github.com/SpoonLabs/nopol/`.

[10] Apache Commons Math, `http://commons.apache.org/math/`.

[11] See `https://fisheye6.atlassian.com/changelog/commons?cs=141473`.

[12] Note that in Fig. 4.1.c, the first two test cases after splitting have extra annotations like `@TestFragment` at Line 2 as well as extra code like `setUp` at line 4 and `tearDown` at Line 15. We add these lines to facilitate the

able to generate a correct patch as expected. The test case splitting motivates our work: refining a test case to cover simpler parts of the control flow during program execution.

Test suite purification can be applied prior to different dynamic analysis techniques and not only to software repair. Section 4.4.2 presents another application scenario, i.e., exception contract analysis.

4.2 Test Suite Refactoring

In this section, we present basic concepts of test suite refactoring, our proposed approach, and important technical aspects.

4.2.1 Basic Concepts

In this chapter, a *program element* denotes an entity in the code of a program, in opposition to a *test constituent* that denotes an entity in the code of a test case. We use the terms *element* and *constituent* for sake of being always clear whether we refer to the applicative program or its test suite. Any node in an Abstract Syntax Tree (AST) of the program (resp. the test suite) can be considered as a program element (resp. a test constituent). For example, an `if` *element* and a `try` *element* denote an `if` element and a `try` element in Java, respectively.

4.2.1.1 Execution Domain

Definition 1 *Let E be a set of program elements in the same type of AST nodes. The execution domain D of a program element $e \in E$ is a set of code that characterizes one execution of e.*

For instance, for an `if` element, the execution domain can be defined as

$$D_{\texttt{if}} = \{\texttt{then-branch}, \texttt{else-branch}\}$$

where `then-branch` and `else-branch` are the execution of the `then` branch and the `else` branch, respectively.

The execution domain is a generic concept. Besides `if`, two examples of potential execution domains are as follows: the execution domain of a method invocation $func(var_a, var_b, ...)$ is $\{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ where $\mathbf{x}_i$ is a vector of actual values of arguments in a method invocation; the execution domain of `switch-case` is $\{case_1, case_2, \ldots, case_n\}$ where $case_i$ is a `case` in the `switch`.

For `try` elements, we define the execution as follows

$$D_{\texttt{try}} = \{\texttt{no-exception}, \texttt{exception-caught}, \\ \texttt{exception-not-caught}\}$$

where `no-exception`, `exception-caught`, and `exception-not-caught` are the execution results of `try` element: no exception is thrown, one exception is caught by the catch block, and one exception is thrown in the catch block but not caught, respectively. The execution domain of `try` will be used in dynamic verification of exception handling in Section 4.4.2.

test execution, which will be introduced in Section 4.2.3.

4.2.1.2 Execution Purity and Impurity

Values in D are mutually exclusive: a single execution of a program element is uniquely classified in D. During the execution of a test case, a program element $e \in E$ may be executed multiple times.

We refer to an execution result of a program element as an *execution signature*. A *pure execution signature* denotes the execution of a program element, which yields a single value in an execution domain D, e.g., only the `then` branch of `if` is executed by one given test case t. An *impure execution signature* denotes the execution of a program element with multiple values in D, e.g., both `then` and `else` branches are executed by t. Given an execution domain, let $D^0 =$ impure be the set of impure execution signatures.

The execution signature of a test case t with respect to an element e is the aggregation of each value as follows. Let T be the test suite, the set of all test cases, we define

$$f : E \times T \to D \cup D^0 \cup \{\perp\}$$

where $\perp$ (usually called "bottom") denotes that the test case t does not execute the program element e. For example, $f(e,t) \in D^0$ indicates both `then` and `else` branches of an `if` element e is executed by a test case t. If the test case executes the same element always in the same way (e.g., the test case always executes `then` in an `if` element), we call it *pure*. Note that for the simplest case, a set of program elements may consist of only one program element.

In this chapter, we consider a test case t as a sequence of test constituents, i.e., $t = \langle c_1, c_2, \ldots, c_n \rangle$. Let C denote the set of c_i ($1 \leq i \leq n$). Then the above function $f(e,t)$ can be refined for the execution of a test constituent $c \in C$. A function g gives the purity of a program element according to a test constituent:

$$g : E \times C \to D \cup D^0 \cup \{\perp\}$$

A test constituent c is pure on E if and only if $(\forall e \in E)\ g(e,c) \in D \cup \{\perp\}$; c is impure on E if and only if $(\exists e \in E)\ g(e,c) \in D^0$.

Definition 2 *Given a set E of program elements and a test case $t \in T$, let us define the impurity indicator function $\delta : \mathcal{E} \times T$, where $\mathcal{E}$ is a set of all the candidate sets of program elements. In details, $\delta(E,t) = 0$ if and only if the test case t is pure (on the set E of program elements) while $\delta(E,t) = 1$ if and only if t is impure. Formally,*

$$\delta(E,t) = \begin{cases} 0 & \textit{pure, iff } (\forall e \in E)\ f(e,t) \in D \cup \{\perp\} \\ 1 & \textit{impure, iff } (\exists e \in E)\ f(e,t) \in D^0 \end{cases}$$

At the test constituent level, the above definition of purity and impurity of a test case can be stated as follows. A test case t is pure if $(\exists x \in D)\ (\forall e \in E)\ (\forall c \in C)\ g(e,c) \in \{x\} \cup \{\perp\}$. A test case t is impure if t contains either at least one impure constituent or at least two different execution signatures on constituents. That is, either $(\exists e \in E)(\exists c \in C) g(e,c) \in D^0$ or $(\exists e \in E)(\exists c_1, c_2 \in C)\ (g(e,c_1) \neq g(e,c_2)) \wedge (g(e,c_1), g(e,c_2) \in D)$ holds.

An *absolutely impure test case* according to a set E of program elements is a test case for which there exists at least one impure test constituent: $(\exists e \in E)\ (\exists c \in C)\ g(e,c) \in D^0$.

Definition 3 *A program element e is said to be purely covered according to a test suite T if all test cases yield pure execution signatures: $(\forall t \in T)\ f(e,t) \notin D^0$. A program element e is impurely covered according to T if any test case yields an impure execution signature: $(\exists t \in T)\ f(e,t) \in D^0$. This concept will be used to indicate the purity of test cases in Section 4.3.*

Note that the above definitions are independent of the number of assertions per test case. Even if there is a single assertion, the code before the assertion may explore the full execution domain of certain program elements.

4.2.1.3 Test Case Purification

Test case refactoring aims to rearrange test cases according to a certain task [96], [97], [98]. *Test case purification* is a type of test case refactoring that aims to minimize the number of impure test cases. In this chapter, our definition of purity involves a set of program elements, hence there are multiple kinds of feasible purification, depending on the considered program elements. For instance, developers can purify a test suite with respect to a set of `if`s or with respect to a set of `try`s, etc.

Based on Definition 2, the task of test case purification for a set E of program elements is to find a test suite T that minimizes the amount of impurity as follows:

$$\arg\min \sum_{t \in T} \delta(E,t) \tag{4.1}$$

The minimum of $\sum_{t\in T} \|\delta(E,t)\|$ is 0 when all test cases in T are pure. As shown later, this is usually not possible in practice. Note that, in this chapter, we do not aim to find the absolutely optimal purified test suite, but finding a test suite that improves dynamic analysis techniques. An impure test case can be split into a set of smaller test cases that are possibly pure.

Definition 4 *A test fragment is a continuous sequence of test constituents. Given a set of program elements and a test case, i.e., a continuous sequence of test constituents, a pure test fragment is a test fragment that includes only pure constituents.*

Ideally, an impure test case without any impure test constituent can be split into a sequence of pure test fragments, e.g., a test case consisting of two test constituents, which covers `then` and `else` branches, respectively. Given a set E of program elements and an impure test case $t = \langle c_1, \ldots, c_n \rangle$ where $(\forall e \in E)\ g(e, c_i) \in D \cup \{\bot\}$ $(1 \leq i \leq n)$, we can split the test case into a set of m test fragments with test case purification. Let φ_j be the jth test fragment $(1 \leq j \leq m)$ in t. Let c_j^k denote the kth test constituent in φ_j and $|\varphi_j|$ denote the number of test constituents in φ_j. We define φ_j as a continuous sequence of test constituents as follows

$$\varphi_j = \langle c_j^1, c_j^2, \ldots, c_j^{|\varphi_j|} \rangle$$

where $(\exists x \in D)\ (\forall e \in E)\ (\forall c)\ g(e,c) \in \{x\} \cup \{\bot\}$.

Based on the above definitions, given a test case without impure test constituents, the goal of test case purification is to generate a minimized number of pure test fragments.

Example of test case purification. In the best case, an impure test case can be refactored into a set of test fragments as above. Table 4.1 presents an example of test case purification

Table 4.1: Example of three test fragments and the execution signature of an `if` element.

Test constituent	c_1	c_2	c_3	c_4	c_5	c_6	c_7
Execution signature	$\perp$	`then-branch`	$\perp$	`else-branch`	$\perp$	`else-branch`	`then-branch`
Test fragment	$\langle c_1, c_2, c_3 \rangle$			$\langle c_4, c_5, c_6 \rangle$			$\langle c_7 \rangle$

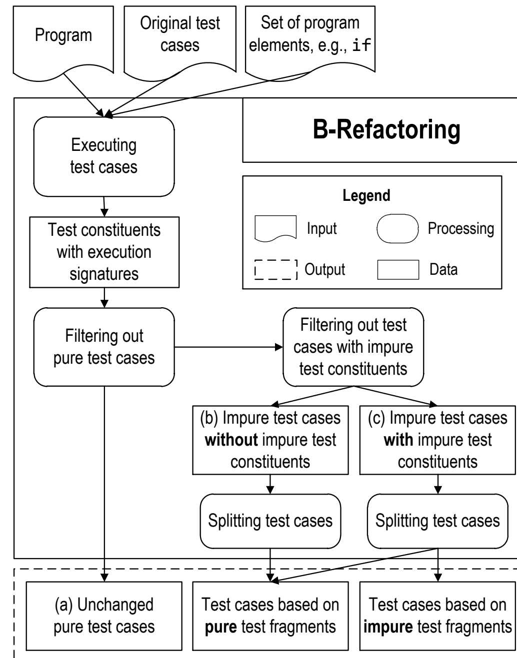

Figure 4.2: Conceptual framework of test case purification. This framework takes a program with test cases and a specific set of program elements (e.g., `if` elements) as input; the output is new test cases based on test fragments. The sum of test cases in (a), (b), and (c) equals to the number of original test cases.

for a test case with seven test constituents $t = \langle c_1, c_2, c_3, c_4, c_5, c_5, c_6, c_7 \rangle$ that are executed on a set of `if` elements consisting of only one `if` element. Three test fragments are formed as $\langle c_1, c_2, c_3 \rangle$, $\langle c_4, c_5, c_6 \rangle$, and $\langle c_7 \rangle$. In test case purification, an absolutely impure test case (a test case with at least one impure test constituent) necessarily results in at least one impure test fragment (one test fragment containing the impure test constituent) and zero or more pure test fragments.

Note that the goal of test case purification is not to replace the original test suite, but to enhance dynamic analysis techniques. Test case purification is done on-demand, just before executing a specific dynamic analysis. Consequently, it has no impact on future maintenance of test cases. In particular, new test cases potentially created by purification are not meant to be read or modified by developers.

4.2.2 B-Refactoring – A Test Suite Refactoring Approach

As mentioned in Section 4.2.1.3, test case purification can be implemented in different ways according to given dynamic analysis techniques. The prior work of my colleague [88] puri-

fies failing test cases based on assertions for fault localization (a dynamic analysis technique of identifying the root cause of a bug).

In this chapter, we present B-Refactoring, our approach to automatic refactoring a whole test suite. B-Refactoring is a generalized approach for dynamic analysis techniques and their various program elements.

4.2.2.1 Framework

B-Refactoring refactors a test suite according to a criterion defined with a set of specific program elements (e.g., `if` elements) in order to purify its execution (according to the execution signatures in Section 4.2.1). In a nutshell, B-Refactoring takes the original test suite and the requested set of program elements as input and generates purified test cases as output.

Fig. 4.2 illustrates the overall structure of our approach. We first monitor the execution of test cases on the requested set E of program elements. We record all the test cases that execute E and collect the execution signatures of test constituents in the recorded test cases. Second, we filter out pure test cases that already exist in the test suite. Third, we divide the remaining test cases into two categories: test cases with or without impure test constituents. For each category, we split the test cases into a set of test fragments. As a result, a new test suite is created, whose execution according to a set of program element is purer than the execution of the original test suite.

In this chapter, we consider a test constituent as a top-level statement in a test case. Examples of test constituents could be an assignment, a complete loop, a `try` block, a method invocation, etc. Our B-Refactoring does not try to split the statements that are inside a loop or a `try` branch in a test case.

4.2.2.2 Core Algorithm

Algorithm 4.3 describes how B-Refactoring splits a test case into a sequence of test fragments. As mentioned in Section 4.2.2.1, the input is a test case and a set of program elements to be purified while the output is a set of test fragments.

Algorithm 4.3 returns a minimized set of pure test fragments and a set of impure test fragments. In the algorithm, each impure test constituent is kept and directly transformed as an atomically impure test case that consists of only one constituent. The remaining continuous test constituents are clustered into several pure test fragments. Algorithm 4.3 consists of two major steps. First, we traverse all the test constituents to collect the last test constituent of each test fragment. Second, based on such collection, we split the test case into pure or impure test fragments. These test fragments can be directly treated as test cases for dynamic analysis applications.

Taking the test case in Table 4.1 as an example, we briefly describe the process of Algorithm 4.3. The traversal at Line 3 consists of only one program element according to Table 4.1. If only one of the `then` and `else` branches is executed, we record this branch for the following traversal of the test case (at Line 12). If a test constituent with a new execution signature appears, its previous test constituent is collected as the last constituent of a test fragment and the next test fragment is initialized (at Line 14). That is, c_3 and c_6 in Table 4.1 are collected as the last constituents. The end constituent of the test case is collected as the last constituent of the last test fragment (at Line 16), i.e., c_7. Lines from 7 to 9 are not run because there is no impure test constituent in Table 4.1. After the traversal of all the test

Figure 4.3: Splitting a test case into a set of test fragments according to a given set of program elements.

Input :
E, a set of program elements;
$t = \langle c_1, \ldots, c_n \rangle$, a test case with n test constituents;
D, an execution domain of the program elements in E.
Output:
Φ, a set of test fragments.

```
1  Let ℂ be an empty set of last constituents in fragments;
2  Let v =⊥ be a default execution signature;
3  foreach program element e ∈ E do
4      v =⊥;
5      foreach test constituent c_i in t (1 ≤ i ≤ n) do
6          if g(e, c_i) ∈ D^0 then   // Impure constituent
7              v =⊥;
8              ℂ = ℂ ∪ c_i − 1; // End of the previous fragment
9              ℂ = ℂ ∪ c_i; // Impure fragment of one constituent
10         else if g(e, c_i) ∈ D then   // Pure constituent
11             if v =⊥ then
12                 v = g(e, c_i);
13             else if v ≠ g(e, c_i) then   // v ∈ D
14                 ℂ = ℂ ∪ c_{i−1};
15                 v = g(e, c_i);
16 ℂ = ℂ ∪ c_n; // Last constituent of the last fragment
17 Let c^+ = c_1;
18 foreach test constituent c_j in ℂ do
19     φ = ⟨c^+, ..., c_j⟩; // Creation of a test fragment
20     Φ = Φ ∪ φ;
21     c^+ = c_{j+1};
```

constituents, Lines from 19 to 21 are executed to obtain the final three test fragments based on the collection of c_3, c_6, and c_7.

4.2.2.3 Validation of the Refactored Test Suite

Our algorithm for refactoring a test suite is meant to be semantics preserving. In other words, the refactored test suite should specify exactly the same behavior as the original one. We use mutation testing to validate that the refactored test suite is equivalent to the original one [99]. The idea of such validation is that all mutants killed by the original test suite must also be killed by the refactored one. Since in practice, it is impossible to enumerate all mutants, this validation is an approximation of the equivalence before and after B-Refactoring. We

Table 4.2: Projects in empirical evaluation.

Project	Description	Source LoC	#Test cases
Lang	A Java library for manipulating core classes	65,628	2,254
Spojo-core	A rule-based transformation tool for Java beans	2,304	133
Jbehave-core	A framework for behavior-driven development	18,168	457
Shindig-gadgets	A container to allow sites to start hosting social apps	59,043	2,002
Codec	A Java library for encoding and decoding	13,948	619
Total		159,091	5,465

will present the validation results in Section 4.3.4.

4.2.3 Implementation

We now discuss important technical aspects of B-Refactoring. Our tool, B-Refactoring, is implemented in Java 1.7 and JUnit 4.11. For test cases written in JUnit 3, we use a converter to adapt them to JUnit 4. Our tool is developed on top of Spoon [100], a Java library for source code transformation and analysis[13]. B-Refactoring handles a number of interesting cases and uses its own test driver to take them into account.

4.2.3.1 Execution Order

To ensure the execution order of test fragments, the B-Refactoring test driver uses a specific annotation `@TestFragment(origin, order)` to execute test fragments in a correct order. Test methods are automatically tagged with the annotation during purification. Examples of this annotation are shown in Fig. 4.1.c. Also, when test fragments use variables that were local to the test method before refactoring, they are changed as fields of the test class. In case of name conflicts, they are automatically renamed in a unique way.

4.2.3.2 Handling `setUp` and `tearDown`

Unit testing can make use of common setup and finalization code. JUnit 4 uses Java annotations to facilitate writing this code. For each test case, a `setUp` method (with the annotation `@Before` in JUnit 4) and a `tearDown` method (with `@After`) are executed before and after the test case, e.g., initializing a local variable before the execution of the test case and resetting a variable after the execution, respectively. In B-Refactoring, to ensure the same execution of a given test case before and after refactoring, we include `setUp` and `tearDown` methods in the first and the last test fragments. This is illustrated in Fig. 4.1.c.

