STAMP

Deliverable D3.3

Prototype of amplification tool for common and anomaly behaviors

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

Deliverable Number	:	D3.3
Title of Deliverable	:	Prototype of amplification tool for common and anomaly behaviors
Dissemination Level	:	Public
Version	:	1.01
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d33_prototype_of_amplification_tool_for_common_and_anomaly_behaviors.pdf
Contractual Delivery Date	:	M24 - November 30, 2018
Contributing WPs	:	WP 7
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 Caroline Landry, INRIA Cedric Thomas, OW2

Abstract

The current state of practice for software debugging is to manually investigate the code based on the inputs provided by the user in the issue tracker. When the application crashes, those inputs include a stack trace, that provides information on the source of the crash. Unfortunately, this information is usually insufficient or incomplete to be able to reproduce the crash in the development environment, resulting in an intensive manual effort for the developer.
The main objective of work package 3 is to amplify the existing test suite of an application by using the observed behavior during operations to create new test cases. To this end, we address the three lines of future work identified in D3.2 by improving the guidance of the search-based crash reproduction algorithm; seed the process with contextual information, including common behaviors; and filtering irrelevant information by preprocessing the stack trace. We devise Botsing, a crash reproduction framework aimed at meeting industrial standards of the STAMP partners.

Revision History

Version	Type of Change	Author(s)
1.00	initial version	Xavier Devroey, TU Delft Pouria Derakhshanfar, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
1.01	updating document information (editor, reference, etc)	Caroline Landry, INRIA

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

Common and anomaly behaviors are two essential elements of any testing activities. Among other things, common behaviors may be used during regression testing to ensure that any evolution of the software does not break previous features. Anomaly behaviors that are uncovered by the existing tests may manifest themselves in various forms. One of the most common ones is an application crash that will eventually be reported to the developers for debugging.

The starting point of any debugging activity is to try to reproduce the problem reported by a user in the development environment [52]. In particular, for Java programs, when a crash occurs, an exception is thrown. A developer strives to reproduce it to understand its cause, then fix the bug, and finally add a (non-)regression test to avoid reintroducing the bug in future versions.

Manual crash reproduction can be a challenging and labor-intensive task for developers: it is often an iterative process that requires setting the debugging environment in a similar enough state as the environment where the crash occurred [52]. Moreover, it requires the developer to have knowledge of the system components involved in the crash. To help developers in their task, several automated crash reproduction methods, relying on different techniques, have been proposed [6, 33, 34, 51, 4, 41, 39].

One of the most promising approaches uses search-based software testing [41, 39] to generate a test case able to reproduce a crash. As for other search-based approaches, search-based crash reproduction may face several challenges. We identified 13 challenges during Task 3.2 and reported them in deliverable D3.2. We also identified 3 future work directions: *taking context into account* during the search; *develop stack trace analysis techniques* to preprocess the stack traces and filter unrelated information; and *enhance the guidance* of the search process. We address those challenges in this deliverable by developing and evaluating new approaches for search-based crash reproduction. We also describe *Botsing*, a brand new search-based crash reproduction framework that meets industrial requirements, including a release under Apache-2.0 license.

The remainder of this document is structured as follows:

Chapter 1 - Background on search-based crash reproduction presents the background on search-based crash reproduction.

Chapter 2 - Fitness function refinement describes a new fitness function, a new multi-objectivized approach to search-based crash reproduction, and their evaluation and comparison to the state of the art.

Chapter 3 - Seeding search-based crash reproduction describes two new seeding approaches for search-based crash reproduction, including one based on the detection of common behaviors.

Chapter 4 - The Botsing framework presents our new search-based crash reproduction framework.

Remark: This deliverable focuses on the tools built during Task 3.3 - *Test amplification for anomalies replication*, it will report on our efforts on test amplification for anomaly behaviors. Since Task 3.4 - *Test amplification for common behaviors* starts at Month 24 (per the plan), the results and tools developed during Task 3.4 will be reported in D3.4 and D3.5.

Summary of Artifacts

1. **Botsing framework** (see Chapter 4)
    - Link: https://github.com/STAMP-project/botsing/releases/tag/1.0.1
    - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD), Luca Andreatta (Eng), Valentina Di Giacomo (Eng)

Chapter 1

Background on seach-based crash reproduction

Crash reproduction tools aim at generating a test case able to reproduce a given crash based on the information gathered during the crash itself. This *crash reproduction test case* can help developers to identify the fault causing the crash [6, 42]. For Java programs, the available information usually consists of a stack trace, i.e., lists of classes, methods and code lines involved in the crash. For instance, the following stack trace has been generated by the test cases of `LANG v9b` from the Defects4J [26] dataset:

```
0 java.lang.ArrayIndexOutOfBoundsException:
    at org.apache.commons.lang3.time.FastDateParser.toArray(FastDateParser.java:413)
2   at org.apache.commons.lang3.time.FastDateParser.getDisplayNames(FastDateParser.
        java:381)
    ...
```

It has a thrown exception (`ArrayIndexOutOfBoundsException`) and different frames (lines 1 to 3), each one pointing to a method call in the source code.

1.1 EvoCrash

Various automated approaches to crash reproduction have been proposed in the literature [4, 25, 51, 34, 41, 6, 38]. Among these, EvoCrash [41] is a search-based approach, which applies a Guided Genetic Algorithm (GGA) to generate a crash-reproducing test. The fitness function (here, a *weighted sum fitness function*) is used to characterize the "quality" of test case generated during each iteration of the guided GA and drives the overall process.

1.1.1 Weighted Sum (WS) Fitness Function

The three components of the WS fitness function are: (i) the coverage of the code line (*target statement*) where the exception is thrown, (ii) the target exception which has to be thrown, and (iii) the similarity between the generated stack trace (if any) and the original one. Formally, the fitness function for a given test t is defined as [41]:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \quad (1.1)$$

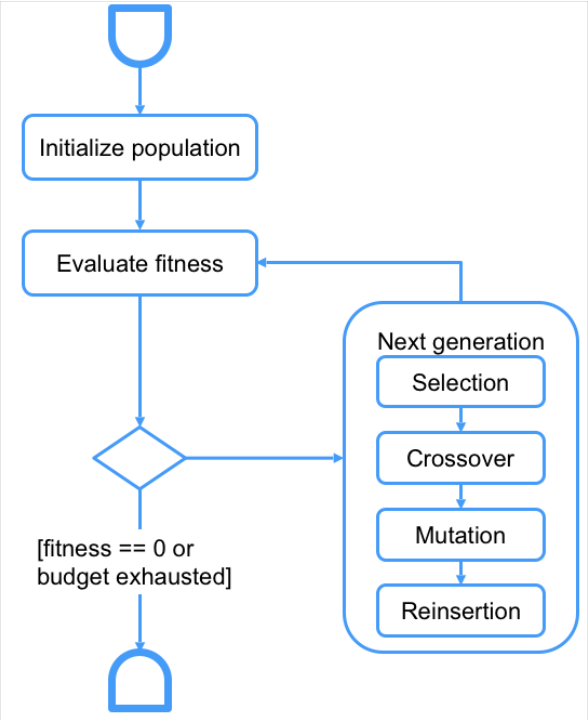

Figure 1.1: Guided Genetic Algorithm overview

where $d_s(t) \in [0, 1]$ denotes how far t is from executing the target statement using two well-known heuristics, *approach level* and *branch distance* [40]. The approach level measures the minimum number of control dependencies between the path of the code executed by t and the target statement s. The branch distance scores how close t is to satisfying the branch condition for the branch on which the target statement is directly control dependent [30]. In Equation 1.1, $d_{except}(t) \in \{0, 1\}$ is a binary value indicating whether the target exception is thrown (0) or not (1); $d_{trace}(t)$ measures the similarity of the generated stack trace with the expected one based on methods, classes, and line numbers appearing in the stack traces; $max(d_{except})$ and $max(d_{trace})$ denote the maximum possible value for d_{except} and d_{trace}, respectively. Therefore, the last two addends of the fitness function (i.e., d_{except} and d_{trace}) are computed upon the satisfaction of two *constraints*. This is because the target exception has to be thrown in the target line s (first constraint) and the stack trace similarity should be computed only if the target exception is actually thrown (second constraint).

1.1.2 Guided Genetic Algorithm (GGA)

EvoCrash (as EvoSuite) generates test cases at the unit level, meaning that test cases are generated by instrumenting and targeting one particular class (the *target class*). Contrary to classical unit test generation, EvoCrash does not seek to maximize coverage by invoking all the methods of the target class, but privileges those involved in the target failure. This is why the GGA algorithm (an overview is provided in Figure 1.1) relies on the stack trace to guide the search and reduces the search space at different steps.

1. A *target frame* is selected by the user amongst the different frames of the input stack trace. Usually, the target frame is the last one in the crash trace as it corresponds to the root method call where the exception was thrown. The class appearing in this target frame corresponds to the target class for which a test case will be generated.

2. The *initial population* of test cases is generated in such a way that the method m of the target frame (the *target method*) is called at least once in each test case [41]: either directly if m is public or protected, or indirectly by calling another method that invokes the target method if m is private.
3. During the search, dedicated *guided crossover* and *guided mutation* operators [41] ensure that newly generated test cases contain at least one call to the target method.
4. The search is guided by the WS *fitness function*.
5. Finally, the algorithm stops if the time budget is consumed or when a zero-fitness value is achieved. In this last case, the test case is minimized by a *post-processing* that removes randomly inserted method calls that do not contribute to reproducing the crash.

1.1.3 Implementation

EvoCrash is designed as an extension (i.e., a *fork*) of EvoSuite. The development of the initial prototype[1] has been abandoned. As part of WP3, we develop Botsing (detailed in Chapter 4), a search-based crash reproduction framework providing a set of tools for developers to debug runtime crashes.

1.2 Related Work

Over the years, various Java crash replication approaches that use stack traces as input have been developed. RECORE [38] is a search-based approach that in addition to crash stack traces, uses core dumps as input data for automated test generation. MUCRASH [51] applies mutation operators on existing test cases, for classes that are present in a reported stack trace, to trigger the reported crash. While BUGREDUX [25] is based on forward symbolic execution, STAR [6] is a more recent approach that applies optimized backward symbolic execution on the method calls recorded in a stack trace in order to compute the input parameters that trigger the target crash. JCHARMING [34] is also based on using crash stack traces as the only source of information about a reported crash. JCHARMING [34] applies directed model checking to identify the pre-conditions and input parameters that cause the target crash. Finally, CONCRASH [4] is a recent approach that focuses on reproducing concurrency crashes, in particular. CONCRASH applies pruning strategies to iteratively look for test code that triggers the target crash in a thread interleaving.

More recently, Soltani et al. have proposed EVOCRASH [41, 42], an evolutionary search-based tool for crash replication built on top of EVOSUITE [16]. EvoCrash uses a novel Guided Genetic Algorithm (GGA), which focuses the search on the method calls that appear in the crash stack trace rather than maximizing coverage as in classical coverage-oriented GAs. Their empirical evaluation demonstrated that EvoCrash outperforms other existing crash reproduction approaches.

[1] Legacy implementation is available at https://github.com/STAMP-project/EvoCrash.

Chapter 2

Fitness function refinement

As any search-based technique, the success of crash reproduction depends on the capability of maintaining a good balance between *exploitation* and *exploration* [9]. The former refers to the ability to visit regions of the search space within the neighborhood of the current solutions (i.e., refining previously generated tests); the latter refers to the ability to generate completely different new test cases. In crash reproduction, the exploitation is guaranteed by the guided genetic operators that focus the search on methods appearing in the crash stack trace [41]. However, such a depth and focused search may lead to a low exploration power. Poor exploration results in low diversity between the generated test cases and, consequently, the search process easily gets trapped in local optima [9].

In this chapter, we investigate two strategies to increase the diversity of generated test cases for crash reproduction. While EvoCrash uses one single-objective fitness function to guide the search, prior studies in evolutionary computation showed that relaxing the constraints [7] or multi-objectivizing the fitness function [27] help promoting diversity. Multi-objectivization is the process of (temporarily) decomposing a single-objective fitness function into multiple sub-objectives to optimize simultaneously with multi-objective evolutionary algorithms. At the end of the search, the global optimal solution of the single-objective problem is one of the points of the Pareto front generated by the multi-objective algorithms. The decomposed objectives should be as independent of each other as possible to avoid getting trapped in local optima [27].

Therefore, we study whether transforming the original weighted scalarized function in EvoCrash into (i) a simple scalarized function via constraint relaxation, and (ii) multiple decomposed objectives, impacts the crash reproduction rate, and test generation time. Our results show that indeed, when crashes are complex and require several generations of test cases, using multi-objectivization reduces the test generation time compared to the weighted scalarized function, and in turn, the weighted scalarized function reduces test generation time compared to the simple scalarized function. Furthermore, we observe that one crash can be fully replicated only by multi-objectivized search and not by the two single-objective strategies. Generally, our results show that problems that are single-objective by nature can benefit from multi-objectivization. We believe that our findings will foster the usage of multi-objectivization in search-based software engineering.

