Basic block coverage for search-based unit testing and crash reproduction

Derakhshanfar, Pouria; Devroey, Xavier; Zaidman, Andy

DOI
10.1007/s10664-022-10155-0

Publication date
2022

Document Version
Final published version

Published in
Empirical Software Engineering

Empirical Software Engineering (2022) 27:192
https://doi.org/10.1007/s10664-022-10155-0

Basic block coverage for search-based unit testing and crash reproduction

Pouria Derakhshanfar[1] · **Xavier Devroey**[2] · **Andy Zaidman**[1]

Accepted: 17 March 2022

Abstract

Search-based techniques have been widely used for white-box test generation. Many of these approaches rely on the *approach level* and *branch distance* heuristics to guide the search process and generate test cases with high line and branch coverage. Despite the positive results achieved by these two heuristics, they only use the information related to the coverage of explicit branches (*e.g.,* indicated by conditional and loop statements), but ignore potential implicit branchings within basic blocks of code. If such implicit branching happens at runtime (*e.g.,* if an exception is thrown in a branchless-method), the existing fitness functions cannot guide the search process. To address this issue, we introduce a new secondary objective, called Basic Block Coverage (*BBC*), which takes into account the coverage level of relevant basic blocks in the control flow graph. We evaluated the impact of *BBC* on *search-based unit test generation* (using the DynaMOSA algorithm) and *search-based crash reproduction* (using the *STDistance* and *WeightedSum* fitness functions). Our results show that for unit test generation, *BBC* improves the branch coverage of the generated tests. Although small (∼1.5%), this improvement in the branch coverage is systematic and leads to an increase of the output domain coverage and implicit runtime exception coverage, and of the diversity of runtime states. In terms of crash reproduction, in the combination of *STDistance* and *WeightedSum*, *BBC* helps in reproducing 3 new crashes for each fitness function. *BBC* significantly decreases the time required to reproduce 43.5% and 45.1% of the crashes using *STDistance* and *WeightedSum*, respectively. For these crashes, *BBC* reduces the consumed time by 71.7% (for *STDistance*) and 68.7% (for *WeightedSum*) on average.

Communicated by: Aldeida Aleti, Annibale Panichella and Shin Yoo

This article belongs to the Topical Collection: *Open Science*
This paper has been awarded the Empirical Software Engineering (EMSE) open science badge

This article belongs to the Topical Collection: *Advances in Search-Based Software Engineering (SSBSE)*

✉ Pouria Derakhshanfar
p.derakhshanfar@tudelft.nl

Extended author information available on the last page of the article.

Published online: 06 October 2022

Keywords Automated crash reproduction · Search-based software testing · Evolutionary algorithm · Secondary objective

1 Introduction

Various search-based techniques have been introduced to automate different white-box test generation activities, *e.g.,* unit testing (Fraser and Acuri 2013b, 2011), integration testing (Derakhshanfar et al. 2020), or system-level testing Arcuri (2019). Depending on the testing level, each of these approaches utilizes dedicated fitness functions to guide the search process and produce a test suite satisfying given criteria (*e.g.,* line coverage, branch coverage, *etc.*).

Fitness functions typically rely on *control flow graphs (CFGs)* to represent the source code of the software under test (McMinn 2004). Each node in a CFG is a *basic block* of code (*i.e.,* maximal linear sequence of statements with a single entry and exit point without any internal branch), and each edge represents a possible *execution flow* between two blocks. Two well-known heuristics are usually combined to achieve high line and branch coverages: the *approach level* and the *branch distance* (McMinn 2004). The former measures the distance between the execution path of the generated test and a target basic block (*i.e.,* a basic block containing a statement to cover) in the CFG. The latter measures, using a set of rules, the distance between an execution and the coverage of a *true* or *false* branch of a particular predicate in a branching basic block of the CFG.

Both *approach level* and *branch distance* assume that only a limited number of basic blocks (*i.e., control dependent* basic blocks Allen 1970) can change the execution path away from a target statement (*e.g.,* if a target basic block is the true branch of a conditional statement). However, basic blocks are not atomic due to the presence of **implicit branches** (Borba et al. 2010) (*i.e.,* branches occurring due to the exceptional behavior of instructions). As a consequence, any basic block between the entry point of the CFG and the target basic block can impact the execution of the target basic block. For instance, a generated test case may stop its execution in the middle of a basic block with a runtime exception thrown by one of the statements of that basic block. In these cases, the search process does not benefit from any further guidance from the approach level and branch distance.

Fraser and Arcuri (Fraser and Arcuri 2015a) introduced testability transformation for **unit testing**, which instruments the code to guide the unit test generation search to cover implicit exceptions happening in the class under test. However, this approach does not guide the search process in scenarios where an implicit branch happens in another class called by the class under test. This is due to the extra cost added to the search process stemming from the calculation and monitoring of implicit branches in all the classes coupled to the class under test. For instance, the class under test may be heavily coupled with other classes in the project, thereby finding implicit branches in all of these classes can be expensive.

In contrast, other test case generation scenarios, like **crash reproduction**, aim to cover only a limited number of paths, and thereby we only need to analyse a limited number of basic blocks (Chen and Kim 2015; Xuan et al. 2015; Nayrolles et al. 2015; Rößler et al. 2013; Soltani et al. 2018). Current crash reproduction approaches rely on information about a reported crash (*e.g.,* a stack trace, a core dump, *etc.*) to generate a crash reproducing test case.

Among these approaches, **search-based crash reproduction** (Rößler et al. 2013; Soltani et al. 2018) takes as input a **stack trace** to guide the generation process. More specifically,

the statements pointed to by the stack trace act as target statements for the approach level and branch distance. Hence, current search-based crash reproduction techniques suffer from a lack of guidance in cases where the involved basic blocks contain implicit branches (which is common when trying to reproduce a crash).

In our prior work we have introduced a novel secondary objective called **Basic Block Coverage (*BBC*)** to address the guidance problem in crash reproduction (Derakhshanfar et al. 2020). The secondary objective guides the search process to differentiate two generated tests with the same fitness values (here, same approach level and branch distance). This paper extends our prior work on *BBC* to the more general unit test case generation context.

BBC helps the search process to compare two generated test cases with the same distance (according to approach level and branch distance) to determine which one is closer to the target statement. In this comparison, *BBC* analyzes the coverage level, achieved by each of these test cases, of the basic blocks in between the closest covered control dependent basic block and the target statement.

To assess the impact of *BBC* on search-based unit test generation, we implemented *BBC* in EvoSuite (Fraser and Arcuri 2011), the state-of-the-art tool for search-based unit test generation, and evaluate its performance against the classical DynaMOSA (Panichella et al. 2018b) for various activation probabilities of *BBC* (11 configurations in total). We applied these eleven configurations to 219 classes under test selected from the last version of Defects4J v.2.0.0 (Just et al. 2014), a collection of existing faults. We compare the performance in terms of effectiveness for branch coverage, weak mutation score, output coverage, and real fault detection capabilities.

Our results show that *BBC* improves the branch coverage of the generated tests when activating *BBC* as a secondary objective in DynaMOSA. Although small on average (from 74.5% for DynaMOSA up to 76.1% for *BBC*), this improvement in the branch coverage leads to an increase of the average output domain coverage (from 54.2% for DynaMOSA up to 55.5% for *BBC*) and implicit runtime exception coverage (from 75.1% when using DynaMOSA up to 80.3% for *BBC*), and of the diversity of runtime states (denoted by an increase of the average weak mutation score from 73.2% for DynaMOSA, up to 74.6% for *BBC*). Our statistical analysis confirms that this improvement is systematic across all *BBC* configurations. Activating *BBC* also significantly improves with a large effect the fault detection rate for 3 real faults out of 92.

Similarly, to assess the impact of *BBC* on search-based crash reproduction, we re-implemented the existing *STDistance* (Rößler et al. 2013) and *WeightedSum* (Soltani et al. 2018) fitness functions and empirically compared their performance with and without using *BBC* (4 configurations in total). We applied these four crash reproduction configurations to 124 hard-to-reproduce crashes introduced in JCrashPack (Soltani et al. 2020), a crash benchmark used by previous crash reproduction studies (Derakhshanfar et al. 2020). We compare the performance in terms of *effectiveness in crash reproduction ratio* (*i.e.,* percentage of times that an approach can reproduce a crash) and *efficiency* (*i.e.,* time required by for reproducing a crash).