4.2.3.3 Shared Variables in a Test Case

Some variables in a test case may be shared by multiple statements, e.g., one common variable in two assertions. In B-Refactoring, to split a test case into multiple ones, a shared variable in a test case is renamed and extracted as a class field. Then each new test case can access this variable; meanwhile, the behavior of the original test case is not changed. Experiments in Section 4.3.4 will also confirm the unchanged behavior of test cases.

[13] Spoon 2.0, `http://spoon.gforge.inria.fr/`.

Table 4.3: Purity of test cases for `if` elements according to the number of test cases, test constituents, and `if` elements.

Project	Test case							Test constituent			`if` element			
	#Total	Pure		Non-absolutely impure		Absolutely impure		Total	Impure		#Total	#Executed	Purely covered `if`	
		#	%	#	%	#	%		#	%			#	%
Lang	2,254	539	23.91%	371	16.46%	1,344	59.63%	19,682	5,705	28.99%	2,397	2,263	451	19.93%
Spojo-core	133	38	28.57%	5	3.76%	90	67.67%	999	168	16.82%	87	79	45	56.96%
Jbehave-core	457	195	42.67%	35	7.76%	227	49.67%	3,631	366	10.08%	428	381	230	60.37%
Shindig-gadgets	2,002	731	36.51%	133	6.64%	1,138	56.84%	14,063	6,610	47.00%	2,378	1,885	1,378	73.10%
Codec	619	182	29.40%	123	19.87%	314	50.73%	3,458	1,294	37.42%	507	502	148	29.48%
Total	5,465	1,685	30.83%	667	12.20%	3,113	56.96%	41,833	14,143	33.81%	5,797	5,110	2,252	44.07%

4.3 Empirical Study on Test Suite Refactoring

In this section, we evaluate our technique for refactoring test suites. This work addresses a novel problem statement: refactoring a test suite to enhance dynamic analysis. To our knowledge, there is no similar technique that can be used to compare against. However, a number of essential research questions have to be answered.

4.3.1 Projects

We evaluate our test suite refactoring technique on five open-source Java projects: Apache Commons Lang (Lang for short),[14] Spojo-core,[15] Jbehave-core,[16] Apache Shindig Gadgets (Shindig-gadgets for short),[17] and Apache Commons Codec (Codec for short).[18] These projects are all under the umbrella of respectful code organizations (three out of five projects by Apache[19]).

4.3.2 Empirical Observation on Test Case Purity

RQ1: What is the purity of test cases in our dataset?

We empirically study the purity of test cases for two types of program elements, i.e., `if` elements and `try` elements. The goal of this empirical study is to measure the existing purity of test cases for `if` and `try` before refactoring. The analysis for `if` will facilitate the study on software repair in Section 4.4.1 while the analysis for `try` will facilitate the study on dynamic verification of exception contracts in Section 4.4.2. We will show that applying refactoring can improve the purity for individual program elements in Section 4.3.3.

4.3.2.1 Protocol

We focus on the following metrics to present the purity level of test cases:

- *#Pure* is the number of pure test cases on all program elements under consideration;

[14] Apache Commons Lang 3.2, `http://commons.apache.org/lang/`.
[15] Spojo-core 1.0.6, `http://github.com/sWoRm/Spojo`.
[16] Jbehave-core, `http://jbehave.org/`.
[17] Apache Shindig Gadgets, `http://shindig.apache.org/`.
[18] Apache Commons Codec 1.9, `http://commons.apache.org/codec/`.
[19] Apache Software Foundation, `http://apache.org/`.

Table 4.4: Purity of test cases for `try` elements according to the number of test cases, test constituents, and `try` elements.

Project	Test case							Test constituent			`try` element			
	#Total	Pure		Non-absolutely impure		Absolutely impure		#Total	#Impure		#Total	#Executed	Purely covered `try`	
		#	%	#	%	#	%		#	%			#	%
Lang	2,254	295	13.09%	1,873	83.1%	86	3.81%	19,682	276	1.40%	73	70	35	50.00%
Spojo-core	133	52	39.10%	81	60.9%	0	0.00%	999	0	0.00%	6	5	5	100.00%
Jbehave-core	457	341	74.62%	91	19.91%	25	5.47%	3,631	29	0.80%	67	57	43	75.44%
Shindig-gadgets	2,002	1,238	61.84%	702	35.06%	62	3.10%	14,063	73	0.52%	296	244	221	90.57%
Codec	619	88	14.22%	529	85.46%	2	0.32%	3,458	2	0.06%	18	16	14	87.50%
Total	5,465	2,014	36.85%	3,276	59.95%	175	3.20%	41,833	380	0.91%	460	392	318	81.12%

- *#Non-absolutely impure* is the number of impure test cases (without impure test constituent);
- *#Absolutely impure* is the number of test cases that consist of at least one impure test constituent.

The numbers of test cases in these three metrics are mapped to the three categories (a), (b), and (c) of test cases in Fig 4.2, respectively.

For test constituents, we use the following two metrics, i.e., *#Total constituents* and *#Impure constituents*. For program elements, we use metric *#Purely covered program elements* as Definition 3) in Section 4.2.1.

We leverage B-Refactoring to calculate evaluation metrics and to give an overview of the purity of test suites for the five projects.

4.3.2.2 Results

We analyze the purity of test cases in our dataset with the metrics proposed in Section 4.3.2.1. Table 4.3 shows the purity of test cases for `if` elements. In the project Lang, 539 out of 2,254 (23.91%) test cases are pure for all the executed `if` elements while 371 (16.46%) and 1,344 (59.63%) test cases are impure without and with impure test constituents. In total, 1,658 out of 5,465 (30.83%) test cases are pure for the all the executed `if` elements. These results show that there is space for improving the purity of test cases and achieving a higher percentage of pure test cases.

As shown in the column *Test constituent* in Table 4.3, 33.81% of test constituents are impure. After applying test suite refactoring, all those impure constituents will be isolated in own test fragments. That is the number of absolutely impure constituents equals to the number of impure test cases after refactoring.

In Table 4.3, we also present the purity of test cases according to the number of `if` elements. In the project Lang, 2,263 out of 2,397 `if` elements are executed by the whole test suite. Among these executed `if` elements, 451 (19.93%) are purely covered. In total, among the five projects, 44.07% of `if` elements are purely covered. Hence, it is necessary to improve the purely covered `if` elements with our test case purification technique.

For `try` elements, we use the execution domain defined in Section 4.2.1.1 and compute the same metrics. Table 4.4 shows the purity of test cases for `try` elements. In Lang, 295 out of 2,254 (13.09%) test cases are always pure for all the executed `try` elements. In total, the percentage of always pure and absolutely impure test cases are 36.85% and 3.20%, respectively. In contrast to `if` elements in Table 4.3, the number of absolutely impure test cases in Spojo-core is zero. The major reason is that there is a much larger number of test cases in

Lang (2254), compared to Spojo-core (133). In the five projects, based on the purity of test cases according to the number of `try` elements, 81.12% `try` elements are purely covered.

Comparing the purity of test cases between `if` and `try`, the percentage of pure test cases for `if` elements and `try` elements are similar, 30.83% and 36.85%, respectively. In addition, the percentage of purely covered `try` elements is 81.12% that is higher than that of purely covered `if`, i.e., 44.07%. That is, 81.12% of `try` elements are executed by test cases with pure execution signatures but only 44.07% of `if` elements are executed by test cases with pure execution signatures. This comparison indicates that for the same project, different execution domains of input program elements result in different results for the purity of test cases. We can further improve the purity of test cases according to the execution domain (implying a criterion for purification) for a specific dynamic analysis technique.

Answer to RQ1: Only 31% (resp. 37%) of test cases are pure with respect to `if` elements (resp. `try` elements).

4.3.3 Empirical Measurement of Refactoring Quality

RQ2: Are test cases purer on individual program elements after applying our test suite refactoring technique?

We evaluate whether our test case purification technique can improve the execution purity of test cases. Purified test cases cover smaller parts of the control flow; consequently, they will provide better support to dynamic analysis tasks.

4.3.3.1 Protocol

To empirically assess the quality of our refactoring technique with respect to purity, we employ the following metrics (see Definition 3):

- *#Purely covered program elements* is the number of program elements, each of which is covered by all test cases with pure execution signatures;
- *#Program elements with at-least-one pure test case* is the number of program elements, each of which is covered by at least one test case with a pure execution signature.

For dynamic analysis, we generally aim to obtain a higher value of those two metrics after test suite refactoring. For each metric, we list the number of program elements before and after applying B-Refactoring as well as the improvement: absolute and relative ($\frac{\#After - \#Before}{\#Before}$).

4.3.3.2 Results

The first part of Table 4.5 shows the improvement of test case purity for `if` elements before and after applying B-Refactoring. For the project Lang, 2,263 `if` elements are executed by the whole test suite. After applying B-Refactoring to the test suite, 1,250 (from 451 to 1,701) `if` elements are changed to be purely covered. The relative improvement reaches 277.16% (1,250/451). Moreover, 884 (from 1,315 to 2,199) `if` elements are changed to be covered with at-least-one pure test case.

Table 4.5: Test case purity by measuring the number of purely covered `if`s and `try`s. The number of purely covered program elements increases after applying test case purification with B-Refactoring.

Project	#Executed `if`	Purely covered `if`				`if` with at-least-one pure test case			
		#Before	#After	Improvement		#Before	#After	Improvement	
				#	%			#	%
Lang	2,263	451	1,701	1,250	277.16%	1,315	2,199	884	67.22%
Spojo-core	79	45	54	9	20.00%	75	78	3	4.00%
Jbehave-core	381	230	262	32	13.91%	347	355	8	2.31%
Shindig-gadgets	1,885	1,378	1,521	143	10.38%	1,842	1,856	14	0.76%
Codec	502	148	208	60	40.54%	411	441	30	7.30%
Total for `if`s	5,110	2,252	3,746	1,494	66.34%	3,990	4,929	939	23.53%

Project	#Executed `try`	Purely covered `try`				`try` with at-least-one pure test case			
		#Before	#After	Improvement		#Before	#After	Improvement	
				#	%			#	%
Lang	70	35	58	23	65.71%	61	68	7	11.48%
Spojo-core	5	5	5	0	0.00%	5	5	0	0.00%
Jbehave-core	57	43	44	1	2.33%	54	54	0	0.00%
Shindig-gadgets	244	221	229	8	3.62%	241	242	1	0.41%
Codec	16	14	16	2	14.29%	16	16	0	0.00%
Total for `try`s	392	318	352	34	10.69%	377	385	8	2.12%

For all five projects, 1,494 purely covered `if` elements as well as 939 at-least-one purely covered `if` elements are obtained by applying B-Refactoring. These results indicate that the purity of test cases for `if` elements is highly improved via test case purification. Note that the improvement on Lang is higher than that on the other four projects. A possible reason is that Lang behaves in a complex implementation and the original design of the test suite is only for software testing and maintenance but not for the usage in a dynamic analysis technique.

Similarly, the second part of Table 4.5 shows the improvement for `try` elements before and after applying B-Refactoring. In Lang, 23 (from 35 to 58) `try` elements are changed to be purely covered after applying B-Refactoring; 7 (from 61 to 68) `try` elements are changed to at-least-one purely covered `try` elements. For all five projects, 34 (from 318 to 352) `try` elements change to be purely covered after test case purification while 8 (from 377 to 385) `try` elements are improved in the sense that they become purely covered by at-least-one pure test cases. Note that for Spojo-core, no value is changed before and after test case purification due to the small number of test cases.

Answer to RQ2: After test suite refactoring, `if` and `try` elements are more purely executed. The purely covered `if` and `try` are improved by 66% and 11%, respectively.

4.3.4 Mutation-based Validation for Refactored Test Suites

RQ3: Do the automatically refactored test suites have the same fault revealing power as the original ones?

In this section, we employ mutation testing to validate that a refactored test suite has the same behavior as the original one [101], [102].

4.3.4.1 Protocol

For each project, we generate mutants by injecting bugs to the program code. A mutant is *killed* by a test suite if at least one test case fails on this mutation. To evaluate whether a refactored test suite behaves the same as the original one, the two test suites should satisfy either of the two following rules: one mutant is killed by both the original test suite and the refactor one; or one mutant is not killed by both test suites. For three projects of our dataset, Lang, JBehave-core, and Codec, we randomly select 100 mutants per project. For each mutant, we individually run the original test suite and the purified test suite to check whether the mutant is killed.

4.3.4.2 Results

Experimental results shows that both the two rules in Section 4.3.4.1 are satisfied for all the mutants. In details, 81 mutants in Lang, 61 mutants in JBehave-core, and 89 mutants in Codec are killed by both original and purified test suites while 18, 33, and 10 mutants are alive in both original and purified test suites, respectively. Moreover, 1, 6, and 1 mutants, respectively in three projects, lead both the original and refactored test suites to an infinite loop. To sum up, mutation-based validation for refactored test suites shows that our technique can provide the same behavior for the refactored test suites as the original test suites.

Answer to RQ3: The test suites automatically refactored by B-Refactoring catch the same mutants as the original ones.

4.4 Improving Dynamic Analysis Using Test Suite Refactoring

We apply our test suite refactoring approach, B-Refactoring, to improve two typical dynamic analysis techniques, automatic repair and exception contract analysis.

4.4.1 Test Suite Refactoring for Automatically Repairing Three Bugs

RQ4: Does B-Refactoring improve the automatic program repair of Nopol [6]?

To repair `if`-condition bugs, Nopol suffers from the ambiguous execution of test cases, each of which covers both `then` and `else` branches. In this section, we leverage test case purification to eliminate the ambiguity of test case execution. In other words, we refactor test cases to convert original impure test cases into purified ones to assist automatic repair.

4.4.1.1 Protocol

We present a case study on three real-world bugs in Apache Commons Lang.[20] All the three bugs are located in `if`-conditions. However, Nopol cannot directly fix these bugs because of the impurity of test cases. Thus, we use B-Refactoring to obtain purified test cases. This enables Nopol to repair those previously-unfixed bugs.

[20] For more details, visit `https://fisheye6.atlassian.com/changelog/commons?cs=137371`, `https://fisheye6.atlassian.com/changelog/commons?cs=137552`, and `https://fisheye6.atlassian.com/changelog/commons?cs=904093`.

Table 4.6: Evaluation of the effect of purification for automatic repair for `if`-condition bugs. Traces of test cases after applying B-Refactoring (last column) enable a repair approach to find patches as the second column.

ID	Patch	#Test cases	
		Before	After
137371	`lastIdx <= 0`	1	2
137552	`len < 0 || pos > str.length()`	1	4
904093	`className == null || className.length() == 0`	2	3

```
1  String chopNewline(String str) {
2    int lastIdx = str.length() -1;
3
4    // PATCH: if (lastIdx <= 0) {
5    if (lastIdx == 0)
6      return "";
7    char last = str.charAt(lastIdx);
8    if (last == '\n')
9      if (str.charAt(lastIdx -1) == '\r')
10       lastIdx--;
11   else
12     lastIdx++;
13   return str.substring(0, lastIdx);
14 }
```

(a) Buggy program

```
1  void testChopNewLine(){
2    ...
3    assertEquals(FOO + "\n" + FOO,
4        StringUtils.chopNewline(FOO
5        + "\n" + FOO));
6    assertEquals(FOO + "b\n",
7        StringUtils.chopNewline(FOO
8        + "b\n\n"));
9
10 // B-refactoring splits here
11
12   assertEquals("",
13       StringUtils.chopNewline("\n"));
14 }
```

(b) Test case

Figure 4.4: Code snippets of a buggy program and a test case. The buggy `if` condition is at Line 5 of Fig. 4.4.a; the test case in Fig. 4.4.b executes both the `then` and `else` branches of the buggy `if`. Then B-Refactoring splits the test case into two test cases (at Line 10 in Fig. 4.4.b).

4.4.1.2 Case Study 1

Table 4.6 recapitulates three bugs, their patches and the created test cases. In total, nine pure test cases are obtained after applying test case purification to the original four test cases. Note that only the executed test cases for the buggy `if`s are listed, not the whole test suite. We show how test suite refactoring influences the repair of the bug with ID 137371 as follows.

Fig. 4.4 shows a code snippet with a buggy `if` condition at Line 5 of bug with ID 137371. In Fig. 4.4.a, the method `chopNewLine` aims to remove the line break of a string. The original `if` condition missed the condition of `lastIdx < 0`. In Fig. 4.4.b, a test case `testChopNewLine` targets this method. We show three test constituents, i.e., three assertions, in this test case (other test constituents are omitted for saving the space). The first two assertions cover the `then` branch of the `if` condition at Line 5 of `chopNewLine` while the last assertion covers the `else` branch. Such a test case will lead the repair approach to an ambiguous behavior; that is, the repair approach cannot find the covered branch of this test

```
1  String mid(String str, int pos, int len) {
2    if (str == null)
3      return null;
4
5    // PATCH:
6    // if (len < 0 || pos > str.length())
7    if (pos > str.length())
8      return "";
9
10   if (pos < 0)
11     pos = 0;
12   if (str.length() <= (pos + len))
13     return str.substring(pos);
14   else
15     return str.substring(pos, pos + len);
16 }
```

(a) Buggy program

```
1  void testMid_String() {
2    ...
3
4  // TODO: Split here
5    assertEquals("b", StringUtils
6        .mid(FOOBAR, 3, 1));
7    ...
8
9  // TODO: Split here
10   assertEquals("", StringUtils
11       .mid(FOOBAR, 9, 3));
12
13 // TODO: Split here
14   assertEquals(FOO, StringUtils
15       .mid(FOOBAR, -1, 3));
16 }
```

(b) Test case

Figure 4.5: Code snippets of a buggy program and a test case in Case study 2. The buggy `if` statement is at Line 7 in Fig. 4.5.a while the test case in Fig. 4.5.b executes the `then`, the `else`, the `then`, and the `else` branches of the buggy statement, respectively. Then B-Refactoring splits the test case into four test cases.

