STAMP

Deliverable D3.2

Log Optimization and Benchmarking

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D3.2
Title of Deliverable	:	Log Optimization and Benchmarking
Dissemination Level	:	Public
Version	:	1.2
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d32_crash_replication_benchmark.pdf
Contractual Delivery Date	:	M12 - November 30, 2017 / Revision: M24 - November 30, 2018
Contributing WPs	:	WP 3
Editor(s)	:	Xavier Devroey, TU Delft
Author(s)	:	Pouria Derakhshanfar, TU Delft Xavier Devroey, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
Reviewer(s)	:	Benoit Baudry, KTH Jesus Gorronogoitia Cruz, Atos Caroline Landry, INRIA

Abstract

Crash reproduction approaches help developers during debugging by generating a test case that reproduces a given crash. Several solutions have been proposed to automate this task. However, the proposed solutions have been evaluated on a limited number of projects, making comparison difficult.
In this deliverable, we enhance this line of research by proposing JCrashPack, an extensible benchmark for Java crash reproduction, together with ExRunner, a tool to simply and systematically run evaluations. JCrashPack contains 200 stack traces from various Java projects, including industrial open source ones, on which we run an extensive evaluation of the state-of-the-art tool for search-based crash reproduction.
The existing tool successfully reproduced 45% of the crashes. Furthermore, we observed that reproducing NullPointerException, IllegalArgumentException, and IllegalStateException is relatively easier than reproducing ArrayIndexOutOfBoundsException and StringIndexOutOfBoundsException. Our results include a detailed manual analysis of the outputs, from which we derive 12 current challenges for crash reproduction. Finally, based on those challenges, we discuss future research directions for search-based crash reproduction for Java.

Revision History

Version	Type of Change	Author(s)
1.0	initial version	Xavier Devroey, TU Delft Pouria Derakhshanfar, TU Delft Arie van Deursen, TU Delft
1.1	updating document information (editor, reference, etc)	Caroline Landry, INRIA
1.2	update after additional review	Xavier Devroey, TU Delft Andy Zaidman, TU Delft

Contents

Acronyms

EC	European Commission
WP	Work Package
TUD	TU Delft
AEon	ActiveEon
Eng	Engineering

Glossary

This glossary presents the terminology used across the different deliverable of work package 3.

Botsing: Meaning *crash* in Dutch, Botsing is a complete re-implementation and extension of the crash replication tool EvoCrash. Whereas EvoCrash was a full clone of EvoSuite (making it hard to update EvoCrash as EvoSuite evolves), Botsing relies on EvoSuite as a (maven) dependency only and provides a framework for various tasks for crash reproduction and, more generally, test case generation. Furthermore, it comes with an extensive test suite, making it easier to extend. The license adopted is Apache, in order to facilitate adoption in industry and academia. Botsing is the name of the framework for online test amplification developed in WP3.

Code instrumentation: Code instrumentation in Botsing consists in the injection of probes into the bytecode of a Java application to monitor and log the runtime behavior of specific classes.

JCrashPack: JCrashPack is a benchmark containing 200 crashes to assess crash replication tools capabilities.

Model seeding: in search-based software testing, seeding consist in providing external information to the search algorithm to help the exploration of the search space. Model seeding, developed within STAMP, is the seeding of transition systems (a formalism similar to state machines describing a dynamic behavior) to a search based test case generation algorithm. The models represent the common usages of classes and allow the generate objects with sequences of method calls, representing a common behavior in the application. In Botsing, models are learned from the source code (static analysis) and from the logs produced by the execution of the system (dynamic analysis using instrumentation) using n-gram inference.

Stack trace: a (crash) stack trace is a piece of log data usually denoting a crash failure. Stack traces provides information on the exception thrown and on the propagation of that exception trough the stack of method calls.

Stack trace preprocessing: stack traces can contain redundant and useless information preventing to automatically reproduce the associated crash. The preprocessing allows to filter stack traces to keep only relevant information.

Introduction

When a user reports a crash, for instance, through an issue tracker, the first step followed by the developer is to try to reproduce the problem in the development environment [41]. For Java programs, the crash occurs when an exception is thrown and not handled by the program. The developer tries to reproduce it to understand its cause, then fix the bug, and add a (non-)regression test to avoid reintroducing the bug in future versions. Manually reproducing a crash may be challenging and labor-intensive as it requires to have knowledge of the system's components involved in the crash.

To help developers in their task, several automated crash reproduction methods, relying on different techniques, have been proposed [13, 30, 31, 39, 11, 37, 36]. One of the most promising approaches uses search-based software testing [37, 36] to generate a test case able to reproduce a crash. This approach has initially been evaluated on a limited number of crashes and outperformed other approaches based on backward symbolic execution [13], test case mutation [39], and model-checking [31]. However, the crashes used in the evaluation were not selected to reflect crashes happening in the current state-of-the-practice.

To facilitate sound empirical evaluation on automated crash reproduction approaches, we devise a new benchmark of real-world crashes, called JCrashPack. It contains 200 crashes from seven actively maintained open-source and industrial projects. These projects vary in their domain application and include:

- crashes extracted from the issue tracker of XWiki using a crawler we developed;
- crashes extracted from the issue tracker of Elasticsearch, a distributed RESTful search engine;
- crashes studied by the state-of-the-art of fault localization [24] coming from several popular open source APIs, and a mocking framework for unit testing Java programs.

JCrashPack is extensible, and can be used for large-scale evaluation and comparison of automated crash reproduction techniques for Java programs.

We evaluated the application of search-based software testing [37, 36] for crash reproduction on JCrashPack, analyzed all failed reproductions, and distilled 12 classes of research and engineering limitations that negatively affected reproducing crashes. The results reported in this deliverable guide our current future developments of search-based crash replication (see D3.3).

The remainder of this document is organized as follows:

Chapter 1 - JCrashPack design details the design and crash collection choices for JCrashPack.

Chapter 2 - Running Experiments with ExRunner details the implementation and usage of ExRunner to benchmark crash reproduction tools with JCrashPack.

Chapter 3 - Evaluation of state-of-the-art search-based crash reproduction details the evaluation using JCrashPack.

Chapter 4 - Log optimization through stack trace preprocessing.

Chapter 5 - Future work presents future work, based on the findings of Chapter 3.

Summary of Artifacts

1. **JCrashPack** (see Chapter 1)
   - Link: https://github.com/STAMP-project/JCrashPack/releases/tag/1.0.0
   - Contact: Xavier Devroey (TUD)
2. **ExRunner** (see Chapter 2)
   - Link: https://github.com/STAMP-project/ExRunner/releases/tag/1.0.0
   - Contact: Pouria Derakhshanfar (TUD)
3. **EvoCrash-JCrashPack-application** (see Chapter 3)
   - Link: https://github.com/STAMP-project/EvoCrash-JCrashPack-application/releases/tag/1.0.0
   - Contact: Xavier Devroey (TUD)
4. **jira-stacktrace-crawler** (see Chapter 1)
   - Link: https://github.com/STAMP-project/jira-stacktrace-crawler/releases/tag/1.0.0
   - Contact: Pouria Derakhshanfar (TUD), Luca Andreatta (Eng)

Chapter 1

JCrashPack design

Benchmarking is a common practice to assess a new technique and compare it to the state of the art [35]. For instance, SF110 [20] is a sample of 100 Java projects from SourceForge, and 10 popular Java projects from Github, that may be used to assess (search-based) test case selection techniques. In the same way, Defects4J [24] is a collection of bugs coming from popular open-source projects: for each bug, a buggy and a fixed version of the projects are provided. Defects4J is used to assess various testing techniques like test case selection or fault localization. To the best of our knowledge, there is no standard benchmark to assess and compare crash reproduction techniques.

In their previous work, Soltani et al. [37], Xuan et al. [39], and Chen and Kim [13] used Apache Commons Collections [4], Apache Ant [3], and Apache Log4j [5] libraries. In addition to Apache Ant and Apache Log4j, Nayrolles et al. [31] used bug reports from 8 other open-source software. Here, we enhance previous efforts to build a benchmark dedicated to crash reproduction by collecting cases coming from both state of the art literature and actively maintained industrial open-source projects with well documented bug trackers.

1.1 Projects selection protocol

To be able to perform analysis of the results of a crash reproduction attempt, we need as much information as possible. The projects considered to be part of the benchmark must have openly accessible binaries, source code, and stack traces (via an issue tracker for instance). Those projects should be under active maintenance to be representative of current software engineering practices and ease communication with developers. We have no restriction on the stack traces to consider, but want some diversity in the kinds of projects: APIs, Web-applications, etc. For this first version of our benchmark, we privilege quality over quantity: a small set of projects with many stack traces representing different types of crashes, as the effort of manual analysis and setting up test environment can be substantial.

To build our benchmark, we took the following approach. First, we investigated projects collected in SF110 [20] and Defects4J [24] as state of the art benchmarks. However, as most projects in SF110 have not been updated since 2010 *or even earlier*, we discarded them from our analysis. From Defects4J, we collected cases where bugs correspond to actual crashes: i.e., the execution of the test case highlighting the bug in a given buggy version of a project generates a stack trace that is not a test case assertion failure.

To reflect the current state-of-the-practice and increase the impact of a benchmark, it is important to include projects that are popular and attractive to end-users. Additionally to projects from Defects4J, we selected two industrial open-source projects: XWiki [40] and Elasticsearch [16]. XWiki is a popular enterprise wiki management system and a reactive use case provider of the STAMP project. And Elasticsearch, a distributed RESTful search and analytic engine, is one of the ten most popular

projects on GitHub[1]. To identify the top ten popular projects from Github, we took the following approach:

1. we queried the top ten projects that had the most number of forks;
2. we queried the top ten projects that had the most number of stars;
3. we queried the top ten most-trending projects; and
4. took the intersection of the three.