Our results show that *BBC* significantly improves the crash reproduction ratio over the 30 runs in our experiment for respectively 10 and 4 crashes when compared to using *STDistance* and *WeightedSum* without any secondary objective. Also, *BBC* helps these two fitness functions to reproduce 3 (for *STDistance*) and 3 (for *WeightedSum*) crashes that could not be reproduced without secondary objective. Besides, on average, *BBC* increases the crash reproduction ratio of *STDistance* and *WeightedSum* by 9% and 4.5%, respectively. Applying *BBC* also significantly reduces the time consumed for crash reproduction guided by

STDistance and *WeightedSum* in 56 (45.1% of cases) and 54 (43.5% of cases) crashes, respectively. In cases where *BBC* has a significant impact on efficiency, this secondary objective improves the average efficiency of *STDistance* and *WeightedSum* by 71.7% and 68.7%, respectively.

The remainder of this paper is organized as follow: Section 2 reports the background and related work on CFG-based guidance. Section 3 describes our novel *BBC* secondary objective and how it can be used for search-based crash reproduction and search-based unit test generation. Section 4 describes our evaluation to assess the importance of implicit branches (**RQ 0**) and the impact of *BBC* on search-based unit test generation (**RQ 1**) and search-based crash reproduction (**RQ 2**). Section 5 presents our results on 219 classes under test selected from the last version of DEFECTS4J and 124 hard-to-reproduce crashes from JCRASHPACK. Sections 6 and 7 discuss our results and their implications for search-based test case generation, and Section 8 concludes the paper.

2 Background and Related Work

2.1 Coverage Distance Heuristics

Many structural-based search-based test generation approaches mix the *branch distance* and *approach level* heuristics to achieve a high line and branch coverage (McMinn 2004). These heuristics measure the distance between a test execution path and a specific statement or a specific branch in the software under test. For that, they rely on the coverage information of *control dependent basic blocks*, *i.e.,* basic blocks that have at least one outgoing edge leading the execution path toward the *target basic block* (containing the targeted statement) and at least another outgoing edge leading the execution path away from the target basic block. As an example, Listing 1 shows the source code of the method `fromMap` from

```
402 public BaseCollection fromMap(Map<[...]> map, BaseCollection
        object){
403     for (PropertyClass property : (Collection<[...]>)
            getFieldList()) {
404         String name = property.getName();
405         Object formvalues = map.get(name);
406         if (formvalues != null) {
407             BaseProperty objprop;
408             if (formvalues instanceof String[]) {
409                 [...]
410             } else if (formvalues instanceof String) {
411                 objprop = property.fromString(formvalues.
                        toString());
412             } else {
413                 objprop = property.fromValue(formvalues);
414             }
415             [...]
416         }
417     }
418     return object;
419 }
```

Listing 1 Method `fromMap` from XWIKI version 8.1 (Soltani et al. 2020)

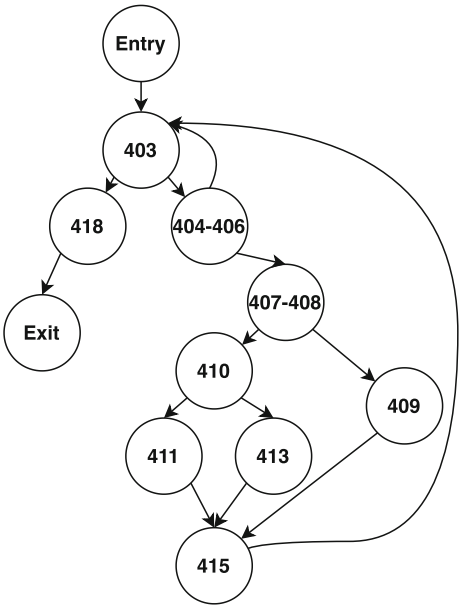


Fig. 1 CFG for method `fromMap`

XWIKI[1], and Fig. 1 contains the corresponding CFG. In this graph, the basic block `409` is control dependent on the basic block `407-408` because the execution of line 409 is dependent on the condition at line 408 (*i.e.,* line 409 will be executed only if elements of array `formvalues` are `String`).

The *approach level* is the number of uncovered control dependent basic blocks for the target basic block between the closest covered control dependent basic block and the target basic block. The *branch distance* is calculated from the predicate of the closest covered control dependent basic block, based on a set of predefined rules. Assuming that the test t covers only line `403` and `418`, and our target line is `409`, the approach level is 2 because two control dependent basic blocks (`404-406` and `407-408`) are not covered by t. The branch distance for the predicate in line `403` (the closest covered control dependency of node `409`) is measured based on the rules from the established technique (McMinn 2004).

To the best of our knowledge, there is no related work studying the extra heuristics helping the combination of approach level and branch distance to improve the coverage. Most related to our work, Panichella et al. (2018b) and Rojas et al. (2015) introduced two heuristics called *infection distance* and *propagation distance*, to improve the weak mutation score of two generated test cases. However, these heuristics do not help the search process to improve the general statement coverage (*i.e.,* they are effective only after covering a mutated statement).

In this paper, we introduce a new secondary objective to improve the statement coverage achieved by fitness functions based on the approach level and branch distance, and analyze the impact of this secondary objective on **search-based unit test generation** and **search-based crash reproduction**.

[1] https://github.com/xwiki

2.2 Search-based Unit Test Generation

Search-based software test generation (SBST) algorithms use meta-heuristic optimization search techniques (*e.g.,* genetic algorithm) to automate the test generation tasks in different testing levels. One of these levels is unit testing, where the search algorithm tries to generate tests satisfying various criteria (such as line coverage and branch coverage) for a given class under test (CUT). SBST techniques are widely used for unit test generation. Prior studies showed that the tests generated by these techniques achieve a high code coverage (Panichella et al. 2018a; Campos et al. 2018) and real-bug detection (Almasi et al. 2017), hence complementing the hand-written test cases.

Dynamic many-objective sorting algorithm (DynaMOSA). Panichella et al. have recently introduced an evolutionary-based algorithm, called DynaMOSA, for unit test generation (Panichella et al. 2018b). Their study (Panichella et al. 2018a), independently confirmed by Campos et al. (2018), shows that DynaMOSA outperforms other unit test generation techniques in terms of structural coverage and mutation coverage. This approach is currently used as the default algorithm in EvoSuite, which is the state-of-the-art tool for search-based unit test generation.

DynaMOSA relies on the hierarchy of dependencies between the coverage targets (*e.g.,* lines and branches) to perform a dynamic selection of the objectives during the search process. For instance, by applying DynaMOSA to generate tests for method `fromMap` (Listing 1), this algorithm, first, try to cover targets that do not have any dependencies. So, first, it tries to generate test cases to cover nodes `403` and `418`. After covering node `403`, it tries to cover the node `404-406`, which is control-dependent on the covered node. DynaMOSA continuously changes the search objectives up to the point that all of the targets are covered.

Since DynaMOSA uses the approach level and branch distance heuristics to guide the search process towards achieving the high line, branch, and weak mutation coverage, *BBC* may help this technique to cover more targets. This study performs an in-depth experiment and analysis to see whether *BBC* can improve DynaMOSA.

Testability Transformation (TT). Testability transformations address the problem of implicit branches in unit test generation (Li and Fraser 2011; Fraser and Arcuri 2015a). This strategy transforms the code to make implicit branches explicit by adding extra branches for error conditions, and bring more guidance for the approach level and branch distance heuristics. For code transformation of each class, TT needs extra bytecode instrumentation. Since instrumenting some classes can be difficult due to several known issues (Fraser and Arcuri 2013a), instrumenting each class, which is coupled with the class under test, may fail. Also, if we limit the testability transformations to the class under test, the search process will not have any extra guidance in cases of facing the implicit branches in the other classes.

2.3 Search-based Crash Reproduction

After a crash is reported, one of the essential steps of software debugging is to write a **crash reproducing test case** to make the crash observable to the developer and help them in identifying the root cause of the failure (Zeller 2009). Later, this crash reproducing test can be integrated into the existing test suite to prevent future regressions. Despite the usefulness of a crash reproducing test, the process of writing this test can be labor-intensive and time-taking (Soltani et al. 2018). Various techniques have been introduced to automate

the reproduction of a crash (Chen and Kim 2015; Xuan et al. 2015; Nayrolles et al. 2015; Rößler et al. 2013; Soltani et al. 2018), and search-based approaches (EvoCrash (Soltani et al. 2018) and ReCore Rößler et al. 2013) yielded the best results (Soltani et al. 2018).