Table 4.7: Test suites after test suite refactoring improves exception contracts by decreasing the number of unknown `try`s.

Project	Before			After			Improvement on #unknown	
	#Source-independent	#Source-dependent	#Unknown	#Source-independent	#Source-dependent	#Unknown	#	%
Lang	23	5	22	37	6	7	15	68.18%
Spojo-core	1	0	0	1	0	0	0	n/a
Jbehave-core	7	2	33	8	2	32	1	3.03%
Shindig-gadgets	30	12	38	31	13	36	2	5.26%
Codec	8	0	2	10	0	0	2	100.00%
Total	69	19	95	87	21	75	20	21.05%

case. Hence, a repair approach cannot generate a patch for this bug.

Based on B-Refactoring, our test case purification technique can split the test case into two test cases, as shown at Line 10 in Fig. 4.4.b. We replace the original test case with two new test cases after B-Refactoring. Then the repair approach can generate a patch, which is the same as the manual patch at Line 4 in Fig. 4.4.a.

Answer to RQ4: B-Refactoring improves the repairability of the Nopol program repair technique on three real-world bugs, which cannot be fixed before applying B-Refactoring.

4.4.1.3 Case study 2

A code snippet in Fig. 4.5 presents an `if`-condition bug with ID 137552 in Apache Commons Lang. In Fig. 4.5.a, the method `mid` is to extract a fixed-length substring from a given posi-

```
1 String getPackageName(Class cls) {
2   if (cls == null)
3     return StringUtils.EMPTY;
4   return getPackageName(cls.getName());
5 }
6
7 String getPackageName(String className){
8
9   // PATCH: if (className == null
10        || className.length() == 0)
11
12  if (className == null)
13    return StringUtils.EMPTY;
14  while (className.charAt(0) == '[')
15    className = className.substring(1);
16  if (className.charAt(0) == 'L' &&
17      className.charAt(className
18      .length() -1) == ';')
19    className = className.substring(1);
20  int i = className.lastIndexOf(
21      PACKAGE_SEPARATOR_CHAR);
22  if (i == -1)
23    return StringUtils.EMPTY;
24  return className.substring(0, i);
25 }
```

(a) Buggy program

```
1 void test_getPackageName_Class() {
2   assertEquals("java.util", ClassUtils
3       .getPackageName(Map.Entry.class));
4   assertEquals("", ClassUtils
5       .getPackageName((Class)null));
6   assertEquals("java.lang", ClassUtils
7       .getPackageName(String[].class));
8   ...
9 }
10
11 void test_getPackageName_String() {
12   ...
13   assertEquals("java.util", ClassUtils
14       .getPackageName(
15       Map.Entry.class.getName()));
16
17   // TODO: Split here
18   assertEquals("", ClassUtils
19       .getPackageName(""));
20 }
21
22   // Manually added test case
23   // to ensure the original condition
24 void test_manually_add() {
25   assertEquals("", ClassUtils
26       .getPackageName(null));
27 }
```

(b) Test cases

Figure 4.6: Code snippets of a buggy program and a test case in Case study 3. The buggy `if` statement is Line 12 of Fig. 4.6.a while two test case in Fig. 4.6.b executes `then` and `else` branches of the buggy statement. B-Refactoring splits the second test case into two test cases and keeps the first test case. The last test case `test_manually_add` is manually added for explanation.

tion. The original `if` condition at Line 7 did not deal with the condition of `len < 0`, which is expected to return an empty string. In Fig. 4.5.b, a test case `testMid_String` targets this method. Three assertions are shown to explain the coverage of branches. Two assertions at Line 5 and Line 14 cover the `else` branch of the `if` condition while the other assertion at Line 10 covers the `else` branch. A repair approach, like Nopol, cannot generate a patch for this bug because the test case `testMid_String` covers both branches of the `if` condition at Line 7 in the method `mid`.

We apply B-Refactoring to split the test case into four test cases, as shown at Lines 4, 9, and 13 in Fig. 4.5.b. Such splitting can separate the coverage of `then` and `else` branches; that is, each new test cases only covers either the `then` or `else` branch. Then the repair approach can generate a patch, `! (pos < str.length() && len >= 0)`, which is equivalent to the manual patch at Line 6 in Fig. 4.5.a.

4.4.1.4 Case study 3

This bug is with ID 904093 in Apache Commons Lang. Fig. 4.6 shows a code snippet with a buggy `if` condition at Line 12. In Fig. 4.6.a, two methods `getPackageName(Class)` and `getPackageName(String)` work on extracting the package name of a class or a string. The original `if` condition missed checking the empty string, i.e., the condition of `className . length() == 0`. In Fig. 4.6.b, two test cases examine the behavior of these two methods. For the first test case `test_getPackageName_Class`, we present three assertions. We do not refactor this test case because this test case is pure (the first and the third assertion execute the `else` branch while the second assertion does not execute any branch). For the second test case `test_getPackageName_String`, two assertions are shown. The first one is passing while the second is failing. Thus, we split this test case into two test cases to distinguish passing and failing test cases.

Based on B-Refactoring, we obtain three test cases, as shown at Line 17 in Fig. 4.6.b. Then the repair approach can generate a patch as `className.length() == 0`. Note that this patch is different from the real patch because the condition `className == null` is ignored. The reason is that in the original test suite, there exists no test case that validates the `then` branch at Line 13.

To generate the same patch as the real patch at Line 9, we manually add one test case `test_manually_add` at Line 24 in Fig. 4.6.b. This test case ensures the behavior of the condition `className == null`. Based on this manually added test case and the test cases by B-Refactoring, the repair approach can generate a patch that is the same as the real one.

Summary. In summary, we empirically evaluate our B-Refactoring technique on three real-world `if`-condition bugs from Apache Commons Lang. All these three bugs cannot be originally repaired by the repair approach, Nopol. The reason is that one test case covers both the `then` and `else` branches. Then Nopol cannot decide which branch is covered and cannot generate the constraint for this test case. With B-Refactoring, we separate test cases into pure test cases to cover only the `then` or `else` branch. Based on the test cases after applying B-Refactoring, the first two bugs are fixed. The generated patches are the same as the manually-written patches. For the last bug, one test case is ignored by developers in the original test suite. By adding one ignored test case, this bug can also be fixed via the test suite after B-Refactoring.

To sum up, we have shown that our test case purification approach enables to automatically repair three previously-unfixed bugs, by providing a refactored version of the test suite that produces traces that are optimized for the technique under consideration.

4.4.2 Test Suite Refactoring for Exception Contract Analysis

RQ5: Does B-Refactoring improve the efficiency of the SCTA contract verification technique [92]?

In this section, we employ an existing dynamic analysis technique of exception contracts called Short-Circuit Testing Algorithm (SCTA), presented in the previous chapter3. SCTA aims to verify an exception handling contract called source-independence, which states that `catch` blocks should work in all cases when they catch an exception. Assertions in a test suite are used to verify the correctness of test cases. The process of SCTA is as follows. To analyze exception contracts, exceptions are injected at the beginning of `try` elements to

trigger the `catch` branches; meanwhile, a test suite is executed to record whether a `try` or `catch` branch is covered by each test case. *SCTA requires that test cases execute only the `try` or the `catch`.*

However, if both `try` and `catch` branches are executed by the same test case, SCTA cannot identify the coverage of the test case. In this case, the logical predicates behind the algorithm state that the contracts cannot be verified because the execution traces of test cases are not pure enough with respect to `try` elements. According to the terminology presented in this chapter, we call such test cases covering both branches *impure*. If all the test cases that execute a `try` element are impure, no test cases can be used for identifying the source-independence. To increase the number of identified `try` elements and decrease the number of unknown ones, we leverage B-Refactoring to refactor the original test cases into purer test cases.

4.4.2.1 Protocol

We apply test case purification on the five projects in Section 4.3.1. The goal of this experiment is *to evaluate how many `try` elements are recovered from unknown ones*. We apply B-Refactoring to the test suite before analyzing the exception contracts. That is, we first refactor the test suite and then apply SCTA on the refactored version.

We analyze exception contracts with the following metrics:

- *#Source-independent* is the number of verified source-independent `try` elements;
- *#Source-dependent* is the number of verified source-dependent `try` elements;
- *#Unknown* is the number of unknown `try` elements, because all the test cases are impure.

The last metric is the key one in this experiment. The goal is to decrease this metric by refactoring, i.e., to obtain less try-catch blocks, whose execution traces are too impure to apply the verification algorithm. Note that the sum of the three metrics is constant before and after applying test suite refactoring.

4.4.2.2 Results

We investigate the results of the exception contract analysis before and after B-Refactoring.

Table 4.7 presents the number of source-independent `try` elements, the number of source-dependent `try` elements, and the number of unknown ones. Taking the project Lang as an example, the number of unknown `try` elements decreases by 15 (from 22 to 7). This enables the analysis to prove the source-independence for 14 more `try` (from 23 to 37) and to prove source-dependence for one more (from 5 to 6). That is, by applying test case purification to the test suite in project Lang, we can detect whether these 68.18% (15/23) `try` elements are source-independent or not.

For all the five projects, 21.05% (20 out of 95) of `try` elements are rescued from unknown ones. This result shows that B-Refactoring can refactor test suites to cover simple branches of `try` elements. Such refactoring helps the dynamic analysis to identify the source independence.

Answer to RQ5: Applying B-Refactoring to test suites improves the ability of verifying the exception contracts of SCTA. 21% of unknown exception contracts are reduced.

4.5 Threats to Validity

We discuss threats to the validity of our B-Refactoring results.

Generality. We have shown that B-Refactoring improves the efficiency of program repair and contract verification. However, the two considered approaches stem from our own research. This is natural, since our expertise in dynamic analysis makes us aware of the nature of the problem. For further assessing the generic nature of our refactoring approach, future experiments involving other dynamic analysis techniques are required.

Internal validity. Test code can be complex. For example, a test case can have loops and internal mocking classes. In our implementation, we consider test constituents as top-level statements, thus complex test constituents are simplified as atomic ones. Hence, B-Refactoring does not process these internal statements.

Construct validity. Experiments in this chapter mainly focus on `if` and `try` program elements. Both of these program elements can be viewed as a kind of branch statements. Our proposed work can also be applied to other elements, like method invocations (in Section 4.2.1.1). To show more evidence of the improvement on dynamic analysis, more experiments could be conducted for different program elements.

4.6 Discussion

In this chapter, we use test case refactoring to better locate the cause of errors. Test refactoring has already been used in other different goals presented below.

Test code refactoring [96] is a general concept of making test code better understandable, readable, or maintainable. Based on 11 test code smells, Deursen et al. [96] first propose the concept of test case refactoring as well as 6 test code refactoring rules, including reducing test case dependence and adding explanation for assertions.

Guerra & Fernandes [103] define a set of 15 representation rules for 3 different categories of test code refactoring (Refactoring inside a test class, refactoring inside a test method and structural refactoring of test classes). The respect of those rules theoretically guarantees the equivalence of the test suites before and after the refactoring.

Extension on this work by Van Deursen & Moonen [104] and Pipka [105] proposes how to refactor test code for the *test first* rule in extreme programming.

Existing known patterns in refactoring are applied to test cases to achieve better-designed test code. Chu et al. [98] propose a pattern-based approach to refactoring test code to keep the correctness of test code and to remove the bad code smells.

Alves et al. [106] presents a pattern-based refactoring technique used on test code. It allows to make better regression testing via test case selection and prioritization instead of running the whole regression test suite each time. For two given versions of a program plus the test suite of the program, they select the test cases impacted by the changes between the two versions, and order them according to the probability of revealing a regression. The new test cases can assist a specific software task, e.g., splitting test cases to execute single

branches in if elements for software repair and to trigger a specific status of try elements for exception handling. They use a case study to demonstrate the feasibility of the approach.

Xuan et al. [88] propose a test-case purification approach for fault localization. This purification technique allows bug-finding tools to have better results when it uses the purified version of the test suite. They perform test code modification to create test cases containing only one assertion. They evaluate their approach using 6 existing localization techniques on 6 real-life programs and by injecting a total of 1800 faults. Their results show that the wasted effort (absolute number of statements to be examined before finding the faulty one) is strongly reduced in most of the cases (from 10% less statement to be examined up to 2 times less).

4.7 Conclusion

This chapter addresses test suite refactoring. We propose B-Refactoring, a technique to split test cases into small fragments in order to increase the efficiency of dynamic program analysis. Our experiments on five open-source projects show that our approach effectively improves the purity of test cases. We show that applying B-Refactoring to existing analysis tasks, namely repairing `if`-condition bugs and analyzing exception contracts, enhances the capabilities of these tasks. The work presented in this chapter improves the capability of the try-catch categorization algorithm by more than 20%. With those first two chapters, we address the characterization of exception behavior and exception handling. Using this knowledge, in the next chapter, we address the problem of debugging exception bugs.

Chapter 5

Casper: Debugging Null Dereferences with Ghosts and Causality Traces

The contributions of this chapter are:

- A definition of causality traces for null dereference errors.
- A set of source code transformations designed and tailored for collecting the causality traces.
- Casper, an implementation in Java of our technique.
- An evaluation of our technique on real null dereference bugs collected over 6 large open-source projects.

Related publications:

- Casper: Debugging Null Dereferences with Ghosts and Causality Traces, poster format, ACM SRC, ICSE 2015.

This work is currently under submission at the main track of the conference Source Code Analysis and Manipulation.

In the previous chapters, we presented a technique to specify exception bugs and a system to improve their detection. The second step of bug fixing is to understand why the bug happens. This is called in the literature "root cause analysis". Hence, in this chapter, considering we know that an exception is a bug and having detecting it, we want to obtain the maximum of information on it before trying to fix it. To do it, we introduce "causality traces" which consists in the information on why a bug happens. We present a tool which automatically constructs those causality traces for a specific kind of exception: null dereferences.

Fixing these errors requires understanding their root cause, a process that we call causality analysis. Computers are mindless accountants of program execution, yet do not track the data needed for causality analysis. This work proposes to harness computers to this task. We introduce *"causality traces"*, execution traces augmented with the information needed to

```
1 Exception in thread "main" java.lang.NullPointerException
2   at [..].BisectionSolver.solve(88)
3   at [..].BisectionSolver.solve(66)
4   at ...
```

Listing 5.1: The standard stack trace of a real null dereference bug in Apache Commons Math

reconstruct a causal chain from a root cause to an execution error. We construct causality traces over *"ghosts"*, an abstract data type that can replace a programming language's special values, like null or NaN. Ghosts replace such values and track operations applied to itself, thereby collecting a causality trace whose analysis reveals the root cause of a bug. To demonstrate the feasibility and promise of causality traces, we have instantiated ghosts for providing developers with causality traces for null deference errors, they are "null ghosts".

We know that null dereferences are frequent runtime errors. Li et al. substantiated this conventional wisdom, finding that 37.2% of all memory errors in Mozilla and and Apache are null dereferences [107]. Kimura et al. [108] found that there are between one and four null checks per 100 lines of code on average. A null dereference runtime error occurs when a program tries to read memory using a field, parameter, or variable that points to nothing — "null" or "none", depending on the language. For example, on October 22 2009, a developer working on the Apache Common Math open source project encountered an null pointer exception and reported it as bug #305[21].

In low-level, unmanaged runtime environments, like assembly or C/C++, null dereferences result in a dirty crash, e.g. a segmentation fault. In a high-level, managed runtime environment such as Java, .NET, etc., a null dereference triggers an exception. Programs often crash when they fail to handle null dereference exceptions properly [39].

When debugging a null dereference, the usual point of departure is a stack trace that contains all the methods in the execution stack at the point of the dereference. This stack trace is decorated with the line numbers where each method was called. Listing 5.1 gives an example of such a stack trace and shows that the null dereference happens at line 88 of BisectionSolver.

Unfortunately, this stack trace only contains a partial snapshot of the program execution when the null dereference occurs, and not its root cause. In Listing 5.1, the stack trace says that a variable is null at line 88, but not when and what assigned "null" to the variable. Indeed, there may be a large gap between the symptom of a null dereference and its root cause [39]. In our evaluation, we present 7 cases where the patch for fixing the root cause of a null dereference error is not in any of the stack trace's method. This gap exactly is an instance of Eisenstadt's cause/effect chasm [109] for a specific defect class: null dereference errors.

This "null dereference cause/effect chasm" has two dimensions. The first is temporal: the symptom may happen an arbitrarily long time after its root cause, e.g. the dereference may happen ten minutes and one million method executions after the assignment to null. In this case, the stack trace is a snapshot of the execution at the time of the symptom, not at the time of the root cause. The second is spatial: the location of the symptom may be arbitrarily far from the location of the root cause. For example, the null dereference may be in package foo and class A while the root cause may be in package bar and class B. The

[21] `https://issues.apache.org/jira/browse/MATH-305`

```
1 Exception in thread "main" java.lang.NullPointerException
2 For parameter : f // symptom
3    at [..].BisectionSolver.solve(88)
4    at [..].BisectionSolver.solve(66)
5    at ...
6 Parameter f bound to field UnivariateRealSolverImpl.f2
7    at [..].BisectionSolver.solve(66)
8 Field f2 set to null
9    at [..].UnivariateRealSolverImpl.<init>(55) // cause
```

Listing 5.2: What we propose: a causality trace, an extended stack trace that contains the root cause.

process of debugging null dereferences consists of tracing the link in space and time between the symptom and the root cause.

A *causality trace* captures the complete history of the propagation of a null value that is incorrectly dereferenced. Listing 5.2 contains such a causality trace. In comparison to Listing 5.1, it contains three additional pieces of information. First, it gives the exact name, here f, and kind, here parameter (local variable or field are other possibilities), of the null variable. Second, it explains the origin of the parameter, the call to solve at line 66 with field f2 passed as parameter. Third, it gives the root cause of the null dereference: the assignment of null to the field f2 at line 55 of class UnivariateRealSolverImpl. Our causality traces contain several kinds of trace elements, of which Listing 5.2 shows only three: the name of the wrongly dereferenced variable, the flow of nulls through parameter bindings, and null assignment. Section 5.1 details the rest.