Four projects were shared among the above top-ten projects, namely: Java-design-patterns [22], Dubbo[15], RxJava [34], and Elasticsearch. To narrow down the scope of the study, we selected Elasticsearch, which ranked the highest among the four shared projects and uses a micro-services architecture, making this choice highly relevant for the STAMP context.

1.2 Stack trace collection

For each project, we collected stack traces to be reproduced as well as the project binaries, with specific versions on which the exceptions happened.

1.2.1 Projects from Defects4J

From the 395 buggy versions of the Defects4J projects, we manually inspected the stack traces generated by the failing tests and collected those which are not JUnit assertion failures (i.e., those which are due to an exception thrown by the code under test and not by the JUnit framework). For instance, we only consider the first and second frames (lines 1 and 2) for the following stack trace from the Joda-Time project:

```
0 java.lang.IllegalArgumentException:
    at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
    at org.joda.time.TestPartialBasics.testWithbaseAndArgHaveNoRange(...)
```

The third and following lines concern testing classes of the project, which are irrelevant for crash reproduction. They are removed from the benchmark, resulting in the following stack trace with two frames:

```
0 java.lang.IllegalArgumentException:
    at org.joda.time.Partial.<init>(Partial.java:224)
2   at org.joda.time.Partial.with(Partial.java:466)
```

We proceeded in the same way for each Defects4J project and collected a total of 73 stack traces coming from five (out of the six) projects: 2 from JFreeChart, 22 from Commons-lang, 27 from Commons-math, 14 from Mockito, and 8 from Joda-Time. All the stack traces generated by the Closure compiler test cases are JUnit assertion failures.

[1]This selection was performed on 26/10/2017.

1.2.2 Elasticsearch

Crashes for Elasticsearch are publicly reported to the issue tracker of the project on GitHub[2]. Therefore, we queried the reported crashes, which were labelled as bugs, using the following string `"exception is:issue label:bug"`. From the resulting issues (600 approx.), we manually collected the most recent ones (reported since 2016), which addressed the following:

1. the version which crashed was reported,
2. the issue was discussed by the developers and approved as a valid crash to be fixed.

The above manual process resulted in 76 crash stack traces.

1.2.3 XWiki

XWiki is an open source project which has a public JIRA issue tracker[3]. We investigated first 1000 issues which are reported for XWIK-7.2 (released in September 2015) to XWIK-9.6 (released in July 2017). We selected the issues where:

1. the stack trace of the crash was included in the reported issue, and
2. the reported issue was approved by developers as a valid crash to be fixed.

Eventually, we selected a total of 51 crashes for XWIKI.

1.3 JIRA issue tracker crawler

During the collection of stack traces, we developed a crawler for JIRA to reduce the effort and automatically extract issues with a reported stack trace (in XWiki for now). Given a specific version of a software, the crawler will search for reported stack traces in a JIRA issue tracking system.

1.3.1 Command line interface

Here is the step-by-step explanation on how to run the crawler:

1. Set the JIRA issue tracker link by modifying the `searchLink` variable in `src/main/java/Crawler.java`.
2. Set the versions of the software to look for by modifying the `versions` array in `src/main/java/Crawler.java`.
3. Compile and run `/src/main/java/Crawler.java`.
4. The collected stack traces are saved in `stack_traces/`.

1.3.2 Implementation

This tool uses Selenium, a library that automates web browsing. The crawler gets the software versions as input arguments and uses the JIRA Query Language to search the issues linked to the given versions. For each issue, it scan its contents to detect stack traces using regular expressions. The crawler cleans and saves each stack trace in a dedicated file, named after the issue identifier.

The JIRA crawler will be reused and extended by Engineering in WP4, as part of the tools integration effort, into a full fledged JIRA plugin. This will allow to spot future stack traces in the JIRA issue tracker and execute the STAMP tools in a continuous integration server.

[2] https://github.com/elastic/elasticsearch/issues
[3] https://jira.xwiki.org/browse/XWIKI/

Table 1.1: Number of stack traces (#), and average number of frames ($\overline{fr.}$) with standard deviation (σ) for Defects4J, Elasticsearch and Xwiki projects for exceptions: NullPointerException (NPE), IllegalArgumentException (IAE), ArrayIndexOutOfBoundsException (AIOOBE), ClassCastException (CCE), StringIndexOutOfBoundsException (SIOOBE), IllegalStateException (ISE), and other exceptions types (Oth.)

Except.	**Defects4J**			**XWiki**			**Elasticsearch**			**Total**		
	#	$\overline{fr.}$	σ	#	$\overline{fr.}$	σ	#	$\overline{fr.}$	σ	#	$\overline{fr.}$	σ
NPE	11	2.3	1.5	20	26.8	33.3	18	12.3	9.8	49	16.0	23.9
IAE	10	2.1	1.7	4	9.8	3.7	10	15.2	9.2	24	8.8	8.5
AIOOBE	8	4.1	4.3	0			6	17.0	18.0	14	9.6	13.3
CCE	4	3.2	4.5	6	21.8	22.2	0			10	14.4	19.3
SIOOBE	7	1.6	1.0	1	8.0	NA	1	15.0	NA	9	3.8	4.8
ISE	0			0			7	19.3	11.9	7	19.3	11.9
Oth.	33	4.7	3.1	20	34.4	47.0	34	21.1	13.4	87	17.9	26.3
Total	73	3.5	3.0	51	27.5	37.0	76	17.7	12.5	200	15.0	22.3

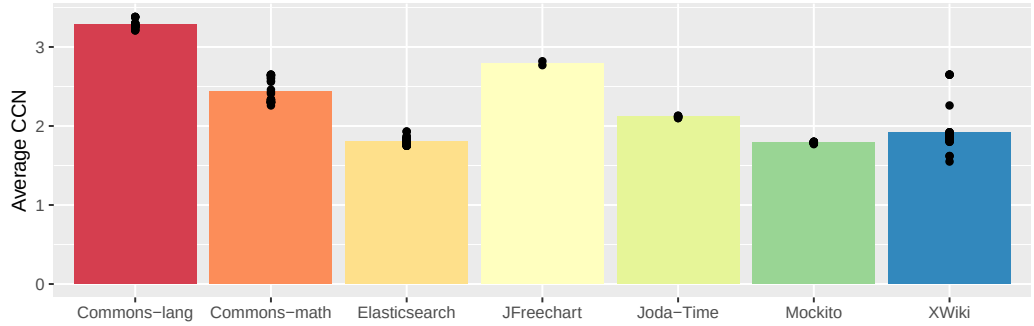

Figure 1.1: Average methods Cyclomatic Complexity Number (CCN) per project

1.4 The JCrashPack benchmark

The result of our selection protocol is an extensible benchmark with 200 stack traces called *JCrashPack*. For each stack trace, based on the information from the issue tracker (or from the Defects4J command line tool), we collected: the *Java project* in which the crash happened, the *version* of the project where the crash happened and (when available) the *fixed version* or the fixing commit reference of the project; the *buggy frame* (i.e., the frame in the stack trace targeting the method where the bug is located or manifests itself); and the *Cyclomatic Complexity Number (CCN)* and the *Non-Commenting Sources Statements (NCSS)* of the project. Filtering, verifying and cleaning up stack traces and issues to produce JCrashPack with its collection of stack traces and binaries involved an important manual effort.

JCrashPack is available on GitHub.[4] We provide guidelines to add new crashes to the benchmark and make a pull request to include them in the JCrashPack master branch. Table 1.1 shows the distribution of stack traces per exception type for the six most common ones, the *Others* category denoting remaining exception types. 73 stack traces come from Defects4J projects. For the remaining stack traces, 51 come from XWiki and 76 come from Elasticsearch. The smallest stack traces have one frame and the largest, a user-defined exception in *Oth.*, has 175 frames.

In search-based crash reproduction, the complexity of the source code has a significant impact. Complex code are harder to handle and may require specific treatments during the search process.

[4] At https://github.com/SERG-Delft/JCrashPack

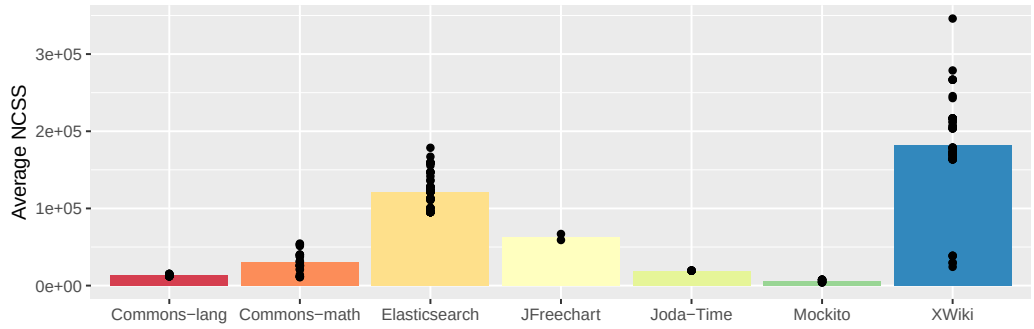

Figure 1.2: Average Non-Commenting Sources Statements (NCSS) per project

Figures 1.1 and 1.2 present the distribution of the average methods Cyclomatic Complexity Number (CCN) per project and the Non-Commenting Sources Statements (NCSS) per project. Those measures are commonly used to give an idea of the complexity of a project. For Defects4J, the least complex project is Mockito (CCN=1.77) and the most complex one is Commons-lang (CCN=3.38). The largest project is JFreeChart (66,938 statements) and the smallest one is Mockito (3,937 statements). It testifies of the diversity of the projects considered for JCrashPack. The detailed numbers for each stack trace and its project are available on the JCrashPack website.

