Delft University of Technology

Good Things Come In Threes
Improving Search-based Crash Reproduction With Helper Objectives

Derakhshanfar, Pouria; Devroey, Xavier; Zaidman, Andy; van Deursen, Arie; Panichella, Annibale

DOI
10.1145/3324884.3416643

Publication date
2020

Document Version
Final published version

Published in
Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020

Citation (APA)
Derakhshanfar, P., Devroey, X., Zaidman, A., van Deursen, A., & Panichella, A. (2020). Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives. In J. Grundy, D. Lo, & C. Le Goues (Eds.), *Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020: Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering* (pp. 211-223). Article 9285999 (Proceedings - 2020 35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020). Association for Computing Machinery (ACM). https://doi.org/10.1145/3324884.3416643

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Copyright
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

Takedown policy
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

This work is downloaded from Delft University of Technology.
For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.

Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives

Pouria Derakhshanfar
p.derakhshanfar@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Xavier Devroey
x.d.m.devroey@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Andy Zaidman
a.e.zaidman@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Arie van Deursen
arie.vandeursen@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Annibale Panichella
a.panichella@tudelft.nl
Delft University of Technology
Delft, The Netherlands

ABSTRACT

Writing a test case reproducing a reported software crash is a common practice to identify the root cause of an anomaly in the software under test. However, this task is usually labor-intensive and time-taking. Hence, evolutionary intelligence approaches have been successfully applied to assist developers during debugging by generating a test case reproducing reported crashes. These approaches use a single fitness function called *Crash Distance* to guide the search process toward reproducing a target crash. Despite the reported achievements, these approaches do not always successfully reproduce some crashes due to a lack of test diversity (premature convergence). In this study, we introduce a new approach, called *MO-HO*, that addresses this issue via multi-objectivization. In particular, we introduce two new Helper-Objectives for crash reproduction, namely *test length* (to minimize) and *method sequence diversity* (to maximize), in addition to *Crash Distance*. We assessed *MO-HO* using five multi-objective evolutionary algorithms (NSGA-II, SPEA2, PESA-II, MOEA/D, FEMO) on 124 non-trivial crashes stemming from open-source projects. Our results indicate that SPEA2 is the best-performing multi-objective algorithm for *MO-HO*. We evaluated this best-performing algorithm for *MO-HO* against the state-of-the-art: single-objective approach (Single-Objective Search) and decomposition-based multi-objectivization approach (*De-MO*). Our results show that *MO-HO* reproduces five crashes that cannot be reproduced by the current state-of-the-art. Besides, *MO-HO* improves the effectiveness (+10% and +8% in reproduction ratio) and the efficiency in 34.6% and 36% of crashes (i.e., significantly lower running time) compared to Single-Objective Search and *De-MO*, respectively. For some crashes, the improvements are very large, being up to +93.3% for reproduction ratio and -92% for the required running time.

This work is licensed under a Creative Commons Attribution International 4.0 License.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-6768-4/20/09.
https://doi.org/10.1145/3324884.3416643

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Search-based software engineering**.

KEYWORDS

crash reproduction, search-based software testing, multi-objective evolutionary algorithms

ACM Reference Format:
Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. 2020. Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives. In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416643

1 INTRODUCTION

When a software application crashes, a report (or issue), including information gathered during the crash, is assigned to developers for debugging [43]. One common practice to identify the root cause of a crash is to provide a test case that reproduces it [45]. This test case can later be adapted and integrated into the test suite to prevent future regressions. However, this test case is not always available in the crash reports. Also, depending on the amount of information available in the report, writing this *crash reproducing test case* can be time-consuming and labor-intensive [39].

Consequently, various approaches have been proposed in the literature to automate *crash reproduction* [4, 6, 23, 31, 32, 36, 39, 44]. These approaches use the information about a crash (*e.g.,* stack traces from crash reports) to generate a crash reproducing test case by utilizing different techniques such as symbolic execution, model checking, *etc.* Among these approaches, two evolutionary-based techniques have been introduced: ReCore [36] and EvoCrash [39]. These two approaches generate test cases able, when executed, to reproduce the target crash using single-objective evolutionary algorithms. The empirical evaluation of EvoCrash [39] shows that it outperforms other, evolutionary-based and non-evolutionary-based approaches in terms of *crash reproduction ratio* (percentage of crashes that could be reproduced) and *efficiency* (time taken to reproduce a given crash successfully). This evaluation also confirms that EvoCrash significantly helps developers during debugging.

EvoCrash relies on a single-objective evolutionary algorithm (*Single-Objective Search* hereafter) that evolves test cases according

to an objective (*Crash Distance* hereafter) measuring how far a generated test is from reproducing the crash. *Crash Distance* combines three heuristics: *line coverage* (how far is the test from executing the line causing the crash?), *exception coverage* (does the test throw the same exception as in the crash?), and *stack trace similarity* (how similar is the exception stack trace from the one reported in the crash?). Although Single-Objective Search performs well compared to the other crash reproduction approaches, a more extensive empirical study [37] evidenced that it is not successful in reproducing complex crashes (*i.e.,* large stack traces). Hence, further studies to enhance the guidance of the search process are required.

Just like any other evolutionary-based algorithm, Single-Objective Search requires to maintain a balance between *exploration* and *exploitation* [42]. The former refers to the generation of completely new solutions (*i.e.,* test cases executing new paths in the code); the latter refers to the generation of solutions in the neighborhood of the existing ones (*i.e.,* test cases with similar execution paths). Single-Objective Search ensures exploitation through Guided Mutation, which guarantees that each solution contains the method call causing the crash (and reported in the stack trace) [39]. However, the low exploration of Single-Objective Search may lead to a lack of diversity, trapping the search in local optima [42].

To tackle this problem, a prior study [38] investigated the usage of *Decomposition-based Multi-Objectivization* (*De-MO*) to decompose the *Crash Distance* in three distinct (sub-)objectives. A target crash is reproduced when the search process fullfils all three sub-objectives at the same time. The empirical evaluation shows that *De-MO* slightly improves the efficiency for some crashes. However, since the sub-objectives are not conflicting, their combined usage can be detrimental for crash reproduction [38]. A recent study [13] also conjectured that increasing diversity via additional objective is a feasible yet unexplored research direction to follow. However, no systematic empirical study has been conducted to draw statistical conclusions.

In this study, we investigate a new strategy to Multi-Objectivize crash reproduction based on Helper-Objectives (*MO-HO*) [13] rather than decomposition. More specifically, we add two additional helper-objectives to *Crash Distance* (first objective): *method sequence diversity* (second objective) and *test case length minimization* (third objective). The second objective aims to increase the diversity in the method sequences; more diverse sequences are more likely to cover diverse paths and, consequently, improve exploration. The third objective aims to address the *bloating effect* (*i.e.,* the generated test cases can become longer and longer after each generation until the all of the system memory is used), as diversity can lead to an unnecessary and counter-productive increase of the test case length [1, 33]. Since these three objectives are *conflicting*, we expect an improvement in the solutions' diversity and, hence, improving the effectiveness (crash reproduction ratio) and efficiency.

To assess the performance of *MO-HO* on crash reproduction, we use five multi-objective evolutionary algorithms (MOEAs): NSGA-II [10], SPEA2 [47], MOEA/D [46], PESA-II [8], and FEMO [25]. We apply them to 124 non-trivial crashes from JCrashPack [37], a crash benchmark used by previous crash reproduction studies [12]. Those crashes can only be reproduced by a test case that brings the software under test to a specific state and invokes the target method with one or more specific input parameters. We performed

```
0    java.lang.ArrayIndexOutOfBoundsException: 4
1      at [...].FastDateParser.toArray(FastDateParser.java:413)
2      at [...].FastDateParser.getDisplayNames([...]:381)
3      at [...].FastDateParser$TextStrategy.addRegex([...]:664)
4      at [...].FastDateParser.init([...]:138)
5      at [...].FastDateParser.<init>([...]:108)
6      [...]
```

Figure 1: LANG-9b crash stack trace [24, 37]

an internal assessment among *MO-HO* algorithms to find the best multi-objective evolutionary algorithm for this optimization problem. According to the results observed in this assessment, *SPEA2* outperforms other MOEAs in crash reproduction using *MO-HO* helper-objectives.

Furthermore, we compared the best-performing *MO-HO* (*MO-HO* + *SPEA2*) against two state-of-the-art approaches (Single-Objective Search [39] and *De-MO* [38]) from the perspectives of *crash reproduction ratio* and *efficiency*. Our results show that *MO-HO* outperforms the state-of-the-art in terms of crash reproduction ratio and efficiency. This algorithm improves the crash reproduction ratio by up to 100% and 93.3% (10% and 8%, on average) compared to Single-Objective Search and *De-MO*, respectively. Also, after five minutes of search, *MO-HO* reproduces five and six crashes (4% and 5% more crashes) that cannot be reproduced by Single-Objective Search and *De-MO*, respectively. In addition, *MO-HO* reproduces crashes significantly faster than Single-Objective Search and *De-MO* in 34.6% and 37.9% of the crashes, respectively.