In this chapter, we present a novel tool, called Casper, to collect null causality traces. The tool is going to be used at debugging time by developers. It takes as input the program under debug and a main routine that triggers the null dereference. It then outputs the causality trace of the null dereference. It first instruments the program under debug by replacing nulls with "ghosts" that track causal information during execution. We have named our tool Casper, since it injects "friendly" ghosts into buggy programs. To instrument a program, Casper applies a set of 11 source code transformations tailored for building causal connections. For instance, o = externalCall() is transformed into o = NullDetector. check(externalCall()), where the method check stores causality elements in a null ghost (subsection 5.1.2) and assigns it to o if externalCall returns null. Section 5.1.3 details these transformations.

We evaluate our contribution Casper by providing and assessing the causality traces of 14 real null dereference bugs collected over six large, popular open-source projects. We collected these bugs from these project's bug reports, retaining those we were able to reproduce. Casper constructs the complete causality trace for 13 of these 14 bugs. For 11 out of these 13 bugs, the causality trace contains the location of the actual fix made by the developer.

Furthermore, we check that our 11 source code transformations do not change the semantics of the program relative to the program's test suite, by running the program against that test suite after transformation and confirming that it still passes. The limitations of our approach are discussed in subsection 5.1.5 and its overhead in paragraph 5.2.3.4.3.

5.1 Null Debugging Approach

Casper tracks the propagation of nulls used during application execution in a causality trace. A *null dereference causality trace* is the sequence of language constructs traversed during execution from the source of the null to its erroneous dereference. By building this trace, Casper generalizes dynamic taint analysis to answer not only whether a null can reach a dereference, but *how* a dereference is reached[22]. These traces speed the localization of the root cause of null dereferences errors.

Our idea is to replace nulls with objects whose behavior, from the application's point of view, is same as a null, except that they store a causality trace, defined in subsection 5.1.1. We called these objects *null ghosts* and detail them in subsection 5.1.2. Casper rewrites the program under debug to use null ghosts and to store a null's causality trace in those null ghosts (subsection 5.1.3). Finally, we discuss Casper's realization in subsection 5.1.5. We instantiated Casper in Java and therefore tailored our presentation in this section to Java.

5.1.1 Null Dereference Causality Trace

To debug a complex null dereference, the developer has to understand the history of a guilty null from its creation to its problematic dereference. She has to know the details of the null's propagation, i.e. why and when each variable became null at a particular location. We call this history the "null causality trace" of the null dereference.

Definition 5.1.1 *A* null dereference causality trace *is the temporal sequence of language constructs traversed by a dereferenced null.*

Developers read and write source code. Thus, source code is the natural medium in which developers reason about programs for debugging. In particular, a null propagates through moves or copies, which are realized via constructs like assignments and parameter binding. This is why Casper defines causal links in a null causality trace in terms of traversed language constructs. Table 5.1 depicts language constructs through which nulls originate, propagate, and trigger null pointer exceptions.

Therefore, nulls can originate in hard-coded null literals (L), and in external library return (P_i) or callbacks (P_e). In our causality abstraction, these links are the root causes of a null dereference. Recall that Java forbids pointer arithmetic, so we do not consider this case.

A null propagates through method parameters, returns (R), and unboxing (U). With the least specificity, a null can propagate through source level assignment (A). **D** denotes the erroneous dereference of a null.

When nulls are passed as parameter, they can be detected at parameter binding bound at a call site (P_i) and method entry (P_e). The reasons are twofold. First, we don't make any assumption on the presence, the observability and the manipulability of library code, so even in the presence of external libraries, we can trace that nulls are sent to them (P_i). Second, it enables to decipher polymorphism in the traces, we trace the actual method that has been called.

Let us consider the snippet "x = foo(bar())); ... x.field" and assume that x.field throws an NPE. The resulting null causality trace is R-R-A-**D** (return return assignment dereference).

[22]We take the source of a null to be a given, and do not concern ourselves with its cause. Under this assumption, Casper's traces, which answer the question of how a dereference is reached, are causal.

Mnemonic	Description	Examples

L	null literal	x = null;
		Object x; // Implicit null
		return null;
		foo(null);

P_e	null at entry	void foo(int x) { . . . }
P_i	null at invocation	foo(e)

R	null return	x = foo()//foo returns null
		foo().bar()
		bar(foo())

U	unboxed null	Integer x = e; int y = x
A	null assignment	x = e;
D	**null dereference**	x.foo() x.field
X	**external call**	lib.foo(e)

Table 5.1: Null-propagating language constructs. We use e to denote an arbitrary expression. In all cases but X, where e appears, a null propagates only if e evaluates to null. A is generic assignment; it is the least specific null-propagating language construct.

Here, the root cause is the return of the method bar[23]. The trace of Listing 5.2, discussed in introduction, is L-A-P_e-P_i-**D**. L and A are redundant in this case, since a single assignment statement directly assigns a null to the field f2; P_e and P_i are also redundant since no library call is involved. Post-processing removes such redundancy in a causality trace before Casper presents the trace to a user.

Casper decorates the L, A, P_i, and U "links" in a null dereference causality trace with the target variable name and the signature of the expression assigned to the target. For each causal link, Casper also collects the location of the language constructs (file, line) as well as the name and the stack of the current thread. Consequently, a causality trace contains a temporally ordered set of information and not just the stack at the point in time of the null dereference. In other words, a causality trace contains a null's root cause and not only the stack trace of the symptom.

A causality trace is any chain of these causal links. A trace *a*) starts with a L or, in the presence of an external library, with R or P_e; *b*) may not end with a dereference (if the null pointer exception is caught, the null can continue to propagate); and *c*) may contain a return, not preceded by a method parameter link, when a void external method returns null. A

[23] The root cause is not somewhere above the return in bar's body or the causality trace would necessarily be longer.

Method	Explanation
nullAssign(x, position)	logs whether x is null at this assignment, returns x if x is a valid object or a ghost, or a new ghost if x is null
nullParam(x, position)	logs whether x is null when passed as parameter, returns x if x is a valid object or a ghost, or a new ghost if x is null
nullPassed(x, position)	logs whether x is null when received as parameter at this position, returns void
nullReturn(x, position)	logs whether x is null when returned at this position, returns x if x is a valid object or a ghost, or a new ghost if x is null
exorcise(x, position)	logs whether x is null when passed as parameter to a library call at this position, returns "null" if x is a ghost, or x
nullUnbox(x, position)	throws a null pointer exception enriched with a causality trace if a null ghost is unboxed
nullDeref(x, position)	throws a null pointer exception enriched with a causality trace if a field access is made on a null ghost

Table 5.2: Explanations of the Methods Injected with Code Transformation.

causality trace can be arbitrarily long.

5.1.2 Null Ghosts

The special value null is the unique bottom element of Java's non primitive type lattice. Redefining null to trace the propagation of nulls during a program's execution, while elegant, is infeasible, since it would require the definition and implementation of a new language, with all the deployment and breakage of legacy applications that entails.

For sake of applicability, we leave our host language, here Java, untouched and we use a vanilla unmodified Java virtual machine. We use rewriting to create a *null ghost* to "haunt" each class defined in a codebase. A null ghost is an object that 1) contains a null causality trace and 2) has the same observable behavior as a null value. To this end, a ghost class contains a queue and overrides all methods of the class it haunts to throw null pointer exceptions.

Listing 5.3 illustrates this idea. Casper creates the ghost class MyGhostClass that extends the application class MyClass. All methods defined in the application type are overridden in the new type (e.g., `sampleMethod`) as well as all other methods from the old class (See subsection 5.1.5). The new methods completely replace the normal behavior and have the same new behavior. First, the call to `computeNullUsage` enriches the causality trace with a causal element of type **D** by stating that this null ghosts was dereferenced. Then, it acts as if one has dereferenced a null value: it throws a CasperNullPointerException (a special version of the Java error thrown when one calls a method on a null value called NullPointerException), which is discussed next. Also, a null ghost is an instance of the marker interface NullGhost. This marker interface will be used later to keep the same execution semantics between real null and null ghosts.

5.1.3 Casper's Transformations

Casper's transformations instrument the program under debug to detect nulls and construct

```
// original type
public MyClass{
  private Object o;
  public String sampleMethod(){
    ...
} }

// corresponding generated type
public MyGhostClass extends MyClass{
  public String sampleMethod(){
    // enriches the causality trace to log
    // that this null ghosts was dereferenced
    computeNullUsage();
    throw new CasperNullPointerException();
  }
  ... // for all methods incl. inherited ones
}
```

Listing 5.3: For each class of the program under debug, Casper generates a null ghost class to replace nulls.

null dereference causality traces dynamically, while preserving its semantics.

5.1.3.1 Overview

Equation 5.1 and Equation 5.2 define Casper's transformations. The former focuses on expressions while the latter on statements (when a language constructs is both an expression and a statement, we choose arbitrarily one): in the figures, e and e_n are Java expressions, s is a statement. For brevity, Figure 5.2 introduces $\langle method_decl \rangle$ for a method declaration. The variable $\langle stmts \rangle$ is a statement list. Since version 5.0, Java automatically boxes and unboxes primitives to and from object wrappers, $unbox(e_1)$ denotes this operation. To apply Equation 5.1 and Equation 5.2 to an entire program, Casper also has a global contextual rewriting rule $T(C[n]) = C[T(n)]$, where T applies either T_e or T_s, n denotes an expression or statement and $C[\cdot]$ denotes a program context. Equations 5.1a–5.1c, in Figure 5.1, define our semantics-preserving expression transformations. subsection 5.1.4 discusses them.

The equations show the injection of the following functions into the program under debug: nullDeref, nullParam, nullPassed, nullAssign, and nullReturn. These functions all check their argument for nullity. If their argument is null, they create a null ghost and add the causality link built into their name, i.e. nullAssign adds A. If their argument is a null ghost, they all append the appropriate causality link to the causality trace. Table 5.2 summarizes these methods.

5.1.3.2 Detection of nulls

To provide the origin and causality of null dereferences, one has to detect null values *before* they become harmful, i.e. before they are dereferenced. This section describes how Casper's

$$T(e) = \begin{cases} e_1 \texttt{ == null ||} \\ \quad e_1 \texttt{ instanceof NullGhost} & \text{if } e = e_1 \texttt{ == null} \quad (5.1a) \\ e_1 \texttt{ instanceof MyClass \&\&} \\ \quad !(e_1 \texttt{ instanceof NullGhost)} & \text{if } e = e_1 \texttt{ instanceof MyClass} \quad (5.1b) \\ lib.m\texttt{(exorcise(}e_1\texttt{),}\cdots\texttt{)} & \text{if } e = lib.m\texttt{(}e_1, \cdots, e_k\texttt{)} \quad (5.1c) \\ \texttt{nullUnbox(}unbox(e_1)\texttt{)} & \text{if } e = unbox(e_1) \quad (5.1d) \\ \texttt{nullDeref(}e_1\texttt{).}f & \text{if } e = e_1.f \quad (5.1e) \\ e & \text{otherwise} \end{cases}$$

Figure 5.1: Casper's local expression transformations: rules 5.1a–5.1c preserve the semantics of the program under debug (subsection 5.1.4); rules 5.1d–5.2c inject calls to collect the U, **D** and P_i causality links (subsection 5.1.1); in Equation 5.1e, f denotes either a function or a field.

$$T(s) = \begin{cases} o \leftarrow \texttt{nullAssign(}e\texttt{);} & \text{if } s = o \leftarrow e \,(assign) \quad (5.2a) \\ o \leftarrow \texttt{nullAssign(null);} & \text{if } s = o \,(var_decl) \quad (5.2b) \\ m\texttt{(nullParam(}p_1, \cdots\texttt{))} & \text{if } s = m\texttt{(}p_1, \cdots, p_n\texttt{)} \; (call) \quad (5.2c) \\ m\texttt{(}p_1, \cdots, p_n\texttt{)} \{ & \text{if } s = \langle method_decl \rangle \quad (5.2d) \\ \quad p_i \leftarrow nullPassed(p_i)\texttt{;} \forall p_i \\ \quad \langle methodbody \rangle \\ \} \\ \texttt{return nullReturn(}e\texttt{);} & \text{if } s = \texttt{return } e\texttt{;} \quad (5.2e) \\ s & \text{otherwise} \end{cases}$$

Figure 5.2: Casper's statement transformations: these rules inject calls into statements to collect the L, A, P_e, and R causality links (subsection 5.1.1); s denotes a statement; $\langle method_decl \rangle$ denotes a method declaration; $\langle stmts \rangle$ denotes a statement list; and p_i binds to a function's formals in a declaration and actuals in a call.

transformations inject helper methods that detect and capture nulls.

In Java, as with all languages that prevent pointer arithmetic, nulls originate in explicit or implicit literals within the application under debug or from an external library. L in Table 5.1 lists four examples of the former case. Casper statically detects nulls appearing in each of these contexts. For explicit null assignment, as in x = null, it applies its Equation 5.2a; for implicit, Object o;, it applies Equation 5.2b. Both of these rewritings inject nullAssign, which instantiates a null ghost and starts its causality trace with L–A. Equation 5.2e injects nullReturn to handle return null, collecting R.

Not all nulls can be statically detected: a library can produce them. An application may be infected by a null from an external library in four cases: 1. assignments whose right hand side involves an external call; 2. method invocations one of whose parameter expressions involves an external call; 3. boolean or arithmetic expressions involving external calls; and 4. callbacks from an external library into the program under debug.

Equation 5.2a handles the assignment case, injecting the nullAssign method to collect the causality links R, A. Equation 5.1c wraps the parameters of internal method calls with nullParam, which handles external calls in a parameter list and adds the R and P_e links to a null ghost. Equation 5.1d handles boolean or arithmetic expressions. It injects the nullUnbox

```
//initial method
void method(Object a, final Object b){
  //method body
}

//is transformed to
void method(Object a, final Object b_dup){
  a = NullDetector.nullPassed(a);
  b = NullDetector.nullPassed(b_dup);
  //method body
}
```

Listing 5.4: Illustration of the Source Code Transformation for Causality Connection "Null Method Parameter" (P_e).

method to check nullity and create a null ghost or update an existing one's trace with R and U. Finally, Equation 5.2d handles library callbacks. A library callback happens when the application under debug provides an object to the library and the library invokes a method of this object, as in the "Listener" design pattern. In this case, the library can bind null to one of the method's parameters. Because we cannot know which method may be involved in a callback, Equation 5.2d inserts a check for each argument at the beginning of every method call, potentially adding P_e to a ghost's causality trace.

Rewriting Java in the presence of its final keyword is challenging. Listing 5.4 shows an example application of Equation 5.2d. The first method is the application method and the second one is the method after instruction by Casper. The use of final variables, which can only be assigned once in Java, requires us to duplicate the parameter as a local variable. Renaming the parameters (b to b_dup), then creating a local variable with the same name as the original parameter, allows Casper to avoid modifying the body of the method.

5.1.4 Semantics Preservation

Using null ghosts instead of nulls must not modify program execution. Casper therefore defines the three transformations in Equations 5.1a–5.1c, whose aim is to preserve semantics. We evaluate the degree to which our transformations preserve application semantics in section 5.2.

5.1.4.1 Comparison Operators

Consider "o == null". When o is null, == evaluates to true. If, however, o points to a null ghost, the expression evaluates to false. Equation 5.1a preserves the original behavior by rewriting expressions, to include the conjunct "!o instanceof NullGhost". Our example "o == null" becomes the expression "o == null && !o instanceof NullGhost". Here, NullGhost is a marker interface that all null ghosts implement. The rewritten expression is equivalent to the original, over all operands, notably including null ghosts.

Java developers can write "o instanceof MyClass" to check the compatibility of a variable and a type. Under Java's semantics, if o is null, no error is thrown and the expression returns

false. When o is a null ghost, however, the expression returns true. Equation 5.1b solves this problem. To preserve behavior, it rewrites appearances of the instanceof operator, e.g. replacing “o instanceof MyClass” with “o instanceof MyClass && !o instanceof NullGhost".

5.1.4.2 Usage of Libraries

During the execution of a program that uses libraries, one may pass null as a parameter to a library call. For instance, o could be null when lib.m(o) executes. After Casper’s transformation, o may be bound to a null ghost. In this case, if the library checks whether its parameters are null, using x == null or x instanceof SomeClass, a null ghost could change the behavior of the library and consequently of the program. Thus, for any method whose source we lack, we modify its calls to “unbox the null ghost”, using Equation 5.1c. In our example, lib.m(o) becomes lib.m(exorcise(o)). When passed a null ghost, the method exorcise returns the null that the ghost wraps.

5.1.4.3 Emulating Null Dereferences

When dereferencing a null, Java throws an exception object NullPointerException. When dereferencing a null ghost, the execution must also results in throwing the same exception. In Listing 5.3, a null ghost throws the exception CasperNullPointerException, which extends Java’s exception NullPointerException. The Casper’s specific exception contains the dereferenced null ghost and overrides the usual exception reporting methods, namely the getCause, toString, and printStackTrace methods, to display the ghost’s causality trace.

Java throws a NullPointerException in three cases: *a*) a method call on null; *b*) a field access on null; or *c*) unboxing a null from a primitive type’s object wrapper. Casper trivially emulates method calls on a null: it defines each method in a ghost to throw CasperNullPointer − Exception, as Listing 5.3 shows. Java does not provide a listener that monitors field accesses. Equation 5.1e overcomes this problem; it wraps expressions involved in a field access in nullDeref, which checks for a null ghost, prior to the field access. For instance, Casper transforms x.f into nullDeref(e).f. Since version 5.0, Java has supported autoboxing and unboxing to facilitate working with its primitive types. A primitive type’s object wrapper may contain a null; if so, unboxing it triggers a null dereference error. For example, Integer a = null; int b = a, a + 3 or a ∗ 3 all throw NullPointerException.

5.1.5 Implementation

Casper requires, as input, the source code of the program under debug, together with the binaries of the dependencies. Its transformations are automatic and produce an instrumented version of the program under debug. We stack source code transformation at compile time and dynamic binary code transformation at load time. The reasons are low-level details that are specific to the Java platform as explained above.