1.5 Extending JCrashPack

JCrashPack can be easily extended. We recap here the procedure described on the JCrashPack website. Future work with Activeeon and OW2 involves adding new crashes from open source projects to JCrashPack coming from their open GitHub issue tracker.[5]

1.5.1 Adding a stack trace

Stack traces must be provided in a text file with the following format: `<issue-name>.log`. To add the file to JCrashPack, you need to follow the following steps:

- Add a `<project>` directory to `crashes/`. For instance, `crashes/XWiki`.
- Add a `<issue-name>` directory to `crashes/<project>`. For instance, `crashes/XWiki/XCOMMONS-928`. The `<issue-name>` should denote the issue in the `<project>` from which the stack trace has been collected.
- Add a `<issue-name>.log` file to `crashes/<project>/<issue-name>`. For instance, `crashes/XWiki/XCOMMONS-928/XCOMMONS-928.log`. File `<issue-name>.log` contains the stack trace without any other information.

Once the file is added, you need to update `jcrashpack.json` with the information on the stack trace:

- Edit `jcrashpack.json`.
- Under crashes, add an entry with `<issue-name>` as the key.
- Set application value to `<project>`.

[5] https://github.com/ow2-proactive/scheduling/issues

- If the information is available, set the `buggy_frame` value to the frame in the stack trace where the bug is, the commit where it was fixed (`fixed_commit`), the `URL` to the issue in an issue tracker, and the version of the project for which the issue has been fixed (`version_fixed`).
- Set the id value to `<issue-name>`.
- Provide a regular expression for `target_frames` designing the target frames that have to be considered in the stack trace for crash replication. For instance, `.*xwiki.*` allows to consider only target frames in the stack trace where `xwiki` appears. This allows to discard frames from other projects and JDK API.
- Set the version value to `<version>`.

1.5.2 Adding projects

If the stack trace requires to add another application to JCrashPack, you have to provide the binaries and sources of this application using the following steps:

- Add a `<project>` directory to `applications/`. For instance, `applications/XWiki`.
- Add a `<version>` directory to `applications/<project>`. For instance, `applications/XWiki/7.2`. `<version>` is the version of the `<project>` that will be used by to replicate the stack trace.
- Put the Jar file of version `<version>` of the `<project>` and all its dependencies in `applications/<project>/<version>/bin` and add a zip file (or a split zip file) with the source code of the application in `applications/<project>/<version>/src.zip` for future analysis.

Once the files are added, you need to update `jcrashpack.json` with the information on the application:

- Edit `jcrashpack.json`.
- Under applications, add an entry with `<project>` as the key.
- Add a `name` and `URL` for the application.
- Add an entry under versions with `<version>` as the key, provide a `URL` for the source code (optional), and set `version` value to `<version>`.

Chapter 2

Running Experiments with ExRunner

Together with JCrashPack, we provide ExRunner, a tool that eases the experimentation with a given stack trace-based crash reproduction tool.

2.1 Architecture

Figure 2.1 gives an overview of its architecture. The *job generator* takes as input the stack traces to reproduce, the path to the Jar files associated to each stack trace, and the configurations to use for the stack trace reproduction tool under study.

For each stack trace, the job generator analyses the stack frames and discards those with a target method that does not belong to the target software, based on the package name. For instance, frames with a target method belonging to the Java API or other external dependencies are discarded from the evaluation. For each pair of configuration and stack trace, the job generator creates a new job description (i.e., a JSON object with all the information needed to run the tool under study) and adds it to a queue.

2.1.1 Multithreading

To speed-up the evaluation, ExRunner multi-threads the execution of the jobs. The number of threads (5 by default) is provided by the user in the configuration of ExRunner and depends on the resources available on the machine and required by one job execution. Each thread picks a job from the waiting queue and executes it. ExRunner users may activate an observer that monitors the jobs and takes care of killing (and reporting) those that do not show any sign of activity (by monitoring the job outputs) for a user-defined amount of time. The outputs of every job are written to separate files, with the generated test case (if any) and the results of the job execution (output results from the tool under study).

For instance, when used with search-based crash reproduction, the log files contain data about the target method, progress of the fitness function value during the execution, and branches covered by the execution of the current test case (in order to see if the line where the exception is thrown is reached). In addition, the results contain information about the progress of search (best fitness function, best line coverage, and if the target exception is thrown), and number of fitness evaluations performed by search-based crash reproduction in an output CSV file. If search-based crash reproduction succeeds to replicate the crash, the generated test is stored separately.

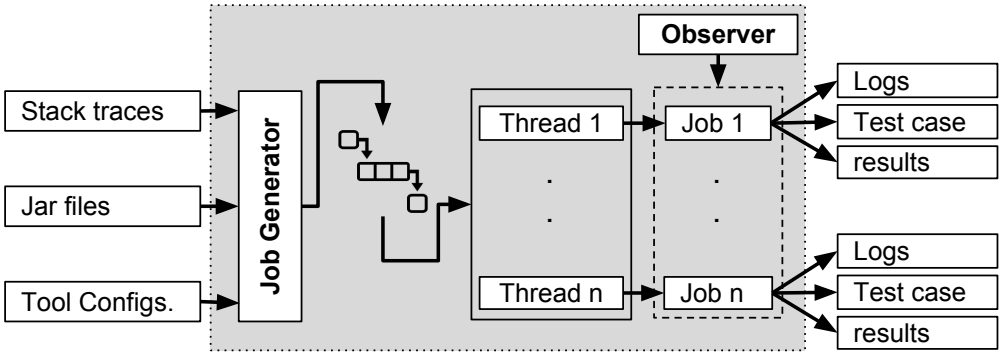

Figure 2.1: ExRunner overview

2.1.2 Timeout

As mentioned by Fraser et al. [17], any research tool developed to handle Java programs may face challenges due to certain peculiarities of Java. One of these is long (or infinite) test execution time. In order to handle this problem, ExRunner creates an observer to check the status of each thread executing an search-based crash reproduction instance. By default, if one search-based crash reproduction execution does not respond for 10 minutes (66% of the expected execution time), the Python script uses a bash command (`thread.terminate()`) to kill the search-based crash reproduction process and all of its spawned threads.

2.1.3 Garbage collection

Another challenge relates to garbage collection: we noticed that, at some point of the execution, one job (i.e., one JVM instance) allocated all the CPU cores for the execution of the garbage collector, preventing other jobs to run normally. Moreover, since search-based crash reproduction allocates a large amount of heap space to each sub-process responsible to generate a new test case (since the execution of the target application may require a large amount of memory) [17], the garbage collection process could not retrieve enough memory and got stuck, stopping all jobs on the machine. To prevent this behavior, we set *-XX:ParallelGCThreads* JVM parameter to 1, enabling only one thread for garbage collection, and limited the number of parallel threads per machine, depending on the maximal amount of allocated memory space. Using the logging mechanism in search-based crash reproduction, we are able to check if a lack of memory happened for each individual execution or not.

ExRunner is available together with JCrashPack. It has been designed to be extensible to other available stack trace reproduction tools.

2.2 Run other crash replication tools with ExRunner

To perform crash replication benchmarking with another tool, ExRunner has to be modified to match input and output formats of this tool.

2.2.1 Implementing an adaptor for the new tool

ExRunner uses an adaptor to handle execution of crash replication tool. This adaptor is a class with a `main` method that is called for each line in `input.csv`. For instance, the adaptor for search-based crash reproduction is defined in `JarFiles/src/Run.java`.

2.2.2 Modifying the input CSV file

Arguments for the crash replication tool are defined in `/pythonScripts/inputs/input.csv`. Each line of this CSV file is one stack trace with specific configurations. Indexes of the columns in `input.csv` that are passed to the adaptor are specified in `useful_indexes` array in `pythonScripts/input.py`. For instance, if the needed configurations are only in 2 first columns of `input.csv`, `useful_indexes` would be `[0,1]`. However, `application`, `case`, and `round` columns are mandatory for CSV file.

2.2.3 Customizing the `csv_results` array in `RunJar.py`

There is a for loop at line 147 of `pythonScripts/RunJar.py` that checks each line of output and saves data to `csv_results` array. Depending on the output of the crash replication tool, this loop has to be modified accordingly.

2.2.4 Collecting execution logs

ExRunner writes results and output logs of the tool in `pythonScripts/outputs/logs`.

Chapter 3

Evaluation of state-of-the-art search-based crash reproduction

We used JCrashPack to perform an extensive evaluation of search-based crash reproduction. Our first research question deals with the capability of search-based crash reproduction to reproduce crashes from JCrashPack:

RQ$_1$ *To what extent can search-based crash reproduction reproduce crashes from JCrashPack?*

Exception types are inherently different. Therefore, reproducing different types of exceptions may be of varying degree of complexity. In addition, different types of projects, and whether they are industrial or not, might have impact on how costly it is to reproduce the reported crashes for them. The second research question studies the influence of the exception type and project type on performance of search-based crash reproduction:

RQ$_2$ *How do project type and exception type influence performance of search-based crash reproduction for crash reproduction?*

The last research question studies open problems and potential future research directions:

RQ$_3$ *What are the open problems that need to be solved to enhance search-based crash reproduction?*

Due to the randomness of Guided Genetic Algorithm in search-based crash reproduction, we executed the tool 10 times on each frame. We ran search-based crash reproduction with the default parameter values [20]. In addition, we chose to keep the reflection mechanisms, used to call private methods, deactivated. The rationale behind this decision is that using reflection leads to generating invalid objects by breaking the class invariant [26] during the search, which results in test cases helplessly trying to reproduce a given crash [13]. Moreover, we activated the implementation of functional mocking in EvoSuite [10] in order to minimize possible risks of environmental interactions on crash reproduction.