A replication package, enabling the full-replication of our evaluation and data analysis of our results is available on Zenodo [14].

2 BACKGROUND AND RELATED WORK

Several approaches have been introduced in the literature that aim to reproduce a given crash. Some of these techniques (*e.g.,* ReCore [36]) use runtime data (*i.e.,* core dumps). However, collecting the runtime data may induce a significant overhead and raises privacy concerns. In contrast, other approaches [4, 6, 32, 44] only require the *stack traces* of the unhandled exception causing the crash, collected from executions logs or reported issues. For Java programs, a stack trace includes the list of classes, methods, and code line numbers involved in the crash. As an example, Figure 1 shows a stack trace produced by a crash (due to a bug) in Apache Commons Lang. This stack trace contains the *type of the exception* (`ArrayIndexOutOfBoundsException`) and *frames* (lines 1-6) indicating the stack of active method calls during the crash.

Among the various approaches solely using a stack trace as input, STAR [6] and BugRedux [23] use backward and forward symbolic execution, respectively; MuCrash [44] mutates the existing test cases of the classes involved in the stack trace; JCharming [31, 32] applies model checking and program slicing for crash reproduction; and ConCrash [4] is designed to use pruning strategies to reproduce the crash-reproducing test case.

EvoCrash is an evolutionary-based approach that applies a Single-Objective Genetic Algorithm (Single-Objective Search) to generate a crash-reproducing test case for a given stack trace and a *target frame* (*i.e.,* the class under test for which the test case is generated). The generated test will trigger a crash with a stack

trace that is identical to the original one, up to the target frame. For instance, for the stack trace in Figure 1 with a target frame at line 3, EvoCrash generates a test case that reproduces the first three frames of this stack trace (*i.e.,* identical from lines 0 to 3). A previous empirical evaluation [39] shows that EvoCrash performs better compared to other crash reproduction approaches relying on model checking and program slicing [31, 32], backward symbolic execution [6], or exploiting existing test cases [44]. The study also confirms that automatically generated crash-reproducing test cases help developers to reduce their debugging effort.

2.1 Single-Objective Search Heuristics

To evaluate the candidate tests, and consequently guide the search process, Single-Objective Search applies a fitness function called the *Crash Distance*. This fitness function contains three components: (i) the **line coverage distance**, indicating the distance between the execution trace and the *target line* (the line number pointed to by the target frame), (ii) the **exception type coverage**, indicating whether the *target exception* is thrown, and (iii) the **stack trace similarity**, indicating whether all frames (from the beginning up to the target frame) are included in the triggered stack trace.

Definition 2.1 (*Crash Distance* [39])**.** For a given test case execution t, the *Crash Distance* (f) is defined as follows:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_e) + max(d_{tr}) & \textit{if line not reached} \\ 3 \times min(d_s) + 2 \times d_e(t) + max(d_{tr}) & \textit{if line reached} \\ 3 \times min(d_s) + 2 \times min(d_e) + d_{tr}(t) & \textit{if exception thrown} \end{cases} \quad (1)$$

Where $d_s(t) \in [0, 1]$ indicates how far the test t is from reaching the *target line* using two heuristics: *approach level* and *branch distance* [27]. The former measures the minimum number of control dependencies between the execution path of t and the target line; the latter indicates how far t is from satisfying the branch condition on which the target line is control dependent. And $d_e(t) \in \{0, 1\}$ indicates whether an exception with the same type as the target exception is thrown (0) or not (1). Finally, $d_{tr}(t) \in [0, 1]$ calculates the similarity between the stack trace produced by t and the expected one, based on classes, methods, and line numbers appearing in both stack traces. Functions $max(.)$ and $min(.)$ denote the maximum and minimum possible values for a function, respectively. Concretely, $d_e(t)$ and $d_{tr}(t)$ are only calculated upon the satisfaction of two *constraints*: *exception type coverage* and *stack trace similarity* are relevant only when we reach the target line (first constraint) and when we have the same type of exception (second constraint), respectively.

2.2 Single-Objective Search

The search process starts with a **guided initialization** during which an initial population of randomly generated test cases is created. The algorithm ensures that each test case calls the *target method* (pointed to by the target frame) at least once. In each generation, the fittest test cases are evolved by applying **guided mutation** and **guided crossover**. Guided mutation applies a classical mutation to the test cases while ensuring that the mutated test contains one or more calls to the target method. Similarly, guided crossover is a variant of the single-point crossover that preserves calls to the target methods in the offsprings. Accordingly, each generated test case contains at least one call to the target method (*i.e.,* the method triggering the crash) [39].

With those operators, Single-Objective Search improves the *exploitation*, but it penalizes *exploration* of new areas of the search space by not generating diverse enough test cases. As a consequence, the search process may get stuck in local optima.

2.3 Decomposition-based Multi-objectivization

To increase diversity during the search, a prior study [38] investigated the usage of *Decomposition-based Multi-Objectivization* (called *De-MO* hereafter) to decompose the *Crash Distance* in three distinct (sub-)objectives. *De-MO* on the *Crash Distance* (temporarily) decomposes the function in three distinct (sub-)objectives: $d_s(t)$, $d_e(t)$, and $d_{tr}(t)$. Then, *De-MO* uses a multi-objective evolutionary algorithm optimizing three objectives to generate one crash-reproducing solution. In the end, the global optimal solution is a test case in the Pareto front produced by MOEAs that satisfies all of the sub-objectives simultaneously. The empirical evaluation shows that *De-MO* increases the efficiency of the crash reproduction process for some specific cases compared to Single-Objective Search. However, it loses efficiency in some other cases.

In particular, in *Multi-objectivization*, search objectives should be conflicting to increase the diversity of generated solutions [22]. However, the three sub-objectives in *De-MO* [38] are tightly coupled and not conflicting: the stack trace similarity ($d_{tr}(t)$) cannot be computed for test case t without executing the target line ($d_s(t) = 0$) and throwing the correct type of exception ($d_e(t) = 0$). Also, the type of exception ($d_e(t)$) is not relevant, while test t does not cover the statement in the target line ($d_s(t) = 0.0$).

3 MULTI-OBJECTIVIZATION WITH HELPER-OBJECTIVES (MO-HO)

Decomposing the *Crash Distance* leads to a set of dependent sub-objectives, which reduces the effect of improving diversity through multi-objectivization [22]. In this study, we focus on using new helper-objectives in addition to the *Crash Distance*, rather than decomposing it. We define two helper-objectives called **method sequence diversity** and **test length minimization** that aim to (i) increase diversity in the population (*i.e.,* generated tests) and (ii) address the *bloating* effect [30, 33]. Then, we use five different evolutionary algorithms belonging to different categories of MOEAs (*e.g.,* decomposition-based and rank-based) to solve this optimization problem. In the remainder of this section, we first discuss the two helper-objectives. Next, we present the MOEAs used to solve this problem.

3.1 Helper-Objectives

As suggested by Jensen *et al.* [22], adding helper-objectives to an existing single objective can help search algorithms escape from local optima. However, this requires that the helper objectives are in conflict with the primary one [22]. Therefore, defining proper helper-objectives is crucial.

Method Sequence Diversity. The first helper-objective seeks to maximize the diversity of the method-call sequences that compose the generated tests because more diverse tests might execute

different paths or behaviors of the *target class*. Notice that each test case is a sequence of statements, where each statement belongs to one of the following five different categories [33]: *primitive* statements, *constructors*, *field statements*, *method calls*, or *assignments*. Furthermore, the length of a test case is variable, *i.e.,* it is not fixed a priori and can vary during the search.

In recent years, several functions have been introduced to measure test case diversity [30]. These functions measure the diversity between two test cases by using a binary encoding function to calculate the distance between the corresponding encoded vectors using the Levenshtein distance [26], Hamming distance [19], *etc.* For three or more test cases, the overall diversity corresponds to the average pairwise diversity of the existing test cases [30]. These metrics have been used in other testing tasks (*e.g.,* automated test selection), but not in crash reproduction.

To measure the value of this helper-objective for the generated solutions, we follow a similar procedure. Let us assume that $F = \{f_1, f_2, ..f_n\}$ is a set of public and protected methods in the target class (*i.e.,* method calls that can be called directly by the generated tests), and $T = \{t_1, t_2, ..t_m\}$ is a set of generated test cases. To calculate the diversity of T, we first need to encode each $t_k \in T$ into a binary vector. We use the same encoding function proposed by Mondal *et al.* [30]: each test case $t_k \in T$ corresponds to a binary vector v_k of length n (*i.e.,* the number of public and protected methods in the target class). Each element $v_k[i]$ of the binary vector denotes whether the corresponding method $f_i \in F$ is invoked by the test case t_k. More formally, for each method $f_i \in F$, the corresponding entry $v_k[i] = 1$ if t_k calls f_i; $v_k[i] = 0$ otherwise.