This chapter has been adapted and published at the *10th Symposium on Search-Based Software Engineering* (SSBSE '18), see [39].

2.1 Constraints Relaxation

As explained in Chapter 1, the crash replication problem has been implicitly formulated in previous studies as a constraint problem. The constraints are handled using *penalties* [41], i.e., the fitness score of a test case is penalized by adding (or subtracting in case of a maximization problem) a certain scalar value proportional to the number of constraints being violated.

For example, in Equation 1.1 all test cases that do not cover the target code line are penalized by the two addends $2 \times max(d_{except})$ and $max(d_{trace})$ as there are two violated constraints (i.e., the line to cover and the exception to throw in that line). Instead, tests that cover the target line but that do not trigger the target exception are penalized by the factor $max(d_{trace})$ (only one constraint is violated in this case).

While adding penalties is a well-known strategy to handle constraints in evolutionary algorithms [7], it may lead to diversity loss because any test not satisfying the constraints has very low probability to survive across the generations. For example, let us assume for example that we have two test cases t_1 and t_2 for the example crash reported in Section 1. Now, let us assume that both test cases have a distance $d_s = 1.0$ (i.e., none of the two could cover the target line), but the former test could generate an exception while the latter does not. Using Equation 1.1, the fitness value for both t_1 and t_2 is $f(t_1) = f(t_2) = 3 \times d_s + 3.0 = 6.0$. However, t_2 should be promoted if it can generate the same target exception of the target crash (although on a different line) and the generated trace is somehow similar to the original one (e.g., some methods are shared).

Therefore, a first alternative to the fitness function in Equation 1.1 consists of relaxing the constraints, i.e., removing the penalties. This can be easily implemented with a *Simple Sum Scalarization* (SSS):

$$f(t) = d_s(t) + d_{except}(t) + d_{trace}(t) \tag{2.1}$$

where $d_s(t)$, $d_{except}(t) \in \{0, 1\}$, and $d_{trace}(t)$ are the same as in Equation 1.1. This relaxed variant —hereafter referred to as *simple sum scalarization*— helps increase test case diversity because test cases that lead to better $d_{except}(t)$ or $d_{trace}(t)$ may survive across the GGA generation independently from the value of $d_s(t)$, which was not the case for the weighted sum, thanks to the constraints from Equation 1.1. This reformulation may increase the number of local optima; therefore, an empirical evaluation of weighted and simple sum variants to the fitness function is needed.

2.2 Multi-objectivization

Knowles et al. [27] suggested to replace the original single-objective fitness function of a problem with a set of new objectives in an attempt to promote diversity. This process, called *multi-objectivization* (MO), can be performed in two ways [27, 23]:

- by decomposing the single-objective function into multiple sub-objectives, or
- by adding new objectives in addition to the original function.

The multi-objectivized problem can then be solved using a multi-objective evolutionary algorithm, such as NSGA-II [10]. By definition, multi-objectivization preserves the global optimal solution of the single-objective problem that, after problem transformation, becomes a Pareto efficient solution, i.e., one point of the Pareto front generated by multi-objective algorithms.

In our context, applying multi-objectivization is straightforward as the fitness function in Equation 1.1 is defined as the weighted sum of three components. Therefore, our multi-objectivized version of the crash replication problem consists of optimizing the following three objectives:

$$\begin{cases} f_1(t) = d_s(t) \\ f_2(t) = d_{except}(t) \\ f_3(t) = d_{trace}(t) \end{cases} \tag{2.2}$$

Test cases in this three-objectivized formulation are therefore compared (and selected) according to the concept of *dominance* and *Pareto optimality*. A test case t_1 is said to *dominate* another test t_2 ($t_1 \prec_p t_2$ in math notation), iff $f_i(t_1) \leq f_i(t_2)$ for all $i \in \{1, 2, 3\}$ and $f_j(t_1) < f_j(t_2)$ for at least one objective f_j. A test case t is said *Pareto optimal* if there does not exist any another test case t_3

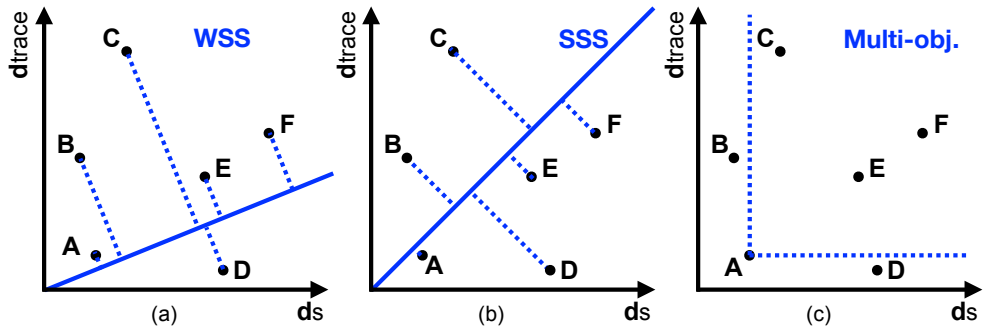

Figure 2.1: A Graphical Interpretation of Different Fitness Functions

such that $t_3 \prec_p t_1$. For instance, for the test cases (i.e., solutions) generated by a multi-objectivized (*Multi-obj.*) search presented in Figure 2.1, A, B, and D dominate C, E, and F.

In our problem, there can be multiple non-dominated solutions within the population generated by GGA at a given generation. These non-dominated solutions represent the best trade-offs among the search objectives that have been discovered/generated during the search so far. Diversity is therefore promoted by considering all non-dominated test cases (trade-offs) as equally good according to the *dominance* relation and that are assigned the same probability to survive in the next generations.

It is worth noting that a test case t that replicates the target crash will achieve the score $f_1(t) = f_2(t) = f_3(t) = 0$, which is the optimal value for all objectives. In terms of optimality, t is the global optimum for the original single-objective problem but it is also the single Pareto optimal solution because it dominates all other test cases in the search space. This is exactly the main difference between classical multi-objective search and multi-objectivization: in multi-objective search we are interested in generating a well-distributed set of Pareto optimal solutions (or optimal trade-offs); in multi-objectivization, some trade-offs are generated during the search (and preserved to help diversity), but there is only one optimal test case, i.e., the one reproducing the target crash.[1]

2.2.1 Non-dominated Sorting Genetic Algorithm II

To solve our multi-objectivized problem, we use NSGA-II [10], which is a well-known multi-objective genetic algorithm (GA) that provides well-distributed Pareto fronts and good performance when dealing with up to three objectives [10]. As any genetic algorithm, NSGA-II evolves an initial population of test cases using crossover and mutation; however, differently from other GAs, the selection is performed using tournament selection and based on the *dominance* relation and the *crowding distance*. The former plays a role during the *non-dominated sorting* procedure, where solutions are ranked in non-dominance fronts according to their dominance relation; non-dominated solutions have the highest probability to survive and to be selected for reproduction. The crowding distance is further used to promote the more diverse test cases within the same non-dominance front.

In this chapter, we implemented a *guided* variant of NSGA-II, where its genetic operators are replaced with the *guided crossover* and *guided mutation* implemented in GGA. We used these operators (i) to focus the search on the method call appearing in the target trace and (ii) to guarantee a fair comparison with GGA by adopting the same operators.

Table 2.1: Crashes used in the study

Exception Type	Defects4J	XWiki
`NullPointerException` (NPE)	9	9
`ArrayIndexOutOfBoundsExceptions` (AIOOBE)	7	0
`ClassCastException` (CCE)	2	3

2.3 Graphical Interpretation

Figure 2.1 shows commonalities and differences among the tree alternative formulations of the crash reproduction problem. For simplicity, let us focus on only two objectives (d_s and d_{trace}) and let us assume that we have a set of generated tests which are shown as points in the bi-dimensional space delimited by the two objectives.

As shown in Figure 2.1(c), points (test cases) in multi-objectivization are compared in terms of non-dominance. In the example, the tests A, B, and D are non-dominated tests and all of them are assigned to the first non-dominance front in NSGA-II, i.e., they have the same probability of being selected. Sum scalarization (either simple or weighted) projects all point to one single vector, i.e., the blue lines in Figures 2.1(a) and 2.1(b). With weighted sum scalarization (WSS), the vector of the aggregated fitness function is inclined to the d_s axis due to the higher weight of the line coverage penalty. In contrast, the vector obtained with simple sum scalarization (SSS) is the bisector of the first quadrant, i.e., both objectives share the same weights. While in both Figure 2.1(a) and 2.1(b), the best solution (point A) is the one closer to the origin of the axes, the order of the solutions (and their selection probability) can vary. For instance, we can see in the Figure that case C is a better choice than case D in the weighted sum because it has a lower value for d_s. But, case D is better than C in the simple sum. These differences in the selection procedure may lead the search toward exploring/exploiting different regions of the search space.

2.4 Evaluation

We conducted an empirical evaluation to assess the impact of the single objective or multi-objectivization fitness functions, answering the following research questions:

RQ$_1$ *How does crash reproduction with simple sum scalarization compare to crash reproduction using weighted sum scalarization?*

RQ$_2$ *How does crash reproduction with a multi-objectivized optimization function compare to crash reproduction using weighted sum scalarization?*

Comparisons for **RQ$_1$** and **RQ$_2$** are done by considering the number of crashes reproduced (*crash coverage rate*) and the time taken to generate a crash reproducing test case (*test generation time*).

2.4.1 Setup

To perform our evaluation, we randomly selected 33 crashes from JCrashPack (see D3.2), presented in Table 2.1: 18 crashes from four projects contained in Defects4J [26], which is a well-known collection of bugs from popular libraries; and 12 crashes from XWiki,[2] a web application project developed by our industrial partner.

[1]Note that there might exist multiple tests that can replicate the target crash; however, these tests are *coincident points* as they will all have a zero-value for all objectives.

[2]http://www.xwiki.org/

We executed the three approaches (weighted sum, simple sum, and multi-objectivization) on 23 virtual machines. Each machine has 8 CPU-cores, 32 GB of memory, and a 1TB shared hard drive. All of them run CentOs Linux release 7.4.1708 as operating system, with OpenJDK version 1.8.0-151.

For each crash c, we run each approach in order to generate a test case that reproduces c and targeting each frame one by one, starting from the highest one (the last one in the stack frame). As soon as one of the approaches is able to generate a test case for the given frame (k), we stop the execution and do not try to generate test cases for the lower frames ($< k$). To address the random nature of the evaluated search approaches, we execute each approach 15 times on each frame for a total number of 12,022 executions independent runs.

Parameter settings

We use the default parameter configurations from EvoSuite with functional mocking to minimize the risk of environmental interactions and increase the coverage [2]. We set the search budget to 10 minutes, which is double of the maximal amount reported by Soltani et al. [41].

2.4.2 Analysis

Since the crash coverage data is a binary distribution (i.e., a crash is reproduced or not), we use the Odds Ratio (OR) to measure the impact of the single or multi-objectivization on the *crash coverage* rate. A value of OR > 1 for comparing a pair of factors (A, B) indicates that the coverage rate increases when factor A is applied, while a value of OR < 1 indicates the opposite. A value of OR = 1 indicates that there is no difference between A and B. In addition, we use Fisher's exact test, with α=0.05 for Type I errors to assess the significance of the results. A p-value < 0.05 indicates the observed impact on the coverage rate is statistically significant, while a value of p-value > 0.05 indicates the opposite.

Furthermore, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [49] to assess the effect size of the differences between the two sum scalarization approaches or between weighted sum and multi-objectivization for *test generation time*. A value of $\hat{A}_{12} < 0.5$ for a pair of factors (A, B) indicates that A reduces the test generation time, while a value of $\hat{A}_{12} > 0.5$ indicates that B reduces the generation time. If $\hat{A}_{12}$ = 0.5, there is no difference between A and B on generation time.

To check whether the observed impacts are statistically significant, we used the non-parametric Wilcoxon Rank Sum test, with α=0.05 for Type I error. P-values smaller than 0.05 indicate that the observed difference in the test generation time is statistically significant.

2.4.3 Results

Results (RQ1)

Table 2.2 presents the crash reproduction results for the 33 crashes used in the experiment. As the table shows, 21 cases were reproduced using the original weighted sum scalarized function, while 20 cases were reproduced using simple sum scalarization. Thus, `MATH-32b` is only reproduced by the weighted sum approach. Both optimization approaches reproduced the crashes at the same frame level.

As Table 2.3 shows, we do not observe any statistically significant impact on the crash reproduction rate, comparing weighted and simple sum scalarization. However, for one case, `XWIKI-13031`, the odds ratio measure is 6.5, which indicates that the rate of crash reproduction using the weighted scalarized function is 6.5 times larger than the reproduction rate of using the simple scalarized function. In this case, the p value is 0.1, therefore we cannot draw a statistically significant conclusion.

For four cases, we see a significant impact on the test generation time. Based on our manual analysis, we observe that when a crash (`XWIKI-13031`) is complex, i.e., it takes several generations to produce a crash reproducing test case, weighted sum reduces execution time. However, when a

Table 2.2: Experiment results for Multi-objectivized (Multi-obj.), Weighted (WSS) and Simple Sum (SSS) Scalarization. "-" indicates that the optimization approach did not reproduce the crash. Bold cases represent the crashes only reproduced by some of the approaches, not all. **Rep., T.,** and **SD** indicate reproduction rate, average execution time, and standard deviation, respectively.