EvoCrash. This approach utilizes a single-objective genetic algorithm to generate a crash reproducing test from a given stack trace and a *target frame* (*i.e.,* a frame in the stack trace that its class will be used as the class under test). The crash reproducing test generated by EvoCrash throws the same stack trace as the given one up to the target frame. For example, by passing the stack trace in Listing 2 and target frame 3 to EvoCrash, it generates a test case reproducing the first three frames of this stack trace (*i.e.,* thrown stack trace is identical from line 0 to 3).

EvoCrash uses a fitness function, called *WeightedSum*, to evaluate the candidate test cases. *WeightedSum* is the sum scalarization of three components: (i) the **target line coverage** (d_s), which measures the distance between the execution trace and the *target line* (*i.e.,* the line number pointed to by the target frame) using *approach level* and *branch distance*; (ii) the **exception type coverage** (d_e), determining whether the type of the triggered exception is the same as the given one; and (iii) the **stack trace similarity** (d_{tr}), which indicates whether the stack trace triggered by the generated test contains all frames (from the most in-depth frame up to the target frame) in the given stack trace.

Definition 1 (*WeightedSum* Soltani et al. 2018) *For a given test case execution t, the WeightedSum (ws) is defined as follows:*

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

Where $d_s(t) \in [0, 1]$ indicates how far t is from reaching the target line and is computed using the normalized approach level and branch distance: $d_s(t) = \|approachLevel_s(t) + \|branchDistance_s(t)\|\|$ ($\|$ indicates the normalized value); $d_e(t) \in \{0, 1\}$ shows if the type of the exception thrown by t is the same as the given stack trace (0) or not (1); $d_{tr}(t) \in [0, 1]$ measures the stack trace similarity between the given stack trace and the one thrown by t. $max(f)$ and $min(f)$ denote the maximum and minimum possible values for a function f, respectively.

In this fitness function, $d_e(t)$ and $d_{tr}(t)$ are only considered in the satisfaction of two *constraints*: (i) *exception type coverage* is relevant only when we reach the target line and (ii) *stack trace similarity* is important only when we both reach the target line and throw the same type of exception.

As an example, when applying EvoCrash on the stack trace from Listing 2 with the target frame 3, *WeightedSum* first checks if the test cases generated by the search process

```
0  java.lang.ClassCastException: [...]
1      at [...].BaseStringProperty.setValue(BaseStringProperty.
           java:45)
2      at [...].PropertyClass.fromValue(PropertyClass.java:615)
3      at [...].BaseClass.fromMap(BaseClass.java:413)
4      [...]
```

Listing 2 XWIKI-13377 crash stack trace (Soltani et al. 2020)

reach the statement pointed to by the target frame (line 413 in class `BaseClass` in this case). Then, it checks if the generated test can throw a `ClassCastException` or not. Finally, after fulfilling the first two constraints, it checks the similarity of frames in the stack trace thrown by the generated test case against the given stack trace in Listing 2.

EvoCrash uses **guided** initialization, mutation and single-point crossover operators to ensure that the target method (*i.e.,* the method appeared in the target frame) is always called by the different tests during the evolution process.

According to a recent study, EvoCrash outperforms other non-search-based crash reproduction approaches in terms of *effectiveness in crash reproduction* and *efficiency* (Soltani et al. 2018). This study also shows the helpfulness of tests generated by EvoCrash for developers during debugging.

In this paper, we assess the impact of *BBC* as the secondary objective in the EvoCrash search process.

ReCore This approach utilizes a genetic algorithm guided by a single fitness function, which has been defined according to the core dump and the stack trace produced by the system when the crash happened. To be more precise, this fitness function is a sum scalarization of three sub-functions: (i) **TestStackTraceDistance**, which guides the search process according to the given stack trace; (ii) **ExceptionPenalty**, which indicates whether the same type of exception as the given one is thrown or not (identical to ExceptionCoverage in EvoCrash); and (iii) **StackDumpDistance**, which guides the search process by the given core dump.

Definition 2 (*TestStackTraceDistance* Rößler et al. 2013) *For a given test case execution t, the TestStackTraceDistance (STD) is defined as follows:*

$$STD(R,t) = |R| - lcp - (1 - StatementDistance(s)) \tag{2}$$

Where $|R|$ is the number of frames in the given stack trace. And lcp is the longest common prefix frames between the given stack trace and the stack trace thrown by t. Concretely, $|R| - lcp$ is the number of frames not covered by t. Moreover, $StatementDistance(s)$ is calculated using the sum of the approach level and the normalized branch distance to reach the statement s, which is pointed to by the first (the utmost) uncovered frame by t: $StatementDistance(s) = approachLevel_s(t) + \|branchDistance_s(t)\|$.

Since using runtime data (such as core dumps) can cause significant overhead (Chen and Kim 2015) and leads to privacy issues (Nayrolles et al. 2015), the performance of ReCore in crash reproduction was not compared with EvoCrash in prior studies (Soltani et al. 2018). Although, two out of three fitness functions in ReCore use only the given stack trace to guide the search process. Hence, this paper only considers *TestStackTraceDistance* + *ExceptionPenalty* (called *STDistance* hereafter).

As an example, when applying ReCore with *STDistance* on the stack trace in Listing 2 with target frame 3, first, *STDistance* determines if the generated test covers the statement at frame 3 (line 413 in class `BaseClass`). Then, it checks the coverage of frame 2 (line 615 in class `PropertyClass`). After covering the first two frames by the generated test case, it checks the coverage of the statement pointed to by the deepest frame (line 45 in class `BaseStringProperty`). For measuring the coverage of each of these statements, *STDistance* uses the approach level and branch distance. After covering all of the frames, this fitness function checks if the the generated test throws `ClassCastException` in the deepest frame.

In this study, we perform an empirical evaluation to assess the performance of crash reproduction using *STDistance* with and without *BBC* as the secondary objective in terms of *effectiveness in crash reproduction* and *efficiency*.

3 Basic Block Coverage

3.1 Motivating Example

During the search process, the fitness of a test case is evaluated using a fitness function. These fitness functions are different according to the given test criteria. However, one of the main components of these fitness functions is the coverage of specific statements and branches. For instance, one of the main goals in the unit test generation is achieving a high structural coverage (*e.g.,* line and branch coverage). For this goal, the search process seeks to cover all of the statements and branches in the given CUT. Similarly, the fitness functions used in search-based crash reproduction (either *WeightedSum* or *STDistance*) require the coverage of specific statements pointed by the given stack trace.

The distance of the test case from the target statement is calculated using the approach level and branch distance heuristics. As we have discussed in Section 2.1, the approach level and branch distance cannot guide the search process if the execution stops because of implicit branches in the middle of basic blocks (*e.g.,* a thrown `NullPointerException` during the execution of a basic block). As a consequence, these fitness functions may return the same fitness value for two tests, although the tests do not cover the same statements in the block of code where the implicit branching happens.

For instance, assume that one of the objectives of a search process (either for unit test generation or crash reproduction) is covering line `413` in method `fromMap` (appeared in Listing 1). This search process generates two test cases T_1 and T_2 for achieving this objective in a population of solutions. However, T_1 stops the execution at line `404` due to a `NullPointerException` thrown in method `getName`, and T_2 throws a `NullPointerException` at line `405` because it passes a null value input argument to `map`. Even though T_2 covers more lines, the combination of approach level and branch distance returns the same fitness value for both of these test cases: approach level is 2 (nodes `407-408` and `410`), and branch distance cannot be helpful in this case as the last covered predicate does not change the execution path away from covering the target line and also the execution stops before covering the next predicate. This is because these two heuristics assume that each basic block is atomic, and by covering line `404`, it means that lines `405` and `406` are covered, as well.