5.1.5.1 Source Code Transformations

We perform our source code transformations using Spoon [100]. This is done at compile time, just before the compilation to bytecode. Spoon performs all modifications on a model representing the AST of the program under debug. Afterwards, Spoon generate new Java

files that contain the program corresponding to the AST after application of the transformations of Equation 5.1 and Figure 5.2.

5.1.5.2 Binary Code Transformations

We create null ghosts with binary code generation using ASM[24]. The reason is the Java final keyword. This keyword can be applied to both types and methods and prevents further extension. Unfortunately, we must be able to override any arbitrary class and all methods to create null ghost classes. To overcome this protection at runtime, Casper uses its our own classloader, which ignores the final keyword in method signatures when the class is loaded. For example, when MyClass must be "haunted", the class loader generates MyClassGhost.class on the fly.

5.1.5.3 Limitations

Casper cannot identify the root cause of a null pointer dereference in two cases. The first is when the root cause is in external library code that we cannot rewrite. This is the price we pay to avoid assuming a closed world, where all classes are known and manipulatable. The second is specific to the fact that we implemented Casper in Java: our classloader technique for overriding final classes and methods does not work for JDK classes, because most of these classes are loaded before the invocation of application-specific class loaders. One consequence is that our implementation of Casper cannot provide causality traces for Java strings.

5.2 Empirical Evaluation

We now evaluate the capability of our approach to build correct causality traces of real errors from large-scale open-source projects. The evaluation answers the following research questions:
$RQ1$: Does our approach provide the correct causality trace?
$RQ2$: Do the code transformations preserve the semantics of the application?
$RQ3$: Is the approach useful with respect to the fixing process?

RQ1 and RQ2 concern correctness. In the context of null deference analysis, RQ1 focuses one one kind of correctness defined as the capability to provide the root cause of the null dereference. In other words, the causality trace has to connect the error to its root cause. RQ2 assesses that the behavior of the application under study does not vary after applying our code transformations. RQ3 studies the extent to which causality traces help a developer to fix null dereference bugs.

5.2.1 Dataset

We built a dataset of real life null dereference bugs. There are two inclusion criteria. First, the bug must be a real bug reported on a publicly-available forum (e.g. a bug repository). Second, the bug must be reproducible.

[24] http://asm.ow2.org/

# Bug Id	Problem summary	Fix summary
McKoi	new JDBCDatabaseInterface with null param -> field -> deref	Not fixed (Artificial bug by [39])
Freemarker #107	circular initialization makes a field null in WrappingTemplateModel -> deref	not manually fixed. could be fixed manually by adding hard-code value. no longer a problem with java 7.
JFreeChart #687	no axis given while creating a plot	can no longer create a plot without axis modifying constructor for fast failure with error message
collection #331	no error message set in a thrown NPE	add a check not null before the throw + manual throwing of NPE
math #290	NPE instead of a domain exception when a null List provided	normalize the list to use empty list instead of null
math #305	bad type usage (int instead of double). Math.sqrt() call on an negative int -> return null. should be a positive double	change the type
math #1117	Object created with too small values, after multiple iterations of a call on this object, it returns null	create a default object to replace the wrong valued one
7 other bugs		add a nullity check

Table 5.3: A dataset of 14 real null dereference errors from large scale open-source projects. The dataset is made publicly available for future replication and research on this problem.

The reproducibility is challenging. Since our approach is dynamic, we must be able to compile and run the software in its faulty version. First, we need the source code of the software at the corresponding buggy version. Second, we must be able to compile the software. Third, we need to be able to run the buggy case. In general, it is really hard to reproduce real bugs in general and null dereferences in particular. Often, the actual input data or input sequence triggering the null dereference is not given, or the exact buggy version is not specified, or the buggy version can no longer be compiled and executed.

We formed our dataset in two ways. First, we tried to replicate results over a published dataset [39] as described below. Second, we selected a set of popular projects. For each project, we used a bag of words over their bug repository (e.g. bugzilla of jira) to identify an under approximate set of NPEs. We then faced the difficult challenge of reproducing these bugs, as bug reports rarely specify the bug-triggering inputs. Our final dataset is therefore conditioned on reproducibility. We do not, however, have any reason to believe that any bias that may exist in our dataset would impact Casper general applicability.

Under these constraints, we want to assess how our approach compares to the closest related work [39]. Their dataset dates back to 2008. In terms of bug reproduction 6 years later, this dataset is hard to replicate. For 3 of the 12 bugs in this work, we cannot find any description or bug report. For 4 of the remaining 9, we cannot build the software because the versions of the libraries are not given or no longer available. 3 of the remaining 5 do not give the error-triggering inputs, or they do not produce an error. Consequently, we were only able to reproduce 3 null dereference bugs from Bond et al.'s dataset.

We collected 7 other bugs. The collection methodology follows. First, we look for bugs in the Apache Commons set of libraries (e.g. Apache Commons Lang). The reasons are the following. First, it is a well-known and well-used set of libraries. Second, Apache commons bug repositories are public, easy to access and search. Finally, thanks to the strong software engineering discipline of the Apache foundation, a failing test case is often provided in the bug report.

To select the real bugs to be added to our dataset we proceed as follows. We took all the bugs from the Apache bug repository[25]. We then select 3 projects that are well used and well known (Collections, Lang and Math). We add the condition that those bug reports must have "NullPointerException" (or "NPE") in their title. Then we filter them to keep only those which have been fixed and which are closed (our experimentation needs the patch). After filters, 19 bug reports remain[26]. Sadly, on those 19 bug reports, 8 are not relevant for our experiment: 3 are too old and no commit is attached (COLL-4, LANG-42 and Lang-144), 2 concern Javadoc (COLL-516 and MATH-466), 2 of them are not bugs at all (LANG-87 and MATH-467), 1 concerns a VM problem. Finally, we add the 11 remaining cases to our dataset.

Consequently, the dataset contains the 3 cases from [39] (Mckoi, freemarker and jfreechart) and 11 cases from Apache Commons (1 from collections, 3 from lang and 7 from math). In total, the bugs come from 6 different projects, which is good for assessing the external validity of our evaluation. This makes a total of 14 real life null dereferences bugs in the dataset. Table 5.3 shows the name of the applications, the number of the bug Id (if existing), a summary of the NPE cause and a summary of the chosen fix. We put only one line for 7 of them because they use the same simple fix (i.e. adding a check not null before the faulty line). The

[25] `https://issues.apache.org/jira/issues`

[26] The link to automatically set those filters is given in `http://sachaproject.gforge.inria.fr/casper`

application coverage of the test suites under study are greater than 90% for the 3 Apache common projects (11 out of the 14 cases). For the 3 cases from [39] (Mckoi, freemarker and jfreechart), we do not have access to the full test suites.

This dataset only contains real null dereference bugs and no artificial or toy bugs. To reassure the reader about cherry-picking, we have considered all null dereferenced bugs of the 3 selected projects. We have not rejected a single null dereference that Casper fails to handle.

5.2.2 Methodology

5.2.2.1 Correctness

*RQ*1*: Does our approach provide the correct causality trace?* To assess whether the provided element is responsible for a null dereference, we manually analyze each case. We manually compare the result provided by our technique with those coming from a manual debugging process that is performed using the debug mode of Eclipse.

*RQ*2*: Do the code transformations preserve the semantics of the application?* To assert that our approach does not modify the behavior of the application, we use two different strategies.

First, we require that the origin program and the transformed program both pass and fail the same tests in the test suite (when it exists). However, this only addresses the correctness of externally observable behavior of the program under debug. To assess that our approach does not modify the internal behavior, we compare the execution traces of the original program (prior to code transformation) and the program after transformation. Here, an "execution trace" is the ordered list of all method calls and of all returned values, executing over the entire test suite. This trace is obtained by logging method entry (injecting *package.class#method(arg.toString . . .)*) and logging return (injecting *package.class#method():returnedValue.toString*). We filter out all calls to the Casper framework, then align the two traces. They must be identical.

5.2.2.2 Effectiveness

*RQ*3*: Is the approach useful with respect to the fixing process?* To assert that our additional data is useful, we look at whether the location of the real fix is given in the causality trace. If the location of the actual fix is provided in the causality trace, it would have helped the developer by reducing the search space of possible solutions. Note that in 7/14 cases of our dataset, the fix location already appears in the original stack trace. Those are the 7 simple cases where a check not null is sufficient to prevent the error. Those cases are valuable in the context of our evaluation to check that: 1) the variable name is given (as opposed to only the line number of a standard stack trace), 2) the causality trace is correct (although the fix location appears in the original stack trace, it does not prevent a real causality trace with several causal connections).

# Bug Id	Fix location	Fix location in the standard stack trace	Addressed by [39]	Fix location in Casper's causality trace	Causality Trace	Exec. time instrumented (ms)
McKoi	Not fixed	No	No	Yes	L-A-R-A-**D**	382
Freemarker #107	Not fixed	No	Yes	Yes	L-A-**D**	691
JFreeChart #687	FastScatterPlot 178	No	No	Yes	L-P_e-P_i-**D**	222
collection #331	CollatingIterator 350	No	No	Yes	L-A-**D**	81
math #290	SimplexTableau 106/125/197	No	No	No	**D**	107
math #305	MathUtils 1624	No	No	No	L-R-A-R-**D**	68
math #1117	PolygonSet 230	No	No	No	L-A-R-A-**D**	191
7 simple cases		Yes	Yes	Yes	L-A-**D** (x6) L-A-U	147 (average)
Total		7 / 14	8/14	11/14		

Table 5.4: Evaluation of Casper: in 13/14 cases, a causality trace is given, in 11/13 the causality trace contains the location where the actual fix was made.

5.2.3 Results

5.2.3.1 RQ1

After verification by manual debugging, in all the cases under study, the element identified by our approach is the one responsible for the error. This result can be replicated since our dataset and our prototype software are made publicly available.

5.2.3.2 RQ2

All the test suites have the same external behavior with and without our modifications according to our two behavior preservation criteria. First, the test suite after transformation still passes. Second, for each run of the test suite, the order of method calls is the same and the return values are the same. In short, our massive code transformations do not modify the behavior of the program under study and provide the actual causality relationships.

5.2.3.3 RQ3

We now perform two comparisons. First, we look at whether the fix locations appear in the standard stack traces. Second, we compare the standard stack trace and causality trace to see whether the additional information corresponds to the fix.

Table 5.4 presents the fix locations (class and line number) (second column) and whether: this location is provided in the basic stack trace (third column); 2) the location is provided by previous work [39] (fourth column); 3) it is in the causality trace (last column). The first column, "# Bug Id", gives the id of the bug in the bug tracker of the project (same as Table 5.3).

In 7 out of 14 cases (the 7 simple cases), the fix location is in the original stack trace. For those 7 cases, the causality trace is correct and also points to the fix location. In comparison to the original stack trace, it provides the name of the root cause variable.

In the remaining 7 cases, the fix location is not in the original stack trace. This means that in 50% of our cases, there is indeed a cause/effect chasm, that is hard to debug [109], because no root cause information is provided to the developer by the error message. We now explain in more details those 7 interesting cases.

The related work [39] would provide the root cause in only 1 out of those 7 cases (according to an analysis, since their implementation is not executable). In comparison, our approach provides the root cause in 4 out of those 7 cases. This supports the claim that our approach is able to better help the developers in pinpointing the root cause compared to the basic stack trace or the related work.

5.2.3.4 Detailed Analysis

5.2.3.4.1 Case Studies There are two different reasons why our approach does not provide the fix location: First, for one case, our approach is not able to provide a causality trace. Second, for two cases, the root cause of the null dereference is not the root cause of the bug.

In the case of Math #290, our approach is not able to provide the causality trace. This happens because the null value is stored in an Integer, which is a final type coming from the jdk. Indeed, java.lang.Integer is a native Java type and our approach cannot modify them (see subsubsection 5.1.5.3).

```
1
2  private static Cluster<T> getNearestCluster(final Collection<Cluster> clusters, final T point) {
3     double minDistance = Double.MAX_VALUE;
4     Cluster<T> minCluster = null; //initialisation
5     for (final Cluster<T> c : clusters) {
6        final double distance = point.distanceFrom(c.getCenter()); //return NaN
7        if (distance < minDistance) { //failing condition
8           minDistance = distance;
9           minCluster = c;
10       }
11    }
12    return minCluster; //return null
13 }
```

Listing 5.5: An excerpt of Math #305 where the causality trace does not contain the fix location.

In the case of Math #305, the root cause of the null dereference is not the root cause of the bug. The root cause of this null dereference is shown in Listing 5.5. The null responsible of the null dereference is initialized on line 4, the method call `distanceFrom` on line 6 return `NaN`, due to this NaN, the condition on line 7 fails, and the null value is returned (line 9). Here, the cause of the dereference is that a null value is returned by this method. However, this is the root cause of the `null` but this is not the root cause of the bug. The root cause of the bug is the root cause of the `NaN`. Indeed, according to the explanation and the fix given by the developer the call `point.distanceFrom(c.getCenter())` should not return `NaN`. Hence, the fix of this bug is in the `distanceFrom` method, which does not appear in our causality chain because no `null` is involved.

In the case of Math #1117, the root cause of the null dereference is not the root cause of the bug. The root cause of this null dereference is shown in Listing 5.6. The null responsible of the dereference is the one passed as second parameter of the constructor call on line 10. This null value is stored in the field minus of this SplitSubHyperplane. Here, the cause of the dereference is that a null value is set in a field of the object returned by this method. Once again, this is the root cause of the `null` but this is not the root cause of the bug. The root cause of the bug is the root cause of the failing condition global $< -1.0e-10$. Indeed, according to the explanation and the fix given by the developer, the Hyperplane passed as method parameter should not exist if its two lines are too close from each other. Here, this Hyperplane comes from a field of a PolygonSet. On the constructor of this PolygonSet they pass a null value as a parameter instead of this "irregular" object. To do that, they add a condition based on a previously existing parameter called `tolerance`, if the distance of the two lines are lower than this tolerance, it returns a null value. (It is interesting that the fix of a null dereference is to return a null value elsewhere.)

5.2.3.4.2 Size of the Traces There are mainly two kinds of traces encountered in our experiment. First, the one of size 3 and of kind L-A-**D** type. The 7 obvious cases (were the fix location is in the stack trace) contains 6 traces of this kind. In all those cases encountered in our experiment, the null literal has been assigned to a field. This means that a field has not been initialized (or initialized to null) during the instance creation, hence, this field is dereferenced latter. This kind of trace is pretty short so one may think that this case is obvious. However, all of those fields are initialized long ago the dereference. In other words, when the dereference occurs, the stack has changed and no longer contains the information of the

```
1
2   public SplitSubHyperplane split(Hyperplane hyperplane) {
3      Line thisLine  = (Line) getHyperplane();
4      Line otherLine = (Line) hyperplane;
5      Vector2D crossing = thisLine.intersection(otherLine);
6     if (crossing == null) { // the lines are parallel
7        double global = otherLine.getOffset(thisLine);
8        return (global < −1.0e−10) ?
9             new SplitSubHyperplane(null, this) :
10            new SplitSubHyperplane(this, null);//initialisation
11     }
12     ...
13  }
```

Listing 5.6: An excerpt of Math #1117 where the causality trace does not contain the fix location.

initialization location.

Second, the one of size >= 4 where the null is stored in a variable then passed as argument in one or multiple methods. In all those case, the null value is either returned by a method at least once or passed as a parameter.

5.2.3.4.3 Execution Time To debug a null dereference error, Casper requires to instrument the code and to run the instrumented version. In all the cases, the instrumentation time is less 30 seconds. At runtime, Casper finds the causality trace of the failing input in less than 1 second (last column of Table 5.4). This seems reasonable from the developer viewpoint: she obtains the causality trace in less than 30 seconds. We have also measured the overhead with respect the original test case trigerring the error: it's a 7x increase. This clearly prevents the technique to be used in production. This is also a limitation for using our technique to debug null dereference errors in concurrent code because the overhead is likely to change the scheduling, mask the error or trigger new so-called "Heisenbugs".

5.2.4 Causality Trace and Patch

Once the causality trace of a null dereference is known, the developer can fix the dereference. There are two basic dimensions for fixing null references based on the causality trace.

First, the developer has to select in the causal elements, the location where the fix should be applied: it is often at the root cause, i.e. the first element in the causality trace. It may be more appropriate to patch the code elsewhere, in the middle of the propagation between the first occurrence of the null and the dereference error.

Second, the developer has to decide whether there should be an object instead of null or whether the code should be able to gracefully handle the null value. In the first case, the developer fixes the bug by providing an appropriate object instead of the null value. In the second case, she adds a null check in the program to allow a null value.

5.2.5 Threats to Validity

The internal validity of our approach and implementation has been assessed through RQ1 and RQ2: the causality trace of the 14 analyzed errors is correct after manual analysis. The threat to the external validity lies in the dataset composition: does the dataset reflect the

complexity of null dereference errors in the field? To address this threat, we took a special care in designing the methodology to build the dataset. It ensures that the considered bugs apply to large scale software and are annoying enough to be reported and commented in a bug repository. The generalizability of our results to null dereference errors in other runtime environments (e.g. .NET, Python) is an open question to be addressed by future work.

5.3 Conclusion

In this chapter, we have presented Casper, a novel approach for debugging null dereference errors. The key idea of our technique is to inject special values, called "null ghosts" into the execution stream to aid debugging. The null ghosts collects the history of the null value propagation between its first detection and the problematic dereference, we call this history the 'causality trace". We define 11 code transformations responsible for 1) detecting null values at runtime; 2) collect causal relations and enrich the causality traces; 3) preserve the execution semantics when null ghosts flow during program execution. The evaluation of our technique on 14 real-world null dereference bugs from large-scale open-source projects shows that Casper is able to provide a valuable causality trace. Being able to automatically understand the root cause of null dereference errors, we propose a method to automatically tolerate them in the next chapter.