			Multi-obj.			WSS			SSS		
Crash ID	**Exception**	**Frame**	Rep.	$\overline{T}$	SD	Rep.	$\overline{T}$	SD	Rep.	$\overline{T}$	SD
CHART-4b	NPE	6	15	16.5	1.4	15	16.6	1.4	15	14.8	1.3
LANG-12b	AIOOBE	2	15	2.5	0.3	15	2.5	0.5	15	2.4	0.5
LANG-33b	NPE	1	15	1.7	0.0	15	1.0	0.2	15	1.0	0.0
LANG-39b	NPE	2	15	2.7	1.0	15	1.1	0.5	15	1.6	1.2
LANG-47b	NPE	1	15	3.4	1.3	15	2.1	1.1	15	1.0	0.7
LANG-57b	NPE	1	11	1.1	0.0	9	185.0	288.0	12	86.1	218.1
LANG-9b	AIOOBE	-	-			-			-		
MATH-100b	AIOOBE	1	15	8.4	13.4	15	7.2	1.7	15	8.2	7.3
MATH-32b	**CCE**	**1**	**15**	**3.9**	**0.9**	**15**	**5.3**	**2.5**	**-**		
MATH-4b	NPE	3	15	27.3	49.2	14	21.7	16.1	14	62.0	150.0
MATH-70b	NPE	3	15	1.7	0.2	15	1.1	0.3	15	1.0	0.0
MATH-79b	NPE	1	15	1.7	0.1	15	1.0	0.2	15	1.0	0.0
MATH-81b	AIOOBE	6	9	82.0	63.0	11	180.7	230.5	15	115.0	114.0
MATH-98b	AIOOBE	1	15	7.7	5.3	14	9.5	5.7	15	9.9	9.7
MOCKITO-12b	CCE	-	-			-			-		
MOCKITO-34b	AIOOBE	-	-			-			-		
MOCKITO-36b	NPE	1	15	10.9	6.9	15	9.2	7.5	15	13.7	11.3
MOCKITO-38b	NPE	-	-			-			-		
MOCKITO-3b	AIOOBE	-	-			-			-		
XRENDERING-418	NPE	-	-			-			-		
XWIKI-12482	NPE	-	-			-			-		
XWIKI-12584	CCE	-	-			-			-		
XWIKI-13031	CCE	3	15	25.8	17.4	15	47.2	67.0	10	249.0	175.0
XWIKI-13096	NPE	-	-			-			-		
XWIKI-13303	NPE	-	-			-			-		
XWIKI-13316	NPE	2	15	37.9	47.7	15	16.6	34.6	15	31.3	86.8
XWIKI-13377	CCE	1	15	10.7	8.6	15	11.8	7.7	15	4.8	3.9
XWIKI-13616	NPE	3	15	4.1	0.1	15	4.0	0.0	15	4.0	0.0
XWIKI-14227	NPE	-	-			-			-		
XWIKI-14319	NPE	1	15	87.0	21.2	15	89.4	17.5	15	87.8	15.2
XWIKI-14475	**NPE**	**1**	**15**	**117.1**	**53.6**	**-**			**-**		
XWIKI-13916	CCE	1	15	59.7	19.8	14	65.0	13.6	15	57.6	13.8
XWIKI-14612	NPE	1	15	8.9	2.0	15	8.7	1.8	15	8.5	2.4

crash, e.g., `XWIKI-13377`, is easy to reproduce, then weighted sum takes longer to find a crash reproducing test.

Results (RQ2)

Table 2.2 shows that 22 cases were reproduced using decomposed crash optimization objectives, while 21 cases were reproduced by the original weighted sum function. `XWIKI-14475` is reproduced by the multi-objectivized approach only.

As Table 2.3 shows, in most cases, we do not observe any impact on the rate of crash coverage. However, for `MATH-81b` and `LANG-57b`, the odds ratio measures are 4.8 and 1.7 respectively, which indicates that the rate of crash reproduction using multi-objectivized optimization is 4.8 times and 1.7 times higher than the rate of reproduction using the weighted sum function. For these cases, the *p*-values are 0.3 and 0.6 respectively, therefore, we cannot draw a statistically significant conclusion

Table 2.3: Comparing coverage rate and test generation time between the optimization approaches, for cases where both optimization approaches in each pair reproduces the crash. P-values for both Wilcoxon tests and odds ratios are reported. Effect sizes and p-values of the comparisons are in bold when the p-values are lower than 0.05.

			Multi-Weighted				Weighted-Simple			
Crash ID	**Exception**	**Fr.**	$\hat{A}_{12}$	p	OR	p	$\hat{A}_{12}$	p	OR	p
CHART-4b	NPE	6	0.3	0.30	0.0	1.0	**0.8**	**<0.01**	0.0	1.00
LANG-12b	AIOOBE	2	0.5	0.50	0.0	1.0	0.4	0.70	0.0	1.00
LANG-33b	NPE	1	**0.9**	**<0.01**	0.0	1.0	0.5	0.30	0.0	1.00
LANG-39b	NPE	2	**0.9**	**<0.01**	0.0	1.0	0.4	0.10	0.0	1.00
LANG-47b	NPE	1	**0.9**	**<0.01**	0.0	1.0	0.4	0.70	0.0	1.00
LANG-57b	NPE	1	0.6	0.20	1.7	0.6	0.5	0.60	0.3	0.40
MATH-100b	AIOOBE	1	**0.1**	**<0.01**	0.0	1.0	0.5	0.40	0.0	1.00
MATH-32b	CCE	2	**0.3**	**<0.01**	0.0	0.5	0.4	0.50	0.0	1.00
MATH-4b	NPE	3	**0.4**	**0.04**	1.0	1.0	0.4	0.70	1.0	1.00
MATH-70b	NPE	3	**0.8**	**<0.01**	0.0	1.0	0.5	0.10	0.0	1.00
MATH-81b	AIOOBE	6	0.5	0.60	4.8	0.3	0.5	0.50	0.0	0.09
MATH-98b	AIOOBE	1	**0.3**	**<0.01**	0.0	1.0	0.6	0.20	0.0	1.00
MOCKITO-36b	NPE	1	0.2	0.60	0.0	1.0	0.3	0.30	Inf	1.00
XWIKI-13031	CCE	3	**0.3**	**0.03**	Inf	1.0	**0.1**	**<0.01**	6.5	0.10
XWIKI-13316	NPE	2	0.6	0.09	0.0	1.0	0.6	0.10	0.0	1.00
XWIKI-13377	CCE	1	0.6	0.50	0.0	1.0	**0.7**	**0.01**	0.0	1.00
XWIKI-13616	NPE	3	**0.5**	**<0.01**	0.0	1.0	**0.5**	**<0.01**	0.0	1.00
XWIKI-14319	NPE	1	**0.4**	**<0.01**	0.0	1.0	0.5	0.70	0.0	1.00
XWIKI-13916	CCE	1	0.3	0.60	0.0	1.0	0.6	0.08	0.0	1.00
XWIKI-14612	NPE	1	0.5	0.40	0.0	1.0	0.4	0.70	0.0	1.00

yet.

Moreover, as Table 2.3 shows, for six cases, namely: `MATH-100b`, `MATH-32b`, `MATH-4b`, `MATH-98b`, `XWIKI-13031`, and `XWIKI-14319`, we observe that using multi-objectivization reduces the time for test generation (as $\hat{A}_{12}$ measures are lower than 0.5). For all these cases, the *p* values are lower than 0.05, which indicates the observed impacts are statistically significant. On the other hand, for four other cases, namely: `LANG-33b`, `LANG-39b`, `LANG-47b`, and `MATH-70b`, we observe an opposite trend, i.e., the weighted sum achieves a lower test generation time (as the $\hat{A}_{12}$ measures are larger than 0.5). Based on our manual analysis, as also indicated by the average execution time values reported in Table 2.2, when a crash is complex and the search requires several generations (e.g., `XWIKI-13031`), multi-objectivization reduces the execution time. On the other hand, when a crash is easy to be reproduced and a few generations of test cases quickly converge to a global optimum, then using the weighted sum approach is more efficient.

2.5 Discussion

As Table 2.3 shows, for only one case, `XWIKI-13031`, the weighted sum is more efficient than the simple sum, while for two other cases, `XWIKI-13377` and `CHART-4b`, the simple sum is more efficient. From our manual analysis of these cases, we see that when the target line is covered in a few seconds (when initializing the first population), the simple sum is more efficient than the weighted sum. However, when more search iterations (generations) are needed to find a test that reaches the target line, like for `XWIKI-13031`, the weighted sum is much faster.

While using weights in single-objective optimization may reduce the likelihood of getting stuck in local optima, it may accept solutions that trigger the target exception but not at the target code

line. Therefore, a possible explanation for these cases is that while maintaining diversity improves efficiency to a small degree, relaxing the constraints may penalize the exploitation. In practice, since it is not possible to know *a priori* when getting stuck in local optima occurs, using weighted sum (that provides more guidance, thanks to the constraints it takes into account) seems a more reliable approach, which might be few seconds less efficient compared to simple sum (in some cases).

As Knowles et. al [27] discussed, when applying multi-objectivization, for a successful search, it is important to derive independent objectives. In our multi-objectivization approach, we decompose the three heuristics in the original scalarized function into three optimization objectives. However, these objectives are not entirely independent of each other; line coverage is interrelated to the stack trace similarity. Thus, if the target line is not covered, the stack trace similarity will never converge to 0.0. This can be one possible explanation for why when the target frame is one, single-objective optimization performed better for most cases in our experiments. The fewer frames to reproduce, the stronger the interrelation between the two objectives is.

Furthermore, we observe that when a crash is complex and requires several generations to be reproduced, the multi-objectivized approach performs more efficiently than single-objective optimization. On the other hand, when crashes can be reproduced in few generations (i.e., the target line is covered by the initial population of GAs and evolution is mostly needed for triggering the same crash), then the single-objective approach is more efficient. This is due to the cost of the fast non-domination sorting algorithm in NSGA-II [10], whose computational complexity is $\mathcal{O}(\mathrm{MN}^2)$, where M is the number of objectives and N is the population size. Instead, the computational complexity of the selection in a single-objective GA is $\mathcal{O}(\mathrm{M})$, where N is the population size. Thus, sorting/selecting individuals is computationally more expensive in NSGA-II and it is worthwhile only when converging to 0.0 requires effective exploration through the enhanced diversity in NSGA-II.

2.6 Threats to validity

We randomly selected 33 crashes from five different open source projects for our evaluation. Those crashes come from Defects4J, a collection of defects from popular libraries, and from the issue tracker of our industrial partner, ensuring diversity in the considered projects.

In addition, the selected crashes contain three types of commonly occurring exceptions. While we did not analyze the exception types, they may be a factor that impacts the test generation time and crash reproduction rate.

2.7 Recommendations

From our results and discussion, we formulate the following insights:

1. **prefer multi-objectivization**, as it substantially reduces the execution time for complex crashes (up to three minutes) and the time loss for simple crashes is small (few seconds on average); furthermore, it allows to reproduce one additional crash that weighted sum could not reproduce;
2. Alternatively, use a **hybrid search** that switches from weighted sum to multi-objectivized search when the execution time is above a certain threshold (20 seconds in our case) or if the target code line is not covered within the first few generations; and finally,
3. Avoid **simple sum scalarization** as it may get stuck into local optima (multi-objectivization).

2.8 Summary

Crash reproduction is an important step in the process of debugging field crashes that are reported by end users. Several automated approaches to crash reproduction have been proposed in the literature

to help developers debug field crashes. EvoCrash is a recent approach which applies a Guided Genetic Algorithm (GGA) to generate a crash reproducing test case. GGA uses a weighted scalarized function to optimize test generation for crash reproduction. In this study, we apply the GGA approach as an extension of EvoSuite and show that using a weighted sum scalarization fitness function improves test generation compared to a simple sum scalarization fitness function when reproducing complex crashes. Moreover, we also investigate the impact of decomposing the scalarized function into multiple optimization functions. Similarly, compared to using the weighted scalarized function, we observe that applying multi-objectivization improves the test generation time when reproducing complex crashes requiring several generations of test case evolution.

In general, we believe that multi-objectivization is under-explored to tackle (by-nature-)single-objective problems in search-based software testing. Our results on multi-objectivization by decomposition of the fitness function for crash reproduction are promising. This calls for the application of this technique to other (by-nature-) single-objective search-based problems.

Chapter 3

Seeding search-based crash reproduction

As other search-based approaches, one of the challenges of search-based crash reproduction is to bring enough information to the fitness function in order to achieve its optimal value. For instance, complex elements (like strings with a particular format or objects with a complex structure) are hard to initialize without additional information. This can lead to two different issues: first, complex elements take more time to be generated, which can prevent to find a solution within the time budget allocated to the search; and second, elements that require complex initialization procedures (*e.g.*, specific sequences of method calls to set up an object) may prevent the search to start if the search-based approach is unable to create an initial population.