3.2 Secondary Objective

The goal of the Basic Block Coverage (*BBC*) secondary objective is to prioritize the test cases with the same fitness value (*i.e.,* same approach level and branch distance) according to their coverage within the basic blocks between the closest covered control dependency and the target statement. At each iteration of the search algorithm, test cases with the same fitness value are compared with each other using *BBC*. Listing 3 presents the pseudo-code of the *BBC* calculation. Inputs of this algorithm are two test cases T_1 and T_2, which both have the same approach level and branch distance values (calculated either using crash reproduction or unit test generation fitness functions), as well as line number and method name of the target statement. This algorithm compares the coverage of basic blocks on the

```
1  input: test T1, test T2, String method, int line
2  output: int
3  begin
4      FCB1 ← fullyCoveredBlocks(T1,method,line);
5      FCB2 ← fullyCoveredBlocks(T2,method,line);
6      SCB1 ← semiCoveredBlocks(T1,method,line);
7      SCB2 ← semiCoveredBlocks(T2,method,line);
8
9      if SCB1 = SCB2 ∧ (FCB1 ⊆ FCB2 ∨ FCB2 ⊆ FCB1) :
10         closestBlock ← closestSemiCoveredBlocks(SCB1, method
               , line);
11         coveredLines1 ← getCoveredLines(T1,closestBlock);
12         coveredLines2 ← getCoveredLines(T2,closestBlock);
13         return size(coveredLines2) - size(coveredLines1);
14     else if (FCB1 ⊆ FCB2 ∧ SCB1 ∈ FCB2) ∨ (FCB2 ⊆ FCB1 ∧ SCB2
           ∈ FCB1):
15         return size(FCB2) - size(FCB1)
16     else:
17         return 0;
18 end
```

Listing 3 *BBC* secondary objective computation algorithm

path between the last control dependent node executed by both of the given tests and the basic block that contains the target statement (called *effective blocks* hereafter). If T_1 and T_2 do not cover any control dependency of the target block, *BBC* uses the entry point of the CFG of the given method instead as the starting point of the effective blocks' path. If *BBC* determines there is no preference between these two test cases, it returns 0. Also, it returns a value < 0 if T_1 has higher coverage compared to T_2, and vice versa. A higher absolute value of the returned integer indicates a bigger distance between the given test cases.

In the first step, *BBC* detects the effective blocks that are fully covered by each given test case (*i.e.,* the test covers all of the statements in the block) and saves them in two sets called $\texttt{FCB}_1$ and $\texttt{FCB}_2$ (lines 4 and 5 in Listing 3). Then, for each of the tests T_1 and T_2, it detects the closest semi-covered effective block (*i.e.,* the closest basic block to the target statement where the test covers the first line but not the last line of the block) and stores them as $\texttt{SCB}_1$ and $\texttt{SCB}_2$, respectively (lines 6 and 7). The semi-covered blocks indicate the presence of implicit branches.

BBC can prioritize given tests in two scenarios: **Scenario 1**, both tests get stuck in the middle of the same basic block (*i.e.,* they both have the same closest semi-covered basic block), or, **Scenario 2**, one of the tests throws an exception in an effective basic block while the other test fully covers this block.

Scenario 1 Line 9 in Listing 3 checks if the first scenario is true by determining two conditions. First, *BBC* checks if both tests have the same semi-covered basic block. Then, it examines if fully covered basic blocks of one of the given tests are equal or the subset of the other test. If the second condition is not fulfilled, it means that each of these tests has one covered block that the other one does not cover, and thereby they achieve their semi-covered basic block from different paths. In this case, *BBC* cannot find the better test as we do not know which path can lead to covering the target statement. If these two conditions are fulfilled, *BBC* checks if one of the tests has a higher line coverage in the identified SCB (lines 10 to 13). If this is the case, *BBC* will return the number of lines in this block covered only by the winning test case. If the lines covered are the same for T_1 and T_2

(*i.e.,* `coveredLines1` and `coveredLines2` have the same size), there is no difference between these two test cases and *BBC* returns value 0 (line 13).

Scenario 2 Line 14 in Listing 3 checks if the effective blocks covered by one test are a subset of the other one. This is true if all of the fully-covered blocks of one test are a subset of fully covered blocks of the other one. Also, the semi-covered block of this test must be among the fully-covered blocks of the test with more coverage (*i.e.,* winner test). In this case, *BBC* returns the number of blocks that are only fully covered by the winner test case (line 15). If *BBC* determines T_2 wins over T_1, the returned value will be positive, and vice versa.

Finally, if each of the given tests has a unique covered block in the given method (*i.e.,* the tests cover different paths in the method), *BBC* cannot determine the winner and returns 0 (lines 16 and 17) because we do not know which path leads to the target block. Even if T_1 and T_2 reach a particular basic block from different paths in the CFG and both throw exceptions in different lines, *BBC* returns 0 and does not select the one with the more coverage in the closest basic block as the winner. The rationale behind this behavior of *BBC* is to provide an equal chance for these two tests to evolve as we do not know which path covered by each of these tests has more potential to help the search process to get closer to the target line. If *BBC* always selects the test with more coverage in the nearest basic block, even if it covers another path, we are negatively impacting the diversity of the tests chosen for the next generation, thereby reducing the search process's exploration ability.

Example When giving two tests with the same fitness value (calculated by the primary objective) T_1 and T_2 from our motivation example to *BBC* with target method `fromMap` and line number 413, this algorithm compares their fully and semi-covered blocks with each other. In this example, both T_1 and T_2 cover the same basic blocks: the fully covered block is `403` and the semi-covered block is `404-406`. So, here the conditions in **Scenario 1** are fulfilled. Hence, *BBC* checks the number of lines covered by T_1 and T_2 in block `404-406`. Since T_1 stopped its execution at line 404, the number of lines covered by this test is 1. In contrast, T_2 managed to execute two lines (404 and 405). Hence, *BBC* returns $size(coveredLines2) - size(coveredLines1) = 1$. The positive return value indicates that T_2 is closer to the target statement, and therefore, it should have a higher chance of being selected for the next generation.

Branchless Methods *BBC* can also be helpful for *branchless methods*. These methods do not contain any branching statement (*e.g.,* if conditions or for loops), and thereby theoretically, covering the first line in these methods leads to covering all of the other lines, as well. In other words, by ignoring the `Entry` and `Exit` nodes, CFGs of branchless methods contain only one node (*i.e.,* basic block) without any edges. For instance, methods from frames 1 and 2 in Fig. 2 are branchless. The absence of branches in these methods means that there are no control dependent nodes in them, and thereby approach level and branch distance cannot guide the search process in these cases if the generated tests throw implicit exceptions in the middle of these methods. However, in contrast with these two heuristics, *BBC* can guide the search process toward covering the most in-depth statement in these cases. As an example, if tests T_1 and T_2 both throws implicit branches in the middle of the only basic block (b_0) of branchless method $m()$, *BBC* enters the Scenario 1 ($FCB_1 = FCB_2 = \emptyset$ and $SCB_1 = SCB_2 = \{b_0\}$) and examines if one of the tests has more lines covered in b_0.

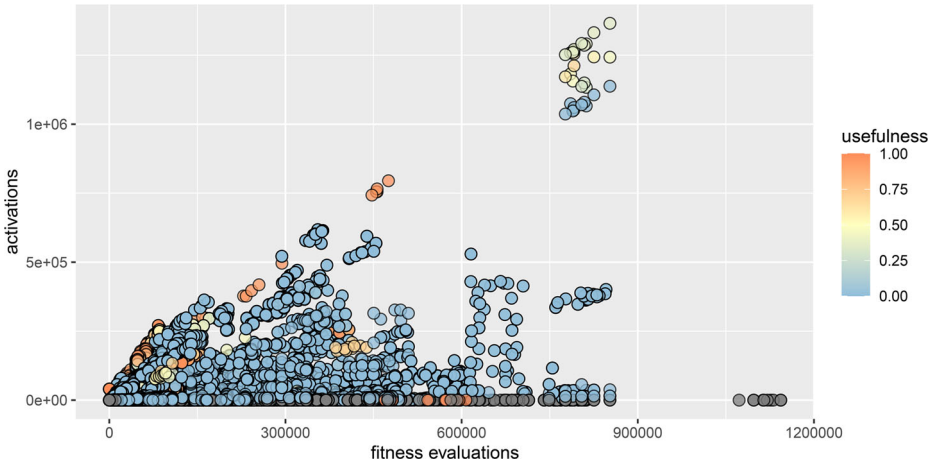


Fig. 2 Distribution of the usefulness of *BBC* activations per fitness evaluations. The usefulness is defined as the number of *BBC* evaluations returning a non-zero value divided by the number of activations. Grey points denote fitness evaluations without any *BBC* activation

3.3 Application of *BBC*

The time complexity of *BBC* is $\mathcal{O}(N \times E \times log\, V)$ where E and V are the numbers of edges and vertices of the CFG of the given method, respectively; and N is the number of semi-covered basic blocks calculated by `semiCoveredBlocks` method at lines `6` and `7` of Listing 3. This complexity stems from the computation of the closest semi-covered basic blocks in Line `12` of Listing 3. In this procedure, *BBC* measures the shortest path between each semi-covered basic block and the target basic block (*i.e.,* the block containing the given target line) using *Dijkstra*'s shortest path algorithm, which has the time complexity of $\mathcal{O}(E \times log\, V)$.