Chapter 6

NpeFix: Tolerating Null Dereference at Runtime with Test Suite Validation

The contributions of this chapter are:

- A set of strategies to tolerate null dereferences.
- A set of source code transformations designed and tailored for allowing the usage of those strategies.
- NpeFix, an implementation in Java of our technique.
- An evaluation of our technique on 14 real null dereference bugs.

This chapter is original work for this thesis, that has not yet been submitted for publication.

In the previous chapters, we presented techniques to improve the knowledge used in bug fixing: Bug characterization and detection in Chapter 3 and 4; Bug understanding with causality traces in Chapter 5.

To go further, in this chapter, we present a system to tolerate a specific class of bugs: null dereferences. In this work, we define "tolerating a bug" by providing a solution to the problematic case at runtime, i.e. at the moment the bug itself happens. To do it, we present a way to detect null dereferences before they happen.

Let us define an exception as "harmful" if this exception will compromise the continuity of the program execution (the most common way is by crashing the application). On the opposite a "non-harmful" exception is an exception for which handling mechanisms have already been set up.

To be sure we do not try to tolerate non harmful exceptions, we use the work in Chapter 3 to attest that the detected exception is a bug. After that, we propose strategies to tolerate it at runtime (such as injecting a new value instead of the null). Those strategies are tested according to the test suite and an algorithm is proposed to execute them at runtime.

We present a new approach for tolerating null pointer dereference at runtime. This ap-

proach automatically propose an alternative execution when an error is going to happen. This approach also asserts the validity of the modified behavior w.r.t. the behavior specified in the test suite.

6.1 Concepts

We want to tolerate the null dereferences which may have lead to a problem (e.g. crash the application). Let us present the different concepts used in our approach. Algorithm 6.1 describes the different steps of this approach. Those steps are detailed in the following sections.

6.1.1 Detecting Null Dereference Bugs

Our final goal is to tolerate null dereferences bugs. There are two main ways to tolerate an error. First, one can wait for the bug to happen, then deal with the error. Second, one can act before the bug happens and avoid the error. We decide to focus on the second way, i.e. to deal with the root cause of the bug.

The first part of our work is to detect harmful null dereferences before it happens. We perform this in two steps. First we detect all the null dereferences which will happen (see Section 6.1.1.1). Then, we check if the dereference that will happen is going to be harmful for the application (see Section 6.1.1.2).

6.1.1.1 Runtime Detection of Null Dereferences

We detect the null dereferences at the exact moment before it happens. Our technique consists in assessing, each time a reference is going to be dereferenced, that this reference is null or not.

6.1.1.2 Detecting Harmful Null Dereferences

Even when a null dereference happens, it does not mean that there is a bug in the application. When a null dereference occurs, an exception is thrown (e.g. NullPointerException in Java). This exception may be caught at some point upper in the execution stack with a try-catch mechanism. In this case, it means that the application already contains a way to handle this error. In other words, the dereference has been anticipated by the developers and we do not consider it as a bug. According to this, we should not modify the behavior of the application.

Hence, when the exception thrown by the null dereference can not be caught by the application, it means that the error is unexpected, and that it will crash the application. In those cases, we perform our tolerating strategies.

For detecting harmful null dereferences, we perform a runtime data acquisition which allow us to know at any moment in the execution which exceptions can be caught. Indeed, the exception that cannot be caught are potentially harmful, and are those one we are interested in. The practical details are given in Section 6.2.1.1.

```
Input:
An Application A,
A test suite TS specifying the behavior of A,
A set of possible strategies S
Output:
A working strategy s
1  begin
2    //before production
3    M ← generate_mapping(A, TS)          ▷ Map of corresponding between the lines
4                                         ▷ and the test cases executing them
5    //in production
6    CS ← newStack                        ▷ initialize the stack of catchable types
7    id ← 0                               ▷ initialize the try-catch dynamic id
8    while true                           ▷ Execution of A
9    do
10     line ← wait_particular_statement()   ▷ locking method
11     switch line.type do
12       case try_start
13         t ← line.corresponding_try
14         t.id ← id++                      ▷ generate and provide a new id to the try-catch
15         CS.add(t.id, t.types)            ▷ add the type(s) of the corresponding
16                                          ▷ catch block(s)to the catchable stack
17       case try_end
18         t ← line.corresponding_try
19         t.remove(t.id)                   ▷ remove the try-catch (and all the type(s) of the
20                                          ▷ corresponding catch block(s)) from the catchable stack
21       case dereference
22         if CS.can_stop_deref()           ▷ if the null dereference is anticipated (catchable)
23         then
24           continue                       ▷ continue the execution (while of line 8)
25         else
26                                          ▷ the null dereference will crash the program
27           for s ∈ S                      ▷ for each possible strategy
28           do
29             viable ← true
30             for t ∈ M.tests_using(s)     ▷ for each test case executing this line
31             do
32               result ← execute(t, s)     ▷ execute the test case with the strategy
                 active
33               if result == failed        ▷ if one test case fail
34               then
35                 viable ← false
36                 break                    ▷ go to the next strategy (for of line 26)
37             if viable                    ▷ if all test cases passed
38             then
39               solution_found(s, line)    ▷ deploy the strategy
40               continue                   ▷ continue the execution (while of line 8)
```


Figure 6.1: The NpeFix Algorithm.

6.1.2 Tolerating

When an harmful null dereference is going to happen, there are two main ways to avoid it. First, one can replace the null reference by a valid reference on this instance, this way it will no longer be a **null** dereference. Second, one can skip the problematic statement, if the dereference is not executed, no null dereference will occur. Listing 6.1 shows samples of those solutions. Based on those two ways, we propose 9 different strategies to tolerate null dereferences. Those strategies are listed in the Table 6.1.

```
//problematic statement
o.doSomething();//o is null -> NPE

//providing instance
o = new Object();
o.doSomething();

//skipping the statement
if(o!=null)
  o.doSomething();
```

Listing 6.1: EXAMPLES OF NULL DEREFERENCE SOLUTIONS

Definition A strategy is a set of actions which modify the behavior of the application in order to avoid the null dereference.

6.1.2.1 Null replacement

One way to avoid a null dereference to happen is to change the reference into a valid instance (see Listing 6.1). This triggers two main questions.

1) What can be provided to replace the null reference? 2) What is the scope of the provided value?

We can inject an existing value (if one can be found) or a new value (if we can provide one). As explained below, we can also inject the value locally or globally. This provide us 4 possible strategies to be applied for the replacement part (see Figure 6.1): use an existing value locally (s1a), use an existing value globally (s1b), use a new value locally (s2a) and use a new value globally (s2b).

Given a reference r and a statement s dereferencing r. In the nominal case, r references a null value. We want r to reference a valid (non-null) value v. Our goal is to find a v. We will consider two main ways to achieve that. First, finding an existing value in the set of the accessible values which corresponds to required one (i.e. same type, ...). Second, creating an new value.

In both cases, the first thing to know is the required type. We know two types, the type of r and the type dereferenced by s. The fact that the program compiles implies that the type of r is the same as, or a subtype of, the required type. But there may be multiple other types corresponding to all the subtypes of r. Following a well known good practice, most of the references are typed as an interface (e.g. List in Java), but there are multiple possible implementations of this type which may be used (e.g. ArrayList, LinkedList, etc).

Strategy			Id	Sample
replacement	variable	local	s1a	b.foo();
		global	s1b	a=b; a.foo();
	new	local	s2a	new A().foo();
		global	s2b	a=new A(); a.foo();
skipping	line		s3	if(a!=null) a.foo();

	method	void	s4	if(a==null)
				return;
				a.foo();

		null	s4a	if(a==null)
				return null;
				a.foo();

		var	s4c	if(a==null)
				return c;
				a.foo();

		new	s4b	if(a==null)
				return new C();
				a.foo();

Table 6.1: List of the proposed strategies.

Because, at this point, we are looking for possible strategies and not for an absolute solution, we will consider in the following of this default initialization strategy all the usable types for r: i.e. all the known subtypes of r.

Let us consider the case of using an existing value. The set of the accessible values is composed of the local variables, the parameters, the fields of the current class and all the static values. Those values are obtain by using a source code modification which register a reference to each of the newly declared values in a method plus a dynamic reflection analysis to retrieve all the accessible fields. We filter all those values to obtain the set of all the well typed and non-null values V. The two s1 strategies consists in testing all those values v *in* V one by one.

Let us consider the case of creating a new value. We statically know all the possible types for r. First, we filter those types to keep only the non-interface, non-abstract ones. Then we try to create a new instance of each of those types (using reflection). The two s1 strategies consists in testing all the successfully created instances one by one.

Our algorithm will try both the injection of existing values and of newly generated values. Section 6.4.4 contains a discussion on the advantages and disadvantages of both of them.

Recall that we want r to reference a well-typed non-null value v. To be exact, we want s not to dereference a null reference, not necessarily r. This let us two possibilities. First, we can make r reference v. In this case, s still dereferences r, but r does no longer references *null*. Second, we can make s dereference a new reference r' which reference v. In that case, s dereferences r', which references v, but r still references *null*. This does not change anything for the execution of s, it uses v in both cases.

Definition Global injection consists in replacing a value and all its references by another.

Definition Local injection consists in replacing one reference to a value by a reference to another one, without modifying the value nor its other references.

This changes for the rest of the execution. Indeed, r is known by the application. In the first case, the application still remembers v has the value referenced by r, which means that all the possible other statements using r will now perform their operations on v instead of on *null*. In the second case, the application still remember *null* has being referenced by v, which means that all the possible other statements using r will still perform their operations on *null*. Indeed, v is not stored in r but in r', which is created for this only statement, it means that v is not stored anywhere else in the application after s ended (except if it has been stored during s).

Our choice here is to choose between modifying the global state or the local state of the program. There are advantages and disadvantages for both of them. The advantage of making a local modification (using r'), is that we change as less as possible the state of the program. If a condition on the nullity of r happen afterwards, the program will still consider r as *null* and acts accordingly. But the corresponding disadvantage of a local modification is that we let a *null* reference (r) in the program, if r is dereferenced again afterwards, we eventually have to do this choice again. This problematic leads us to keep the two possible cases as possible strategies.

6.1.2.2 Statement skipping

The second proposed way is to skip the statement where a null dereference would happen.

The strategy s3 consists in skipping the problematic statement and allows us to avoid the null dereference at this location (as shown in Figure6.1). This strategy consists in the insertion of a check not null before the execution of the faulty statement.

Other strategies may skip more statements that only the problematic one. In particular, we will see the s4 family of strategies which consists in skipping the rest of the method. Considering the skipping of the rest of the method, there are two possibilities to consider. Either the method does not have to return anything ("void"), and we can just return to the caller (strategy s4), or the method expects a return value. If the method expects a return value, we have to choose what to return to the caller. We consider three possibilities. First, we could return null (adding null values to tolerate null dereferences may look meaningless, this is discussed in Section 6.4). Second, we could search in the set of the accessible values one which corresponds to the expected return type and return it. Third, we could return a new instance of the expected type.

We decided to implement those strategies, respectively called s4a, s4c and s4b.

We have presented different strategies which allow the application not to execute the null dereference. In both cases, the nominal behavior of the application is changed. The following section show how we assess that the behavior modification conforms with the application specifications.

6.1.3 Viability

How to assert that a tolerating strategy is viable? Our goal is to assert that the strategy, which is going to be executed on production, is viable. In other words, assert that this strategy respects the specifications of the program. As well as numerous other works on the repair field (e.g. Weimer et al. [55]), we consider the test suite as the given specifications of the program. In this case, executing the test suite with the applied strategy for the particular location is a way to validate that strategy respects the specifications of the program. For example, using a regression-oriented test suite, asserting that our strategy corresponds to the test suite means that after applying our strategy, we do not modify the expected behavior of the program.

We apply two basic rules during this validation step. First, our framework only executes the test cases which executes the modified line. Indeed, because this validation happens at runtime (during production) this allow us to save time. The fact that a test case t executes or not a statement s is statically determined before the run of the program. Second, we consider that our strategy is only applied when the faulty reference is null. For example, if the strategy consists in replacing the value v by another one $newV$ (s2b), for all test cases executed for testing this strategy we replace v by $newV$ if and only if v is null.

6.2 Methodology

In this section, we present the main different parts of the presented framework. Algorithm 6.1 recalls the main steps of it.

6.2.1 Detection

```
//before modification
try{
   //codeA
}catch(TypedException te){
   //codeB
}catch(AnotherTypedException ate){
   //codeC
}

//after modification
int tryId = catchStack.getFreeId();
try{
   catchStack.add(tryId, TypedException.class, AnotherTypedException.class);
   //codeA
}catch(TypedException te){
   catchStack.remove(tryId);
   //codeB
}catch(AnotherTypedException ate){
   catchStack.remove(tryId);
   //codeC
}finally{
   catchStack.remove(tryId);
}
```

Listing 6.2: CATCH STACK CONSTRUCTION

6.2.1.1 code transformation for baby foot

For a given location in the execution, we want to know if a null dereference may harm the application. We consider harmful an exception that is not caught in the current thread. Hence, our goal is to know if a NPE will be caught somewhere in the execution stack. To do it, we use a Stack which corresponds to all the exceptions that may be caught. To know if a given exception will be caught at a given moment in the execution, we have to look at whether this exception type corresponds to one of the types in the stack. Figure 6.2 shows how this works in practice. The method call "catchStack.add" alerts the framework that we enter in a try body which is able to catch the types given as parameter. The method calls "catchStack.remove" alert the frameworks that we exited the body of the given try. At the beginning of every try, we add the caught type(s) to the stack, associated with a generated try id (the next available id in the map). At the end of every try, we remove the caught type from the stack. There are three possibilities to exit the body of a try block: 1/ no exception is thrown: the end of the try is the end of the execution of the try (after codeA), 2/ a caught exception is thrown: the end of the try is the beginning of one catch (in the middle of codeA, just before codeB or codeC), 3/ an uncaught exception is thrown: the end of the try is the beginning of the finally block (in the middle of codeA). To know when the try is exited we add the "remove" calls at the beginning of every catch and at the beginning of the finally. In

the cases 1/ and 3/, the "remove" call on the finally block allows us to know that the try is finished. In the case 2/ the call on the corresponding catch block alerts the framework that the try is finished. In this case, the "remove" call on the finally block will also be executed, that is why we added a try-id to each try on the stack. Even if the remove method is called twice, the framework knows that the try is no longer in the stack and will not remove another try from the stack.

```
//before modification
o.doSomething();

//after modification
check(o).doSomething();

//with static method
public static Object check(Object o){
  if(o==null)
    //null dereference detected
    if(cannotCatchNPE())
      //problematic null dereference
  return o;
}
```

Listing 6.3: DEREFERENCE ENCAPSULATION

6.2.1.2 Code Transformation for NullPointerException

We transform each possible null dereference location (see Figure 6.3). The method call executed before the possible dereference does multiple things. It first assesses that a null dereference will occur. If no, the program can continue its execution. If yes, our framework looks at the "try-stack". If the exception is anticipated, the program can continue its execution. If the exception will crash the execution, we apply a tolerance strategy.

6.2.2 Tolerance

We create a set of code transformations which allow us to execute any of the presented strategies. For seek of clarity we present the transformations linked to a strategy. In practice, the code is modified once, then it is able to execute any strategy. We present the different strategies in the two following sections.

6.2.2.1 Value Replacement Strategies

Our goal is to be able to change the dereferenced variable, either changing its value, or replacing the variable itself. The code transformation given in Listing 6.3 is also used to execute the strategies s1a, s1b, s2a and s2b, after adding the type of the variable is given as a parameter. Figure 6.5 shows the difference between the 4 given strategies. We use polymorphism to deal with the different strategies, in the figure, each line below a comment corresponds to a strategy. The "getVar" method must return a well-typed accessible variable of the application. The "newVar" method must return a new value of the right type.

For the "getVar" method, we register each variable initialization and assignment inside the application methods. Listing 6.6 shows how we deal with those variables; we use a stack to store all the variables of each method. In addition, we use reflection to access all the fields of the current instance, we access the instance with the "this" parameter in the "startMethod" call. When the "getVar" is called, we look at all accessible variables (i.e. all the variables of the method plus all the fields of the instance) and we assess if one of them has the good type.

For the "newVar" method, we use reflection to access to all the constructors of the given type. Listing 6.4 shows how the "newVar" method works. We try to create a new instance of the class with each constructor. Given a constructor, we know the parameter types used in the constructor. We try to create a new instance for each of the parameter recursively. The stopping condition is when a constructor does not need parameters. Note that the primitive types are not recognized as class with constructors, so we have to add some checks at the beginning of the method.

```
public static Object newVar(Class clazz){
  for(Constructor constructor : clazz.getDeclaredConstructors()){
    Object[] params = new Object[constructor.getParamTypes().length];
    for(int i =0; i < constructor.getParamTypes().length; i++){
      params[i] = newVar(constructor.getParamTypes()[i]);
    }
    constructor.newInstance(params);
  }
  return null;
}
```

Listing 6.4: INSTANCE CREATION

```
public static Object check(Object o, Class clazz){
  if(o==null && cannotCatchNPE())
    //s1a
    return getVar(clazz);
    //s1b
    o = getVar(clazz);
    //s2a
    return newVar(clazz);
    //s2b
    o = newVar(clazz);
  return o;
}
```

Listing 6.5: DEREFERENCE ENCAPSULATION

```
public void method(){
//...
Object a = {expr};
//...
a = {expr2};
```

```
//...
}

public void method(){
int id = getFreeId();
startMethod(id, this);
//...
Object a = initVar({expr}, id, ``a'');
//...
a = modifVar({expr2}, id, ``a'');
//...
endMethod(id);
}
```

Listing 6.6: DEREFERENCE ENCAPSULATION

6.2.2.2 Line skipping

The strategy s3 necessitates to know if a null dereference will happen in a line, before the execution of the line. Indeed, the previous transformation with the “check” method implies that the execution of the line has started. Listing 6.7 shows the concept used here. The “skipLine” method assesses, before the line execution, if the dereferenced value is null or not. If it is null and the program cannot stop the error, it returns false, so the if condition will skip the line.