Rojas *et al.* [37] demonstrated that *seeding* is beneficial for the search process. More specifically, by analyzing source code (collecting information that relates to numeric values, string values, and class types) and existing tests (collecting information about the states of the objects in the test) and making them available for the search process, the overall coverage of the generated test improves. However, current seeding strategies focus on collecting and reusing values and object states as-is.

In this chapter, we define, implement, and evaluate a new seeding strategy, called *behavioral model seeding*, which abstracts behavior observed in the source code and test cases using transition systems. Behavioral model seeding takes advantage of the advances made by the model-based testing community [48] and uses them to enhance search-based software testing. The transition systems represent the (observed) usages of the classes and are used during the search to generate objects and sequences of method calls on those objects.

We apply behavioral model seeding to the problem of search-based crash reproduction. Contrarily to search-based test case generation aiming at generating a test suite that maximizes statement and branch coverage, the goal of search-based crash reproduction is to generate a single test that reproduces a crash. In this context, behavioral model seeding is used to focus the search, as the execution steps leading to a crash are likely to be close to the execution steps of valid executions of the system.

We implemented behavioral model seeding and *test seeding* [37] and use the latter as a baseline since it has never been applied to search-based crash reproduction. We performed an evaluation to answer the following research questions:

RQ1 What is the influence of *test seeding used during initialization* on search-based crash reproduction?

RQ2 What is the influence of *behavioral model seeding used during initialization* on search-based crash reproduction?

From the perspective of *effectiveness* (of initializing the population and reproducing crashes) and *efficiency*. We also investigate the factors that influence the test and model seeding approaches. Our results show that behavioral model seeding can improve the search process by increasing the number

of searches actually started by 14% and the number of crash reproductions by 13%, without any impact on efficiency.

This chapter has been adapted and submitted to the *41st ACM/IEEE International Conference on Software Engineering* (ICSE '19).

3.1 Background and related work

3.1.1 Seeding strategies for search-based testing

Seeding strategies for search-based testing use related knowledge to help the generation of test cases and optimize the fitness of the population. We focus here on the usage of the source code and the available tests as primary sources of information. Other approaches use, for instance, the internet [31] or the existing test corpus [45] to mine relevant formatted string values (*e.g.*, XML or SQL statements).

Seeding from the source code

Three main seeding strategies are using the source code for search-based testing [14, 37]:

1. *constant seeding* uses static analysis to collect and reuse constant values appearing in the source code (*e.g.*, constant values appearing in boundary conditions);
2. *dynamic seeding* complements constant seeding by using dynamic analysis to collect numerical and string values, observed only during the execution of the software, and reuse them for seeding; and
3. *type seeding* is used to determine the object type that should be used as an input argument, based on a static analysis of the source code (*e.g.*, by looking at `instanceof` conditions or generic types for instance).

Seeding from the existing tests

Rojas *et al.* [37] suggest two test seeding strategies, using dynamic analysis on existing test cases: *cloning* and *carving*. For cloning, the execution of an existing test case is copied and used as a member of the initial population of a search process. For carving, the state of an object (*i.e.,* the sequences of methods called on that object) is reused during the initialization of the population (as part of a newly created test case) and mutation of the individuals (*e.g.*, when a new object is created in the test case).

The integration of seeding strategies to the crash reproduction is also illustrated in Figure 3.2, box 5. As shown, the test cases (resp. objects) to be used by the algorithm are stored in test cases (resp. objects) pool from which they can be used according to user-defined probabilities. For instance, if a `LinkedList` is created (using `new`) and filled using two `add` method calls in a test, this sequence (`new`, `add`, `add`) may be used as-is in the initial population (cloning) or inserted by a mutation into other test cases (carving).

3.1.2 Behavioral model-based testing

Model-based testing [48] relies on abstract specifications (models) of the system under test to support the generation of relevant test cases. *Transition systems* [3] have been used as a fundamental formalism to reason on test case generation and support the definition of formal test selection criteria [47]. Once selected from the model, *abstract test cases* are concretized (by mapping the transition system's paths to concrete sequences of method calls) into *executable test cases* (simply called test cases in this chapter) to be run on the system. Figure 3.1 shows an example of a transition system representing the

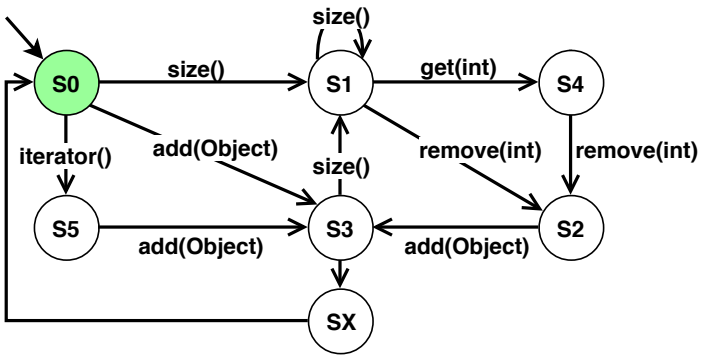

Figure 3.1: Transition system for `java.util.LinkedList` from Apache commons math source code and test cases.

possible *sequences* of method calls on `java.util.List` objects. Figure 3.1 illustrates usages of methods in `java.util.List` objects found in code and tests in terms of a transition system, from which *sequences* of methods calls can be derived.

A transition system is composed of a set of states (with s_0 being the initial state) and transitions. Each transition may be labeled with an action, representing here a method call.

The abstract test cases are selected from the transition system following criteria defined by the tester. In the remainder of this chapter, we use *dissimilarity* as selection criteria [5, 19]. Dissimilarity selection, which aims to maximize the fault detection rate by increasing diversity among test cases, has been shown to be an interesting and scalable alternative to other classical selection criteria [19, 32]. This diversity is measured using a dissimilarity distance (here, 1 - the Jaccard index [22]) between the actions of two abstract test cases.

The model may be directly specified (and in this case will generally focus on specific aspects of the system) [48], or automatically inferred from observations of the system [20, 29, 44, 43, 46, 50]. In the latter case, the model will be incomplete and only contain the *observed behavior* of the system. This behavior can be obtained via static analysis (like in this chapter) or dynamically (e.g., [28]). Model inference may be used for visualization [29, 50], or generation [20, 44, 43, 36] and prioritization [11, 12] of test cases. We use n-gram inference to build the transition systems used for model seeding. N-gram inference takes a set of sequences of actions as input to produce a transition system where the n^{th} action depends on the $n-1$ previously executed actions. As a small value of n enables better diversity in the behavior allowed by the model (ending up in more diverse abstract test cases), requires less observations to reach stability of the model, simplifies the inference, and results in a more compact model [44, 43], we use 2-gram inference to build our models.

The test case selection from patterns of common object usages, defined by Fraser *et al.* [17], is the most related approach. In their approach, Fraser *et al.* extract a single API usage model from the client code using static analysis and select test cases from that model. Since the API usage model directly represents source code, the concretization is trivial (*i.e.,* a one-to-one mapping). Our approach differs by also considering dynamic analysis of the test cases to have a better accuracy of the observed usages. Additionally, we do not build a unique usage model of a whole API. Instead, we generate models for classes involved in a stack trace and use those models to generate objects. Since we are in a crash reproduction context, this approach allows a better diversity of the behaviors, hopefully leading to an uncommon one that reproduces the crash.

3.2 Behavioral model seeding approach

The goal of behavioral model seeding (denoted model seeding hereafter) is to abstract the behavior of the software under test using models and use that abstraction during the search. At the unit

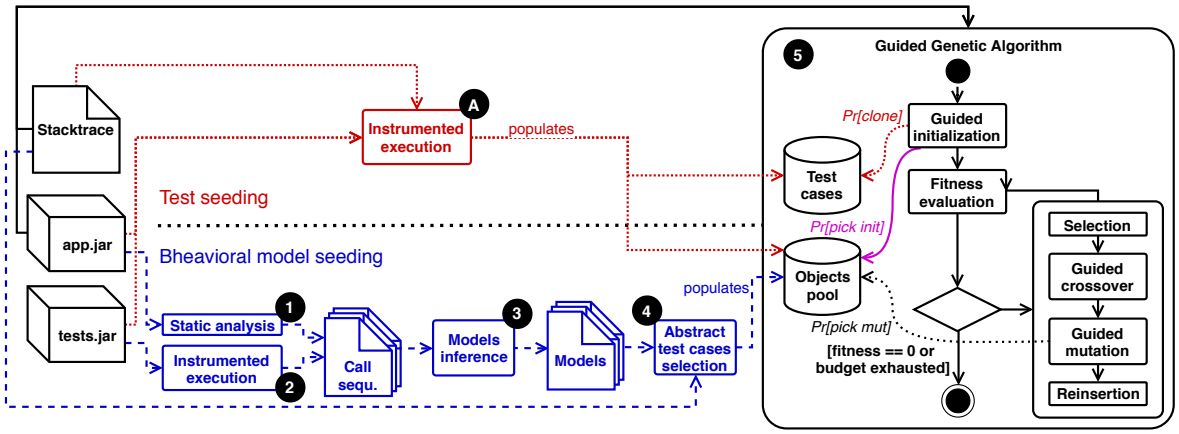

Figure 3.2: General overview of model seeding and test seeding for search-based crash reproduction

test level, each model is a transition system, like in Figure 3.1, and represents possible usages of a class: *i.e.,* possible sequences of method calls observed for objects of that class. For instance, the transition system model in Figure 3.1 has been partially generated from the following call sequences: (`<size(), remove(), add(Object), size(), size(), size(), get(), remove(), add(Object)>`; `<iterator(),add(Object)>`; *etc.*). Each transition in Figure 3.1 corresponds to a method call in one of the sequences, and for each sequence, there exists a path in the model.

The main steps of our model seeding approach, presented in Figure 3.2, are: the *inference* of the individual models (3) from the *call sequences* collected through *static analysis* (1) performed on the application code, and *dynamic analysis* (2) of the test cases; and for each model, the *selection of abstract test cases* (4), that are concretized into Java objects, stored in an *object pool* from which the guided genetic algorithm (5) can randomly pick objects to build test cases during the search process.

3.2.1 Model inference

Call sequences are obtained by using static analysis on the bytecode of the application and by instrumenting and executing the existing test cases. For each class, the model is obtained using a 2-gram inference method using the call sequences of that class. For 2-gram inference, the next action to execute only depends on the current state in the transition system. For instance, in the transition system of Figure 3.1, the action `size()`, executed from state s_3 at step k only depends on the fact that the action `add(Object)` has been executed at step $k-1$, independently of the fact that there is a step $k-2$ during which the action `iterator()` has been executed. Increasing the value of n for the n-gram inference would result in wider transition systems with more states and less incoming transitions, representing a more constrained behavior and producing less diverse test cases.

Static analysis of the application

The static analysis is performed on the bytecode of the application. To keep the process within a reasonable amount of time, we only apply this analysis to the classes involved in the stack trace. In each method of these classes, we collect the sequences of method calls for all of the available classes. Since a stack trace represents a call hierarchy, this analysis ensures collecting relevant call sequences for at least the class at frame level k in the methods of the class at frame level $k+1$. For instance, in the stack trace of Listing 3.1, call sequences of `BaseObject` objects are collected (at least) in class `XWikiDocument`, which calls methods of `BaseObject`.

Listing 3.1: Stack trace of the XWIKI-13372 crash

```
java.lang.NullPointerException: null
  at com[...]BaseProperty.equals([...]:96)
  at com[...]BaseStringProperty.equals([...]:57)
  at com[...]BaseCollection.equals([...]:614)
  at com[...]BaseObject.equals([...]:235)
  at com[...]XWikiDocument.equalsData([...]:4195)
  [...]
```

Dynamic analysis for the test cases

Dynamic analysis of the existing tests is done in a similar way to the carving approach of Rojas *et al.* [37]: instrumentation adds log messages to indicate when a method is called, and the sequences of method calls are collected after execution. In similar fashion to static analysis, we collect call sequences of any observed object (even objects which are not defined in the software under test). The quality of the collected sequences depends on the quality of the existing tests.

Dynamic analysis for the test cases complements static analysis by collecting more call sequences for internal, but also publicly exposed methods. Dynamic analysis is also more precise in the collected method calls since it executes the code. For instance, considering two consecutive `if` statements with exclusive conditions, dynamic analysis collects only the method calls effectively executed, while static analysis will collect all the method calls appearing in the source code.

3.2.2 Abstract test cases selection

Abstract test cases are selected from the transition systems and concretized to populate the objects pool used during the search. Each abstract test case corresponds to a sequence of method calls on one object: *i.e.,* a path in the transition system starting from the initial state and ending in the initial state, a commonly used convention to deal with finite behaviours [11]. To limit the number of objects in the pool, we only select abstract test cases from two categories of models: models of internal classes and models of dependency classes that are involved in the stack trace. Since we do not seek to validate the implementation of the application, the states are ignored during the selection process.