Given the complexity of *BBC*, applying this secondary objective for any generated tests with the same approach level and branch distance may negatively impact the search process's efficiency. In the following paragraphs, we discuss this potential negative impact on search-based crash reproduction and unit test generation.

3.3.1 Search-Based Crash Reporduction

The crash reproduction search process can be guided by either *WeightedSum* or *STDistance*. As discussed in Section 2.3, both of these fitness functions heavily rely on approach level and branch distance. Hence, *BBC* can be helpful in the crash reproduction search process. Since the crash reproduction search process's goal is to cover a specific path in the control dependent graph, which is indicated by the given stack trace, we apply *BBC* without any limitation on any case that includes two test cases with the same (and nonzero) approach level and branch distance.

3.3.2 Search-Based Unit Test Generation

In contrast with crash reproduction, the unit test generation search process has multiple statements and branches to cover simultaneously. In DynaMOSA, each line or branch to

cover is an objective of the search. Hence, the number of times that *BBC* is applied as the secondary objective is higher compared to crash reproduction. Therefore, we should limit the number of times that *BBC* is applied in this algorithm. We introduce two parameters to bring this limitation: SLEEP TIME and USAGE RATE.

SLEEP TIME When DYNAMOSA adds a target to the active search objectives, the target will stay active until the search process covers it. Some of the targets are easy to cover, and thereby, approach level and branch distance can simply cover them without *BBC*. However, *BBC* can help in harder cases where approach level and branch distance cannot cover them in a certain time. SLEEP TIME makes sure that *BBC* is only applied for the hard-to-cover search objectives. If we set this parameter to t seconds, DYNAMOSA uses *BBC* secondary objective only for search objectives that are active for more than t seconds.

USAGE RATE Like any other evolutionary-based algorithm, the unit test generation search process needs to maintain a balance between the *exploration* and *exploitation*. The former indicates the diversity in the solutions (*i.e.,* generated tests execute new paths in the code); the latter indicates searching the solutions in the existing ones' neighborhood (*i.e.,* the search process should generate tests similar to the existing ones). By applying *BBC*, we improve the exploitation ability of the search process. However, the over-application of *BBC* may negatively impact the exploration ability of the search process. USAGE RATE makes sure that *BBC* does not hinder this balance. Higher USAGE RATE means that there is a higher chance of *BBC* application during the search process. Assume we set $p \in [0, 1]$ as our USAGE RATE. Any time that the search process generates two test cases with the same approach level and branch distance for a hard-to-cover target (*i.e.,* target which stays as an active objective in DYNAMOSA for more than SLEEP TIME), *BBC* will be used with the probability of p.

Moreover, by default, EVOSUITE has eight types of search objectives (Rojas et al. 2015): *line coverage*, which aims to cover maximum lines in the given CUT; *branch coverage*, which aims to cover maximum branches in the CUT; *exception coverage*, which aims to maximize the number of exceptions captured by the generated tests; *weak mutation*, which aims to generate tests that kill the maximum number of mutants (in weak mutation, a mutant is considered killed if executing one of the generated tests on the mutant leads to a different state compared to the execution on the given CUT); *output coverage*, that aims for generating tests that drive the most diverse outputs; *method coverage*, which aims to cover all of the methods in the given CUT; *no-exception Method Coverage*, checks if each of the methods in the CUT is called directly by one of the tests and this invocation does not lead to any exception; and *direct branch coverage* that makes sure that each branch in the public methods of CUT is covered by a direct call from one of the generated tests.

Since *BBC* aims to help the search process relying on the approach level and branch distance in covering lines and branches that cannot be executed with the tests generated by DYNAMOSA, this secondary objective is only triggered when two tests have the same fitness value either for a non-covered line coverage or branch coverage objective. Hence, *BBC* is not involved in segments of the search process in which two tests are getting the same fitness value for other kinds of objectives such as exception coverage. Thereby, despite the fact that *BBC* prioritizes tests without throwing implicit exceptions, since this secondary objective is not triggered for objectives other than line coverage and branch coverage, it does not have any negative impact on covering other search objectives (*e.g.,* exception coverage).

4 Empirical Evaluation

Before evaluating the impact of *BBC*, we want to assess its potential usefulness by answering the following research question:

RQ 0 How frequent are implicit branches in a search-based test case generation process?

This research question serves as a preliminary analysis before the full evaluation of the impact of *BBC* on search-based unit test generation and search-based crash reproduction. To answer it, we consider a special configuration of DynaMOSA, currently the best algorithm for unit test generation, where the executions of the *BBC* algorithm described in Listing 3 are monitored. We choose DynaMOSA, a many-objectives algorithm, because, unlike search-based crash reproduction, it targets each line and branch of a class under test independently, allowing us to collect more data about the execution of *BBC* for the different objectives.

To assess the impact of *BBC* on search-based unit test generation, we perform an empirical evaluation to answer the following research questions:

RQ 1 What is the impact of *BBC* on search-based unit test generation?
- **RQ 1.1** What is the impact of *BBC* on the structural coverage effectiveness of the unit tests?
- **RQ 1.2** What is the impact of *BBC* on the output coverage of the unit tests?
- **RQ 1.3** What is the impact of *BBC* on the fault finding capabilities of the unit tests?
- **RQ 1.4** What is the impact of *BBC* on the structural coverage efficiency of the unit tests?

In these RQs, we want to evaluate the effect of *BBC* on DynaMOSA. As for other algorithms, DynaMOSA relies on the approach level and branch distance to evaluate the progress of the search process. Previous research has shown that it outperforms other search-based and guided random approaches (Campos et al. 2018; Devroey et al. 2020; Kifetew et al. 2019; Molina et al. 2018; Panichella et al. 2018a, b). We compare DynaMOSA for 11 different configurations of *BBC* in terms of structural coverage effectiveness (RQ 1.1). Since a change in the structural coverage of a class might impact the data flow, we also study the outputs produced by the different tests (RQ 1.2). Then, we look at the fault finding capabilities using weak mutation and real faults from the Defects4J collection (RQ 1.3). Finally, we study the structural coverage efficiency of *BBC* (RQ 1.4).

Similarly, for search-based crash reproduction, we answer the following research questions:

RQ 2 What is the impact of *BBC* on search-based crash reproduction?
- **RQ 2.1** What is the impact of *BBC* on the crash reproduction effectiveness?
- **RQ 2.2** What is the impact of *BBC* on the crash reproduction efficiency?

In these two RQs, we want to evaluate the effect of *BBC* on the existing fitness functions, namely *STDistance* and *WeightedSum*, from two perspectives: the crash reproduction ratio of the different configurations (RQ 2.1) and the time required to reproduce a crash (RQ 2.2).

In Sections 4.1 and 4.2 we will detail the experimental setup for respectively the study on unit test generation (RQ 0 and RQ 1) and crash reproduction (RQ 2).

4.1 Setup for search-based unit test generation (RQ 0 and RQ 1)

4.1.1 Implementation

We implemented *BBC* as a secondary objective (called `BBCOVERAGE`) in EvoSuite (Fraser and Arcuri 2011), the state-of-the-art tool for search-based unit test generation. As discussed in Section 3.3.2, since *BBC* impacts the exploration-exploitation

trade-off and efficiency of the search process, we also defined two additional parameters for SLEEP TIME (BBC_SLEEP with a default value of 60 seconds) and USAGE RATE (BBC_USAGE_PERCENTAGE with a default probability of 0.5). Our implementation is openly available at https://github.com/pderakhshanfar/evosuite.