Then, we calculate the diversity for each pair of test cases t_k and t_i as the Hamming distance between the corresponding binary vectors v_k and v_i [19]. The Hamming distance (*Hamming*) between two vectors corresponds to the number of mismatches[1] over the total length of the binary vectors. For instance, the Hamming distance between $A = \langle 1, 1, 0, 1, 0 \rangle$ and $B = \langle 0, 1, 0, 1, 1 \rangle$ equals to $2/5 = 0.4$.

Definition 3.1 (Method Sequence Diversity)**.** *Given an encoding function $V(.)$, the method sequence diversity (MSD) of a test $t \in T$ corresponds to the average Hamming distance of that test from the other test cases in T:*

$$MSD(t) = \frac{\sum_{t_i \in T \setminus \{t\}} Hamming(V(t), V(t_i))}{|T| - 1} \quad (2)$$

In our approach, *MSD* should be maximized to increase the chance of the generated test to execute new paths or behaviors in the *target class*. Since our tool (see Section 4.1) is designed for minimization problems, we minimize the method sequence similarity using the formula:

$$f_{MSD}(t) = 1 - MSD(t) \quad (3)$$

Test Length Minimization While increasing method sequence diversity can help to execute diverse paths of the target class, a previous study [1] also showed that *test diversity metrics* (such as call sequence diversity) can reduce coverage. This is due to the *bloating effect*, *i.e.,* diversity will also promote larger test cases over short ones. Let us assume that we have a set of short test cases with few method calls in our population (most of the elements in their binary vectors are 0). A lengthy test case t_L that calls all the methods of the target class will have a binary vector containing only 1 values. As a consequence, t_L will have a large Hamming distance from the existing test cases.

[1]The number of positions at which the corresponding bits are different.

Larger tests introduce two potential issues: (i) they are likely more expensive to run (extra overhead), and (ii) they may contain spurious statements that do not help code coverage (which is a part of *Crash Distance*). In the latter case, mutation can become less effective as it may mutate spurious statements rather than the relevant part of the chromosomes. Therefore, test diversity is in conflict with *Crash Distance*. To avoid the *bloating* effect, our second helper-objective is *test length minimization*, which counts the number of statements in a given test:

Definition 3.2 (Test Length Minimization)**.** *For a test case t with a length $|t|$, the fitness function is:*

$$f_{len}(t) = |t| \quad (4)$$

3.2 Multi-Objective Evolutionary Algorithms

In this study, our goal is to solve a multi-objectivized problem by minimizing the three objective functions (*Crash Distance*, f_{MSD}, and f_{len}). In theory, we could consider various MOEAs, each coming with different advantages and disadvantages over different optimization problems (*e.g.,* multimodal, convex, *etc.*). However, we cannot establish upfront what type of MOEA works better for crash reproduction as the shape of the Pareto Front (*i.e.,* type of problem) for crash reproduction is unknown. Hence, we chose five MOEAs from different categories to determine the best algorithm for *MO-HO*: *NSGA-II* uses the non-dominated sorting procedure; *SPEA2* is an archive-based algorithm that selects the best solutions according to the fitness value; *PESA-II* divides the objective space to hyper-boxes and selects the solutions from the hyper-boxes with the lower density; *MOEA/D* decomposes the problem to multiple sub-problems; and *FEMO*, is a (1+1) evolutionary algorithm that evolves tests solely with mutation and without crossover.

We use the same stopping conditions for all search algorithms, which is a maximum search budget, or when the target crash is successfully reproduced, *i.e.,* a solution with a *Crash Distance* of 0.0 is found. Also, to increase *exploitation* during the search, all algorithms use the *guided crossover* and *guided mutation* operators.

In the following subsections, we briefly describe the selected search algorithms and their core characteristics.

3.2.1 Non-dominated Sorting Genetic Algorithm II (NSGA-II) [10]. In NSGA-II, offspring tests are generated, from given a population of size N, using genetic operators (crossover and mutation). Next, NSGA-II unions the offspring population with the parent population into a set of size $2N$ and applies a *non-dominated sorting* to select the N individuals for the next generation. This sorting is performed based on the *dominance* relation and *crowding distance*: the solutions are sorted into subsequent dominance fronts. The non-dominated solutions are in the first front ($Front_0$). These solutions have a higher chance of being selected. Furthermore, *crowding distance* is used to raise the chance of the most diverse solutions within the same front to be selected for the next generation. In each generation, parent test cases are selected for reproduction using the *binary tournament selection.*

3.2.2 *Strength Pareto Evolutionary Algorithm 2 (SPEA2) [47].*

Besides the current population, SPEA2 contains an external archive that collects the non-dominated solutions among all of the solutions considered during the search process. SPEA2 assigns a *fitness value* to each solution (test) in the archive. The *fitness value* of solution i is calculated by summing up two values: *Raw fitness* ($R(i) \in \mathbb{N}_0$), which represents the dominance relation of i; and *Strength value* ($S(i) \in [0, 1]$), which estimates the density of solutions in the same Pareto front (solutions that are not dominating each other). A solution with lower fitness value is "better" and has a higher chance of being selected. For instance, the non-dominated solutions have a $R(i) = 0$, and their fitness values are lower than 1.

The external archive has a fixed size, which is given at the beginning of the search process. After updating the archive in each iteration, the algorithm checks if the size of the archive exceeds this given size. If the size of the archive is smaller than the given size, SPEA2 fills the archive with the existing dominated solutions. In contrast, if the size of the archive is bigger than the given size, this algorithm uses a *truncation operator* to remove the solutions with a high *fitness value* from the archive. After updating the archive, SPEA2 applies *binary tournament selection* based on the calculated *fitness values*, selects parent solutions, and generates offspring solutions via *crossover* and *mutation.*

3.2.3 *Pareto Envelop-based Selection Algorithm (PESA-II) [8].*

Similar to SPEA2, PESA-II benefits from an external archive. In each generation, the archive is updated by storing the non-dominated solutions in the archive and the current population. However, the difference is in the selection strategy and archive truncation. In this algorithm, instead of assigning a fitness value to each of the solutions in the archive, the objective space is divided, based on the existing solutions, into *hyper-boxes* or grids. Non-dominated solutions in a hyper-box with lower density have a higher chance of being selected and a lower chance of being removed.

3.2.4 *Multi-objective Evolutionary Algorithm Based on Decomposition (MOEA/D) [46].*

This algorithm decomposes the M-objectives problem into K single-objective sub-problems and optimizes them simultaneously. Each sub-problem has different weights for the optimization objectives. The K sub-problems $g(x|w_1), \ldots, g(x|w_K)$ are obtained using a scalarization function $g(x|w)$ and a set of uniformly-distributed weight vectors $W = \{w_1, \ldots, w_k\}$. The decomposition can be done with several techniques such as *weighted sum* [29], *Tchebycheff* [29], or *Boundary Intersection* [9, 28]. In each generation, MOEA/D maintains the best individuals for each subproblem $g(x|w_i)$, while the reproduction (based on crossover and mutation) is allowed only among solutions (tests) within the same neighbourhood (*mating restriction*).

3.2.5 *Fair Evolutionary Multi-objective Optimizer (FEMO) [25].*

This algorithm is a local (1+1) evolutionary algorithm. It means that in each iteration, only one solution is evolved by the mutation operator to have only one offspring solution for the next generation. FEMO contains an archive. In the first iteration, it generates a random solution and places it in the archive. In the next generations, it selects one individual from the archive and evolves it by mutation operator to generate a new solution. Finally, if the new solution dominates at least one of the solutions in the archive, it adds the new solution to the archive and removes the dominated solutions.

Each solution in the archive has a weight (w) that indicates the number of times that a solution was selected from the archive. So, the initial weight of a newly generated test case is 0. During the selection, FEMO selects a solution randomly from the solutions in the archive that have the lowest w.

4 EMPIRICAL EVALUATION

To assess the impact of *MO-HO* on crash reproduction, we performed an empirical evaluation and answered the following research questions.

RQ_1: *Which Multi-Objective algorithm performs better with MO-HO's search objectives in terms of crash reproduction?*

RQ_2: *What is the impact of the MO-HO algortihm on crash reproduction compared to Single-Objective Search and De-MO?*

RQ_3: *How does MO-HO's efficiency compare to Single-Objective Search and De-MO?*

4.1 Implementation

Since other crash reproduction approaches are not openly available, we implemented a new open-source evolutionary-based crash reproduction framework, called Botsing.[2] Botsing is well-tested and designed to be easily extensible for new techniques (new evolutionary algorithms, new genetic operators, *etc.*). It relies on EvoSuite [17], an evolutionary-based unit test generation tool, for code instrumentation and for the internal representation of an individual (*i.e.,* a test case) by using `evosuite-client` as a dependency.

