Exploring means to facilitate software debugging

SOLTANI, M.S.

4

Fitness Function Evaluation

EvoCrash is a recent search-based approach to generate a test case that reproduces reported crashes. The search is guided by a fitness function that uses a weighted sum scalarization to combine three different heuristics: (i) code coverage, (ii) crash coverage and (iii) stack trace similarity. In this study, we propose and investigate two alternatives to the weighted sum scalarization: (i) the simple sum scalarization and (ii) the multi-objectivization, which decomposes the fitness function into several optimization objectives as an attempt to increase test case diversity. We implemented the three alternative optimizations as an extension of EvoSuite, a popular search-based unit test generator, and applied them on 33 real-world crashes. Our results indicate that for complex crashes the weighted sum reduces the test case generation time, compared to the simple sum, while for simpler crashes the effect is the opposite. Similarly, for complex crashes, multi-objectivization reduces test generation time compared to optimizing with the weighted sum; we also observe one crash that can be replicated only by multi-objectivization. Through our manual analysis, we found out that when optimizing the original weighted function gets trapped in local optima, optimization for decomposed objectives improves the search for crash reproduction. Generally, while multi-objectivization is under-explored, our results are promising and encourage further investigations of the approach.

4.1 Introduction

Crash reproduction is an important step in debugging field crashes. Therefore, various automated approaches to crash reproduction [65, 81, 173, 194, 202, 215] have been proposed in the literature. Among these, EvoCrash [202] is a search-based approach, which applies a Guided Genetic Algorithm (GGA) to generate a crash-reproducing test. To optimize test generation for crash reproduction, the GGA uses a weighted-sum scalarized function, which is a sum of three heuristics, namely: (i) line coverage, (ii) exception coverage, and (iii) stack trace similarity rate. The function resulting from the sum scalarization is further subject to the constraint that the target exception has to be thrown at the code line reported in the crash stack trace. Depending on how close a generated test case may come to trigger a reported crash, its fitness value may be between 0.0 (i.e., each of the three heuristics evaluates to 0.0), and 6.0 (i.e., none of the heuristics is satisfied by the generated test). Soltani et. al [202] evaluated EvoCrash on 50 real-world crashes and showed that the search-based approach improved over other non-search-based approaches proposed in the related literature [81, 173, 215].

As any search-based technique, the success of EvoCrash depends on its capability of maintaining a good balance between *exploitation* and *exploration* [88]. 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 [202]. 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 [88].

In this paper, 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 [87] or multi-objectivizing the fitness function [140] 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 [140].

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. EvoCrash [202] relies on EvoSuite [103] for test generation, and as such, we implemented the original weighted function as an extension of EvoSuite. Similarly, we implemented the alternative optimization functions by extending EvoSuite. We evaluated the alternatives on 33 real-world crashes from four open source projects. 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.

The remainder of the chapter is structured as follows: Section 4.2 provides background and related work. Section 4.3 describes single and multi-objectivization for crash reproduction. Sections 4.4 and 4.5 present the evaluation and results, respectively. Discussion follows in Section 4.6. Section 4.7 concludes.

4.2 Background and Related Work

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 [81]. 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 [135] dataset:

```
0 java.lang.ArrayIndexOutOfBoundsException:
1   at org.apache.commons.lang3.time.FastDateParser.toArray(FastDateParser.java:413)
2   at org.apache.commons.lang3.time.FastDateParser.getDisplayNames(FastDateParser
3     .java:381)
4   ...
```

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

4.2.1 Related Work

Over the years, various Java crash replication approaches that use stack traces as input have been developed. RECORE [194] is a search-based approach that in addition to crash stack traces, uses core dumps as input data for automated test generation. MUCRASH [215] applies mutation operators on existing test cases, for classes that are present in a reported stack trace, to trigger the reported crash. While BUGREDUX [134] is based on forward symbolic execution, STAR [81] 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 [173] is also based on using crash stack traces as the only source of information about a reported crash. JCHARMING [173] applies directed model checking to identify the pre-conditions and input parameters that cause the target crash. Finally, CONCRASH [65] 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 [202], an evolutionary search-based tool for crash replication built on top of EVOSUITE [105]. 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.

4.2.2 EvoCrash

To design EvoCrash, Soltani et al. [202] defined a fitness function (*weighted sum fitness function*) and a search algorithm (*guided genetic algorithm*) dedicated to crash reproduction. The fitness function is used to characterize the "quality" of test case generated during each iteration of the guided GA.

4.2.2.1 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 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 [202]:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \quad (4.1)$$

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* [201]. 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 [160]. In Equation 4.1, $d_{except}(t) \in \{0, 1\}$ 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).

4.2.2.2 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 relies on the stack trace to guide the search and reduces the search space at different steps. (i) 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. (ii) 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 [202]: either directly if m is public or protected, or indirectly by calling another method that invokes the target method if m is private. (iii) During the search, dedicated *guided crossover* and *guided mutation* operators [202] ensure that newly generated test cases contain at least one call to the

target method. (iv) The search is guided by the WS *fitness function*. (v) 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.

4.3 Single-Objective and Multi-Objectivization for Crash Reproduction

A key limitation of evolutionary algorithms (and metaheuristics in general) is that they may become trapped in local optima due to *diversity loss* [88], a phenomenon in which no modification (with crossover and mutation) of the current best solutions will lead to discovering a better one. This phenomenon is quite common in white-box unit-level test case/suite generation, as shown by previous studies in search-based software testing [42, 99, 121, 138]. Many strategies have been investigated by the evolutionary computation community to alleviate the problem of diversity loss, including (i) combining different types of evolutionary algorithms [88, 121], (ii) defining new genetic operators to better promote diversity [88, 93, 121], (iii) altering the fitness function [88, 113, 140], and (iv) relaxing the constraints of the problem [87].