4.1.2 Classes under test selection

We selected classes under test from the latest version of DEFECTS4J (v.2.0.0) (Just et al. 2014), a collection of reproducible failures coming from open source projects with the identification of the corresponding faulty classes. DEFECTS4J has been used in other studies to assess the coverage and the effectiveness of unit-level test case generation (Ma et al. 2015; Panichella et al. 2018b; Shamshiri et al. 2015), program repair (Smith et al. 2015; Martinez and Monperrus 2016), fault localization (Pearson et al. 2017; Le et al. 2016), and regression testing (Noor and Hemmati 2015; Lu et al. 2016).

We selected the ten most recent bugs from the 17 available projects for a total of 225 faulty classes, used as classes under test in our evaluation. This offers a good balance between the number of repetitions (*i.e.,* statistical power) of each configuration and number of cases (*i.e.,* generalization) (Arcuri and Briand 2014).

Since EVOSUITE may face inevitable challenges for generating tests for some particular classes (Xiao et al. 2011; McMinn 2011; Fraser and Arcuri 2014), we performed a trial with default parameters, on all of the classes to filter out the ones for which EVOSUITE cannot generate any test, as recommended by related work (Campos et al. 2018; Molina et al. 2018; Panichella et al. 2018b). We filtered out six classes according to our trial experiment results. In three of these classes, EVOSUITE could not finish the class instrumentation. For the other two, DYNAMOSA could not find any search objective. Finally, EVOSUITE failed to generate tests for a class because of missing classes. By filtering these classes, we performed our main experiment on the 219 remaining cases. Table 1 provides more information about the classes selected for the evaluation.

4.1.3 Parameter settings

To evaluate the impact of *BBC* secondary objective on search-based unit test generation, first, we should set values for SLEEP TIME and USAGE RATE (explained in Section 3.3.2). To find the optimum SLEEP TIME, we performed a pre-analysis on a subset of subjects. We have randomly selected 45 classes (20% of our subjects) for this pre-analysis. We ran DYNAMOSA on each of the sampled classes for 30 times and collected the time required by the search process for covering each objective. These collected results indicate that DYNAMOSA can cover more than 85% of the objectives in 60 seconds. For this reason, we have set SLEEP TIME to 60 seconds for our experiments.

For our pre-analysis (**RQ 0**), we have enabled *BBC* (USAGE RATE= 1.0) after 60 seconds (with an additional setting to record the execution results of *BBC*) to evaluate the number of implicit branches occurring during the search and the number of times *BBC* could help overcoming those implicit branches. Furthermore, to draw a comparison between setting different USAGE RATE, we have used ten different values of this parameter in our main experiment (**RQ 1**): USAGE RATE $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$.

Hence, for the main experiment, we have executed DYNAMOSA and one plus ten configurations of *BBC* on 219 classes for 30 rounds of execution with the search budget of 10 minutes. Also, we have executed DYNAMOSA on 45 classes with the same number of repetition and search budget for finding the optimum SLEEP TIME. In total, we ran 80,190

Table 1 Classes under test used for the evaluation of *BBC* for unit testing (**RQ 0 RQ 1**): number of classes under test (**CUTs**), number of non-commented source statements per class (**NCSS**), number of methods per class (**Methods**), weighted methods per class (**WMC**), and cyclomatic complexity per method (**CCN**)

Project	CUTs	NCSS		Methods		WMC		CCN	
		$\overline{x}(\sigma)$	range	$\overline{x}(\sigma)$	range	$\overline{x}(\sigma)$	range	$\overline{x}(\sigma)$	range
Chart	10	471.0(570.9)	[5,1917]	54.1(64.7)	[2,229]	199.7(245.1)	[2,817]	3.7(6.7)	[1,110]
Cli	14	140.9(77.5)	[45,236]	25.4(13.8)	[8,44]	66.9(39.5)	[17,111]	2.6(3.3)	[1,19]
Closure	11	589.4(389.2)	[88,1276]	63.6(69.4)	[22,265]	265.6(188.4)	[52,601]	4.2(10.4)	[1,230]
Codec	11	152.2(158.1)	[42,564]	19.6(14.1)	[4,50]	72.8(83.5)	[10,304]	3.7(5.1)	[1,42]
Collections	4	375.8(440.1)	[49,1021]	67.5(61.0)	[14,153]	204.5(234.2)	[23,542]	3.0(3.8)	[1,28]
Compress	10	257.3(192.7)	[24,569]	35.9(24.8)	[4,75]	117.7(81.1)	[10,226]	3.3(3.6)	[1,27]
Csv	11	225.9(127.6)	[53,460]	36.5(24.2)	[15,78]	119.7(75.8)	[29,250]	3.3(6.4)	[1,44]
Gson	12	319.1(242.0)	[56,933]	38.0(21.9)	[10,80]	160.9(121.8)	[32,464]	4.2(6.0)	[1,64]
JacksonCore	13	852.2(645.4)	[125,2121]	67.5(22.0)	[32,109]	386.8(312.7)	[56,1012]	5.7(7.7)	[1,71]
JacksonDatabind	32	214.1(196.0)	[19,911]	34.0(29.7)	[1,126]	106.8(103.7)	[8,446]	3.1(4.2)	[1,62]
JacksonXml	6	290.7(166.0)	[104,526]	36.7(23.8)	[11,68]	126.0(66.5)	[49,214]	3.4(5.0)	[1,40]
Jsoup	18	273.1(338.6)	[5,1348]	38.8(37.4)	[2,125]	116.4(143.5)	[2,583]	3.0(7.7)	[1,176]
JxPath	14	239.9(194.4)	[22,488]	26.0(18.5)	[3,45]	131.7(108.5)	[9,291]	5.1(7.1)	[1,61]
Lang	10	274.1(190.0)	[29,455]	34.6(25.2)	[2,75]	153.7(121.4)	[10,329]	4.4(9.9)	[1,76]
Math	18	195.7(182.1)	[29,579]	23.0(17.7)	[4,54]	78.5(69.3)	[13,198]	3.4(4.6)	[1,49]
Mockito	13	68.7(65.8)	[10,220]	18.5(24.0)	[2,74]	39.8(45.5)	[3,151]	2.1(2.5)	[1,31]
Time	12	273.2(130.0)	[71,442]	51.8(25.3)	[18,103]	123.9(53.8)	[45,195]	2.4(3.0)	[1,28]

independent executions to answer **RQ 0** and **RQ 1**. These executions took about 12 days overall.

4.1.4 Data collection

To evaluate the potential impact of *BBC* (**RQ 0**), we collected for each line and branch objective: the number of times its *fitness* has been evaluated, and the number of times *BBC* has been *called*, *activated* (*i.e.,* the call effectively led to an evaluation of the *BBC*, line 13 or 15 in Listing 3), and *useful* (*i.e.,* the call to *BBC* has returned a non-zero value). When *BBC* is useful, it indicates that at one or both of the test throw an implicit exception in the middle of a basic block in the method of search objective (*i.e.,* line or branch coverage objective).

We compare *BBC* to DynaMOSA using *branch coverage* for **RQ 1.1** and **RQ 1.4** for 30 rounds of execution. Branch coverage provides an indication about the structural coverage by looking at the percentage of branches covered by the executions of the test cases in the class under test. We recorded the value of the branch coverage every ten seconds to see how it evolves over time and answer **RQ 1.4**.

For **RQ 1.2**, we consider *output coverage* and *implicit exceptions*. Output coverage (Alshahwan and Harman 2014) denotes the diversity of the outputs of the different methods of the class under test. It provides information about the data output coverage of the generated tests by looking at how many pre-defined abstract values (*i.e.,* partitions of the output domain) are returned by the methods of the class under test (Rojas et al. 2015). For instance, a method returning integer value has to return negative, zero, and positive values (when the tests are executed) to satisfy the output coverage criterion.

In addition to (expected) outputs, we consider *implicit exceptions* by looking at the number (e) of top-level methods in the class under test throwing an undeclared (*i.e.,* runtime) exception implicitly (*i.e.,* without any `throw new` instruction). For one execution, we compute the *implicit exception coverage* as the ratio between e and the highest value of e among the all the executions of the different *BBC* configurations for that class.