3.1 Evaluation setup

Since our evaluation is executed in parallel on 2 different machines, we choose to express the budget time in terms of number of fitness evaluations: i.e., the number of times the fitness function is called to evaluate a generated test case during the execution of the guided generic algorithm. We set this number to 62,328. It corresponds to the average number of fitness evaluations performed by search-based crash reproduction when running it during 15 minutes on each frame of a subset of 4 randomly

Table 3.1: Five types of execution outcomes for each stack trace frame.

Abbrev.	Description
reproduced	search-based crash reproduction successfully reproduced the crash at this frame level
line reached	the target line is covered, but no exception is thrown
ex. thrown	the target line was reached, and the target exception was thrown but the generated stack trace does not contain all original frames
failed	search-based crash reproduction did not produce any tests
crashed	search-based crash reproduction crashed

selected stack traces on one out of our two machines. Both of the machines have the same configuration: a cluster running Linux Ubuntu 14.04.4 LTS with 20 CPU-cores, 384 GB memory, and 482 GB hard drive.

We partitioned the evaluation into 2, one per available machine: all the stack traces with the same kind of exception have been run on one machine for 10 rounds.

For each run, we measure the number of fitness evaluations needed to achieve reproduction (or the exhaustion of the budget if search-based crash reproduction fails to reproduce the crash) and the best fitness value achieved by search-based crash reproduction (0 if the crash is reproduced and higher otherwise).

The whole process is managed using ExRunner and the setup of the evaluation infrastructure took about 1 month. The evaluation itself was executed during 10 days on our 2 powerful machines.

3.2 Evaluation results

In this section we present the results of our evaluation. Thereby, we answer the first two research questions on

1. the extent to which the selected crashes were reproduced with search-based crash reproduction, and
2. the impact of project and exception type on performance of search-based crash reproduction.

For each stack trace, we executed search-based crash reproduction on each frame 10 times to account for randomness of the algorithm. Therefore, to analyze the results for each frame, we consider the most frequent outcome from the 10 rounds of executions, namely: *reproduced*, *line reached*, *ex. thrown*, *failed*, and *crashed*. Table 3.1 summarizes the description for these labels. In addition, to answer the second research question, we only consider the *reproduced* frames, and measure the average number of fitness evaluations for them.

3.2.1 Crash Reproduction Outcomes (RQ1)

Figure 3.1 presents an overview of the results, grouped by exceptions and applications. Overall, search-based crash reproduction reproduced 154 frames (out of 1,885), from 77 different crashes (out of 200). For 180 frames (from 87 crashes), search-based crash reproduction produced a crash-reproducing test at least once. In total, search-based crash reproduction crashed for 1,027 frames, 581 of which were from Elasticsearch.

For the projects from Defects4J, in total, 82 (out of 247) of the frames (33%) were reproduced. In these projects, the total number of *crashed* and *failed* frames are 61 and 23, respectively. In particular, only 4 frames out of 63 frames for Mockito were successfully reproduced. For instance, search-based

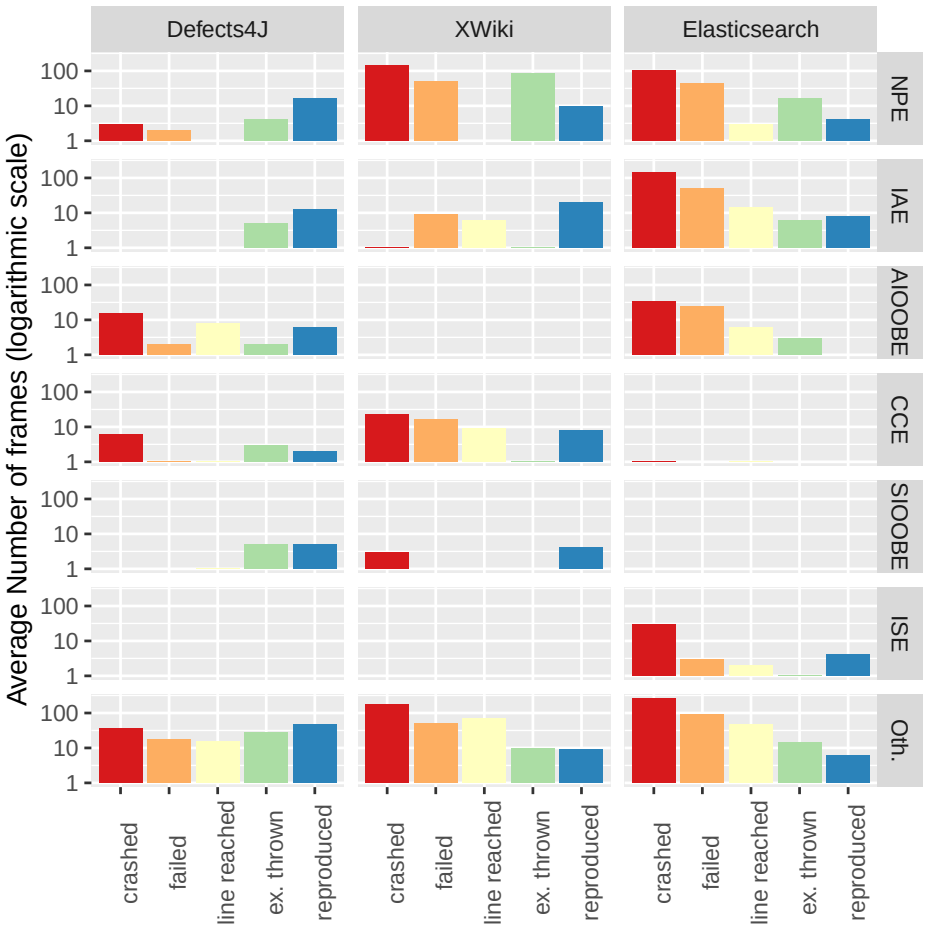

Figure 3.1: An overview of the reproduction outcome, grouped by exceptions and applications.

crash reproduction could not reproduce `MOCKITO-4b`, which has only one frame. From our evaluation, we observe that one very common problem when trying to reproduce a *ClassCastException* is to find which class should be used to trigger the exception.

```
public void noMoreInteractionsWantedInOrder(Invocation undesired){
  throw new VerificationInOrderFailure(join( ...,
    "..." + undesired.getMock() + "':", ...));
}
```

The exception happens when the `undesired.getMock()` call returns an object that cannot be cast to `String`. During the search, search-based crash reproduction mocks the `undesired` object and assigns some random value to return when the `getMock` method is called. But, since the signature of this method is `Object getMock()`, search-based crash reproduction assigns only random `Object` values to return, where, from the original stack trace, a `Boolean` value is required to trigger the exception.

XWiki is one of the industrial open source cases in the evaluation, for which 51 (out of 706) frames (8%) were successfully reproduced. For 80 frames (11%), search-based crash reproduction generated

Table 3.2: Statistics for the average number of fitness evaluations for the *reproduced* frames (**Fr.**) belonging to different crashes (**Cr.**), grouped by applications. Confidence Interval (CI) is calculated for median with bootstrapping with *100,000* runs, at 95% confidence level.

Applications	Cr.	Fr.	Min	Lower Quart.	Median CI	Median	Upper Quart.	Max
Commons-lang	19	213	1	2	[5,22]	15.00	237.0	52,240
Commons-math	24	471	1	13	[124,211]	178.00	1,046.5	58,731
Mockito	2	40	1	1	[1,1]	1.00	5.2	138
Joda-Time	6	138	1	15.5	[79,372]	253.50	1,290.2	40,189
JFreechart	1	41	1	10	[-292,350]	221.00	1,188.0	20,970
XWiki	23	506	1	2	[10.5,23]	19.00	171.2	34,089
Elasticsearch	15	218	1	3	[6,30]	20.00	128.2	17,461

Table 3.3: Statistics for the average number of fitness evaluations for the *reproduced* frames (**Fr.**) belonging to different crashes (**Cr.**), grouped by exceptions. Confidence Interval (CI) is calculated for median with bootstrapping with *100,000* runs, at 95% confidence level.

Exception kind	Cr.	Fr.	Min	Lower Quart.	Median CI	Median	Upper Quart.	Max
NPE	24	308	1	5	[12,49]	33.50	174.2	34,089
IAE	17	406	1	2	[7,12]	10.00	49.8	38,907
AIOOBE	5	58	1	15.5	[252,1104.5]	675.00	1,671.2	53,644
CCE	6	103	1	6.5	[74,210]	120.00	560.0	10,197
SIOOBE	8	95	1	12.5	[122,945]	505.00	2,326.0	52,240
ISE	2	40	1	1	[0.5,2]	1.50	3.2	130
Oth.	28	617	1	8	[102,160]	140.00	962.0	58,731

tests marked as *ex. thrown*, which means that the tests covered the target line and triggered the target exception, however, the generated stack traces were not entirely similar to the original traces. XWiki also has, 385 and 111 frames, which are labeled as *crashed* and *failed*, respectively.