Selection

There exist various criteria to select abstract test cases from transition systems [48]. To successfully guide the search, we need to establish a good ratio between *exploration* (the ability to visit new regions of the search space) and *exploitation* (the ability to visit the neighborhood of previously visited regions) [9]. The guided genetic operators guarantee the exploitation by focusing the search on the methods in the stack trace. However, depending on the stack trace, focusing on particular methods may reduce the exploration. Poor exploration decreases the diversity of the generated tests and may trap the search process in local optima.

To improve the exploration, we use *dissimilarity* as the criterion to select the abstract test cases. Compared to classical structural coverage criteria that seek to cover as many parts of the transition system as possible, dissimilarity tries to increase diversity among the test cases by maximizing a distance: 1 - the ratio between the actions common to two abstract test cases and the total number of actions appearing in the two test cases (the Jaccard index).

Concretization

Each abstract test case has to be concretized to an object and method calls before being added to the objects pool. In other words, for each abstract test case, if the constructor invocation is not the first

Listing 3.2: Concretized abstract test case for `LinkedList`

```
int[] t = new int[7];
t[3] = (-2147483647);
EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
LinkedList<[...]> lst = new LinkedList<>();
lst.add(ep);
lst.add(ep);
```

Listing 3.3: Test generated for frame 2 of MATH-79b

```
public void testCluster() throws Exception{
 int[] t = new int[7];
 t[3] = (-2147483647);
 EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
 LinkedList<[...]> lst = new LinkedList<>();
 lst.add(ep);
 lst.add(ep);
 KMeansPlusPlusClusterer<[...]> kmean = new KMeansPlusPlusClusterer<>(12);
 lst.offerFirst(ep);
 kmean.cluster(lst, 1, (-1357));
}
```

action (this is usually the case when the call sequences used to infer the model have been captured from objects that are parameters or attributes of a class), one constructor is randomly called; and the methods are called on this object in the order specified by the abstract test case with randomly generated parameter values. Due to the randomness, each concretization may be different from the previous one. For each abstract test case, n concretizations (default value is $n = 1$ to balance scalability and diversity of the objects in the object pool) are done for each abstract test case and saved in the objects pool.

For instance, Listing 3.2 shows the concretized abstract test case `<add(Object), add(Object)>` derived from the transition system model of Figure 3.1. The type of the parameters (`EuclideanIntegerPoint`) is randomly selected during the concretization and created with required parameter values (an integer array here).

3.2.3 Guided Initialization and Guided Mutation

Objects are instantiated during two main steps of the guided genetic algorithm: guided initialization, where objects are used to create the initial set of test cases; and guided mutation, where objects may be required as parameters when adding a method call. When no seeding is used, those objects are randomly created (as in concretization) by calling the constructor and random method calls.

To preserve exploration in model seeding, objects are picked in the object pool during guided initialization (resp. guided mutation) according to a user-defined probability $Pr[pick\ init]$ (resp. $Pr[pick\ mut]$), and randomly generated otherwise. The value of the probability depends on the number and diversity of the concretized abstract test cases. For instance, a small model will deliver a few different abstract test cases. In this case, lower probability values allow more randomly generated objects with calls to methods that do not appear in the model. In contrast, a large model, denoting complex behavior for the objects, may require higher probability values to avoid incorrect sequences of method invocations.

For instance, during our evaluation, model seeding generated the test case in Listing 3.3 for the

second frame of the following stack trace (crash MATH-79b from the Apache commons math project):

```
java.lang.NullPointerException
 at ...KMeansPlusPlusClusterer.assignPointsToClusters()
 at ...KMeansPlusPlusClusterer.cluster()
```

The target method is the last method called in the test (line 10) and throws a `NullPointerException`, reproducing the input stack trace. The first parameter of the method has to be a `Collection<T>` object. In this case, the guided genetic algorithm picked the list object from the object pool (from Listing 3.2) and inserted it in the test case (lines 2 to 7). The algorithm also modified that object (during guided mutation) by invoking an additional method on the object (line 9).

3.2.4 Test seeding

As for model seeding, test seeding starts by executing the test cases (Figure 3.2 box A). The observed behavior is used as-is to populate the test cases and objects pools. The test cases pool is used only during guided initialization to clone test cases that contain the target class, according to a user-defined $Pr[clone]$ probability. If the target method is not called in the cloned test case, the guided initialization also mutates the test case to add a call to the target method.

As for model seeding the object pool is used during the guided initialization and guided mutation to pick objects, according to the same user-defined probabilities: $Pr[pick\ init]$ and $Pr[pick\ mut]$.

3.2.5 Comparison between behavioral model seeding and test seeding

Behavioral model seeding exploits both test cases and source code, thereby *subsuming* test seeding regarding the observed behavior of the application that is reused during the search. For instance, to initialize a list like in Listing 3.3, test seeding can only seed the call sequences which are observed during the execution of the test cases, while model seeding can use any path in the transition system of Figure 3.1 for seeding. However, by using cloning, test seeding benefits from (fixed) readily usable objects during initialization at the expense of diversity.

3.3 Evaluation

In order to assess the usage of test seeding applied to crash reproduction and our new model seeding approach during the guided initialization, we performed an empirical evaluation to answer our two research questions:

RQ1 *What is the influence of test seeding used during initialization on search-based crash reproduction?* To answer this research question, we compare executions with *test seeding* enabled to executions where no additional seeding strategy is used (denoted *no seeding* hereafter), from their effectiveness to reproduce crashes and start the search process, the factors influencing this effectiveness, and the impact of test seeding on the efficiency. We divide **RQ1** into four sub-research questions:

RQ1.1 Can test seeding help to initialize the search process?

RQ1.2 Does test seeding help to reproduce more crashes?

RQ1.3 Does test seeding impact the efficiency of the search process?

RQ1.4 Which factors in test seeding help the search process compared to no seeding?

RQ2 *What is the influence of behavioral model seeding used during initialization on search-based crash reproduction?* To answer this question, we compare executions with *model seeding* to executions with *test seeding* and *no seeding*. We also divide **RQ2** into four sub-research questions:

Table 3.1: Projects used for the evaluation

Application	Cr.	$\overline{\text{frm}}$	$\overline{\text{CCN}}$	$\overline{\text{LOC}}$	$\overline{\text{LC}}$	$\overline{\text{BC}}$
JFreeChart	1	6.00	3.49	66,938.00	65%	53%
Commons-lang	6	2.83	4.21	13,021.00	80%	78%
Commons-math	8	3.38	3.18	27,039.25	91%	86%
Mockito	5	3.20	2.78	5,128.00	100%	93%
XWiki	25	11.40	1.96	44,541.48	50%	49%

RQ2.1 Can behavioral model seeding help to initialize the search process compared to no seeding and test seeding?

RQ2.2 Does behavioral model seeding help to reproduce more crashes comparing to no seeding and test seeding?

RQ2.3 Does behavioral model seeding impact the efficiency of the search process collating to no seeding and test seeding?

RQ2.4 Which factors in behavioral model seeding help the search process compared to no seeding and test seeding?

3.3.1 Setup

We extend the 33 crashes used in Chapter 2 to 45 crashes from additional versions of the same projects. Table 3.1 presents the list of applications and number of crashes that may happen on different versions of this application (**Cr.**) used for the evaluation, with, for each one: the average number of frames in the stack traces of the crashes ($\overline{\text{frm}}$), the average cyclomatic complexity ($\overline{\text{CCN}}$), the average number of lines of code ($\overline{\text{LOC}}$), the average line coverage of the existing test cases ($\overline{\text{LC}}$), and the average branch coverage of the existing test cases ($\overline{\text{BC}}$).

For each frame, we executed the search-based crash reproduction 10 times for each configuration of the 3 seeding strategies: *no seeding* (i.e., no additional seeding compared to the default parameters), *test seeding*, and *model seeding*. We used a budget of 62,328 fitness evaluations (corresponding on average to 15 minutes execution with no seeding on our infrastructure) to avoid side effects on execution time when executing search-based crash reproduction on different frames in parallel. We also fixed the population size to 100 individuals. All other configuration parameters have their default value [37], and we used the default weighted sum scalarisation fitness function from Soltani *et al.* [39].

Test seeding

During the guided initialization of the algorithm, test seeding can clone test cases according to a user-defined probability $Pr[clone]$. Additionally to the default 0.2 value, we executed model seeding with values 0.5, 0.8, and 1.0 for $Pr[clone]$. In the current EvoSuite implementation, there is no difference between $Pr[pick\ init]$ and $Pr[pick\ mut]$ (both called `p_object_pool` in the list of configuration parameters). Since in this study we focus on seeding during the initialization, we left the default value of 0.3 for those two parameters. Assessing the influence of test seeding on mutation is part of our future work.

Model-seeding

Our implementation of model seeding makes a distinction between using the object pool during guided initialization and guided mutation (as shown in Figure 3.2). This distinction enables to study

the influence of seeding during the different steps of the algorithm independently. In this evaluation, we focus on guided initialization and use the following values for $Pr[pick\ init]$: 0.2, 0.5, 0.8, 1.0. The value of $Pr[pick\ mut]$ is fixed to 0.3.

Infrastructure

We used 2 clusters (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. For each stack trace, we executed an instance of search-based crash reproduction for each frame which points to a class of the application. We discarded other frames to avoid generating test cases for external dependencies. We ran 351 frames from 45 stack traces 10 times for each seeding strategy configuration, for a total of 28,080 independent executions.

3.3.2 Analysis

To check if the search process can achieve a better state using seeding strategies, we analyze the status of the search process after executing each of the cases (each run in one frame of a stack trace). We define 5 states:

1. **not started**, the initial population could not be initialized, and the search did not start;
2. **failed**, the target line could not be reached;
3. **line reached**, the target line has been reached, but the target exception could not be thrown;
4. **ex. thrown**, the target line has been reached, and an exception has been thrown but produced a different stack trace; and
5. **reproduced** the stack trace could be reproduced.

Since we repeat each execution 10 times, we use the majority of outcomes for a frame reproduction result. For instance, if a frame could be reproduced in the majority of the 10 runs, we count that frame as a *reproduced*.

To check whether each strategy is statistically different from each other in different aspects, we use the Friedman test [18]: a non-parametric version of the ANOVA test [13] (*i.e.,* there is no assumption about the distribution of the data). We use it to compare: the number of executions where the search could start to answer **RQ1.1** and **RQ2.1**; the number of executions that could reproduce a frame to answer **RQ1.2** and **RQ2.2**; and the number of fitness function evaluations needed by the executions for the frames that could be reproduced to answer **RQ1.3** and **RQ2.3**. For this last case, since the number of reproductions could be different from one seeding configuration to another, executions that could not reproduce the frame simply reached the maximum allowed budget (62,328).

Additionally, based on the results of the tests, we rank the seeding configurations using the standard method of Panichella *et al.* [35, 24]: to produce a global ranking, we produce sub-rankings for each seeding configuration according to their performance for each individual frame. The average of a seeding configuration sub-rankings gives the ranking of that configuration.

For Friedman's test, we use a level of significance $\alpha = 0.01$. If the p-values obtained from Friedman's test are significant (p-values $<= 0.01$), we apply pairwise multiple comparison using Conover's post-hoc procedure [8]. To correct for multiple comparisons, we adjust the p-values from Conover's procedure using Holm-Bonferroni [21].

To answer **RQ1.4** and **RQ2.4**, we performed a manual analysis on the logs and crash reproducing test case (if any) produced by the executions of the frames:

1. for which the search could start in one seeding configuration but not in the others;
2. that could be reproduced by one of the seeding configuration but not the others.

Based on the manual analysis, we used a card sorting strategy by assigning keywords to each frame result and grouping those keywords to identify influencing factors.

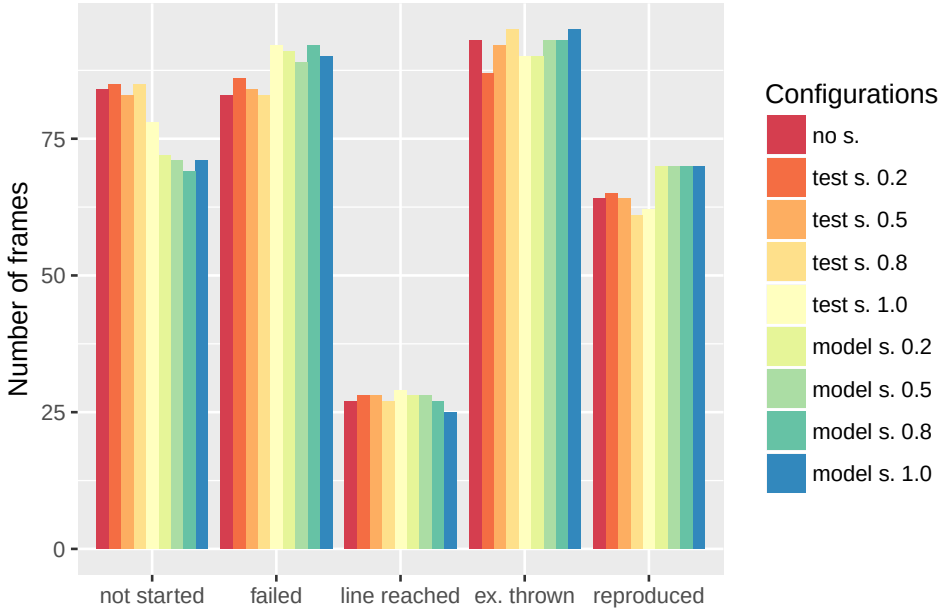

Figure 3.3: Evaluation results for each frame of each crash.