Since *BBC* addresses the challenge of handling implicit branches for search-based unit test generation, we expect it to impact both the output coverage and the number of methods throwing an implicit exception.

We rely on *weak mutation* and *real faults* to assess the fault findings capabilities of the generated tests (**RQ 1.3**). Weak mutation score (Howden 1982; Papadakis and Malevris 2011) gives the percentage of mutants (*i.e.,* artificially injected faults) for which at least one test triggers a different program state, compared to the original program, directly after the execution of the mutated statement. Weak mutation is a viable cheaper alternative to strong mutation, which requires an additional propagation of the erroneous state to the output of the program (Offutt and Lee 1994). For our evaluation, weak mutation allows us to assess the diversity of runtime states, allowing to catch more faults, when using *BBC*. We use the default set of weak mutation operators available in EvoSuite (Fraser and Arcuri 2015b): delete call, delete field, insert unary operator, replace arithmetic operator, replace bitwise operator, replace comparison operator, replace constant, and replace variable.

Additionally, we use real faults from the Defects4J benchmark to compare the effective fault finding capabilities of tests generated using *BBC*. We executed all of the 11 configurations on the buggy versions of the software, and next, we check if the tests generated by each configuration can throw the same exception as the bug exposing stack traces, which are indicated by Defects4J. The rationale behind running all of the configurations only

on the buggy versions, and not the fixed versions, is to have a realistic scenario. In a realistic scenario, developers are neither aware of the bug, nor have access to the fixed version. In this scenario, an automated test generation tool can help developers if it generates tests that throw an exception revealing the bug. Since EvoSuite can detect the assertion-based failures only by running it on the fixed version (Fraser and Arcuri 2015a), we limited our comparison for fault detection only on the 92 faults that a non-assertion error can expose.

4.1.5 Data analysis

For each class under test, we use the Vargha-Delaney $\hat{A}_{12}$ statistic (Vargha and Delaney 2000) to examine the effect size of differences between using and not using *BBC* for branch, output, and implicit exception coverage, and weak mutation score (**RQs 1.1-1.3**). For a pair of factors (A, B) a value of $\hat{A}_{12} > 0.5$ indicates that A is more likely to achieve a higher coverage or mutation score, while a value of $\hat{A}_{12} < 0.5$ shows the opposite. Also, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. We used the standard thresholds (Vargha and Delaney 2000) for interpreting the $\hat{A}_{12}$ magnitude: 0.56 (small), 0.64 (medium), and 0.71 (large). To assess the significance of effect sizes ($\hat{A}_{12}$), we apply the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.01$ for the Type I error.

We also rank the different configurations of *BBC*, based on their coverage and weak mutation score, using Friedman's non-parametric test for repeated measurements with a significance level $\alpha = 0.05$ (García et al. 2009) (**RQs 1.1-1.3**). This test is used to test the significance of the differences between groups (treatments) over the dependent variable (here, coverage and weak mutation score). We further complement the test for significance with Nemenyi's post-hoc procedure (Japkowicz and Shah 2011; Panichella 2021).

Finally, since fault coverage (**RQ 1.3**) has a dichotomic distribution (*i.e.,* a generated test exposes the fault or not), we use the Odds Ratio (OR) to measure the impact of each *BBC* configuration on the *real faults coverage*. A value $OR > 1$ in a comparison between a pair of factors (A, B) indicates that the application of factor A increases the fault coverage, while $OR < 1$ indicates the opposite. Also, a value of $OR = 1$ indicates that both of the factors have the same performance. We apply Fisher's exact test, with $\alpha = 0.01$ for the Type I error, to assess the significance of results.

4.2 Setup for search-based crash reproduction (RQ 2)

4.2.1 Implementation

Since ReCore and EvoCrash are not openly available, we implement *BBC* in Botsing, an extensible, well-tested, and open-source search-based crash reproduction framework already implementing the *WeightedSum* fitness function and the guided initialization, mutation, and crossover operators. We also implement *STDistance* (ReCore fitness function) in this tool. Botsing relies on EvoSuite for code instrumentation and test case generation by using `evosuite-client` as a dependency. We also implement the *STDistance* fitness function used as baseline in this paper.

4.2.2 Crash selection

We select crashes from JCrashPack (Soltani et al. 2020), a benchmark containing hard-to-reproduce Java crashes. We apply the two fitness functions with and without using *BBC*

as a secondary objective to 124 crashes, which have also been used in a recent study (Derakhshanfar et al. 2020). These crashes stem from six open-source projects: JFreeChart, Commons-lang, Commons-math, Mockito, Joda-time, and XWiki. For each crash, we apply each configuration on each frame of the crash stack traces. We repeat each execution 30 times to take randomness into account, for a total of 114,120 independent executions. We run the evaluation on two servers with 40 CPU-cores, 128 GB memory, and 6 TB hard drive. In total, these executions took about 5 days.

4.2.3 Parameter settings

We run each search process with five minutes time budget and set the population size to 50 individuals, as suggested by previous studies on search-based test generation (Panichella et al. 2018b). Moreover, as recommended in prior studies on search-based crash reproduction (Soltani et al. 2018), we use the *guided mutation* with a probability $p_m = 1/n$ ($n =$ length of the generated test case), and the *guided crossover* with a probability $p_c = 0.8$ to evolve test cases. We do note that prior studies do not investigate the sensitivity of the crash reproduction to these probabilities. Tuning these parameters should be undertaken as future work.

4.2.4 Data collection

To evaluate the crash reproduction ratio (*i.e.,* the ratio of success in crash reproduction in 30 rounds of runs) of different assessed configurations (**RQ 2.1**), we follow the same procedure as previous studies (Derakhshanfar et al. 2020; Soltani et al. 2018): for each crash C, we detect the highest frame that can be reproduced by at least one of the configurations (r_{max}). We examine the crash reproduction ratio of each configuration for crash C targeting frame r_{max}.

To evaluate the efficiency of different configurations (**RQ 2.2**), we analyze the time spent by each configuration on generating a crash reproducing test case. We do note that the extra pre-analysis and basic block coverage in *BBC* is considered in the spent time. Since measuring efficiency is only possible for the reproduced crashes, we compare the efficiency of algorithms on the crashes that are reproduced at least once by one of the algorithms. We assume that the algorithm reached the maximum allowed budget (5 minutes) in case it failed to reproduce a crash.

4.2.5 Data analysis

As for real fault coverage (**RQ 1.3**), crash reproduction data (**RQ 2.1**) has a dichotomic distribution (*i.e.,* an algorithm reproduces a crash C from its r_{max} or not), we use the Odds Ratio (OR) to measure the impact of each algorithm in crash reproduction ratio for each crash. A value $OR > 1$ in a comparison between a pair of factors (A, B) indicates that the application of factor A increases the crash reproduction ratio, while $OR < 1$ indicates the opposite. Also, a value of $OR = 1$ indicates that both of the factors have the same performance. We apply Fisher's exact test, with $\alpha = 0.01$ for the Type I error, to assess the significance of results.

For **RQ 2.2**, we use the Vargha-Delaney $\hat{A}_{12}$ statistic (Vargha and Delaney 2000) with the non-parametric Wilcoxon Rank Sum test to examine differences between using and not using *BBC* for efficiency. For a pair of factors (A, B) a value of $\hat{A}_{12} > 0.5$ indicates that A reproduces the target crash in a longer time, while a value of $\hat{A}_{12} < 0.5$ shows the opposite.

Also, $\hat{A}_{12} = 0.5$ means that there is no difference between the factors. We used the standard thresholds (Vargha and Delaney 2000) for interpreting the $\hat{A}_{12}$ magnitude: 0.56 (small), 0.64 (medium), and 0.71 (large).

4.3 Replicability

We enable the replicability of our results by providing replication packages on Zenodo (https://zenodo.org) for **RQ 0** and **RQ 1** (Derakhshanfar and Devroey 2021) and **RQ 2** (Derakhshanfar and Devroey 2020). Those replication packages include the classes under test and crashes used for the evaluation, the evaluation infrastructure (including documentation and scripts to re-run the evaluation), and the data analysis procedure used to produce the graphs, tables, and numbers reported in this paper.