The majority of the *crashed* frames are from Elasticsearch (581 out of 1027, 57%). In addition, no frame from this project was reproduced. Based on our observations, frequent use of *Java generics* and *static initialization*, and most commonly, automatically generating suitable input data that resembles `http` requests are among the major reasons for the encountered challenges for reproducing Elasticsearch crashes. In Section 3.2.3 we will describe **12** categories of challenges that we identified as the underlying causes for the presented execution outcomes.

Moreover, 28 frames (out of 461), 6%, of NPEs were reproduced, 80 frames, 17%, were labeled as *ex. thrown*, and 264 frames crashed. As for IAE, 41 (out of 278) frames, 15%, were reproduced, 59 frames, 21%, failed, and 146 frames, 53%, crashed. In the case of AIOOBE, 6 (out of 99) frames were reproduced, 26 frames failed, and 48 frames crashed. For CCE, 11 (out of 72) frames were reproduced, while 8 frames failed, and 46 frames crashed. For SIOOBE, 7 (out of 19) frames were reproduced, no frame failed, and 4 frames crashed. For ISE, 4 (out of 24) frames were reproduced, 3 frames failed, and 14 frames crashed. Finally, for the other kinds of exceptions, 58 (out of 932) frames were reproduced, 169 frames failed, and 505 frames crashed.

3.2.2 Impact of Exception Type and Project on Performance (RQ2)

To identify the distribution of fitness evaluations per exception type and project, we filtered the *reproduced* frames, for which the fitness value turned 0.0 the most frequently. Tables 3.2 and 3.3 present the statistics for these executions, grouped by application and exception types, respectively.

We filtered out the frames that were not reproduced to analyze the impact of project and exception types on the average number of fitness evaluations. Following Arcuri and Briand [8] recommendations, we replaced the test of statistical difference by a confidence interval. For both groups, we calculated confidence intervals with 95% confidence level for medians with bootstrapping with $100,000$

Listing 3.1: Stack trace for the crash ES-25878

```
java.lang.NullPointerException
 at org.[...].ElasticsearchException.guessRootCauses([...]:618)
 at org.[...].ElasticsearchException.generateFailureXContent([...]:563)
[...]
```

Listing 3.2: Code excerpt from Elasticsearch `ElasticsearchException.java`

```
public static ElasticsearchException[] guessRootCauses(Throwable t) {
  Throwable ex = ExceptionsHelper.unwrapCause(t);
  if (ex instanceof ElasticsearchException) {
    return ((ElasticsearchException) ex).guessRootCauses();
  }
  return new ElasticsearchException[]{new ElasticsearchException(t.getMessage(), t
      ) {
    @Override
    protected String getExceptionName() {
      return getExceptionName(getCause());
    }
  }};
}
```

runs.[1]

As Table 3.2 shows, for four projects, namely: Commons-lang, Mockito, XWiki, and Elasticsearch, the medians for cost of fitness evaluations is low. On the other hand, the cost of crash reproductions for Commons-math, Joda-Time, and JFreechart are higher in comparison to the rest of projects. Those trends need to be further confirmed, giving the low number of crashes reproduced for Mockito, Joda-Time, and JFreechart.

As Table 3.3 shows, we observe that for *CCE*, *SIOOBE*, and *AIOOBE*, the cost of generating a crash-reproducing test case is high, while for *NPE*, *IAE*, and *ISE*, the cost is lower. One possible explanation could be that generating input data which is in a suitable state for causing cast conflicts, or an array which is in the right state to be accessed by an illegal index is often non-trivial. In contrast, to trigger an NPE, it is often enough to pass `null` as input parameters to the target method. For example, Listing 3.1 shows an excerpt of the crash stack trace that is generated for the Elasticsearch code snippet shown in Listing 3.2. To reproduce the first frame (line 1 in Listing 3.1) of the stack trace, the generated test case in Listing 3.3 calls the method `guessRootCauses` with a `null` parameter value.

Considering the presented results in Figure 3.1 and Table 3.2, crash replication for various exceptions may be dependent on project type. Figure 3.2 presents the results of crash reproduction grouped both by applications and exception types. As the table shows, the cost of reproducing NPE is lower for Elasticsearch, compared to XWiki and JFreechart, and the cost of reproducing IAE is lower for Commons-lang than for Elasticsearch. We also observe differences in terms of costs of reproducing AIOOBE and SIOOBE for different projects. Our results suggest that further investigation in terms of possible correlations between application types and exception types are needed.

[1] We used the *boot* function from the *boot* library in R to compute the *basic* intervals with bootstrapping. See https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results to reproduce the statistical analysis.

Listing 3.3: The test case generated by EvoCrash for reproducing Elasticsearch-25878

```
@Test
public void testNPE() throws Throwable {
     ElasticsearchException.guessRootCauses((Throwable) null);
}
```

3.2.3 Challenges for crash reproduction (RQ3)

To identify open problems and future research directions, we manually analysed the execution logs of 316 frames that could not be reproduced in at least one of the 10 executions. This analysis includes a description of the reason why a frame could not be reproduced. [2] Based on those descriptions, we grouped the reason of the different failures into 12 categories and identify future research directions. This manual process took 9 person-weeks in total.

We detail each category hereafter and provide the number of frames classified in each one between braces.

Input data generation

Trying to replicate a crash for a target frame requires to set the input arguments of the target method and all the other calls in the sequence properly such that when calling the failing method, the crash happens. Since the input space of a method is usually large, this is a very challenging task. Search-based crash reproduction uses randomly generated *input arguments* and mock objects as inputs for the target method. As we described in Section 3.2.2, we observe that a very common problem when reproducing a *ClassCastException* (CCE) is to identify which types to use as input parameters such that CCE is thrown.

We also noticed that some stack traces involving Java *generic types* make search-based crash reproduction crash in its attempts to inject the failing method in every generated test during the guided initialisation phase. Generating *generic type* parameters is a recognized challenge for automated testing tools for Java [19]. To handle these parameters, search-based crash reproduction, based on EvoSuite's implementation [19], collects candidate types that are randomly assigned to the type parameter, based on `castclass` and `instanceof` operators in Java bytecode. Since the candidate types may themselves have generic type parameters, a threshold is used to avoid large recursive calls to generic types. One possible explanation for the crashes in these cases could be that the threshold is not properly tuned for the kind of classes involved in the recruited projects. Thus, the tool fails to set up the target method to inject into the tests. Based on the results of our evaluation, handling Java generics in search-based crash reproduction needs further investigation to identify the root cause(s) of the crashes and devise efficient strategies to address them.

Environmental dependencies

Search-based crash reproduction extends EvoSuite [18], which is a search-based *unit* test generation tool. As Arcuri et al. [9] discuss, generating unit tests for classes which interact with the environment leads to

1. difficulty in covering certain branches which depend on the state of the environment, and
2. generating flaky tests, which may sometimes pass, and sometimes fail, depending on the state of the environment.

[2] Available at https://github.com/STAMP-project/EvoCrash-JCrashPack-application/tree/master/results/manual-analysis.

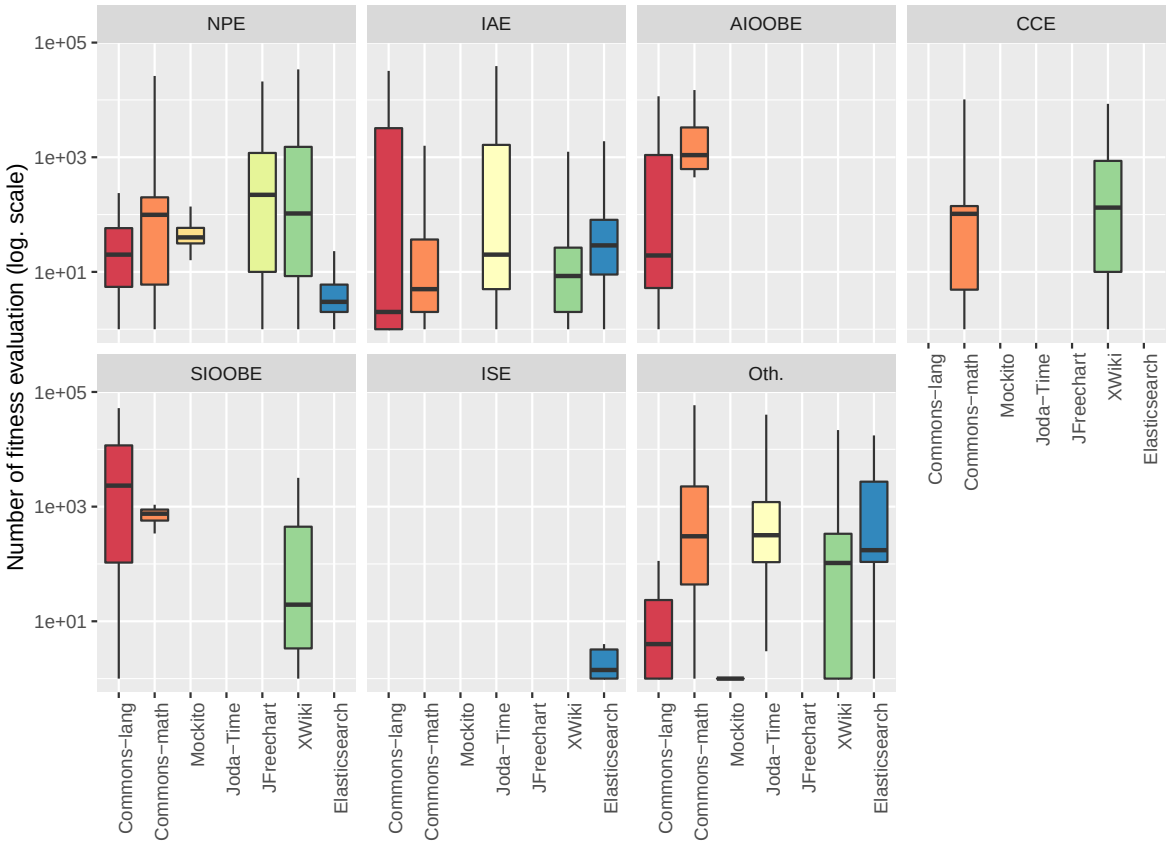

Figure 3.2: Average number of fitness evaluations for the *reproduced* frames for each applications and exception type.