3.3.3 Results

Figure 3.3 presents the general results of our evaluation: for each frame and each seeding configuration, we report the outcome observed in the majority of the 10 executions. Configurations refer to the configuration used for seeding: no seeding (no s.), test seeding with $Pr[clone] \in \{0.2, 1.0\}$ (*test s.*), and behavioral model seeding with $Pr[pick\ init] \in \{0.2, 0.5, 0.8, 1.0\}$ (*model s.*).

Figure 3.4 provides the distribution of the frames that could be reproduced out of 10 execution rounds for each seeding configuration.

Figure 3.5 presents the distribution of the number of fitness evaluations for the frames that could be reproduced by each seeding configuration in the 10 rounds of execution. The number of fitness evaluations varies between 0 (*i.e.,* the crash is reproduced by one of the test cases of the initial population) and 62,328 (*i.e.,* the maximum budget allocated for the search). For each box, we also provide the mean (represented as a white diamond) and number of observations (*i.e.,* the number of executions with a reproduction) at the top.

3.3.4 RQ1 Influence of test seeding

Guided initialization effectiveness (RQ1.1)

We observe in Figure 3.3 that *test s. 1.0* helps to start more search processes for 6 frames compared to no seeding (light yellow bar in *not started* is lower). *Test s. 0.5* only helped with 2 more frames in the majority of the cases compared to no seeding, and *test s. 0.2* and *test s. 0.8* started 1 frame less than no seeding (thus doing worse). To answer **RQ1.1**, we use Friedman's test to rank the different seeding configurations. Results are reported in Table 3.2, smaller values of rank indicate better effectiveness. For each configuration, we also report whether it is significantly better according to the post-hoc procedure. We observe that only *test s. 1.0* and *test s. 0.5* significantly help to start more search processes during guided initialization compared to no seeding. Other configurations are either worse (*test s. 0.2*) or not significant (*test s. 0.8*).

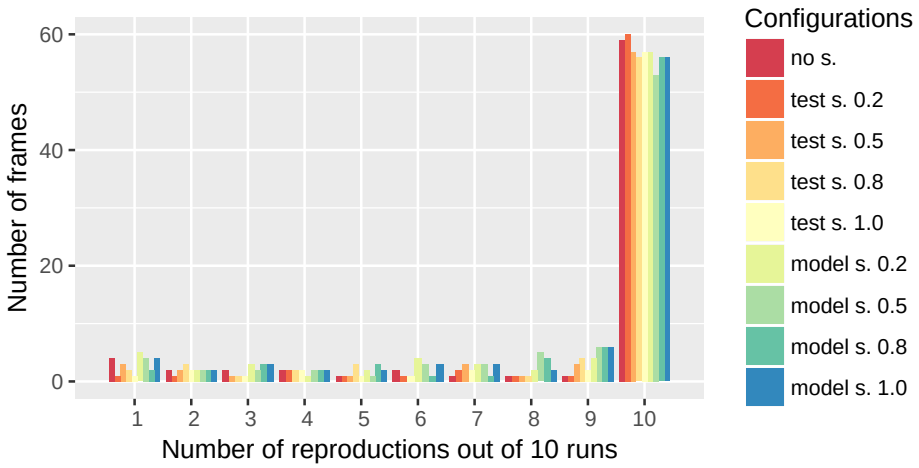

Figure 3.4: Distribution of the numbers of reproduced frames in 10 rounds of execution.

Table 3.2: Starting search ratio ranking.

Rank	Strategy	Rank value	Significantly better than
1	model s. 1.0	4.91	(4), (5), (6), (7), (8), (9)
2	model s. 0.5	4.96	(8), (9)
3	test s. 1.0	4.96	(8), (9)
4	test s. 0.5	4.98	(8), (9)
5	model s. 0.2	4.99	(8), (9)
6	model s. 0.8	5.00	(8), (9)
7	test s. 0.8	5.01	(9)
8	no_seeding	5.06	(9)
9	test s. 0.2	5.14	

Crash reproduction effectiveness (RQ1.2)

Compared to no seeding, *test s. 0.2* achieved to reproduce 1 more frame in a majority of runs (orange bar is one higher in *reproduced* columns in Figure 3.3), while *test s. 0.5* achieved to reproduce the same number of frames. *Test s. 0.8* and *test s. 1.0* reproduced resp. 3 and 2 frames less than no seeding.

Table 3.3 ranks the different configurations regarding their crash reproduction capability (**RQ1.2**). Only *test s. 0.2* ranks better than no seeding, but not significantly. All other configurations are not significantly worse. Only test seeding with *test s. 0.2* performs significantly better than *test s. 0.5* and *test s. 1.0*.

Crash reproduction efficiency (RQ1.3)

We observe that there is no major difference in the number of fitness evaluations between test seeding and no seeding (orange / yellow numbers in figure 3.5 are similar). As the value of $Pr[clone]$ increases, the number of fitness evaluations slightly decreases from 1.7% of the search budget on average for *test s. 0.2* to 1.5% on average for *test s. 1.0*.

To answer **RQ1.3**, Table 3.4 ranks the different configurations according to the number of fitness evaluations used by the guided generic algorithm for the frames that have been reproduced (in majority) by one of the configurations. Executions that could not achieve to reproduce a frame reached the maximum number of fitness evaluations: 62,328. Only *test s. 0.2* performs significantly better than

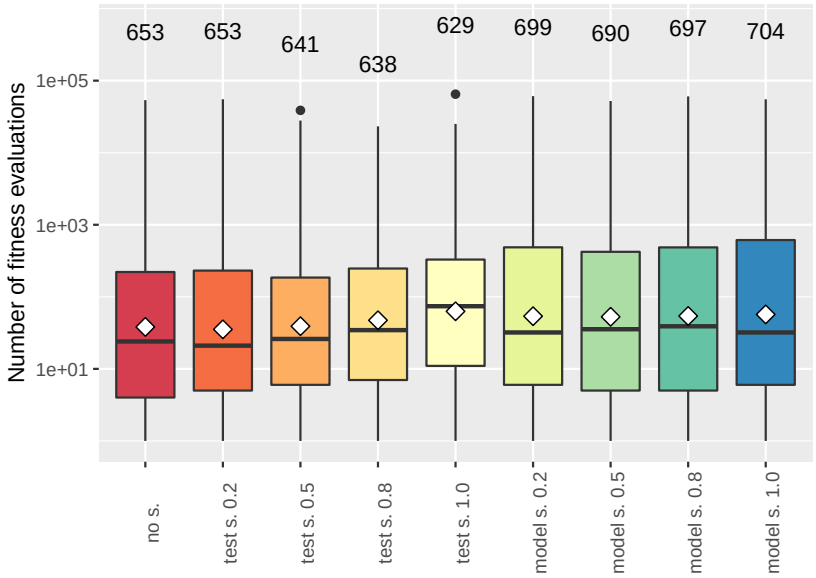

Figure 3.5: Distribution of the numbers of fitness evaluations for the reproduced frames in the 10 rounds of execution.

Table 3.3: Crash reproduction ranking.

Rank	Strategy	Rank value	Significantly better than
1	model s. 1.0	4.57	(4), (5), (6), (7), (8), (9)
2	model s. 0.2	4.63	(4), (5), (6), (7), (8), (9)
3	model s. 0.8	4.81	(7), (8), (9)
4	test s. 0.2	4.97	(8), (9)
5	no s.	5.05	
6	model s. 0.5	5.05	
7	test s. 0.8	5.26	
8	test s. 0.5	5.32	
9	test s. 1.0	5.33	

the three other test seeding configurations, but not significantly better than no seeding.

Influencing factors (RQ1.4)

From our manual analysis, we identified two factors of the test seeding process that helps the search:

- **clone and mutate tests** (3 frames) used during guided initialization; and
- **seed call sequences** (3 frames) that are used during the search process.

For the first factor, we observe that *cloning existing test cases* in the initial population enables to *start more search processes* when the target class is difficult to access. For instance, frames 7 and 8 of the crash LANG-9b point to an anonymous class and the underlying test generation facilities have no access to these types of classes [15]. By seeding test cases that use this anonymous class, *test s. 1.0* (which always clones test cases) was the only configuration able to start the search 10 out of 10 times for frames 7 and 8 (but could not achieve reproduction). No other seeding configuration could start the search process.

Table 3.4: Number of fitness evaluations ranking.

Rank	Strategy	Rank value	Significantly better than
1	test s. 0.2	5.83	(4), (5), (6), (7), (8), (9)
2	no_seeding	5.59	(7), (8), (9)
3	model s. 0.2	5.29	(7), (8), (9)
4	model s. 0.8	5.21	(8), (9)
5	model s. 1.0	5.18	(8), (9)
6	model s. 0.5	5.09	(8), (9)
7	test s. 0.5	4.73	(8), (9)
8	test s. 0.8	4.10	
9	test s. 1.0	3.98	

For the second factor, we observe that, despite the fixed value of $Pr[pick\ mut]$ for test seeding, the objects with call sequences carved from the existing tests and stored in the object pool help during the search. For instance, for frame 1 of the crash MATH-79b, the search process needs to initialize a `List` object with at leasts two elements before calling the target method in order to reproduce the crash. In our case, such an object had been carved from the existing tests and allowed test seeding to reproduce the crash for that frame 6.5 times on average against 4 for no seeding.

Summary (RQ1)

Test seeding with proper values of $Pr[clone]$ can reduce non-started searches (up to 7%) and significantly outperforms the no seeding strategies by cloning and mutating existing tests and allowing to access more target classes.

However, improper values can be detrimental to initialization (even worse than no seeding). While generally not significantly improving the crash reproduction ratio, despite a fixed value for $Pr[pick\ mut]$, test seeding can outperform no seeding in some cases (*e.g.*, first frame of MOCKITO-12b), thanks to the call sequences carved from the existing tests.

3.3.5 RQ2 Influence of behavioral model seeding

Guided initialization effectiveness (RQ2.1)

As depicted in Figure 3.3, model seeding could not start the search for between 69 (*model s. 0.8*) and 72 (*model s 0.8*) frames, performing better than no seeding (84 frames) and test seeding (between 78 and 85 frames). Table 3.2 (answering **RQ2.1**) shows that all model seeding configurations perform significantly better than no seeding and *test s. 0.2*. Yet, for *test s. 1.0* the improvement is not significant.

Crash reproduction effectiveness (RQ2.2)

All the model seeding configurations achieved the reproduction of 70 frames, compared to no seeding (64 frames) and test seeding (between 61 and 65 frames), in a majority of the runs. However, among the 10 runs of executions, *model s. 1.0* and *model s. 0.2* performed the best by achieving reproduction in respectively 704 and 599 executions.

This is confirmed by results in Table 3.3, answering **RQ2.2**: both *model s. 1.0* and *model s. 0.2* are significantly better than no seeding and all test seeding configurations. *Model s. 0.8* performed significantly better than all test seeding configurations but *test s. 0.2*. And only *model s. 0.5* does not rank significantly better than no seeding and test seeding configurations.

Crash reproduction efficiency (RQ2.3)

On average, model seeding takes slightly more fitness function evaluations, compared to test seeding and no seeding, but also has more executions with reproduction: between 2.6% of the search budget for *model s. 0.5* and 2.8% for *model s. 1.0*, between 1.3% for *test s. 0.8* with the lowest number of successful executions and 1.7% for *test s. 0.2*, and 2.1% for no seeding.

Table 3.3 helps to answer **RQ2.3**: we observe that all *model s.* perform worse than no seeding, but not significantly. Only *model s. 0.5* and *model s. 1.0* performed significantly worse than *test s. 0.2*. *Model s. 0.2* ranks significantly better than *test s. 0.5* and *test s. 1.0*.

Influencing factors (RQ2.4)

From our manual analysis, in addition to the **seed call sequences** factor also present in model seeding, we identified three factors in the test seeding process that help the search:

- using **dissimilar call sequences** (2 frames) for guided initialization;
- having **multiple information sources** (18 frames) to infer the behavioral models; and
- **prioritize the call sequences** (3 frames) for seeding by focusing on the classes involved in the stack trace.

Using *dissimilar call sequences* to populate the object pool in model seeding seems particularly useful to initialize the search compared to test seeding. In particular if the number of test cases is large, model seeding enables to (re)capture the behavior of those tests in the model and regenerate a smaller set of call sequences which maximize diversity, augmenting the probability to have more diverse objects used during the initialization. For instance, for the second and third frames of the crash CHART-4b, test seeding has to carve objects from 33 test suites, each one containing on average 10 test cases, making test seeding unable to start the search. In contrast, model-seeding is always able to start the search.