For this study, we implemented the techniques used in previous studies for crash reproduction (Single-Objective Search and *De-MO*) in Botsing. Moreover, we implemented all of the *MO-HO* approaches, which include the two fitness functions for our new helper-objectives (*method sequence diversity* and *test length*) and the five MOEAs mentioned above.

4.2 Setup

Crash Selection. We selected our crashes from JCrashPack [11, 37], a collection of crashes from open-source projects and created for crash reproduction benchmarking. Based on the reported results of the prior studies about search-based crash reproduction [37, 38], we know that Single-Objective Search and *De-MO* face various challenges to reproduce many of the crashes in this benchmark. For this study, we apply our approach and state-of-the-art algorithms to 124 crashes from JCrashPack, which are used in the recent search-based crash reproduction study [12]. These crashes stem from six open-source projects: JFreeChart, a framework for building interactive charts; Commons-lang, a library providing extra utilities to the `java.lang` API; Commons-math, a library for mathematical and statistical usages; Mockito, a testing framework for mocking objects; Joda-time, a library for date and time manipulation; XWiki, a large-scale enterprise wiki management system.

Algorithm Selection. We attempted to reproduce the selected crashes using seven evolutionary algorithms: Single-Objective Search, *De-MO*, and *MO-HO* with five MOEAs (NSGA-II, SPEA2, PESA-II, MOEA/D, and FEMO). For each crash, we ran each algorithm on

[2] Available at https://github.com/STAMP-project/botsing

each frame of crash stack traces. We repeated each execution 30 times to take randomness into account, for a total number of 199,710 independent executions. We ran the evaluation on servers with 40 CPU-cores, 128 GB memory, and 6 TB hard drive.

Evaluation procedure. In RQ_1, we perform an internal assessment of *MO-HO* by comparing all MOEAs to determine the best-performing one when optimizing the search objectives in *MO-HO*. Then, to answer RQ_2 and RQ_3, we use the best-performing *MO-HO* configuration (MOEA) to evaluate its effectiveness and efficiency against the state-of-the-art crash reproduction approaches.

Parameter Settings. We set the search budget to five minutes, as suggested by previous studies on evolutionary-based crash reproduction [39]. Also, we fixed the population size and archive size (if needed) to 50 individuals, as recommended in prior studies on test case generation [33]. For *MO-HO* with PESA-II, the number of bisections for gridding is set to the default value of five grids. In *MO-HO* with MOEA/D, the weight vectors are obtained using a variant *simplex-lattice design* [40] and using the *Tchebycheff approach* as the aggregation function. Finally, we set the *neighborhood selection probability* to 0.2 (set to the default value [15]) and the maximum number of *solutions that can be replaced* in each generation to 50. For all MOEAs, we use the *guided mutation* with mutation probability $p_m = 1/n$ (n is the length of the test case), and *guided crossover* with crossover probability $p_c = 0.8$ (the same parameters used for the suggested baselines).

4.3 Data Analysis

To evaluate the crash reproduction ratio (*i.e.,* the percentage of successful crash reproduction attempts in 30 rounds of runs) of different algorithms, we follow the same procedure as the previous studies [12, 38]: for each crash C, we find the highest frame that can be reproduced by at least one of the algorithms (r_{max}). We analyze the crash reproduction ratio of each algorithm for a target crash C targeting frame r_{max}.

To check whether the performance (reproduction ratio) of MOEAs significantly differs from one another, we use the Friedman test [18]. The Friedman test is a non-parametric version of the ANOVA test [16], *i.e.,* it does not make any assumption about the data distribution. It is a multiple-problem statistical test and has been widely used in the literature to compare randomized algorithms [21, 34]. Friedman's test allows to rank and statistically compare different MOEAs over multiple independent problems, i.e., crashes in our case. For Friedman's test, we use a level of significance $\alpha = 0.05$. If the p-values obtained from Friedman's test are significant (p-values $<= 0.05$), we apply pairwise multiple comparison using Conover's post-hoc procedure [7]. To correct for multiple comparison errors, we adjust the p-values from Conover's procedure using Holm-Bonferroni [20].

To answer RQ_2, we need to determine whether an algorithm reproduces a crash. Since we repeat each execution 30 times, we use the majority of outcomes for a crash reproduction result. In other words, if an algorithm could reproduce a crash in ≥ 15 runs (*i.e.,* reproduction ratio of $\geq 50\%$), we count that frame as *reproduced.*

To compare the number of reproduced crashes by each algorithm, we used the same procedure used by Almasi *et al.* [2] and Campos *et al.* [5]: we check crash reproduction status and reproduction ratio of the best-performing *MO-HO* algorithm (according to the results of RQ_1), Single-Objective Search, and *De-MO* at five time intervals: 1, 2, 3, 4 and 5 minute.

Table 1: MOEAs ranking (in *MO-HO*) in terms of crash reproduction ratio (Friedman's test) and results of the pairwise comparison (p-value ≤ 0.05)

Rank	MOEA	Rank value	Significantly better than
1	SPEA2	2.63	(2), (3), (4), (5)
2	PESA-II	2.86	(4), (5)
3	NSGA-II	2.90	(4), (5)
4	MOEAD	4.97	(5)
5	FEMO	5.05	

To evaluate the efficiency of the algorithms (RQ_3), we analyze the time spent by the best *MO-HO* algorithm, Single-Objective Search, and *De-MO* for generating a crash reproducing test cases. Since efficiency is only applicable to the reproduced crashes, we compare the efficiency of algorithms on the crashes that are reproduced at least once by one of the algorithms. If, for one execution, an algorithm was not able to reproduce the crash, it means that it consumed the maximum allowed time budget (5 minutes). To assess the effect size of differences between algorithms, we use the Vargha-Delaney $\hat{A}_{12}$ statistic [41]. A value of $\hat{A}_{12} < 0.5$ for a pair of factors (A, B) shows that A reproduced the target crash in a shorter time, while a value of $\hat{A}_{12} > 0.5$ indicates the opposite. Besides, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. To evaluate the significance of effect sizes ($\hat{A}_{12}$), we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for the Type I error.

A replication package of our evaluation is available on Zenodo [14]. It contains the selected crashes, the results and data analysis presented in this paper, as well as the implementation of MOEAs in Botsing and a Docker-based infrastructure to enable the full-replication of our evaluation.

5 RESULTS

This section presents the results of our empirical evaluation and answers, one by one, our research questions.

5.1 Best MOEA for *MO-HO* (RQ1)

Figure 2 presents the crash reproduction ratio of the MOEAs applied to our *MO-HO* framework. For this analysis, we consider the number of times (in percentage) each MOEAs could reproduce a given crash across 30 runs and using a search budget of five minutes. On average (the squares in Figure 2), the best algorithm for *MO-HO* is *SPEA2*, with an average and median of 76% and 100% of successful reproductions, respectively. *SPEA2* is Followed by *PESA-II*, *NSGA-II*, and *MOEAD.* Also, this figure shows that the first quartile of the crash reproduction ratio of *SPEA2* is, at least, about 25% higher than other MOEAs.

According to Friedman's test, the differences in reproduction ratios are statistically significant (p-value ≤ 0.05). This means that some MOEAs are significantly better than others within our *MO-HO* framework. For completeness, Table 1 reports the ranking produced by the Friedman test. To better understand for which pairs

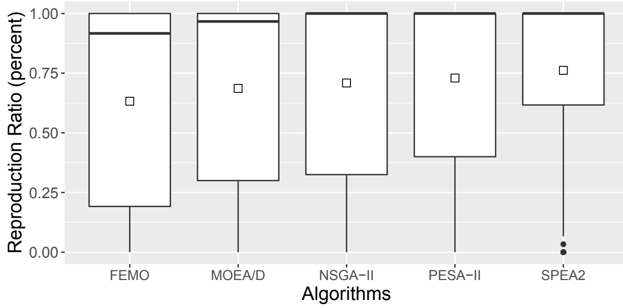

Figure 2: Crash reproduction ratio (out of 30 executions) of *MO-HO* algorithms. The upper and lower edge of each box present the upper and lower quartile, respectively. (□) denotes the arithmetic mean and (—) is the median.

of MOEAS the statistical significance holds, we applied the post-hoc Conover's procedure for the pairwise comparison. The results of the comparison are also reported in Table 1. According to this table, the best-performing algorithm is *MO-HO + SPEA2*, which has a significantly higher crash reproduction ratio compared to other *MO-HO* algorithms. The next algorithms are *MO-HO + PESA-II* and *MO-HO + NSGA-II*. These two algorithms are significantly better than *MO-HO + MOEAD* and *MO-HO + FEMO*. Finally, the worst algorithm in terms of crash reproduction is *FEMO*, which is significantly worse than other MOEAs.