Despite the numerous advances made by the search-based testing community in handling environmental dependencies [9, 20], we noticed that having such dependencies in the target class hampers the search process. Listing 3.4 shows an example of a private method in which the content of a received `HTTP` request is checked. If the input is not well-formatted, a `ServletException` is thrown, which indicates that the request is malformed. To trigger a `NullPointerException` at line 6, search-based crash reproduction needs to make a call to the method which invokes `getRequestURL`. However, reaching line 6 to trigger the exception is challenging, since every time line 4 is reached, due to malformed $HTTP$ requests that are generated by the test, a `MalformedURLException` is thrown.

Frames with `try/catch`

During our manual inspection of the frames that could not be reproduced, we noticed that some frames have a line number that designates a call inside a `try/catch` block. When the exception is caught, it is no longer thrown at the specific line given in the trace, rather it is typically handled inside the associated `catch` blocks. From what we observed, often catch blocks either

1. re-throw a checked exception, or
2. log the caught exception.

Listing 3.4: An example of having environmental dependency (the content of an input HTTP request)

```
private URL getRequestURL(HttpServletRequest request) throws ServletException{
  URL url;
  try {
      StringBuffer requestURL = request.getRequestURL();
      String qs = request.getQueryString();
      if (!StringUtils.isEmpty(qs)) {
          url = new URL(requestURL.toString() +'?' + qs);
      } else {
          url = new URL(requestURL.toString());
      }
  } catch (MalformedURLException e) {
      throw new ServletException(
          String.format("Failed to reconstruct URL from HTTP Servlet Request");
           ... }
  ...}
```

If the caught exception results in re-throwing another exception, chained exceptions are generated, which yield chained stack traces with information that is not exactly as the input stack trace. On the other hand, search-based crash reproduction only considers uncaught exceptions that are generated as the result of running the generated test cases during the search, thus the logged stack traces would be of no use for crash reproduction.

Generally, if an exception is caught in one frame, it cannot be reproduced (as it cannot be observed) from higher level frames. This property of a crash stack trace implies that depending on where in the trace such frames exist, only a fraction of the input stack traces could actually be used for automated crash reproduction.

Complex code

The first component of the fitness function that is used in search-based crash reproduction encodes how close the algorithm is to actually reaching the line where the exception is thrown. Therefore, frames of a given stack trace that have high code complexity are difficult and therefore more costly to reproduce. Complex code in our evaluation includes:

1. failing methods which often have over 100 lines of code[3], and high cyclomatic complexity,
2. methods with nested predicates [27, 21], and
3. methods with the *flag problem* [27, 21], which include (at least one) branch predicate with a binary (boolean) value.

Abstract classes and methods

In several cases from Elasticsearch, the majority of the frames from a given stack trace point to an abstract class, which may also include at least one abstract method. In Java, abstract classes cannot be instantiated, rather they are supposed to be extended. When a target class is abstract, for test generation, (similar to EvoSuite) search-based crash reproduction looks for classes on the classpath that extend the abstract class. Therefore, even if search-based crash reproduction finds a test case that satisfies the first two conditions of the fitness function, the generated stack trace would not include matching frames, as the class names and line numbers would differ. Therefore, the fitness function

[3] In some cases for Elasticsearch, the failing methods actually have near 300 lines of source code

Listing 3.5: An example for a crash stack trace with missing line numbers

```
at org.apache.xerces.parsers.XMLParser.parse(Unknown Source)
at org.apache.xerces.parsers.AbstractSAXParser.parse(Unknown Source)
at org.xml.sax.helpers.XMLFilterImpl.parse(XMLFilterImpl.java:357)
```

would yield a value between 0 and 1, but it may never be equal to 0. Moreover, we observed several cases where in a given trace, abstract classes were chained, such that all those frames that were from packages of Elasticsearch in the trace, pointed to abstract classes. Thus, the ones which remained pointed to concrete implementation of classes which came from external packages such as `Apache Lucene`[4].

Static initialisation

It may be that frames of a given stack trace point to classes that have static initializers. In Java, static initializers are invoked only once when the class containing them is loaded. As explained by Fraser and Arcuri [20], these blocks may have dependencies to static fields from other classes on the class-path that have not been initialized yet, and thus cause exceptions such as `NullPointerException` to be thrown. In addition, they may involve environmental dependencies that are restricted by the security manager, which may also lead to unchecked exceptions being generated. Therefore, when such frames are used for crash reproduction with search-based crash reproduction, the tool simply crashes. As Fraser and Arcuri [20] discuss, automatically determining and solving all possible kinds of dependencies in static initializers is a non-trivial task that warrants dedicated research.

Anonymous classes

As we observed, when the target frame from a given crash stack trace points to an anonymous object or a lambda expression, guided initialization in search-based crash reproduction fails. One possible explanation for this is that the target method that needs to be set up to be injected to the tests, is not directly accessible in these cases[18].

Private inner classes

In some cases, the target frame from a given trace points to a failing method inside a private inner class. Since it is not possible to access this class, and therefore, not possible to directly instantiate it, it is difficult for search-based crash reproduction to create an object of this class. Therefore, it is not possible to inject the failing method from this class during the guided initialization phase in search-based crash reproduction.

Unknown source

Occasionally, stack traces have frames with a missing line number, as shown in Listing 3.5. Since search-based crash reproduction requires a line number to compute the fitness values during the search, those frames have been discarded from our evaluation. Yet, a total of 31 frames with an unknown line number are in JCrashPack, so we decided to mention this challenge here.

Interfaces

In some cases, target frames point to an interface. In Java, similar to abstract classes, interfaces may not be instantiated. They serve to specify behaviour of a type which can be implemented by any

[4] https://lucene.apache.org/core/

other classes. In these cases also, search-based crash reproduction randomly selects the classes on the classpath that implement the interface. However, similar to the case with abstract classes, using the classes that implement the target interface, the fitness function may never evaluate to 0.0 because the generated stack trace would not point to the same class, and line number at that particular level of the input stack trace.

Irrelevant frames

In some cases, the target frame points to a line in the source code where the target class or target method is defined. We observed that this happens when the previous frame points to an anonymous class or a lambda expression. Such frames practically cannot be used for crash reproduction as the location they point to is irrelevant to where exactly the target exception occurs.

Nested private calls

In multiple cases, the target frame points to a private method. As we mentioned in Section 3.1, those private methods are not directly accessible by search-based crash reproduction. To reach them, search-based crash reproduction detects other public or protected methods which invoke the target method directly or indirectly and randomly choose during the search. If the private method is called by a large number of public methods, or if the chain of method calls is too long, search-based crash reproduction may fail to pick the right method during the search.

Unknown

Despite our investigation efforts, we were unable to identify why search-based crash reproduction failed to reproduce 21 frames.

3.3 Discussion

3.3.1 Extending JCrashPack

To the best of our knowledge, JCrashPack is the first benchmark dedicated to crash reproduction. We deliberately made a biased selection when choosing Elasticsearch as the most popular, trending, and frequently-forked project from Github. While such selection methodology has the advantage of making an informed decision, it is at the same time biased. In addition, Elasticsearch was among several other highly ranked projects, which addressed other application domains, and thus were interesting to explore. To address such limitations in a benchmark, and thereby improve the validity of studies using it, it is important that the benchmark is easy to extend. In the future, further effort should be spent in such a way that JCrashPack is extended, possibly by: (i) using a *random selection* methodology for choosing projects, and (ii) involving industrial projects from other application domains. Furthermore, building JCrashPack required substantial manual effort. Since we want it to be representative of current crashes, we need to automate this effort as much as possible: for instance, by mining stack traces from issue tracking systems [29].

3.3.2 Experimental setup

Optional dependencies

Running search-based crash reproduction on a project requires to have all its dependencies available on the classpath. During the test of our evaluation protocol, we faced some issues with optional dependencies. Maven and Gradle building systems allow to declare optional dependencies (*provided* in Maven and *compileOnly* in Gradle) in a project [38, 6]. Those dependencies are used only during

compilation (like annotation processors) and, if needed, provided by the environment at runtime (by a Java EE application server). As they are usually not packaged by default with the binaries, one has to be very cautious during collection of the binaries to perform crash replication.

Line mismatch in some stack traces

During the manual analysis of our results, we found out that three frames in two different stack traces, coming from Defects4J projects, target the wrong lines in the source code: the line numbers in the stack traces point to lines in the source code that cannot throw the targeted exception. Since the stack traces were collected directly from the Defects4J data (which reports failing tests and their outputs), we tried to regenerate them using the provided test suite and found indeed a mismatch between the line numbers of the stack traces. We reported those two projects to the Defects4J developers: this mismatch is caused by a bug in JDK7 [23]. Since search-based crash reproduction relies on line numbers to guide its search, it could not reproduce the crashes. We recompiled the source code, updated the stack trace accordingly in JCrashPack, and rerun the evaluation for those two stack traces.