Having *multiple sources* to infer the model helps to select diversified call sequences compared to test seeding. For instance, the third frame of the crash XWIKI-13372 points to a class called `BaseObject` and could be reproduced 5.75 times on average with model seeding compared to 0.5 times with test seeding and 2 times with no seeding. By examining the generated crash reproducing test cases, we observe that they all contain two specific call sequences on a `BaseObject` instance that is also a path in the transition system generated for that class. Those specific parts of the model have been learned from static analysis of the source code.

Finally, by prioritizing classes involved in the stack trace for the abstract test cases selection, the object pool contains more objects likely to help to reproduce the crash. For instance, for the 10th frame of the crash LANG-9b, model seeding could achieve reproduction 6.75 times on average, compared to 0 for test and model seeding, by using the class `FastDateParser` appearing in the stack trace.

Summary (RQ2)

Model seeding achieves a better search initialization ratio from 14% (worst case) to 18 % (best case) with respect to no seeding. With respect to the best achievement of test seeding (*test s. 1.0*), it decreases the number of not started searches from 12% to 8%.

Compared to no seeding, model seeding increases the number of frames that can be reproduced in the majority of times to 13%. This is notable since test seeding only improves crash reproduction cases by less than 1%.

Model seeding takes on average slightly more fitness function evaluations, compared to test seeding. However, we argue that the difference in the number of fitness evaluations (1.5% at most, around 13.5 seconds in our case) is minor, especially considering the higher effectiveness of model seeding.

The helping factors include the *seeding of call sequences* (as for test seeding), the diversity of the sources used to build the behavioral model and the selection of dissimilar call sequences as well as their prioritization, based on the stack trace.

3.4 Discussion

3.4.1 Practical implication on the cost

Generating seeds comes with a cost. For our worst case, XWIKI-13916, we collected 286K call sequences from static and dynamic analysis and generated 7,880 models from which we selected 6K abstract test cases. We repeated this process 10 times and found the average time for call sequence collection to be 14.2 seconds; model inference took 77.8 seconds; and abstract test case selection and concretization took 51.5 seconds. We do note however that the model inference is a one-time process that could be done offline (in a continuous integration environment). After the initial inference of models, any search process can utilize model seeding. To summarize, the total initial overhead is ~ 2.5 minutes, and the total nominal overhead is around ~ 1.25 minute.

We argue that **the overhead of model seeding is affordable giving its increased effectiveness.** The initial model inference can also be incremental, to avoid complete regeneration for each update of the code, or limited to subparts of the application (like in our evaluation where we only applied static and dynamic analysis for classes involved in the stack trace). Similarly, abstract test case selection and concretization may be prioritized to use only a subset of the classes and their related model. In our current work, this prioritization is based on the content of the stack traces. Other prioritization heuristics, based for instance on the size of the model (reflecting the complexity of the behavior), is part of our future work.

3.4.2 Applicability and effectiveness

In general, test seeding is less effective than model seeding and could lead to worse results than no seeding when poorly configured. We observe that **test seeding is very sensitive to parameter setting**, which requires a good knowledge of the application and existing tests. Also, its performance depends on the existence and quality of the existing test cases. We did not observe this behavior with model seeding as settings have a minimal influence on the results. Thus, **model seeding can be more appealing for maintainers with less knowledge**. Since model seeding also exploits test cases, thereby subsuming test seeding regarding the observed behavior of the application that is reused during the search, greater performance can be attributed to the analysis of the source code translated in the model.

In our experiments, model seeding also reproduced frames (frame 6 in XWIKI-12482, frames 7 and 8 in XWIKI-13546, frame 4 in XWIKI-13372, frame 4 in XWIKI-13916, and frame 2 in MATH-79b) that neither test seeding nor no seeding strategies could reproduce. However, it did not occur the majority of times, and those frames were therefore not considered as reproduced in our results. Additionally, **only model seeding could reproduce stack traces with more than seven frames** (*e.g.*, XWIKI-13546 and LANG-9b).

We, therefore, recommend to use behavioral model seeding as the seeding overhead is compensated by quantitative (more reproduced crashes and fewer failures) and qualitative (*e.g.*, by using relevant objects in the test cases) improvements.

3.5 Threats to validity

Regarding *internal validity*, we selected 45 crashes from different open source projects. Those crashes have previously been studied [25], or come from the issue tracker of an industrial application, ensuring

diversity in the considered subjects.

Since we focused on the effect of seeding during guided initialization, we fixed the $Pr[pick\ mut]$ value (which, due to the current implantation, is also used as $Pr[pick\ init]$ value in test seeding) to 0.3, the default value used in EvoSuite for unit test generation. The effect of this value for crash reproduction, as well as the usage of test and model seeding in guided initialization, is part of our future work.

Finally, each frame has been run 10 times for each seeding configuration to take randomness into account and we derive our conclusion, based on standard statistical tests [1, 35].

For *external validity*, we cannot guarantee that our results are generalizable to all crashes. However, we recall the diversity of the applications and associated crashes. Variations in the performance of the approaches also suggest mitigation of this threat.

3.6 Future work

Currently, the behavioral models are generated from the source code and the existing tests. In our future work, we will consider other sources of information, like logs of the running environment, to collect relevant call sequences and additional information about the actual usage of the application. That additional information would enable using full-fledged *behavioral usage models* (*i.e.,* a transition system with probabilities on the transition reflecting the usage of the application) to select and *prioritize* abstract test cases according to that usage.

In this study, we focused on the impact of seeding during guided initialization by using different values for $Pr[pick\ init]$ and $Pr[clone]$ and setting $Pr[pick\ mut]$ to the default value (0.3). However, our results show that even with the default value 0.3, using seeded objects during the search process helps to reproduce several crashes. Our future work includes a thorough assessment of that factor.

Furthermore, in the current version of model seeding, the abstract test case selection from the model is an independent task from the search process. We will study the integration of those two processes to guide the seeding (*e.g.*, the abstract test case selection) using the current status of the search process.

Finally, we believe that this seeding strategy may be useful for other search-based software testing applications and will implement and assess it for other classical coverage criteria.

3.7 Summary

Manual crash reproduction is labor-intensive for developers. A promising approach to alleviate them from this challenging activity is to automate crash reproduction using search-based techniques. In this chapter, we augment this technique using both test and behavioral model seeding. We implement both test seeding and the novel model seeding.

For practitioners, the implication is that more crashes can be automatically reproduced, with a small cost. In particular, our results show that behavioral model seeding outperforms test seeding and no seeding without a major impact on efficiency. The different behavioral model seeding configurations reproduce 13% more cases compared to no seeding, while test seeding improves only by 1%. Behavioral model seeding could also start the search process for 14% more cases in the evaluation compared to no seeding, while test seeding could only achieve 8% more than no seeding.

From the research perspective, by abstracting behavior through models and taking advantage of the advances made by the model-based testing community, we can enhance search-based crash reproduction. Our analysis reveals that (1) using collected call sequences, together with (2) the dissimilar selection (3) and prioritization of abstract test cases, as well as (4) the combined information from source code and test execution, enable more search processes to get started, and ultimately more crashes to be reproduced.

In our future work, we will explore whether behavioral model seeding has further ranging implications for the broader area if search-based software testing. Furthermore, we aim to study the effect of changing the seeding probabilities on the search process, explore other sources of data to generate the model and try different abstract test case selection strategies.

Chapter 4

The Botsing framework

Botsing is a framework for Java crash reproduction developed in the context of the STAMP project. It includes a fresh implementation of the search-based crash reproduction approach described in Chapter 1. Botsing largely extends the legacy search-based crash reproduction tool EvoCrash by providing additional tools to help the developers to preprocess stack traces and include Botsing in a continuous integration (CI) lifecycle or an integrated development environment (IDE). Contrarily to EvoCrash (which is a fork of EvoSuite), Botsing relies on EvoSuite as an external dependency. It enables STAMP to develop Botsing independent from EvoSuite (i.e., we are not impacted by the EvoSuite release cycle), and release Botsing under *Apache-2.0*, a business-friendly license (EvoSuite being released under a contaminating *LGPL* license).

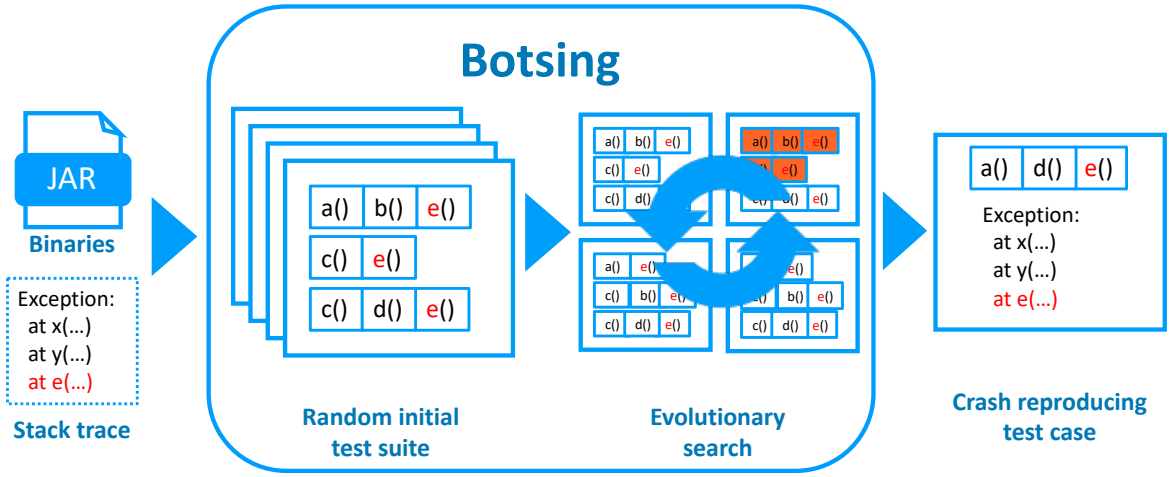

Figure 4.1: Botsing overview

Figure 4.1 presents an overview of the crash reproduction engine of Botsing. It takes as input a stack trace and the Java binaries required to generate a crash reproducing test case for that stack trace. The initial population is randomly initialized before starting the evolutionary search. If the crash could be reproduced, the engine outputs the test case able to do so.

4.1 Developer documentation

4.1.1 Maven modules

Botsing is designed as a framework for crash reproduction. The Maven projects is organized as follows:

- `botsing`, the parent module that contains only a `pom.xml` file with the configuration common to all the modules:
  - `botsing-reproduction`, the reproduction engine.
  - `botsing-preprocessing`, a tool to preprocess and clean stack traces before calling the reproduction engine (see Chapter **??**).
  - `botsing-maven`, the Botsing Maven plugin (see WP4).
  - `botsing-examples`, code examples that could be reused to try Botsing and that are also used for Botsing integration tests.

4.1.2 Architecture

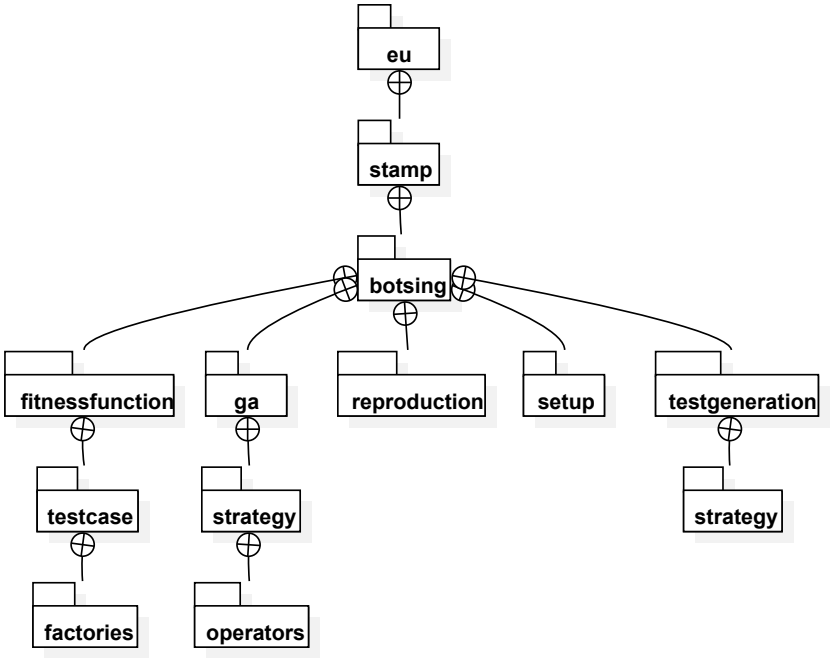

Figure 4.2: Package hierarchy of `botsing-reproduction`

Figure 4.2 presents the packages of the crash reproduction engine (`botsing-reproduction`). The main packages are the `eu.stamp.botsing.ga` and `eu.stamp.botsing.fitness-function` packages. They contain the classes used to execute the guided genetic algorithm and compute the fitness function, both presented in Chapter 1.

Figure 4.3 details the classes implementing the guided genetic algorithm. A `GeneticAlgorithm` object uses a `TestFitnessFunction` to drive the test case generation process. In Botsing, the `GuidedGeneticAlgorithm` uses a `WeightedSum` fitness function and `GuidedSinglePointCrossover` and `GuidedMutation` operators to perform the crash reproduction search.