Summary (RQ$_1$). *MO-HO + SPEA2 achieved the highest performance in terms of crash reproduction ratio compared to MO-HO + other MOEAs. The next best-performing MOEAs, in terms of crash reproduction, are PESA-II and NSGA-II.*

5.2 Crash Reproduction (RQ2)

Figure 3 depicts the crash reproduction ratio of the best-performing *MO-HO* configuration (i.e., with *SPEA2*), Single-Objective Search, and *De-MO* at five time intervals (search budgets). As indicated in this figure, the average crash reproduction ratio of *MO-HO* is higher than other algorithms at all of the time intervals. Also, the median crash reproduction ratio for this algorithm is always 100%. Furthermore, the maximum improvement achieved by *MO-HO* with the five-minutes search budget is in *XWIKI-14599* (with 100% improvement) and *MATH-3b* (with 93.3% improvement) compared to Single-Objective Search and *De-MO*, respectively. In contrast, the largest reduction in reproduction ratio by *MO-HO* (with the five-minutes budget) is in *XCOMMONS-1057* (with 30% drop) and *XWIKI-13616* (with 40% reduction) compared to Single-Objective Search and *De-MO*, respectively. We will explain the negative factors in *MO-HO*, which lead to negative results for this algorithm in some corner cases, in Section 5.4.

Moreover, we can see that *De-MO* is the second-best algorithm in all of the time intervals. In the first 60 seconds of the crash reproduction process, on average, its crash reproduction ratio is 4% better than Single-Objective Search. However, in contrast to the other two algorithms, the crash reproduction ratio of this algorithm changes only slightly after the first 120 seconds. Hence, at the end of the search process, the average crash reproduction ratio of *De-MO* is only 2% better than Single-Objective Search. In contrast, since the crash reproduction ratio of *MO-HO* keeps growing, on average, it remains more effective than Single-Objective Search (about 10%) even after 300 seconds. The other interesting point in Figure 3 is the first quantile of *MO-HO*. In the first 60 seconds, this value is lower than 12%, but it grows up to 62% after 300 seconds. This improvement is not observable in state-of-the-art algorithms.

Furthermore, *MO-HO* is more stable in crash reproduction after 300 seconds budget compared to the other algorithms. Figure 3 demonstrates that the interquartile range (i.e., the difference between first and third quartile) of crash reproduction ratio in *MO-HO* with the 300 seconds budget is 46% smaller than the interquartile range of other algorithms (being 38.3% for *MO-HO*, 76.6%. for Single-Objective Search, and 70.8% for *De-MO*).

Also, Figure 4 shows the number of crashes, which are reproduced by *MO-HO*, but not by the state-of-the-art algorithms and vice versa in different time intervals. As indicated in this figure, in all of the time intervals, the number of crashes that are reproduced by *MO-HO* is higher than the crashes that it cannot reproduce. In the best case (after 1 minute of search), *MO-HO* reproduces eight and seven new crashes that cannot be reproduced by Single-Objective Search and *De-MO*, respectively. In contrast, there is only one crash that can be reproduced by *De-MO* and not by *MO-HO*. Also, after five minutes, *MO-HO* still reproduces more crashes than the baselines: it reproduces five and six new crashes that cannot be reproduced by Single-Objective Search and *De-MO*, respectively.

The crashes that are reproduced by *MO-HO* after five minutes but not by Single-Objective Search are: `TIME-10b` frame 5, `XCOMMONS-928` frame 2, `XWIKI-14227` frame 2, `XWIKI-14475` frame 1, and `XWIKI-14599` frame 1. And the crashes that are reproduced by *MO-HO* after five minutes but not by *De-MO* are: `MOCKITO-16b` frame 4, `TIME-5b` frame 3, `XWIKI-13377` frame 3, `XWIKI-14227` frame 2, `MATH-3b` frame 1, and `MOCKITO-10b` frame 1.

Figure 5 shows the crash's stack trace reported in the issue `XWIKI-14227`. *MO-HO* is the only approach that can reproduce the first two frames of this stack trace. Here, the target method is `useMainStore` (Figure 6), which does not have any input argument. Hence, to reproduce this crash, the crash reproducing test generated by *MO-HO* (depicted in Figure 8) should invoke specific methods (*e.g.,* `setWiki`, `setWikiId`) to set different local variables in the `xwikiContext0` object, and then, pass this object to the class under test (here, `ActivityStreamConfiguration`). Since the crash reproducing test case generated by *MO-HO* does not add any plugin to the `xWiki0` object, the execution of this test indeed leads to a `NullPointerException` thrown at line 5619 of the `getPlugin` method in Figure 7. Generating such a specific test case requires a search process with high exploration ability, which can generate diverse test cases.

We do note that Single-Objective Search cannot even generate a test case covering the target line (line 85 of the `useMainStore` method). However, *De-MO* can cover the target line thanks to more test generation diversity delivered by the application of multi-objectivization.

Moreover, Single-Objective Search and *De-MO* reproduces two crashes that cannot be reproduced by *MO-HO* after five minutes. We will analyze these corner cases later in Section 5.4.

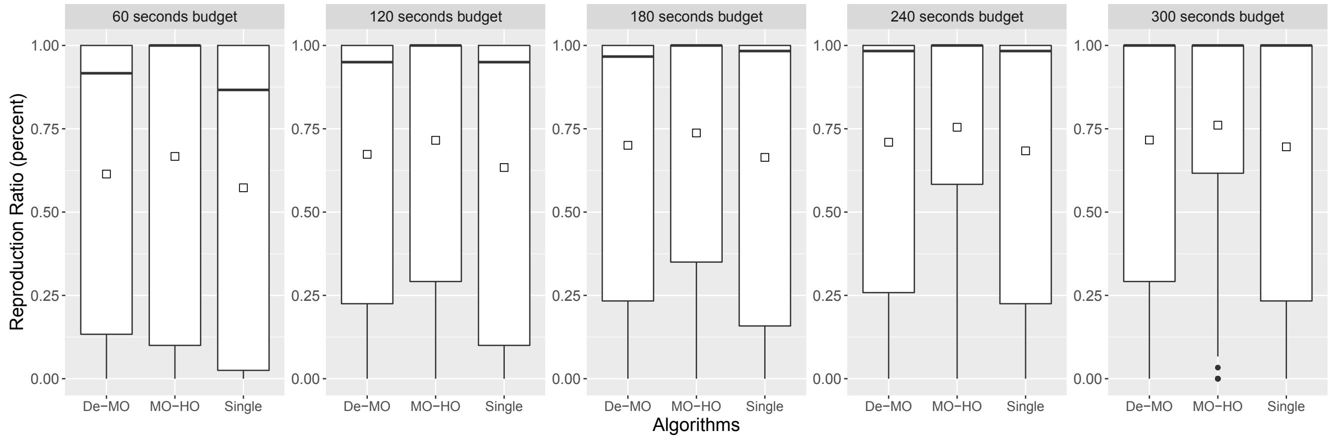

Figure 3: Crash reproduction ratio (out of 30 executions) of *MO-HO* against state-of-the-art in five different time intervals. (□) denotes the arithmetic mean and (—) is the median.

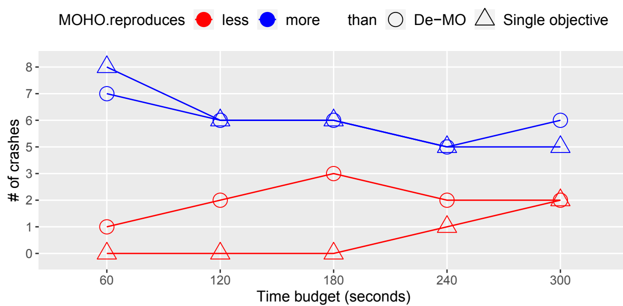

Figure 4: Number of reproduced crashes only by *MO-HO* or only by one of the state-of-the-art algorithms.

```
0   java.lang.NullPointerException: null
1     at [...].XWiki.getPlugin(XWiki.java:5619)
2     at [...].ActivityStreamConfiguration.useMainStore([...]:85)
3       [...]
```

Figure 5: XWIKI-14227 crash's stack trace [37].

```
82   public boolean useMainStore(){
83       XWikiContext context = contextProvider.get();
84       if (context.isMainWiki()) {return false;}
85       ActivityStreamPlugin plugin = (
             ActivityStreamPlugin) context.getWiki().
             getPlugin[...] context); // <-- target line
86   }
```

Figure 6: Method `useMainStore` appears in the second frame of the `XWIKI-14227` crash's stack trace.