There are numerous cases where one can not easily skip one line. For example, one cannot skip a “return” or a “throw” statement, if the method has no longer its termination points, the program will not compile. Another case is the variable declaration, skipping the declaration (and the initialization) of a variable leads to unaccessible variables later on. Those cases are tagged as not-skipable by our framework.

```
value.dereference();
//becomes
if(skipLine(value)){
  value.dereference();
}

public static boolean skipLine(Object o){
  if(o==null && cannotCatchNPE())
    return false
  return true;
}
```

Listing 6.7: DEREFERENCE ENCAPSULATION

6.2.2.3 Method skipping

The strategies s4, s4a, s4c and s4b consist in skipping the end of the method after a null dereference occurs. In other words, it consists in returning to the caller. To do it, we add try-

catch around all the method bodies. Those try-catch handle a particular type of exception (ForceReturnError). This exception is thrown by the "skipLine" method. Listing 6.8 shows a minimalist sample of this transformation.

```
public Object method(){
  //...
  value.dereference();
  //...
  return X;
}

//becomes

public Object method(){
  try{
    //...
    if(skipLine(value)){
      value.dereference();
    }
    //...
    return X;
  }catch(ForceReturnError f){
    // s4a
    return null;
    //or s4b
    return getVar(Object.class);
    //or s4c
    return newVar(Object.class);
  }
}

//with
public static boolean skipLine(Object o){
  if(o==null && cannotCatchNPE())
    thow new ForceReturnError();
  return true;
}
```

Listing 6.8: DEREFERENCE ENCAPSULATION

bug	s1a	s1b	s2a	s2b	s3	s4a	s4b	s4c	s4	nbKO	NbOK
col331	NoVar	NoVar	OK	OK	OK	test fail	OK	NoVar		4	4
lang304	NoVar	NoVar	OK	OK	OK	NPE	OK	NoVar		2	4
lang587	NoVar	NoVar	OK	OK	return	OK	OK	generic		4	4
lang703	NoVar	NoVar	OK	OK	OK	test fail	OK	NoVar		4	4
math290	NoVar	NoVar	OK	OK	cannot init				OK	3	3
math305	NoVar	NoVar	OK	OK	OK				Div by 0	3	3
math369	NoVar	NoVar	NoInst	NoInst	OK	test fail	OK	OK		5	3
math988a	NoVar	NoVar	OK	OK	cannot	NPE	OK	NoVar		5	1
math988b	NoVar	NoVar	OK	OK	cannot	NPE	OK	NoVar		5	1
math1115	NoVar	NoVar	NoInst	NoInst	cannot int	NPE	NoInst	NoVar		8	0
math1117	NoVar	NoVar	NoInst	NoInst	NPE	AOB	NoInst	NoVar		8	0
nbKO	11	11	3	3	6	8	2	8	1		
nbOK	0	0	8	8	5	1	6	1	1		

Table 6.2: Efficiency of the proposed strategies.

6.3 Evaluation

6.3.1 Dataset

We use an extended version of the dataset presented in the previous chapter 5.

We built a dataset of 11 real life null dereference bugs. There are two inclusion criteria. First, the bug must be a real bug reported on a publicly-available forum (e.g. a bug repository). Second, the bug must be reproducible.

Let us dwell on bug reproducibility. Since our approach is dynamic, we must be able to compile and run the software in its faulty version. First, we need the source code of the software at the corresponding buggy version. Second, we must be able to compile the software. Third, we need to be able to run the buggy case.

The collection methodology follows. First, we look for bugs in the Apache Commons set of libraries (e.g. Apache Commons Lang). The reasons are the following. First, it is a well-known and well-used set of libraries. Second, Apache commons bug repositories are public, easy to access and to be searched. Finally, thanks to the strong software engineering discipline of the Apache foundation, a failing test case is often provided in the bug report.

To select the real bugs to be added to our dataset we proceed as follows. We took all the bugs from the Apache bug repository[27]. We then select 3 projects that are well used and well known (Collections, Lang and Math). We add the condition that those bug reports must have “NullPointerException" (or “NPE") in their title. Then we filter them to keep only those which have been fixed and which are closed (our experimentation needs the patch). Those filters let us 19 bug reports. Sadly, on those 19 bug reports, 8 are not relevant for our experimentation: 3 are too olds and no commit is attached (COLL-4, LANG-42 and Lang-144), 2 concern Javadoc (COLL-516 and MATH-466), 2 of them are not bugs at all (LANG-87 and MATH-467), 1 concerns a VM problem. Finally, we add the 11 remaining cases to our dataset.

Consequently, the dataset contains 11 cases from Apache Commons (1 from collections, 3 from lang and 7 from math).

This dataset only contains real null dereference bugs and no artificial or toy bugs. To reassure the reader about cherry-picking, we have considered all null dereferenced bugs of the 3 selected projects. We have not rejected a single null dereference.

6.3.2 Overall Efficiency

Is our approach able to tolerate null dereferences? This section is about the efficiency of our approach to globally tolerate null dereferences. Our algorithm 6.1 tries each of the presented strategies (see Table 6.1). Table 6.2 presents the results of each of the strategies. The last column of the table is the number of viable strategies, if this number is higher or equals to 1, there is at least one functioning strategy. In other words, if this number is not zero, our global tolerance algorithm works (it finds at least one solution). On our dataset, our algorithm finds a valid tolerance strategy in 9 out of the 11 cases. At runtime, 9 out of 11 null dereferences are tolerated and 2 out of 11 are not. Remind that in the other 2 cases, our algorithm sends the same error as if nothing has been done. In other words, it means that our approach can not worsen the problem.

[27] `https://issues.apache.org/jira/issues`

6.3.3 Replacement Efficiency

In this section, we discuss the efficiency of the replacement strategies (s1a, s1b, s2a and s2b), which correspond to replacing the null value by another one. The efficiency of each strategy is displayed in the two last rows of Table 6.2, they are summarized as the number of failing cases (nbKO) and the number of working cases (nbOK).

6.3.3.1 From Var

As shown in Table 6.2, the strategies s1a and s1b, which correspond to replacing the null value by a corresponding existing non null value, are not efficient. The reason is that in none of the studied cases it exists such a value. In all of those cases, the method is short and does not contain such a variable, and the class does not contain a field of this type, or the field is null also. For example, in lang-304, the required type is a Set, the method is 1 line-long, so there is no other values, and the only field of the class which is a Set is null. In addition, in 4 out of those 11 cases, the type of the value is domain specific. For example, in math-988a, the required type is org.apache.commons.math3.geometry.euclidean.twod.Vector2D.

6.3.3.2 From New

As shown in Table 6.2, the strategies s2a and s2b, which correspond to replacing the null value by a corresponding new value, are 2 of the 3 more efficient strategies. In 8 out of the 11 cases, this strategy is able to provide a valid substitute to the null value. They are the most efficient of our strategies, according to our dataset. In the 3 remaining cases, we are not able to construct a valid new instance. This may have multiple reasons, explained in Section 6.4.1.2.

However, one may note that there is no case where injecting a new value leads to a failing case. Either the injection works as a valid strategy, or we are not able to inject a valid object. According to this, being able to inject new values at runtime looks an interesting solution to tolerate null dereferences.

6.3.3.3 Local or Global

As shown by the columns s2a and s2b of Table 6.2, there is no difference between replacing a null value locally or globally. Recall that global injection consists in modifying the value of the dereferenced variable where local injection consists in giving another new variable which will be dereferenced. On the first hand, injecting globally may have lead to side effects error, indeed, we modify the application state. In our cases, those side effects neither happen (i.e. the value is neither reused), or are neglectable (no impact on the results). On the other hand, injecting locally may have lead to other null dereferences if the given null value is reused later on. In our cases, this neither happens, it means that either those values are not used anywhere else, or they are initialized before being reused.

6.3.4 Skipping Efficiency

6.3.4.1 Line Skipping

As shown in Table 6.2, the strategies s3, which corresponds to skipping the faulty line, is efficient in 5 out of 11 cases. In those 5 cases, skipping the execution of the faulty line, then continuing the execution of the program looks an efficient strategy to tolerate the null dereference.

Let us have a look at the 6 failing cases. In 5 out of the 6 failing cases, we can not skip the faulty line: four of them are variable initialization which are used later on, just putting a if not null before the faulty line leads to a compilation error (see Section 6.4.1.3 to the implementation details). One of them is a return statement, once again, skipping it will lead to compilation errors.

There is only one case (math-1117) in our dataset in which skipping the faulty line leads to an error. In this case, the skipped line is an assignment of a returned value, this value having already been assigned before. Skipping this line leads the returned value to be incorrect. This returned value failed to execute a later on treatment (return false instead of true), this leads to a badly construct instance of PolygonSet (it has no Plan). Later on the execution, a method call on this badly construct PolygonSet (getSize), dereferences its Plan, which leads to another null dereference error. In other terms, we replace one null dereference by another. This is a good example of the difficulty to prevent every possible side effect when modifying the nominal behavior of an application.

6.3.4.2 Method Skipping

6.3.4.2.1 Return Void There are two cases in our dataset where the null dereference occurs inside a method which returns void. Those cases correspond to the strategy s4.

In math-290, we are able to return instead of executing the faulty statement, and this does not look problematic. This cancels the execution of the "initialize" method called at the end of the SimplexTableau constructor. Either this method is not critical (e.g. performance gain), or its behavior is not necessary in this particular case.

In the second case, in math-305, skipping the assignPointsToClusters method leads to a division by zero later in the program execution. This method was clearly critical, and skipping its whole execution is not a valid substitute. Note that only skipping the faulty line (s3) is a valid substitute, which means that this particular line is not critical, but one cannot skip the whole method containing it.

With only those two cases we cannot say that returning to the caller when executing a method without return value is a viable strategy or not.

6.3.4.2.2 Return Something There are 9 cases in our dataset where the null dereference occurs inside a method which returns something. Those cases correspond to the strategies s4a, s4c and s4b.

Table 6.2 shows that returning null (s4a) is a valid strategy in only one out of the 9 cases. In most of the cases, it leads to another error (null dereference in 4 cases, test failure in 3 cases and an array out of bound access in one case). Those errors are all consequences of the newly injected null value. There is one case where returning null is working, the method takes an array as input and creates another array which corresponds to the types of the values in

the parameter array. This behavior is not tested, so returning a null array instead of a array containing a null value is considered as similar.

Table 6.2 shows that returning an existing value of the returned type (s4c) is a valid strategy in one out of the 9 cases. We encounter the same problems as replacing the dereferenced value by an existing one (s1a & s1b), in 8 out of the 9 cases, no corresponding value can be found. We do not know if this strategy may be considered as viable because it "always" (in one case) works when a value is found, or not because it does not find values most of the time.

Finally, Table 6.2 shows that returning a new value of the return type (s4b) is a valid strategy in 7 out of the 9 cases. In our cases, either returning a new value is sufficient to complete the process successfully, or the injected side effects are negligible (no impact on the tested results).

6.4 Discussion

6.4.1 Limitations

6.4.1.1 Null Dereference Location

Because our approach uses source code modification, we have to have access to the source code of the location of the null dereference. If the dereference happens in an archived library, our approach is not able to detect the null dereference. The workaround to this situation is to add the source code of the library in the experiment, considering the library as a part of the application.

6.4.1.2 Cannot Instantiate

There are 3 cases where we are not able to construct a valid instance to a given type. We could have use workarounds to manually construct the required instances but we want the results to be as near as possible as the runtime execution of this framework. In those 3 cases, the required type is an application-specific interface. We implemented most of the used interface inside the JDK but we do not yet have a way to automatically identify and instantiate the application specific ones. The way to identify the required types could have been to remember all the application specific types, then to find those which implement the required interface, but this implementation detail is not in the scope of this work.

6.4.1.3 Cannot Skip Line

Most of the time we are able to skip a single statement but it creates other problems. For example, if the statement is a variable declaration and/or initialization, we cannot totally skip it because other locations may use the declared variable. In this case, we still can keep the variable declaration but skip the initialization, in other words, adding null values in the execution. Adding null values to tolerate null dereference looks meaningless to us, so we don't do it, we prefer to say we cannot skip this statement. Other problems occur if the problematic statement is inside a condition. For example, if the problematic statement is contained in the condition of a if, should we execute the then branch? the else branch? or none? If it is in a loop condition do we execute the loop once? n times? or not at all? We could

create multiple meaningless strategies which follow each one of those theoretical rules. Once again, we prefer to keep only re-usable results, instead of providing an application-specific solution.

6.4.2 False Positive Cases

Our approach dynamically detects the null dereferences at runtime. With this way, we do not take into account false positive possible null dereferences as a static analysis may give us.

6.4.3 Typing

In the case where the type of r is a subtype of the required one, we loose some possibilities for v, but we are nearer of the nominal behavior.

6.4.4 Value Creation or Reuse

We consider here both strategies because both of them have advantages and disadvantages. Using an existing value is more meaningful than using an artificial one. But it is not always possible, it must exist such a value. In addition, it also may lead to problems. What if the value is modified by s? We may loose some data of the program state. There are also cases where a fresh value is preferable. For example, consider a linked list, each element knows a reference to the following one. The current element is known by the program, and it tries to attach the next one, which may lead to a null dereference. If we replace this null value by the current element, it will link to itself, which will probably create problems such as infinite loops. A solution may be to clone this value into a new one, but not all values are cloneable, even when using reflection.

6.5 Conclusion

In this chapter, we have presented npeFix, a novel approach for tolerating harmful null dereferences. We first use dynamic analysis to detect harmful null dereferences, skipping the non-problematic ones. Then we proposed a set of strategies able to tolerate this error. We define code transformations to 1) detect harmful null dereferences at runtime; 2) allow a behavior modification to execute strategies; 3) attest the corresponding between the modified behavior and the specifications. The evaluation of our technique on 11 real-world null dereference bugs from large-scale open-source projects shows that npeFix contains valuable and applicable strategies.

Chapter 7

Conclusion

7.1 Summary

In this thesis, we addressed two major problems regarding software errors. **Problem #1**: There is a lack of debug information for the bugs related to exceptions. **Problem #2**: There are unexpected exceptions at runtime for which there is no error-handling code. We presented four different contributions to improve bug fixing and resilience w.r.t. the exceptions.

First, we have explored the concept of software resilience against exceptions. We have contributed with different results that, to our knowledge, were not discussed in the literature. We have shown to what extent test suites specify exception-handling. In addition, we have formalized two formal resilience properties: source-independence and pure-resilience as well as an algorithm 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. We discover that numerous cases are undecidable due to the behavior of the test cases.

To overcome this undecidability problem, we proposed a technique to increase the efficiency of dynamic program analysis. To do it, we presented B-Refactoring, a technique to split test cases into small fragments. Our experiments on five open-source projects show that our approach effectively improves the purity of test cases. We show that applying B-Refactoring to existing analysis tasks, namely repairing `if`-condition bugs and analyzing exception contracts, enhances the capabilities of these tasks. The work presented in this chapter improves the capability of the try-catch categorization algorithm by more than 20%.

Contribution #1 and **#2** are generic, they target any kind of exceptions. In order to further contribute to bug fixing and resilience, we need to focus on specific types of exceptions. This focus enables us to exploit the knowledge we have about them and further improve bug fixing and resilience. Hence, in the rest of this thesis, we focused on a more specific kind of exception: the null pointer dereference exceptions (NullPointerException in Java).

We have presented Casper, a novel approach for debugging null dereference errors. The key idea of this technique is to inject special values, called "null ghosts" into the execution stream to aid debugging. The null ghosts collects the history of the null value propagation between its first detection and the problematic dereference, we call this history the 'causality trace". We defined 11 code transformations responsible for 1) detecting null values at runtime; 2) collect causal relations and enrich the causality traces; 3) preserve the execution

semantics when null ghosts flow during program execution. The evaluation of our technique on 14 real-world null dereference bugs from large-scale open-source projects shows that Casper is able to provide a valuable causality trace.

In the last contribution of this thesis, we have presented npeFix, a novel approach for tolerating harmful null dereferences. We first use dynamic analysis to detect harmful null dereferences, skipping the non-problematic ones. Then we proposed a set of strategies able to tolerate this error. We defined code transformations to 1) detect harmful null dereferences at runtime; 2) allow a behavior modification to execute strategies; 3) assess the corresponding between the modified behavior and the specifications. The evaluation of our technique on 11 real-world null dereference bugs from large-scale open-source projects shows that npeFix contains valuable and applicable strategies.

7.2 Perspectives

In this section we present some research ideas related to this thesis that look worth to be explored in the future.

All contributions of this thesis are supported by empirical evaluations on open-source software applications. In future work we want to know the resilience capabilities of commercial projects and compare them to open-source projects. Hence, we aim at replicating those experiments in commercial projects. To validate our results at a larger scale we also plan on using tools to be able to automatize experiments on more numerous projects.

Finally, when the contribution concerns existing bugs (Contribution #3 and #4), we had to find corresponding bugs, find and set up their source code, and be able to compile and to reproduce them. When it comes to real life bugs, this is time consuming due to the numerous problems which may occur in any of those steps. However, datasets of bugs are now publicly available (such as Defects4J [110]). We plan on setting our experiments in such dataset to be able to confirm or to review our results.

7.2.1 Study of Exception Errors

In Chapter 3 we presented two contracts to categorize the resilience capabilities of try-catch blocks. Both contracts relies on the location of the thrown exception. To simulate worst-case exception we inject exceptions as the first statement of the try-catch block. We plan on setting experiments for injecting exceptions at all the possible locations (i.e. between each statement of the try-catch). This experiment may change the categorization of some of the try-catch blocks.

In Chapter 3, we have shown that 10% of the try-catch under study are source-dependent. Recall that a source-dependent try-catch implies a lower resilience capabilities (as shown in Section 3.1.1). We plan on proposing a solution to set those try-catch source-independent. When a try-catch block is proven to violate the source-independence contract, our approach gives a precise counter-example. We have explained that the cause of the violation is that the code of the catch block depends on the execution of parts of the try block. The solution would be to decrease the scope of the try block so that the catch block only depends on the state at the beginning of the try block only. In order to keep all test cases passing, the decrease stops at the statement for which the specified exception occurs (and not the injected ones).