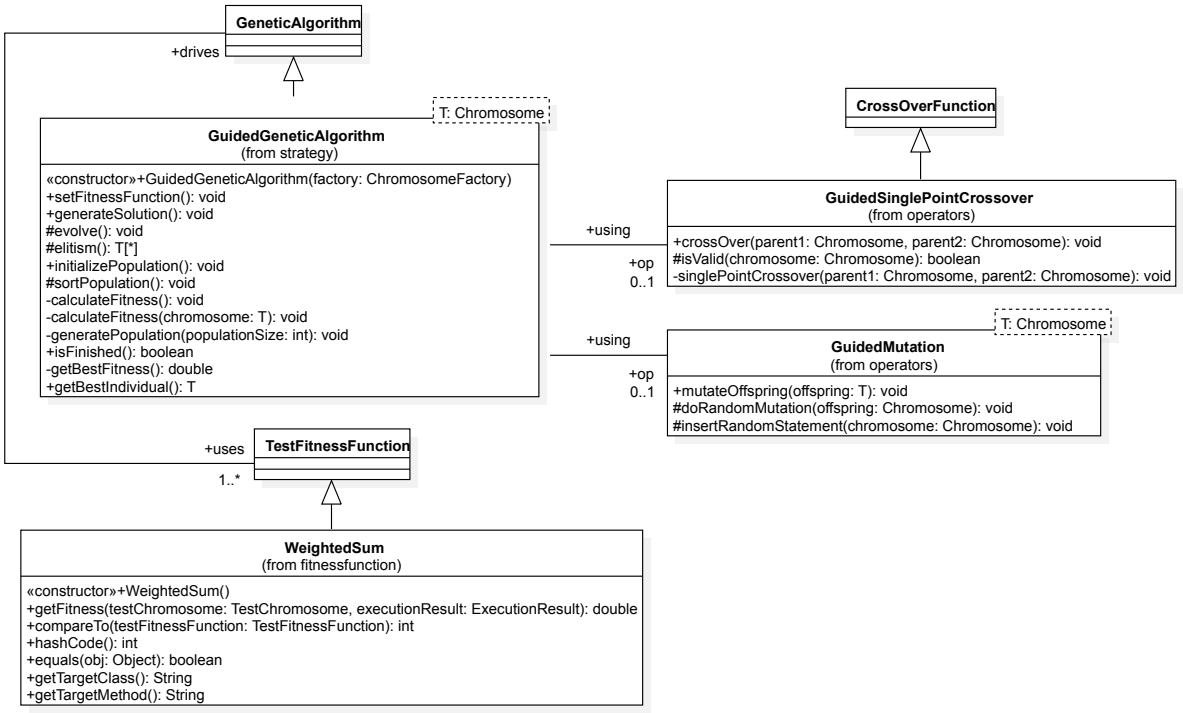

Figure 4.3: Core classes of `botsing-reproduction`

The guided genetic algorithm of Figure 1.1 is implemented in the `GuidedGeneticAlgorithm` class by the following method:

```
@Override
public void generateSolution() {
  currentIteration = 0;

  // generate initial population
  initializePopulation();

  LOG.debug("Starting evolution");
  int starvationCounter = 0;
  double bestFitness = getBestFitness();
  double lastBestFitness = bestFitness;

  LOG.debug("Best fitness in the initial population is: {}", bestFitness);
  while (!isFinished()){
    // Create next generation
    evolve();
    sortPopulation();

    bestFitness = getBestFitness();
    LOG.debug("New fitness is: {}", bestFitness);

    // Check for starvation
    if (Double.compare(bestFitness, lastBestFitness) == 0) {
      starvationCounter++;
```

```
25      } else {
          LOG.debug("Reset starvationCounter after {} iterations", starvationCounter);
27        starvationCounter = 0;
          lastBestFitness = bestFitness;
29      }
        updateSecondaryCriterion(starvationCounter);
31
        LOG.debug("Current iteration: {}", currentIteration);
33      this.notifyIteration();
      }
35    LOG.debug("Best fitness in the final population is: {}", lastBestFitness);
  }
```

The population is initialized at line 6. The algorithm then loops until the fitness is 0.0 or until the budget is exhausted. The next generation is created at lines 16 and 17. The algorithm has an additional check to prevent starvation at line 23, before starting the next iteration.

4.1.3 Coding style

The coding style is described in a `checkstyle.xml` file available at the root of the project. The command `mvn checkstyle:check` must succeed for any pull request to be accepted.

4.1.4 Adding dependencies

The Botsing reproduction engine (`botsing-reproduction`) relies on EvoSuite to instrument the Java code during the evolutionary search. EvoSuite and other dependencies are managed at the module level. Each module declares a list of Maven dependencies, if you want to add one, simply add it to the list. However, dependency version must be declared as a property in the parent `pom.xml` file using the following syntax:

```
<properties>
  ...
  <!-- Dependencies versions -->
  <depdencendy-artifactId.version>1.1.1</depdencendy-artifactId.version>
  ...
</properties>
```

And referenced in the dependencies of the module using the following syntax:

```
<dependencies>
  <dependency>
    <groupId>com.groupId</groupId>
    <artifactId>depdencendy-artifactId</artifactId>
    <version>${depdencendy-artifactId.version}</version>
  </dependency>
</dependencies>
```

Please check in the list of properties that the dependency version is not already there before adding a new one.

4.1.5 License

Botsing is available under a business friendly license: *Apache-2.0*.

4.2 User documentation

We provide hereafter the user documentation to use the Botsing crash reproduction engine with the command line interface, the Maven plugin, and the Eclipse IDE plugin.

4.2.1 Command line interface

The latest version of Botsing crash reproduction engine with a command line interface (`botsing-reproduction-X-X-X.jar`) is available at https://github.com/STAMP-project/botsing/releases. The interface has three mandatory parameters:

- `-crash_log` the file with the stack trace. The stack trace should be clean (no error message) and cannot contain any nested exceptions.
- `-target_frame` the target frame to reproduce. This number should be between 1 and the number of frames in the stack trace.
- `-projectCP` the classpath of the project and all its dependencies. The classpath can be a folder containing all the .jar files required to run the software under test.

Additional parameters can be set. By default, the engine uses the following parameter values:

- `-Dsearch_budget=1800`, a time budget of 30 min. This value can be modified by specifying an additional parameter in format `-Dsearch_budget=60` (here, for 60 seconds).
- `-Dpopulation=100`, a default population with 100 individuals. This value may be modified using `-Dpopulation=10` (here, for 10 individuals).
- `-Dtest_dir=crash-reproduction-tests`, the output directory where the tests will be created (if any test is generated). This value may be modified using `-Dtest_dir=new-outputdir`.

To check the list of options, use `java -jar botsing-reproduction.jar -help`:

```
$ java -jar botsing-reproduction.jar -help
usage: java -jar botsing-reproduction.jar -crash_log stacktrace.log -target_frame
    2 -projectCP dep1.jar;dep2.jar
 -crash_log <arg>       File with the stack trace
 -D <property=value>    use value for given property
 -help                  Prints this help message.
 -projectCP <arg>       classpath of the project under test and all its
                        dependencies
 -target_frame <arg>    Level of the target frame
```

Example

```
java -jar botsing-reproduction.jar -crash_log LANG-1b.log -target_frame 2 -
    projectCP ~/bin
```

4.3 Development status

4.3.1 Fitness function refinement

For now, the Botsing reproduction engine uses the weighted sum fitness function. The multi-objectivization search described in Chapter 2 is part of our future development. Given its poor results, the simple sum scalarization fitness function will not be implemented in the Botsing reproduction engine.

4.3.2 Behavioral model seeding

Behavioral model seeding as described in Chapter 3 has been partially implemented. For now, it includes:

- the generation of behavioral models to use as seed from the static and dynamic analysis of the source code and the test cases.

See pull request https://github.com/STAMP-project/botsing/pull/25.

Future developments

Future developments include:

- the seeding of existing tests;
- the seeding of the behavioral models in the Botsing reproduction engine;
- the generation of models from runtime logs.

4.3.3 Stack trace preprocessing

Preprocessing of the stack traces as described in D3.2 has been partially implemented. For now, it includes:

- the conversion of chained stack traces to a single one, that can be processed by the Botsing reproduction engine;
- the removal of the error messages from the stack trace.

See pull request https://github.com/STAMP-project/botsing/pull/12.

Future developments

Future developments include:

- the detection of irrelevant frames;
- the detection of frames pointing to `try`/`catch` blocks.

Conclusion

Manual crash reproduction is labor-intensive for developers. A promising approach to alleviate them from this challenging activity is to automate crash reproduction using search-based techniques. In this deliverable, we augment this technique by refining the guidance of the search using multi-objectivization; developing and using both test and behavioral model seeding for search-based crash reproduction; and filtering out irrelevant information and provide guidance for debugging by preprocessing the stack traces.

For practitioners, the implication is that more crashes can be automatically reproduced, with a small cost compared to the state-of-the-art. We devise Botsing, a search-based crash reproduction framework aimed at including the different advances made within the STAMP project.

From the research perspective, by abstracting common behavior through models and taking advantage of the advances made by the model-based testing community, we can enhance search-based crash reproduction.

In Task 3.4, we will explore whether behavioral model seeding has further ranging implications for the broader area of search-based software testing.

Bibliography

[1] 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.

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

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[4] 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.

[5] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2011.

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

[7] C. A. Coello Coello. Constraint-handling techniques used with evolutionary algorithms. In *Proc. of the Genetic and Evolutionary Computation Conference Companion (GECCO Companion)*, pages 563–587. ACM, 2016.

[8] W. J. Conover and R. L. Iman. Rank transformations as a bridge between parametric and nonparametric statistics. *The American Statistician*, 35(3):124–129, 1981.

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

[10] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *TEVC*, 6(2):182–197, 2002.

[11] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. Statistical prioritization for software product line testing: an experience report. *Software & Systems Modeling*, 16(1):153–171, feb 2017.

[12] W. Dulz and Fenhua Zhen. MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In *Third International Conference on Quality Software, 2003. Proceedings.*, pages 336–342. IEEE, 2003.

[13] R. A. Fisher. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*, 7(2):179–188, 1936.

[14] G. Fraser and A. Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130. IEEE, apr 2012.

[15] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb 2013.

[16] G. Fraser and A. Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology*, 24(2):1–42, dec 2014.

[17] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, pages 80–89, 2011.

[18] S. García, D. Molina, M. Lozano, and F. Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the cec'2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617, May 2008.

[19] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–42, feb 2013.

[20] S. Herbold, P. Harms, and J. Grabowski. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. *International Journal on Software Tools for Technology Transfer*, 19(3):309–324, jun 2017.

[21] S. Holm. A simple sequentially rejective multiple test procedure. *Scandinavian Journal of Statistics*, 6:65–70, 1979.

[22] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[23] M. Jähne, X. Li, and J. Branke. Evolutionary algorithms and multi-objectivization for the travelling salesman problem. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 595–602. ACM, 2009.

[24] S. Jan, A. Panichella, A. Arcuri, and L. Briand. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering*, (i):1–27, 2017.

[25] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 474–484. IEEE, jun 2012.

[26] R. Just, D. Jalali, and M. D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proc. of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 437–440. ACM, 2014.

[27] 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.

[28] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA, 2010. ACM.

[29] M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. The Statechart Workbench: Enabling scalable software event log analysis using process mining. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–506. IEEE, mar 2018.

[30] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, jun 2004.

[31] P. McMinn, M. Shahbaz, and M. Stevenson. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*, pages 141–150. IEEE, apr 2012.

[32] D. Mondal, H. Hemmati, and S. Durocher. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '15, pages 1–10. IEEE, apr 2015.

[33] 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.

[34] 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.

[35] A. Panichella and U. R. Molina. Java unit testing tool competition - Fifth round. *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017*, pages 32–38, 2017.

[36] S. Prowell and J. Poore. Computing system reliability using Markov chain usage models. *Journal of Systems and Software*, 73(2):219–225, oct 2004.

[37] 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.

[38] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 114–123. IEEE, 2013.

[39] 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 '18)*. Springer, 2018.

[40] M. Soltani, A. Panichella, and A. van Deursen. Evolutionary testing for crash reproduction. In *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*, pages 1–4, 2016.

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

[42] M. Soltani, A. Panichella, and A. van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. *Software Engineering, IEEE Transactions on*, 2018.

[43] S. Sprenkle, L. Pollock, and L. Simko. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 230–239. IEEE, mar 2011.

[44] S. E. Sprenkle, L. L. Pollock, and L. M. Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *Software Testing, Verification and Reliability*, 23(6):439–464, 2013.

[45] L. D. Toffola, C.-A. Staicu, and M. Pradel. Saying 'Hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49. IEEE, oct 2017.

[46] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 562–572. ACM Press, 2014.

[47] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.

[48] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[49] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[50] S. Verwer and C. A. Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In L. O'Conner, editor, *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642. IEEE, sep 2017.

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

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