In addition, after five minutes of crash reproduction, *De-MO* reproduced six crashes, which are not reproduced by Single-Objective Search. Still, there are more crashes (seven) that can be reproduced by Single-Objective Search but not by *De-MO*. This

```
5617   public XWikiPluginInterface getPlugin([...]){
5618       XWikiPluginManager plugins = getPluginManager();
5619       Vector<String> pluginlist = plugins.getPlugins();
5620       [...]
5621   }
```

Figure 7: Method `getPlugin` appears in the first frame of the `XWIKI-14227` crash's stack trace.

```
1   public void test0() throws Throwable {
2       ActivityStreamConfiguration ac0 = new
            ActivityStreamConfiguration();
3       XWikiContext xWikiContext0 = new XWikiContext();
4       XWiki xWiki0 = new XWiki();
5       xWikiContext0.setWiki(xWiki0);
6       xWikiContext0.setWikiId("4~YRlfI>.U{ib");
7       Provider<XWikiContext> provider0 = (Provider<
            XWikiContext>) mock([...]);
8       doReturn(xWikiContext0).when(provider0).get();
9       Injector.inject(ac0,[...], "contextProvider", (
            Object) provider0);
10
11      // Undeclared exception!
12      ac0.useMainStore();
13  }
```

Figure 8: Crash-reproducing test case generated by *MO-HO* for the XWIKI-14227 crash.

result shows that despite the new crashes reproduced by *De-MO*, this algorithm was counter-productive with respect to the total number of reproduced crashes.

Summary (RQ$_2$). *On average, MO-HO has the highest crash reproduction ratio independently from the search budgets.*

5.3 Efficiency (RQ3)

Figure 9 shows the time (in seconds) needed by the *MO-HO* and the state-of-the-art algorithms to successfully reproduce the crashes in our benchmark. On average, the fastest algorithm is *MO-HO*, with

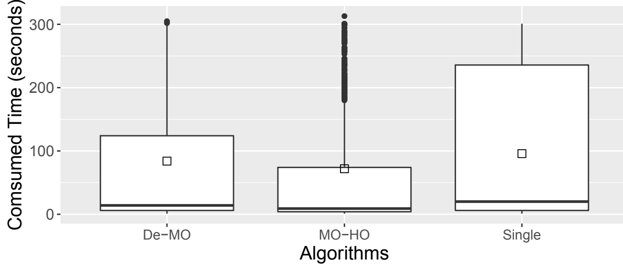

Figure 9: Overall budget consumption in seconds (log. scale). (□) denotes the arithmetic mean and (—) is the median.

Table 2: Pairwise comparison of the budget consumption with a small (S), medium (M), and large (L) effect size $\hat{A}_{12} < 0.5$ and a statistical significance < 0.05.

#($\hat{A}_{12} < 0.5$)	Single			De-MO			MO-HO		
	L	M	S	L	M	S	L	M	S
Single	-	-	-	7	-	4	1	-	2
De-MO	13	7	2	-	-	-	3	2	-
MO-HO	35	6	2	33	10	4	-	-	-

an average search time of 71 seconds per crash replication. The median of its running time is lower than 10 seconds. The second fastest algorithm is *De-MO* that, on average, uses 84 seconds to reproduce the crashes. The slowest algorithm is Single-Objective Search, which demands, on average, about 100 seconds.

Moreover, the biggest improvements achieved by *MO-HO* in terms of efficiency are for *XWIKI-14599*, in which *MO-HO* requires only 3% of the time required by Single-Objective Search to achieve crash reproduction, and *MATH-3b*, in which *MO-HO* requires only 7% of the time required by *De-MO* to finish the crash reproduction task. However, the biggest efficiency losses by *MO-HO* are in *MATH-81b* with 45 seconds drop (15% of time budget) and *XRENDERING-481* with 145 seconds drop (48% of time budget) compared to Single-Objective Search and *De-MO*, respectively.

Table 2 compares the budget consumption of the algorithms from a statistical point of view, *i.e.,* according to the effect sizes ($\hat{A}_{12} < 0.5$) and statistical significance (*p-value* < 0.5). According to this table, *MO-HO* is the fastest algorithm: it significantly reproduced 43 (34.6% of crashes) and 47 (37.9% of crashes) crashes faster than Single-Objective Search and *De-MO*, respectively. Most of these significant improvements have large effect sizes (35 against Single-Objective Search and 33 against *De-MO*). In cases that *MO-HO* improves efficiency, on average, this algorithm decreases the time required for crash reproduction by 47% and 58% compared to *De-MO* and Single-Objective Search, respectively.

Furthermore, Table 2 shows a few cases, in which *MO-HO* increases the consumed time compared to the state-of-the-art: 3 against Single-Objective Search and 5 against *De-MO*. In most of these cases (7 out of 8), the crash reproduction process needs to reproduce a crash with only one frame. Even the exceptional case is a stack trace with three frames. In contrast, in cases that *MO-HO* wins, we have many crashes with more frames (six frames, for

```
0   java.lang.ArrayIndexOutOfBoundsException: 2
1       at org.apache.commons.math.linear.BigMatrixImpl.operate(
            BigMatrixImpl.java:997)
```

Figure 10: `MATH-98b` crash's stack trace [24, 37].

```
991   public BigDecimal[] operate(BigDecimal[] v) {
992       final int nRows = this.getRowDimension();
993       final int nCols = this.getColumnDimension();
994       final BigDecimal[] out = new BigDecimal[v.length];
995       for (int row = 0; row < nRows; row++) {
996           ...
997           out[row] = sum; // <-- target line
998       }
999       ...
1000  }
```

Figure 11: Method `operate` appears in the first frame of the `MATH-98b` crash's stack trace [24, 37].

instance). Also, this table shows that *De-MO* is significantly slower than Single-Objective Search in 11 crashes. Meanwhile, *MO-HO* is only slow in reproducing three crashes. Hence, our proposed algorithm reduces the cases in which the multi-objectivization search process is slower than the single objective search by 73%.

Summary (RQ$_3$). *The fastest crash reproduction algorithm is MO-HO with an average improvement in running time in 34.6% of the crashes compared to the state of the art.*

5.4 Corner cases analysis

Despite the notable improvements achieved by *MO-HO*, there are few specific cases, in which Single-Objective Search or *De-MO* outperform *MO-HO*. For instance, in Section 5.2, Single-Objective Search and *De-MO* reproduce two crashes that are not reproduced by *MO-HO*. Also, we observed in Section 5.3 that the efficiency of these two algorithms is higher than *MO-HO* in 8 crashes.

To understand why *MO-HO* is counter-productive in a few cases, we performed a manual analysis to analyze the factors in *MO-HO* that negatively impact the crash reproduction process. Results of our analysis point to two adverse factors: **extra overhead in calculating the objectives (fitness evaluation)** and **helper-objectives misguidance**.

Extra calculation in fitness evaluation. In some cases, crash reproduction is trivial, and the search process reproduces it in a few seconds. For instance, in `TIME-8b` [24, 37], Single-Objective Search and *De-MO* reproduce the crash in about a second. The time required by *MO-HO* to reproduce this crash is three seconds (3 times more). This stems from the fact that fitness function evaluation in *MO-HO* is more time-consuming than the state-of-the-art: Single-Objective Search and *De-MO* need to calculate only the crash distance for each test case evaluation, while *MO-HO* needs to calculate the call diversity, as well. This extra calculation lengthens the search process by a couple of seconds. In these cases, the increased crash reproduction time is lower than 5 seconds, and it is negligible in practice.

Helper-objectives misguidance. In some other cases, the scenario, which leads to crash reproduction, needs a simple sequence

of methods calls to the target class. Still, the complexity of this scenario stems from the input arguments used for the method calls. In these cases, since crash reproduction does not need the call diversity, *method sequence diversity* objective misguides the search process. Alternatively, we need another objective for method input argument diversity (*i.e.,* improves the diversity of the input arguments for method calls). Adding new helper-objectives to consider other aspects of diversity is part of our future agenda.

As an example, let us analyze `MATH-98b` (Figure 10), in which MO-HO doubled the time consumed by the crash reproduction search process against state-of-the-art. This crash concerns an `ArrayIndexOutOfBoundsException` . Also, this crash has only one frame. For reproducing this crash, the generated test case needs to instantiate a class called `BigMatrixImpl` and call a method named `operate` (Figure 11) with precise input values. Method `getColumnDimension` used in `operate` returns the number of rows in the data variable, which has been set in the constructor. To reproduce this crash, the generated test case should pass an array with a size smaller than the passed size to the constructor. In this case, method argument diversity could help the search process, and the method call diversity is not helpful.

6 DISCUSSION

6.1 Effectiveness and applicability

Generally, *De-MO* reproduces some crashes that cannot be reproduced by Single-Objective Search due to its improved exploration ability, resulting from the multi-objectivization of the crash distance. However, since the decomposed objectives in this approach depend on one another (*e.g.,* the stack trace similarity is not helpful if the generated test does not throw the given type of exception), they may misguide the search process in various cases. For instance, as we saw in Section 5.2, Single-Objective Search reproduces six crashes that are not reproducible by *De-MO*.