7.2.2 Repair

In Chapter 5 we provide information on the null pointer dereference cause to ease the debug process. This contribution addresses the problem of the errors happening for the first time in production mode. In addition of the causality trace, we plan on improving this technique to be able to automatically provide a test case corresponding to the runtime error. Indeed, it is possible to set the ghosts to store additional data (such as function parameter and field values). This technique would help developers, they will automatically have a way to replicate the buggy situations instead of having only the logs of the error.

As said before, to valuably contribute to bug fixing we needed to focus on specific types of errors. This focus enabled us to exploit the knowledge we have about null dereferences and further improve their bug fixing. Our future work consists in further exploring the idea of "ghost" for debugging other kinds of runtime errors such as arithmetic overflows. For example, we could follow the zero values in numeric variables to provide the causality traces of division by zero errors.

7.2.3 Resilience

In Chapter 6, we presented techniques to tolerate null pointer dereferences and to assess the validity of the tolerating technique w.r.t. the test suite. We proposed multiple techniques to tolerate the error and we try them in a random order. Instead of trying the strategies in a random order, we want to be able to explore them in the order of probable effectiveness. Hence, the most suitable strategy should be used first. One solution would be to first use Casper (Chapter 5) to obtain causality traces of null pointer dereferences bugs. Second, we set up a table of the different kinds of bug fix (corresponding to a strategy) according to the kind of causality trace. Third, we envision to modify npeFix (Chapter 6) to include Casper: we want to know the causality trace of the incoming error. Finally, when we have to tolerate a null pointer dereference, we will have the causality trace provided by the null ghost. We will use this causality trace to search in the created table the most similar known bug. The corresponding strategy will be the most probably suitable strategy.

Bibliography

[1] Z. Merali, "Computational science: Error, why scientific programming does not compute," *Nature*, vol. 467, no. 7317, pp. 775–777, 2010.

[2] "CNSC Fukushima Task Force Report," Tech. Rep. INFO-0824, Canadian Nuclear Safety Commission, 2011.

[3] J. B. Goodenough, "Exception handling: Issues and a proposed notation," *Commun. ACM*, vol. 18, no. 12, pp. 683–696, 1975.

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

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

[6] 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*, pp. 30–39, ACM, 2014.

[7] P. Rovner, "Extending modula-2 to build large, integrated systems," *Software, IEEE*, vol. 3, no. 6, pp. 46–57, 1986.

[8] J.-C. Laprie, "From dependability to resilience," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[9] 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, (New York, NY, USA), pp. 74–77, ACM, 2009.

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

[11] 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*, pp. 391–400, IEEE, 2011.

[12] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *Software Engineering, IEEE Transactions on*, vol. 27, no. 10, pp. 929–948, 2001.

[13] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th international conference on Software engineering*, pp. 467–477, ACM, 2002.

[14] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *Software Engineering, IEEE Transactions on*, vol. 27, no. 2, pp. 99–123, 2001.

[15] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[16] S. Sinha and M. J. Harrold, "Analysis and testing of programs with exception handling constructs," *Software Engineering, IEEE Transactions on*, vol. 26, no. 9, pp. 849–871, 2000.

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

[18] M. P. Robillard and G. C. Murphy, "Static analysis to support the evolution of exception structure in object-oriented systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 12, no. 2, pp. 191–221, 2003.

[19] E. A. Barbosa, "Improving exception handling with recommendations," in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), pp. 666–669, ACM, 2014.

[20] N. Cacho, T. César, T. Filipe, E. Soares, A. Cassio, R. Souza, I. Garcia, E. A. Barbosa, and A. Garcia, "Trading robustness for maintainability: An empirical study of evolving c# programs," in *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, (New York, NY, USA), pp. 584–595, ACM, 2014.

[21] S. Entwisle, H. Schmidt, I. Peake, and E. Kendall, "A model driven exception management framework for developing reliable software systems," in *Enterprise Distributed Object Computing Conference, 2006. EDOC '06. 10th IEEE International*, pp. 307–318, Oct 2006.

[22] F. Souchon, C. Urtado, S. Vauttier, and C. Dony, "Exception handling in component-based systems: a first study," in *Proceedings of ECOOP'03 Workshop on Exception Hanling in Object-Oriented Systems*, pp. 84–91, 2003.

[23] C. Dony, C. Urtado, and S. Vauttier, "Exception handling and asynchronous active objects: Issues and proposal," in *Advanced Topics in Exception Handling Techniques*, pp. 81–100, Springer, 2006.

[24] M. Karaorman, U. Hölzle, and J. Bruno, "jcontractor: A reflective java library to support design by contract," in *Meta-Level Architectures and Reflection*, pp. 175–196, Springer, 1999.

[25] C. Fu and B. G. Ryder, "Exception-chain analysis: Revealing exception handling architecture in java server applications," in *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[26] J. Mercadal, Q. Enard, C. Consel, and N. Loriant, "A domain-specific approach to architecturing error handling in pervasive computing," in *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

[27] H. Ohe and B.-M. Chang, "An exception monitoring system for java," in *Rapid Integration of Software Engineering Techniques*, pp. 71–81, Springer, 2005.

[28] B. Bokowski, A. Spiegel, *et al.*, "Barat-a front end for java," 1998.

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

[30] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin, "Fiat-fault injection based automated testing environment," in *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing*, pp. 102–107, IEEE, 1988.

[31] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek, "Fault injection experiments using fiat," *IEEE Transactions on Computers*, vol. 39, no. 4, pp. 575–582, 1990.

[32] W. lun Kao, R. K. Iyer, and D. Tang, "Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults," *IEEE Trans. Software Eng.*, vol. 19, no. 11, pp. 1105–1118, 1993.

[33] M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "A comprehensive survey of trends in oracles for software testing," *University of Sheffield, Department of Computer Science, Tech. Rep. CS-13-01*, 2013.

[34] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "Holmes: Effective statistical debugging via efficient path profiling," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 34–44, IEEE, 2009.

[35] A. Zeller, "Isolating cause-effect chains from computer programs," in *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pp. 1–10, ACM, 2002.

[36] C. Zamfir and G. Candea, "Low-overhead bug fingerprinting for fast debugging," in *Runtime Verification* (H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. Pace, G. Roşu, O. Sokolsky, and N. Tillmann, eds.), vol. 6418 of *Lecture Notes in Computer Science*, pp. 460–468, Springer Berlin Heidelberg, 2010.

[37] C. Fetzer, P. Felber, and K. Hogstedt, "Automatic detection and masking of nonatomic exception handling," *Software Engineering, IEEE Transactions on*, vol. 30, no. 8, pp. 547–560, 2004.

[38] N. Tracey, J. Clark, K. Mander, and J. McDermid, "Automated test-data generation for exception conditions," *Software-Practice and Experience*, vol. 30, no. 1, pp. 61–79, 2000.

[39] M. D. Bond, N. Nethercote, S. W. Kent, S. Z. Guyer, and K. S. McKinley, "Tracking bad apples: reporting the origin of null and undefined value errors," *ACM SIGPLAN Notices*, vol. 42, no. 10, pp. 405–422, 2007.

[40] D. Romano, M. Di Penta, and G. Antoniol, "An approach for search based testing of null pointer exceptions," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pp. 160–169, IEEE, 2011.

[41] D. Hovemeyer, J. Spacco, and W. Pugh, "Evaluating and tuning a static analysis to find null pointer bugs," in *ACM SIGSOFT Software Engineering Notes*, vol. 31, pp. 13–19, ACM, 2005.

[42] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 153–164, ACM, 2009.

[43] M. Nanda and S. Sinha, "Accurate interprocedural null-dereference analysis for java," in *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pp. 133–143, May 2009.

[44] N. Ayewah and W. Pugh, "Null dereference analysis in practice," in *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, (New York, NY, USA), pp. 65–72, ACM, 2010.

[45] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, (New York, NY, USA), pp. 467–477, ACM, 2002.

[46] M. Bruntink, A. van Deursen, and T. Tourwé, "Discovering faults in idiom-based exception handling," in *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, (New York, NY, USA), pp. 242–251, ACM, 2006.

[47] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, *et al.*, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 87–102, ACM, 2009.

[48] A. Smirnov and T.-c. Chiueh, "Dira: Automatic detection, identification and repair of control-hijacking attacks.," in *NDSS*, 2005.

[49] A. Smirnov, R. Lin, and T.-C. Chiueh, "Pasan: Automatic patch and signature generation for bufferoverflow attacks," 2006.

[50] S. Thummalapenta and T. Xie, "Mining exception-handling rules as sequence association rules," in *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, (Washington, DC, USA), pp. 496–506, IEEE Computer Society, 2009.

[51] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[52] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 772–781, IEEE Press, 2013.

[53] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, (New York, NY, USA), pp. 1427–1434, ACM, 2011.

[54] K.-P. Vo, Y.-M. Wang, P. E. Chung, and Y. Huang, "Xept: a software instrumentation method for exception handling," in *Software Reliability Engineering, 1997. Proceedings., The Eighth International Symposium on*, pp. 60–69, IEEE, 1997.

[55] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 364–374, IEEE Computer Society, 2009.

[56] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Software Engineering, IEEE Transactions on*, vol. 38, no. 1, pp. 54–72, 2012.

[57] 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*, ICSE 2014, (New York, NY, USA), pp. 254–265, ACM, 2014.

[58] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, (Washington, DC, USA), pp. 550–554, IEEE Computer Society, 2009.

[59] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *Software Engineering, IEEE Transactions on*, vol. SE-13, pp. 23–31, Jan 1987.

[60] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *Proceedings of the 19th international symposium on Software testing and analysis*, pp. 61–72, ACM, 2010.

[61] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pp. 356–366, Nov 2013.

[62] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*, pp. 162–168, IEEE, 2008.

[63] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, "Jagr: an autonomous self-recovering application server," in *Autonomic Computing Workshop. 2003. Proceedings of the*, pp. 168–177, June 2003.

[64] F. Long, S. Sidiroglou-Douskos, and M. Rinard, "Automatic runtime error repair and containment via recovery shepherding," *SIGPLAN Not.*, vol. 49, pp. 227–238, June 2014.

[65] K. Dobolyi and W. Weimer, "Changing java's semantics for handling null pointer exceptions," in *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pp. 47–56, IEEE, 2008.

[66] Y.-M. Wang, Y. Huang, and C. Kintala, "Progressive retry for software failure recovery in message-passing applications," *Computers, IEEE Transactions on*, vol. 46, pp. 1137–1141, Oct 1997.

[67] Y.-M. Wang, Y. Huang, and W. Fuchs, "Progressive retry for software error recovery in distributed systems," in *Fault-Tolerant Computing, 1993. FTCS-23. Digest of Papers., The Twenty-Third International Symposium on*, pp. 138–144, June 1993.

[68] S. Sidiroglou, O. Laadan, C. Perez, N. Viennot, J. Nieh, and A. D. Keromytis, "Assure: Automatic software self-healing using rescue points," *SIGPLAN Not.*, vol. 44, pp. 37–48, Mar. 2009.

[69] A. Avizienis, G. Gilley, F. P. Mathur, D. Rennels, J. Rohr, and D. Rubin, "The star (self-testing and repairing) computer: An investigation of the theory and practice of fault-tolerant computer design," *Computers, IEEE Transactions on*, vol. C-20, pp. 1312–1321, Nov 1971.

[70] J. Rohr, "Starex self-repair routines: Software recovery in the jpl-star computer," in *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*, pp. 201–, Jun 1995.

[71] N. Wang, M. Fertig, and S. Patel, "Y-branches: when you come to a fork in the road, take it," in *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pp. 56–66, Sept 2003.

[72] A. D. Keromytis, "Characterizing self-healing software systems," in *Computer Network Security: Fourth International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS 2007, St. Petersburg, Russia, September 13-15, 2007: Proceedings*, pp. 22–33, Springer, 2007.

[73] J. Zhao, Y. Jin, K. S. Trivedi, and R. Matias, "Injecting memory leaks to accelerate software failures," in *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, pp. 260–269, IEEE, 2011.

[74] F. Souchon, C. Dony, C. Urtado, and S. Vauttier, "Improving exception handling in multi-agent systems," in *Software Engineering for Multi-Agent Systems II* (C. Lucena, A. Garcia, A. Romanovsky, J. Castro, and P. Alencar, eds.), vol. 2940 of *Lecture Notes in Computer Science*, pp. 167–188, Springer Berlin Heidelberg, 2004.

[75] G. Munkby and S. Schupp, "Automating exception-safety classification," *Science of Computer Programming*, vol. 76, no. 4, pp. 278 – 289, 2011. Special issue on library-centric software design (LCSD 2006).

[76] M. Rinard, "Acceptability-oriented computing," *ACM SIGPLAN Notices*, vol. 38, no. 12, pp. 57–75, 2003.

[77] T. Ogasawara, H. Komatsu, and T. Nakatani, "A study of exception handling and its dynamic optimization in java," *SIGPLAN Not.*, vol. 36, pp. 83–95, Oct. 2001.

[78] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, “Efficient and precise modeling of exceptions for the analysis of java programs,” in *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE ’99, (New York, NY, USA), pp. 21–31, ACM, 1999.

[79] N. Cacho, F. Dantas, A. Garcia, and F. Castor, “Exception flows made explicit: An exploratory study,” in *Software Engineering, 2009. SBES ’09. XXIII Brazilian Symposium on*, pp. 43–53, Oct 2009.

[80] N. Cacho, A. Garcia, E. Figueiredo, *et al.*, “Ejflow: taming exceptional control flows in aspect-oriented programming,” in *Proceedings of the 7th international conference on Aspect-oriented software development*, pp. 72–83, ACM, 2008.

[81] B. Meyer, “Applying design by contract,” *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

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

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

[84] 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.

[85] G. Candea, M. Delgado, M. Chen, and A. Fox, “Automatic failure-path inference: A generic introspection technique for internet applications,” in *Proceedings of the 3rd IEEE Workshop on Internet Applications*, pp. 132–141, IEEE, 2003.

[86] S. Ghosh and J. L. Kelly, “Bytecode fault injection for java software,” *Journal of Systems and Software*, vol. 81, no. 11, pp. 2034–2043, 2008.

[87] B. Baudry, F. Fleurey, and Y. Le Traon, “Improving test suites for efficient fault localization,” in *Proceedings of the 28th international conference on Software engineering*, pp. 82–91, ACM, 2006.

[88] J. Xuan and M. Monperrus, “Test case purification for improving fault localization,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, 2014.

[89] P. Wadler, “The essence of functional programming,” in *Proceedings of the 19th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 1–14, ACM, 1992.

[90] N. Carriero and D. Gelernter, “Linda in context,” *Communications of the ACM*, vol. 32, no. 4, pp. 444–458, 1989.

[91] N. Heintze and J. G. Riecke, “The slam calculus: programming with secrecy and integrity,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 365–377, ACM, 1998.

[92] B. Cornu, L. Seinturier, and M. Monperrus, "Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions," *Information and Software Technology*, vol. 57, pp. 66–76, 2015.

[93] 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 *Software Engineering (ICSE), 2012 34th International Conference on*, pp. 3–13, IEEE, 2012.

[94] Z. Qi, F. Long, S. Achour, and M. Rinard, "Efficient automatic patch generation and defect identification in kali," in *Proceedings of ISSTA*, ACM, 2015.

[95] M. Monperrus, "A critical review of "automatic patch generation learned from human-written patches": An essay on the problem statement and the evaluation of automatic software repair," in *Proc. of the Int. Conf on Software Engineering (ICSE)*, (Hyderabab, India), 2014.

[96] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)* (M. Marchesi, ed.), pp. 92–95, University of Cagliari, 2001.

[97] T. Mens and T. Tourwé, "A survey of software refactoring," *Software Engineering, IEEE Transactions on*, vol. 30, no. 2, pp. 126–139, 2004.

[98] P.-H. Chu, N.-L. Hsueh, H.-H. Chen, and C.-H. Liu, "A test case refactoring approach for pattern-based software development," *Software Quality Journal*, vol. 20, no. 1, pp. 43–75, 2012.

[99] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, 2011.

[100] R. Pawlak, M. Monperrus, N. Petitprez, C. Noguera, and L. Seinturier, "Spoon v2: Large scale source code analysis and transformation for java," Tech. Rep. hal-01078532, INRIA, 2006.

[101] S. C. P. F. Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation testing applied to validate specifications based on statecharts," in *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pp. 210–219, IEEE, 1999.

[102] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *Software Engineering, IEEE Transactions on*, vol. 38, no. 2, pp. 278–292, 2012.

[103] E. M. Guerra and C. T. Fernandes, "Refactoring test code safely," in *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, pp. 44–44, IEEE, 2007.

[104] A. Van Deursen and L. Moonen, "The video store revisited–thoughts on refactoring and testing," in *Proc. 3rd Int'l Conf. eXtreme Programming and Flexible Processes in Software Engineering*, pp. 71–76, Citeseer, 2002.

[105] J. U. Pipka, "Refactoring in a "test first"-world," in *Proc. Third Int'l Conf. eXtreme Programming and Flexible Processes in Software Eng*, 2002.

[106] E. L. Alves, P. D. Machado, T. Massoni, and S. T. Santos, "A refactoring-based approach for test case selection and prioritization," in *Automation of Software Test (AST), 2013 8th International Workshop on*, pp. 93–99, IEEE, 2013.

[107] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, pp. 25–33, ACM, 2006.

[108] S. Kimura, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Does return null matter?," in *IEEE Conference on Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE)*, pp. 244–253, 2014.

[109] M. Eisenstadt, "My hairiest bug war stories," *Communications of the ACM*, vol. 40, no. 4, pp. 30–37, 1997.

[110] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, (San Jose, CA, USA), pp. 437–440, July 23–25 2014.