3.4 Replication of the results

All the results may be replicated using the **EvoCrash-JCrashPack-application** artifact available online at `https://github.com/STAMP-project/EvoCrash-JCrashPack-application/releases/tag/1.0.0`. The artifact includes a Docker file and a a step-by-step explanation on how to run the evaluation and the analysis of the results.

3.4.1 Reproducing the evaluation

To run the evaluation, run `Init.py` from the `evaluation/pythonScripts` directory:

```
python Evaluation/pythonScripts/Init.py <Number of Threads>
```

Attention: to reproduce evaluation in a reasonable amount of time, one need a powerful machine with a large amount of memory to be able to increase the `<Number of Threads>`. In our evaluation, we distributed the execution over several machines.

Results of the execution are saved in `evaluation/pythonScripts/outputs/csv/result.csv`. In addition, the generated test cases (if any) are saved in `evaluation/JarFiles/G-GA-tests`.

3.4.2 Reproduce analysis

The `results` folder contains different sub-folders with the complete results of our evaluation. A `Makefile` is available in `results/` to launch the R analysis (requires R to be installed and `Rscripts` command to be available). The different goals (to be launched from the `results/` folder) are:

- `make analysebenchmark` prints various statistics on the JCrashPack benchmark;
- `make analyseresults` prints various statistics on the results of our evaluation;
- `make plots` produces the `.pdf` plots describing the benchmark and the evaluation results in `plots/`;
- `make tables` produces the `.tex` tables describing the benchmark and the evaluation results in `tables/`;
- `make all` all of the above goals;

- `make clean` deletes `plots/` and `tables/`.

Other files in `results/` include:

- `csv/` with CSV files containing data on the benchmark and the results of our evaluation;
- `logs.zip` with the logs produced by search-based crash reproduction for each round of each case;
- `manual-analysis/` the manual analysis and classification performed on the unreproducible cases;
- `tests.zip` the tests generated by search-based crash reproduction (if any) for the different cases.

Chapter 4

Stack trace preprocessing

The selection of a relevant frame from a stack trace for crash reproduction is one of the challenges identified in Section 3.2.3. For instance, frames targeting code in a private inner class, or irrelevant source code location (like class header or annotation) should be discarded before performing the selection. Similarly, frames targeting code in abstract classes or interfaces may contain the bug, but makes it very hard for search-based crash reproduction as abstract classes and interfaces may not be directly instantiated. In this case, considering higher level frames (i.e., frames that are lower in the stack trace) may help to pick the right subclass.

Those reasons motivate the need to develop stack trace analysis techniques in order to help the selection of a target frame. This analysis will discard irrelevant and unknown source location frames, and identify frames pointing to: a `try/catch` block; to an interface of abstract class of methods; or to a private class of method.

In Sections 4.1 to 4.5 we will discuss some of the stack preprocessing techniques that we have devised.

4.1 Nested exceptions

Nested exceptions occur when an exception is thrown at some point in the code and caught in a higher frame to be encapsulated and re-thrown. For instance, the issue `XWIKI-12482`[1] contains the following nested exception:

```
com.xpn.xwiki.XWikiException: [...]
  at com.xpn.xwiki.XWiki.evaluateTemplate(XWiki.java:1828)
  at com.xpn.xwiki.XWiki.parseTemplate(XWiki.java:1801)
  at com.xpn.xwiki.api.XWiki.parseTemplate(XWiki.java:820)
    [...]
  ... 75 more
Caused by: org.apache.velocity.exception.MethodInvocationException: [...]
  at org.apache.velocity.runtime.parser.node.ASTMethod.handleInvocationException(
      ASTMethod.java:243)
  at org.apache.velocity.runtime.parser.node.ASTMethod.execute(ASTMethod.java:187)
  at org.apache.velocity.runtime.parser.node.ASTReference.execute(ASTReference.
      java:280)
  [...]
  ... 83 more
```

[1] See https://jira.xwiki.org/browse/XWIKI-12482.

```
13 Caused by: java.lang.NullPointerException
     at com.xpn.xwiki.internal.store.hibernate.query.HqlQueryUtils.
         isSelectExpressionAllowed(HqlQueryUtils.java:223)
15   at com.xpn.xwiki.internal.store.hibernate.query.HqlQueryUtils.
         isSelectItemAllowed(HqlQueryUtils.java:204)
     at com.xpn.xwiki.internal.store.hibernate.query.HqlQueryUtils.isSafe(
         HqlQueryUtils.java:153)
17   [...]
     ... 102 more
```

In this case, a `NullPointerException` (line 13) has been thrown in the `isSelectExpressionAllowed` method and propagated to be caught, encapsulated in a `MethodInvocationException` (line 7) and re-thrown in method `handleInvocationException`. This exception is in turn propagated to be catch, encapsulate in a `XWikiException` (line 1) and thrown by method `evaluateTemplate`. The effective exception in this case is the `NullPointerException`, and the stack trace for search-based crash reproduction (and the one that has been included in JCrashPack, see D3.2) is the nested one (starting at line 13).

Nested exceptions are easy to identify, thanks to the `Caused by` keywords appearing in the stack trace (at lines 1 and 7). The preprocessing of the stack trace can thus easily remove encapsulating exceptions to focus on the deepest nested exceptions thrown by the application.

4.2 Frames with `try/catch`

During our manual inspection in D3.2, we noticed that some frames have a line number that designates a call inside a `try/catch` block. When the exception is caught, it is no longer thrown at the specific line given in the trace, rather it is typically handled inside the associated catch blocks. From what we observed, often catch blocks either:

- re-throw a checked exception, or
- log the caught exception.

If the caught exception results in re-throwing another exception, chained exceptions are generated. In the latter case, current search-based crash reproduction only considers uncaught exceptions that are generated as the result of running the generated test cases during the search, thus the logged stack traces would be of no use for crash reproduction.

Generally, if an exception is caught in one frame, it cannot be reproduced (as it cannot be observed) from higher level frames. This property of a crash stack trace implies that depending on where in the trace such frames exist, only a fraction of the input stack traces could actually be used for automated crash reproduction. For instance, considering the following stack trace from the XWiki issue XWIKI-13345:[2]

```
0 java.lang.NullPointerException: null
    at com.xpn.xwiki.XWiki.getSkinPreference(XWiki.java:2101)
2   at com.xpn.xwiki.api.XWiki.getSkinPreference(XWiki.java:988)
    at org.apache.velocity.util.introspection.UberspectImpl$VelMethodImpl.doInvoke(
        UberspectImpl.java:395)
4   at org.apache.velocity.util.introspection.UberspectImpl$VelMethodImpl.invoke(
        UberspectImpl.java:384)
    at org.apache.velocity.runtime.parser.node.ASTMethod.execute(ASTMethod.java:173)
```

[2] See https://jira.xwiki.org/browse/XWIKI-13345.

The method `getSkinPreference` in the first frame throws a `NullPointerException` but also catches it, making it impossible for current implementation of search-based crash reproduction to observe that exception. In general, is an exception is caught by a `try/catch block` at frame n, this exception cannot be observed for all the frames $\geq n$.

The filtering process may spot such cases using static analysis and indicate to the developer which are the frames pointing 'to a `try/catch block`.

4.3 Interfaces, abstract classes and methods

Frames may also point to locations in abstract classes or methods, or even interfaces since Java 8 and higher allow code blocks in interfaces. In Java, abstract classes cannot be instantiated, rather they are supposed to be extended. When a target class is abstract search-based crash reproduction looks for classes on the classpath that extend the abstract class and instantiate them to proceed to the search. For instance, considering the stack trace from the XWiki issue `XWIKI-13544`:[3]

```
org.xwiki.rendering.macro.MacroExecutionException:
  at org.xwiki.rendering.macro.script.AbstractScriptMacro.execute(
      AbstractScriptMacro.java:178)
  at org.xwiki.rendering.macro.script.AbstractScriptMacro.execute(
      AbstractScriptMacro.java:58)
  at org.xwiki.rendering.internal.transformation.macro.MacroTransformation.
      transform(MacroTransformation.java:269)
    [...]
```

The two first frames point to an abstract class `AbstractScriptMacro` partially implemented. If one of those two frames is targeted for search-based crash reproduction, the evolutionary process will randomly pick a subclass implementing `AbstractScriptMacro`. This could lead to local optima (if the line could be reached in the abstract class for instance) or require a large search budget (if there is a large number of classes implementing `AbstractScriptMacro`).

Stack trace preprocessing can pinpoint frames pointing to interfaces, abstract classes and methods and recommend to the developer to select higher level frames pointing to concrete implementations (`MacroTransformation` in our example).

4.4 Private classes and methods

Similarly to interfaces and abstract classes and methods, frames pointing to private inner classes or private methods do not allow to directly call the target method.[4] Instead, the search-based crash reproduction process will try to find an indirect way to call the method (by scanning accessible methods calling the private class or method). Again, this could lead to local optima or require a large search budget (if there is a large number of public methods calling the private method).

Stack trace preprocessing can also pinpoint such frames and methods and recommend to the developer to select higher level frames pointing to accessible methods.

[3] https://jira.xwiki.org/browse/XWIKI-13544.

[4] See for instance `XWIKI-14302` in JCrashPack and the evaluation done in D3.2 at https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Xwiki/XWIKI-14302.md.

4.5 Irrelevant frames

In some cases, we observed in D3.2 that frames in a stack trace may point to an irrelevant line in the source code.[5] Those irrelevant line numbers include annotations, methods declarations, etc. Or sometimes, it denotes a mismatch between the version of the code and the version of the application where the stack trace has been generated.

In any case, stack trace preprocessing can report those stack traces to the developer for further investigation.