In contrast, *MO-HO* has three *conflicting* search objectives. From the theory [22], the objective function must be conflicting to increase the overall exploration ability. Our results confirm the theory: the chance of the search process getting trapped in a local optimum is lower by using *MO-HO* objectives compared to the ones used in *De-MO*. As we observed in Section 5.2, after 1 minute of search, *MO-HO* reproduces 8 and 7 crashes more than Single-Objective Search and *De-MO*, respectively. Also, it continues outperforming with larger search budgets (2, 3, 4, and 5 minutes) until the end of the search process. It reproduces 5 and 6 crashes more than Single-Objective Search and *De-MO*, respectively, while it cannot reproduce only two crashes, reproduced by the other algorithms.

Note that reproducing each crash needs a particular test case which drives the software under test to a particular state, and then, it calls a method with proper input variables. To achieve this goal, each crash reproducing test case needs to create multiple complex objects. Hence, reproducing five new crashes (4% of crashes available in our benchmark) is a significant improvement for *MO-HO*.

6.2 Factors in the benchmark crashes that impact the Success of *MO-HO*

There are multiple factors/characteristics of the crashes in our benchmark that might impact the performance of our approach positively. We identify the following relevant factors: (1) the type of the exception (e.g., null pointer exception), (2) the size the stack frames, (3) the number of classes involved in the crashes, (4) the number of methods of the deepest class in the crash stack. To verify whether these factors influence the performance of our algorithm, we used the two-way permutation test [35]. The permutation test is a well-established non-parametric to assess the significance of factor interactions in multi-factorial analysis of variance (non-parametric ANOVA). We use a significance level *alpha*=0.05 and a very large number of iterations (1,000,000) to ensure the stability of the results over multiple executions of the procedure [35].

For the sake of our analysis, we considered the difference in crash reproduction rate between *MO-HO* and the baselines as the dependent variable, while the co-factors are our independent variables. According to the permutation test, the type of exception (p-value=0.006) and the number of crash stack frames (p-value=0.001) significantly impact the performance of *MO-HO* compared to Single Objective Search. We can also observe similar results when considering the improvements of *MO-HO* against *De-MO*: p-values=$< 10^{-12}$ for both exception type and the number of frames). In other words, there are certain types of exceptions and stack trace sizes for which *MO-HO* is statistically better than the state-of-the-art approaches.

From a deeper analysis, we observe that for `NullPointerException` and `org.joda.time.IllegalFieldValueException`, *MO-HO* achieves a higher reproduction ratio than Single Objective Search when the stack traces contain up to three frames for `NPE` (+22% in reproduction rate) and up to five frames for `IllegalFieldValueException` (+50% in reproduction rate). Instead, for stack traces with more frames, the differences in reproduction ratio are negligible (±1% on average) or negative (-10% in reproduction ratio). Besides, *MO-HO* achieves better reproduction ratios for the following exceptions independently of the stack size: `XWikiExceptions` (+23% on average), `UnsupportedOperationException` (+6% on average), `MathRuntimeException` (+14% on average).

Finally, *MO-HO* outperforms *De-MO* when reproducing `NullPointerException` with 1-3 frames (+8% on average), `ClassCastException` (+8% on average), `StringOutOfBoundsException` (+18% with more than 2 frames, on average), `IllegalFieldException` (+8% on average), `UnsupportedOperationException` (+23% on average), `MockitoException` (+83% for short traces, on average), and `MissingMethodInvocation` (+80% on average).

6.3 Crash reproduction cost

In this study, we observed that since *MO-HO* increases the diversity of the generated test cases, it can dramatically improve the efficiency of crash reproduction. This algorithm significantly improved the speed of the search process in more than 36% of crashes compared to Single-Objective Search and *De-MO*. In cases in which *MO-HO* had a significant impact, it improves the crash reproduction speed by more than 47%.

The prior studies on search-based crash reproduction [37, 38] suggested 5 minutes as the search budget because the search process cannot reproduce more after 5 minutes. However, we observed that despite the high efficiency of *MO-HO*, this algorithm continues to reproduce more crashes in the second half of the time budget.

Section 5.2 shows that *MO-HO* keeps increasing the crash reproduction ratio even in the last minutes of the search process, while the previous multi-objectivization approach (*De-MO*) changes only slightly after the first 2 minutes of crash reproduction. Hence, increasing the search budget for *MO-HO* can lead to a higher crash reproduction ratio.

6.4 Extendability

The improvement achieved by the proposed helper-objectives shows the impact of suitable objectives on increasing the diversity of the generated test cases and result in improving the effectiveness and efficiency of the crash reproduction search process. Hence, we hypothesize that this approach can be extended by adding new relevant helper-objectives.

7 THREATS TO VALIDITY

Internal validity. We cannot ensure that our implementation of BOTSING is without bugs. However, we mitigated this threat by testing our tool and manually analyzing some samples of the results. We used a previously defined benchmark for crash reproduction, which contains 124 non-trivial crashes from six open-source projects and applications. Moreover, we explained how we parametrized the evolutionary algorithms in Section 4.2. We used the default values of these algorithms in the other open-source implementations like EVOSUITE and *JMetal*. The effect of these values for crash reproduction is part of our future work. Finally, to take the randomness of the search process into account, we followed the guidelines of the related literature [3] and executed each evolutionary crash reproduction algorithm for 30 times.

External validity. We report our results for only 124 crashes introduced by JCRASHPACK [37], which is an open-source crash reproduction benchmark collected from six open-source projects. However, we recall here that we cannot guarantee that our results are generalizable to all crashes. Evaluation *MO-HO* on a larger benchmark from more projects is part of our future work.

Reproducibility. We provide BOTSING as an open-source publicly available tool. Also, the data and the processing scripts used to present the results of this paper, including the subjects of our evaluation (inputs), the evolution of the best fitness function value in each generation of each execution, and the produced test cases (outputs), are openly available as a docker image [14].

8 CONCLUSION AND FUTURE WORK

Crash reproduction can ease the process of debugging for developers. Evolutionary approaches have been successfully used to automate this process. Existing evolutionary-based approaches use one single objective (*i.e., Crash Distance*) to guide the search and rely on guided genetic operators. Later strategies applied multi-objectivization via decomposition (*De-MO*) in an attempt to improve diversity (and, therefore, exploration). However, the latter strategy may misguide the search process because the sub-objectives are not strongly conflicting.

In this study, we apply a new approach called Multi-Objectivization using Helper-Objectives (*MO-HO*) to tackle the problems of the former techniques. In *MO-HO*, multi-objectivization is performed by adding two helper-objectives that are in conflict with *Crash Distance*. We evaluated *MO-HO* with five MOEAs, which are selected from different categories of multi-objective algorithms. Our results indicate that *MO-HO* is the most efficient algorithm, significantly outperforming Single-Objective Search and *De-MO*. Also, this algorithm is able to reproduce 8 and 5 more crashes in 1 and 5 minutes, respectively, compared to the state-of-the-art. Moreover, in contrast to the previous multi-objectivized crash reproduction approach (*De-MO*), the crash reproduction ability of *MO-HO* increases with large search budgets (*i.e.,* above two minutes).

We performed an additional analysis to find the correlation between the different aspects of the crashes and the ability of *MO-HO* in reproducing them. The result of this analysis shows that two factors in crashes significantly impact the performance of *MO-HO*: (i) type of exception and (ii) the number of crash stack frames.

Furthermore, we observed that Single-Objective Search and *De-MO* could outperform *MO-HO* but only in a few cases. We performed a manual analysis to characterize the negative factors leading to the adverse results in these cases. Our analysis reveals that two negative factors are at play in these cases: (i) extra calculations in fitness evaluation and (ii) helper-objectives misguidance. We also showed in Section 5.4 that while the differences in *extra calculations in fitness evaluation* are significant, they are often negligible in practice.

The contributions of the paper are as follows:

1. An open-source implementation of seven crash reproduction techniques (Section 4.1).
2. An empirical comparison of seven search-based crash reproduction approaches (Section 4).
3. An analysis of the benefits of multi-objectivization with helper objectives in terms of reproduction ratio and efficiency (Section 5).
4. The identification of the special situations in which *MO-HO* can be counter-productive (Section 5.4).
5. The identification of a strong correlation between the ability of *MO-HO* in improving the efficiency and effectiveness of crash reproduction for combinations of exception types and the number of frames in the stack trace of the target crash (Section 6.2).

In our future work, we will investigate additional helper-objectives for crash reproduction. For instance, the current helper-objectives in *MO-HO* concern the test length and method sequence diversity. However, further objectives can be added, such as test input/data diversity. Increasing the number of objectives will require to evaluate their performance using different many-objective evolutionary algorithms. We will also analyze the evolution of the fitness values of existing and new objective to further investigate the root causes of good and bad performances of *MO-HO* and other objectives for different crashes and different MOEAs.

Moreover, the search objectives introduced by *De-MO* is only optimized by *NSGA-II* MOEA. As future work, we will investigate the impact of utilizing other MOEAs for optimizing *De-MO* objectives.