In the context of crash replication, most attention has been devoted to improving the genetic operators [201, 202] to better focus the search on method calls related to the target crash. However, to the best of our knowledge, no previous study investigated alternative formulations to the fitness function in Equation 4.1 and how they are related to diversity and convergence to local optima. The original equation by Soltani et al. [202] (i.e., Equation 4.1) combines three different factors into one single scalar value based on some constraints. Given this type of equation, there are two possible alternatives to investigate: (i) relaxing the constraints and (ii) split the fitness function into three search objectives to optimize simultaneously. The next subsections describe these two alternative formulations of the crash replication problem and how they are related to test case diversity.

4.3.1 Constraints Relaxation

As explained in Section 4.2, the crash replication problem has been implicitly formulated in previous studies as a constraint problem. The constraints are handled using *penalties* [202], 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 4.1 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 [87], it may lead to diversity loss because any test not satisfying the constraints have 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 4.2. 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 4.1, 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 4.1 consists of relaxing the constraints, i.e., removing the penalties. This can be easily implemented with a *Simple Sum Scalarization* (SSS):

$$f(t) = d_s(t) + d_{except}(t) + d_{trace}(t) \tag{4.2}$$

where $d_s(t)$, $d_{except}(t) \in \{0, 1\}$, and $d_{trace}(t)$ are the same as in Equation 4.1. This relaxed variant —hereafter referred 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 4.1. On the other hand, 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.

4.3.2 Multi-objectivization

Knowles et al. [140] 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 [127, 140]: (i) by decomposing the single-objective function into multiple sub-objectives, or (ii)

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 [93]. 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 4.1 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:

$$\begin{cases} f_1(t) = d_s(t) \\ f_2(t) = d_{except}(t) \\ f_3(t) = d_{trace}(t) \end{cases} \tag{4.3}$$

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 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 4.1, 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]

[1]Note that there might exist multiple tests that can replicate the target crash; however, these tests are

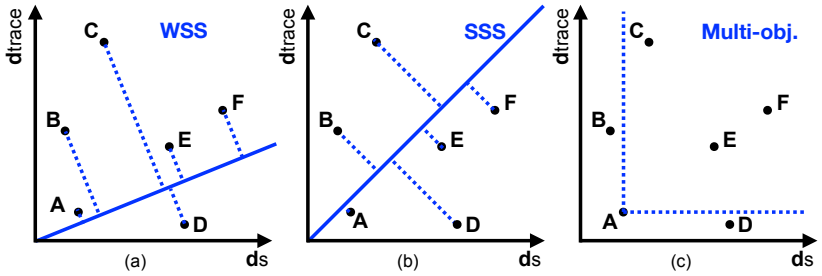

Figure 4.1: A Graphical Interpretation of Different Fitness Functions

Non-dominated Sorting Genetic Algorithm II. To solve our multi-objectivized problem, we use NSGA-II [93], 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 [93]. 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 paper, 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.

4.3.3 Graphical Interpretation

Figure 4.1 shows commonalities and differences among the tree alternative formulations of the crash reproduction problem (see sections 4.3.1 and 4.3.2). 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 4.1(c), points (test cases) in multi-objectivization are compared in terms of non-dominance. In the example, the tests

coincident points as they will all have a zero-value for all objectives.

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. On the other hand, sum scalarization (either simple or weighted) projects all point to one single vector, i.e., the blue lines in Figures 4.1(a) and 4.1(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 4.1(a) and 4.1(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.

4.4 Empirical 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 by EvoCrash to generate a crash reproducing test case (*test generation time*).

4.4.1 Setup

To perform our evaluation, we randomly selected 33 crashes from five open source projects: 18 crashes from four projects contained in Defects4J [135], 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.

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

Table 4.1: Crashes used in the study.

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

We execute the EvoSuite extensions, with 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 [56]. We set the search budget to 10 minutes, which is double of the maximal amount reported by Soltani et al. [202].

4.4.2 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 [207] 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.

4.5 Results

In this section, we present the results of the experiments. Thereby, we answer the two research questions on comparing simple and weighted sum aggregation functions as well as weighted sum and multi-objectivization for crash reproduction.

Results (RQ1). Table 4.2 (please see the end of the chapter) 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 4.3 (please see the end of the chapter) 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 crash, e.g., `XWIKI-13377`, is easy to reproduce, then weighted sum takes longer to find a crash reproducing test.

Results (RQ2). Table 4.2 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 4.3 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 yet.

Moreover, as Table 4.3 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 4.2, 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.

4.6 Discussion

As Table 4.3 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. As indicated in Section 4.3, 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 [140] discussed, when applying multi-objectivization, for a successful search, it is important to derive independent objectives. In our multi-objectivization approach, as presented in Section 4.3, 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 [93], 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.

Insights. From our results and discussion, we formulate the following insights: (i) **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; (ii) 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, (iii) Avoid **simple sum scalarization** as it may get stuck into local optima (multi-objectivization).

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 reproduc-

tion rate. Finally, our extension to EvoSuite may contain unknown defects. To mitigate this risk, in addition to testing the extensions, the first three authors reviewed the artifacts independently.

4.7 Conclusion

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.

Table 4.2: 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

Table 4.3: 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