4.6 Summary

The purpose of stack trace preprocessing is twofold. First, is cleans the stack trace before its processing by a search-based crash reproduction tool. This includes removing the encapsulating exceptions to keep only the nested one and removing irrelevant frames from the stack trace so it is not taken into account by the search-based crash reproduction process. Second, it identifies frames pointing to `try`/`catch` blocks, interfaces, abstract or private classes and methods, and reports such frames to the developer to help him in his choice of a target frame.

[5] See for instance `XWIKI-12798` in JCrashPack and the evaluation done in D3.2 at https://github.com/STAMP-project/EvoCrash-JCrashPack-application/blob/master/results/manual-analysis/Xwiki/XWIKI-12798.md.

Chapter 5

Future work

From the evaluation and the challenges derived from our manual analysis, we devise the following future work.

5.1 Context matters

While search-based crash-reproduction [37] outperformed other approaches based on backward symbolic execution [13], test case mutation [39], and model-checking [31], our evaluation shows that the extent to which crashes are reproduced varies. These results indicate the need for taking various types of contexts and properties of software applications into account when devising an approach to a problem. Thus, we show that indeed, rather than seeking a universal approach to search-based crash reproduction, it is important to find out and address challenges specific to various types of application domains [7]. Furthermore, search-based crash replication comes down to seeking the execution path that will reproduce a given stack trace. As with other search-based testing approaches, it faces challenges about *input data generation* during the search when the input space is large. Previous research on *mocking* and *seeding* [10, 33] address this problem by using functional mocking and extracting objects and constants from the bytecode.

We believe that *taking context into account* should go one step further for crash replication. With the development of DevOps [32] and continuous integration and delivery pipelines, there is an increasing available amount of data on the execution of the software. Those data can be used to guide the search more accurately. For instance, by seeding the search using values observed in the execution logs.

5.2 Guided search

Besides usage of contextual information to enhance the generation of test cases during the search process, we also consider to enhance the guidance itself by improving the fitness function. For instance the fitness function could use different weights in the formula or benefit from multi-objectivization [25] to avoid the algorithm to become trapped in local optima due to diversity loss [14].

We plan to reuse our evaluation infrastructure to compare those different approaches and devise guidelines on search-based crash reproduction settings to maximize crash reproduction for a given stack trace and its project.

5.3 Adding probes to logs

Finally, dynamic probes and log messages can be added to the production code to improve its testability, improve the guidance of the search, and ease the debugging process. We identify the following potential directions to decide which are the proper places and the right timing for probing the system and collect additional log data used in the subsequent tasks of WP3:

- by extending the current code instrumentation that dynamically injects probes into the bytecode of a Java application, to monitor and log the runtime behavior of specific classes, which provides additional information for the search process and improves its guidance;
- by using a Java agent [1, 2] dynamically attached to the Java Virtual Machine to collect information on the running system;
- by defining a set of rules for developers describing the optimal locations for log messages and log messages content. For instance, by using spectrum-fault localization [12] to determine the best place that maximizes fault detection capability; or
- since the insertion of log statements in the source code is a multi-objective problem where one wants to optimize testability while minimizing the execution overhead induced by logging, by using a multi-objective search-based approach to introduce log statements in the source code that would satisfy the concurrent objectives (for instance, using genetic programming and program repair [28]).

Conclusion

Experimental evaluation of crash reproduction research is challenging, due to the computational resources needed by reproduction tools, the difficulty of finding suitable real life crashes, and the intricacies of executing a complex system so that the crash can be reproduced at all.

To remedy this problem, this paper sets out to create a benchmark of Java crashes, that can be reused for experimental purposes. To that end we propose JCrashPack and ExRunner, a curated benchmark of 200 real life crashes, and a tool to conduct massive experiments on these crashes. This benchmark is publicly available and can be used to compare existing and new tools against each other, as well as to analyse how proposed improvements to existing reproduction techniques actually constitute an improvement.

We applied the state of the art search-based Java crash reproduction tool, search-based crash reproduction, to JCrashPack. Our findings include that the state of the art can reproduce 90 crashes out of 200, that crash reproduction for industry-strength systems is substantially harder, and that `NullPointerException`s are generally easiest to reproduce. Furthermore, we identified 13 challenges that crash reproduction research needs to address to strengthen uptake in practice, as well a future research directions to address those challenges.

Bibliography

[1] btrace. https://github.com/btraceio/btrace, 2019. [Online; accessed 30-April-2019].

[2] Glowroot. https://glowroot.org, 2019. [Online; accessed 30-April-2019].

[3] Apache. Ant. http://ant.apache.org/, 2017. [Online; accessed 25-January-2018].

[4] Apache. Commons Collections. https://commons.apache.org/proper/commons-collections/, 2017. [Online; accessed 25-January-2018].

[5] Apache. Log4j. https://logging.apache.org/log4j/2.x/, 2017. [Online; accessed 25-January-2018].

[6] Apache Maven Project. Introduction to the Dependency Mechanism. https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html, 2018. [Online; accessed 25-January-2018].

[7] A. Arcuri. RESTful API Automated Test Case Generation. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 9–20. IEEE, jul 2017.

[8] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[9] A. Arcuri, G. Fraser, and J. P. Galeotti. Automated unit test generation for classes with environment dependencies. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering - ASE '14*, pages 79–90, Vasteras, Sweden, 2014. ACM Press.

[10] A. Arcuri, G. Fraser, and R. Just. Private api access and functional mocking in automated unit test generation. In *Software Testing, Verification and Validation (ICST), 2017 IEEE International Conference on*, pages 126–137, Tokyo, Japan, 2017. IEEE, IEEE Computer Society.

[11] F. A. Bianchi, M. Pezzè, and V. Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 705–716. ACM Press, 2017.

[12] C. Chen, H.-G. Gross, and A. Zaidman. Analysis of service diagnosis improvement through increased monitoring granularity. *Software Quality Journal*, 25(2):437–471, jun 2017.

[13] N. Chen and S. Kim. STAR: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Transactions on Software Engineering*, 41(2):198–220, 2015.

[14] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):35, 2013.

[15] Dubbo. A high-performance, java based, open source RPC framework. http://dubbo.io, 2018. [Online; accessed 25-January-2018].

[16] Elastic. Elasticsearch: RESTful, Distributed Search and Analytics. https://www.elastic.co/products/elasticsearch, 2018. [Online; accessed 25-January-2018].

[17] G. Fraser and A. Arcuri. Evosuite: On the challenges of test case generation in the real world. In *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pages 362–369, Luxembourg, Luxembourg, 2013. IEEE, IEEE Computer Society.

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

[19] G. Fraser and A. Arcuri. Automated test generation for java generics. In *International Conference on Software Quality*, pages 185–198, Vienna, Austria, 2014. Springer, Springer.

[20] G. Fraser and A. Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(2):8, 2014.

[21] M. Harman, L. Hu, R. Hierons, J. Wegener, H. Sthamer, A. Baresel, and M. Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004.

[22] Java Design Patterns. Design patterns implemented in Java. http://java-design-patterns.com, 2018. [Online; accessed 25-January-2018].

[23] JDK. Stack trace has invalid line numbers. https://bugs.openjdk.java.net/browse/JDK-7024096, 2016. [Online; accessed 25-January-2018].

[24] 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 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 437–440, San Jose, CA, USA, 2014. ACM Press.

[25] J. D. Knowles, R. A. Watson, and D. W. Corne. Reducing local optima in single-objective problems by multi-objectivization. In *International Conference on Evolutionary Multi-Criterion Optimization*, pages 269–283. Springer, 2001.

[26] B. Liskov and J. Guttag. *Program development in JAVA: abstraction, specification, and object-oriented design*. Pearson Education, London, England, UK, 2000.

[27] J. Malburg and G. Fraser. Combining search-based and constraint-based testing. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 436–439, Lawrence, KS, USA, 2011. IEEE, IEEE Computer Society.

[28] M. Martinez and M. Monperrus. ASTOR: a program repair library for Java (demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*, pages 441–444. ACM Press, 2016.

[29] M. Nayrolles and A. Hamou-Lhadj. BUMPER: A Tool for Coping with Natural Language Searches of Millions of Bugs and Fixes. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 649–652, Suita, Osaka, Japan, mar 2016. IEEE.

[30] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, mar 2015.

[31] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, 29(3):e1789, mar 2017.

[32] J. Roche. Adopting DevOps practices in quality assurance. *Communications of the ACM*, 56(11):38–43, 2013.

[33] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, aug 2016.

[34] RxJava. Reactive Extensions for the JVM. https://github.com/ReactiveX/RxJava, 2018. [Online; accessed 25-January-2018].

[35] S. E. Sim, S. Easterbrook, and R. C. Holt. Using Benchmarking to Advance Research: A Challenge to Software Engineering. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 74–83, Portland, Oregon, USA, 2003. IEEE Computer Society.

[36] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen. Single-objective versus Multi-Objectivized Optimization for Evolutionary Crash Reproduction. In *Proceedings of the 10th Symposium on Search-Based Software Engineering (SSBSE)*. Springer, 2018.

[37] M. Soltani, A. Panichella, and A. van Deursen. A Guided Genetic Algorithm for Automated Crash Reproduction. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 209–220. IEEE, may 2017.

[38] M. Vieira. Introducing Compile-Only Dependencies. https://blog.gradle.org/introducing-compile-only-dependencies, 2016. [Online; accessed 25-January-2018].

[39] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 910–913. ACM Press, 2015.

[40] XWiki. The Advanced Open Source Enterprise and Application Wiki. http://www.xwiki.org/, 2018. [Online; accessed 25-January-2018].

[41] A. Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.