ACKNOWLEDGMENTS

This research was partially funded by the EU Project STAMP ICT-16-10 No.731529.

REFERENCES

[1] Nasser M Albunian. 2017. Diversity in search-based unit test suite generation. In *International Symposium on Search Based Software Engineering*. Springer, 183–189.

[2] M. Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE, 263–272. https://doi.org/10.1109/ICSE-SEIP.2017.27

[3] Andrea Arcuri and Lionel Briand. 2014. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250. https://doi.org/10.1002/stvr.1486

[4] Francesco A. Bianchi, Mauro Pezzè, and Valerio Terragni. 2017. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, 705–716. https://doi.org/10.1145/3106237.3106292

[5] José Campos, Rui Abreu, Gordon Fraser, and Marcelo d'Amorim. 2013. Entropy-based test generation for improved fault localization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 257–267.

[6] Ning Chen and Sunghun Kim. 2015. STAR: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. on Software Engineering* 41, 2 (2015), 198–220. https://doi.org/10.1109/TSE.2014.2363469

[7] W. J. Conover and Ronald L. Iman. 1981. Rank Transformations as a Bridge between Parametric and Nonparametric Statistics. *The American Statistician* 35, 3 (1981), 124–129. https://doi.org/10.1080/00031305.1981.10479327

[8] David W. Corne, Nick R. Jerram, Joshua D. Knowles, and Martin J. Oates. 2001. PESA-II: Region-Based Selection in Evolutionary Multiobjective Optimization. In *Proceedings of the 3rd Annual Conference on Genetic and Evolutionary Computation* (San Francisco, California) *(GECCO 01)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 283–290.

[9] Indraneel Das and J. E. Dennis. 1998. Normal-Boundary Intersection: A New Method for Generating the Pareto Surface in Nonlinear Multicriteria Optimization Problems. *SIAM J. on Optimization* 8, 3 (March 1998), 631–657. https://doi.org/10.1137/S1052623496307510

[10] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[11] Pouria Derakhshanfar and Xavier Devroey. 2020. *JCrashPack: A Java Crash Reproduction Benchmark*. Zenodo. https://doi.org/10.5281/zenodo.3766689

[12] Pouria Derakhshanfar, Xavier Devroey, Gilles Perrouin, Andy Zaidman, and Arie Deursen. 2020. Search-based crash reproduction using behavioural model seeding. *STVR* 30, 3 (may 2020), e1733. https://doi.org/10.1002/stvr.1733

[13] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. 2020. Crash Reproduction Using Helper Objectives. In *Genetic and Evolutionary Computation Conference Companion (GECCO '20 Companion)*. ACM, Cancún, Mexico. https://doi.org/10.1145/3377929.3390077

[14] Pouria Derakhshanfar, Xavier Devroey, Andy Zaidman, Arie van Deursen, and Annibale Panichella. 2020. *Replication package of "Good Things Come In Threes: Improving Search-based Crash Reproduction With Helper Objectives"*. https://doi.org/10.5281/zenodo.3979097

[15] Juan J Durillo, Antonio J Nebro, and Enrique Alba. 2010. The jMetal framework for multi-objective optimization: Design and architecture. In *IEEE congress on evolutionary computation*. IEEE, 1–8.

[16] Ronald A Fisher. 1936. The use of multiple measurements in taxonomic problems. *Annals of Eugenics* 7, 2 (1936), 179–188. https://doi.org/10.1111/j.1469-1809.1936.tb02137.x

[17] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. ACM, New York, NY, USA, 416–419. https://doi.org/10.1145/2025113.2025179

[18] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. 2008. 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 (14 May 2008), 617. https://doi.org/10.1007/s10732-008-9080-4

[19] R. W. Hamming. 1950. Error Detecting and Error Correcting Codes. *Bell System Technical Journal* 29, 2 (apr 1950), 147–160.

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

[21] Sadeeq Jan, Annibale Panichella, Andrea Arcuri, and Lionel Briand. 2017. Automatic Generation of Tests to Exploit XML Injection Vulnerabilities in Web Applications. *IEEE Transactions on Software Engineering* i (2017), 1–27. https://doi.org/10.1109/TSE.2017.2778711

[22] Mikkel T Jensen. 2004. Helper-objectives: Using multi-objective evolutionary algorithms for single-objective optimisation. *Journal of Mathematical Modelling and Algorithms* 3, 4 (2004), 323–347.

[23] Wei Jin and Alessandro Orso. 2012. BugRedux: reproducing field failures for in-house debugging. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 474–484. https://doi.org/10.1109/ICSE.2012.6227168

[24] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[25] Marco Laumanns, Lothar Thiele, Eckart Zitzler, Emo Welzl, and Kalyanmoy Deb. 2002. Running time analysis of multi-objective evolutionary algorithms on a simple discrete optimization problem. In *International Conference on Parallel Problem Solving from Nature*. Springer, 44–53.

[26] Vladimir Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet Physics Doklady*, Vol. 10. 707–710.

[27] Phil McMinn. 2004. Search-based software test data generation: A survey. *Software Testing Verification and Reliability* 14, 2 (2004), 105–156. https://doi.org/10.1002/stvr.294

[28] Achille Messac, Amir Ismail-Yahaya, and Christopher A Mattson. 2003. The normalized normal constraint method for generating the Pareto frontier. *Structural and multidisciplinary optimization* 25, 2 (2003), 86–98.

[29] Kaisa Miettinen. 1999. *Nonlinear Multiobjective Optimization: Kaisa Miettinen* (1st ed.). Springer US.

[30] Debajyoti Mondal, Hadi Hemmati, and Stephane Durocher. 2015. 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)*. IEEE, 1–10. https://doi.org/10.1109/ICST.2015.7102588

[31] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiene Tahar, and Alf Larsson. 2015. 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)*. IEEE, 101–110. https://doi.org/10.1109/SANER.2015.7081820

[32] Mathieu Nayrolles, Abdelwahab Hamou-Lhadj, Sofiène Tahar, and Alf Larsson. 2017. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process* 29, 3 (mar 2017), e1789. https://doi.org/10.1002/smr.1789 arXiv:1408.1293

[33] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2018), 122–158. https://doi.org/10.1109/TSE.2017.2663435

[34] Annibale Panichella and Urko Rueda Molina. 2017. Java unit testing tool competition - Fifth round. *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017* (2017), 32–38. https://doi.org/10.1109/SBST.2017.7

[35] Fortunato Pesarin and Luigi Salmaso. 2010. *Permutation tests for complex data: theory, applications and software*. John Wiley & Sons.

[36] Jeremias Rößler, Andreas Zeller, Gordon Fraser, Cristian Zamfir, and George Candea. 2013. Reconstructing core dumps. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 114–123. https://doi.org/10.1109/ICST.2013.18

[37] Mozhan Soltani, Pouria Derakhshanfar, Xavier Devroey, and Arie van Deursen. 2020. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering* 25, 1 (jan 2020), 96–138. https://doi.org/10.1007/s10664-019-09762-1

[38] Mozhan Soltani, Pouria Derakhshanfar, Annibale Panichella, Xavier Devroey, Andy Zaidman, and Arie van Deursen. 2018. Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction. In *Symposium on Search-Based Software Engineering. SSBSE 2018. (LNCS)*, Thelma Elita Colanzi and Phil McMinn (Eds.), Vol. 11036. Springer, Montpellier, France, 325–340. https://doi.org/10.1007/978-3-319-99241-9_18

[39] Mozhan Soltani, Annibale Panichella, and Arie Van Deursen. 2018. Search-Based Crash Reproduction and Its Impact on Debugging. *IEEE Transactions on Software Engineering* (2018). https://doi.org/10.1109/TSE.2018.2877664

[40] Yan-Yan Tan, Yong-Chang Jiao, Hong Li, and Xin-Kuan Wang. 2012. A modification to MOEA/D-DE for multiobjective optimization problems with complicated Pareto sets. *Information Sciences* 213 (2012), 14–38.

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

[42] Matej Črepinšek, Shih-Hsi Liu, and Marjan Mernik. 2013. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.* 45, 3 (2013), 33. https://doi.org/10.1145/2480741.2480752

[43] Martin White, Mario Linares-Vásquez, Peter Johnson, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Generating reproducible and replayable bug reports from android application crashes. In *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 48–59.

[44] Jifeng Xuan, Xiaoyuan Xie, and Martin Monperrus. 2015. Crash reproduction via test case mutation: Let existing test cases help. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, New York, New York, USA, 910–913. https://doi.org/10.1145/2786805.2803206

[45] Andreas Zeller. 2009. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging* (2nd ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[46] Qingfu Zhang and Hui Li. 2007. MOEA/D: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on evolutionary computation* 11, 6 (2007), 712–731.

[47] Eckart Zitzler, Marco Laumanns, and Lothar Thiele. 2001. SPEA2: Improving the strength Pareto evolutionary algorithm. *TIK-report* 103 (2001).