5 Results

5.1 Potential impact of *BBC* (RQ 0)

Table 2 provides the general statistics of the preliminary analysis answering **RQ 0** per project. The number of branch and line objectives ranges from 526 for `Codec` to 8,108 for `JacksonCore`. In total, the number of fitness evaluations per objective ranges between 1 and 1,143,620 with an average of 30,111.81 evaluations. *BBC* has been called between 1 and 1,681,329 times per objective with an average of 34,988.58 calls. It is interesting to note that, since the evaluation of an objective may require to compare multiple test cases, *BBC* can be called multiple times for each fitness evaluation. *BBC* has been effectively activated up to 1,365,526 (average of 9,472.140) times per objective, and has been useful up to 798,005 (average of 354) times per objective.

Figure 2 provides a summary of the usefulness of *BBC*. Each data point corresponds to the percentage of useful calls to *BBC* per fitness evaluation, measured for one objective and one execution out of 30. On average, *BBC* has been useful 2.5 times ($\sigma = 3.17$ times) per fitness evaluation, with a maximum of 4,0145 times for a single fitness evaluation (which happens when multiple test cases have to be compared).

Summary (RQ 0) Implicit branches are quite common. Our results show that on average, *BBC* has been activated (*i.e.,* the call to *BBC* effectively led to an evaluation) 9,472.140 times with a standard deviation $\sigma = 40,567.40$, denoting big variations of the activation among the different objectives. The usefulness rate per activation is 2.39% on average ($\sigma = 12.09\%$), confirming that not all activations can effectively lead to a distinction between two test cases *w.r.t.* to their partial coverage of basic blocks. Those results tend to confirm our design choice to parameterize the activation of *BBC* using an activation probability.

5.2 Search-based unit test generation (RQ 1)

We first discuss the results of applying *BBC* as a secondary objective for unit test generation using DynaMOSA. Contrarily to crash reproduction, which seeks to cover only a small number of branches, unit test generation targets all the branches in a class under test.

Branch coverage effectiveness (RQ 1.1) Figure 3a reports the branch coverage of the different classes under test for all the 30 test suites for the different configurations of *BBC*.

Table 2 Statistics about the number of objectives (**Obj.**), fitness evaluations (**Fitness eval.**), calls to *BBC* evaluations (**BBC calls**), calls effectively leading to an evaluation of the *BBC* (**BBC active**), and evaluations returning a non-zero value (**BBC useful**)

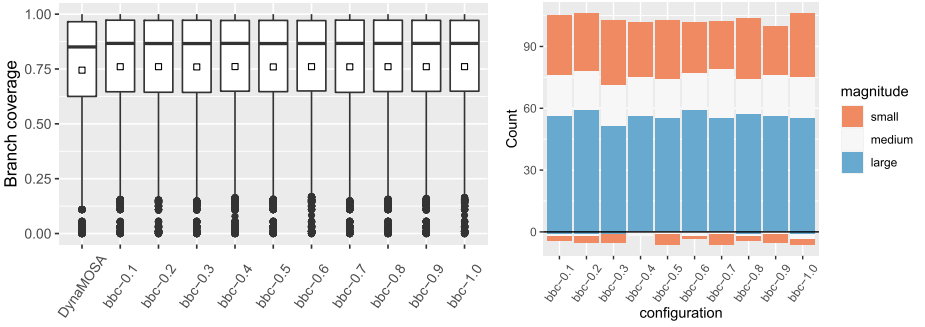
Project	Obj.	Fitness eval.		BBC calls		BBC active		BBC useful	
		$\overline{count}$	σ	$\overline{count}$	σ	$\overline{count}$	σ	$\overline{count}$	σ
Chart	3492	17,522.59	69,478.23	15,769.44	108,896.01	2,267.34	15,409.45	133.66	4,980.58
Cli	963	46,395.51	144,057.34	39,927.05	179,255.85	5,300.18	37,189.29	2.74	43.16
Closure	4779	23,864.65	33,537.30	34,880.69	59,787.43	8,716.67	28,410.00	446.23	6,556.16
Codec	526	85,859.14	138,087.36	118,522.38	249,495.43	49,434.50	161,610.84	0.00	0.07
Collections	915	41,404.66	40,811.89	78,162.33	80,603.58	2,391.87	13,382.00	713.11	6,706.91
Compress	1602	27,870.01	56,441.02	25,610.46	58,955.84	10,477.92	35,881.90	0.06	2.13
Csv	1279	21,797.16	66,812.74	21,892.09	89,951.27	1,617.00	16,831.40	51.66	561.60
Gson	2272	50,307.24	105,668.55	47,428.92	143,743.06	12,515.74	69,460.49	972.59	22,547.11
JacksonCore	8108	16,546.99	32,507.93	16,406.25	49,033.41	10,233.04	34,686.78	240.63	5,202.30
JacksonDatabind	4932	19,779.36	44,533.60	26,837.34	72,399.41	6,323.01	21,387.24	436.74	6,523.39
JacksonXml	1130	30,898.57	29,490.09	55,675.38	64,763.15	35,723.20	47,364.58	195.75	1,210.80
Jsoup	2458	58,216.14	117,964.61	82,136.18	168,880.68	2,080.66	17,089.75	87.03	3,178.65
JxPath	2348	51,578.30	103,321.87	29,519.46	104,762.47	7,402.75	42,828.18	4.72	64.64
Lang	1749	37,868.96	93,794.17	20,247.58	60,978.13	1,338.74	12,510.84	2.91	38.91
Math	1309	27,917.29	48,262.64	49,197.32	84,697.47	21,353.59	45,462.28	2,710.62	19,146.62
Mockito	584	91,840.19	113,787.23	156,256.50	216,605.91	42,901.56	95,736.14	608.66	4,312.73
Time	1891	19,180.13	45,616.90	21,628.31	68,101.74	1,331.23	11,072.58	90.19	2,319.25
(all)	40337	30,111.81	71,396.34	34,988.58	100,703.53	9,472.14	40,567.40	354.12	7,913.20

Generally, the average branch coverage slightly improves when activating *BBC* as a secondary objective (from 74.5% for DynaMOSA up to 76.1% for *BBC* 0.2, 0.4, 0.6, and 1.0). Although small, this improvement is systematic across all *BBC* configurations according to the effect sizes reported in Fig. 3b. *BBC* 0.6 gives the best results with a *large* positive ($\hat{A}_{12} > 0.5$) effect size for 59 classes under test (against 0 *large* negative, $\hat{A}_{12} < 0.5$, effect size), followed by *BBC* 0.2 with 59 classes (against 1 classes), and *BBC* 0.8 with 57 classes (against 1 class).

Figure 4 provides a graphical representation of the ranking (*i.e.,* mean ranks with confidence interval) of the different *BBC* configurations. According to Friedman's test, the different treatments *BBC* 0.1 to 1.0 achieve significantly different branch coverage (p-values < 0.01) compared to DynaMOSA. Furthermore, the differences between the average ranks of *BBC* 0.1 to 1.0 and the average rank of the baseline are larger than the critical distance $CD = 1.375$ determined by Nemenyi's post-hoc procedure. This indicates that *BBC* 0.1 to 1.0 achieves a significantly higher branch coverage than DynaMOSA.

We analyzed the correlation between the effect sizes ($\hat{A}_{12}$) of the best performing *BBC* configuration (according to Friedman's test with Nemenyi's post-hoc procedure) and *BBC* usefulness (presented in RQ 0). The result of this analysis indicates that there is a positive correlation between the number of times that *BBC* could be useful (*i.e.,* select a winner between two given tests with the same approach level and branch distance) and the effect that this secondary objective has on branch coverage improvement (Spearman's $\rho = 0.4$ with a p-value $< 0.6e - 10$). Hence, in any case that *BBC* exposes that one generated test is closer to the target line than another test with the same approach level and branch distance (due to the implicit branch occurrence), there is a considerable chance that it helps the search-based test generation process to generate tests with higher branch coverage.

To confirm if this observed correlation stems from the connection between the potential implicit branches in the middle of basic blocks and improvement in the branch coverage, we manually analyzed some cases in which *BBC* application leads to statistically significant improvement in branch coverage achieved by the generated test. In this manual analysis, we identified multiple potential implicit exceptions before the target lines and branches, which are only covered by tests generated by utilizing *BBC* as a secondary objective.



(a) Branch coverage.

(b) $\hat{A}_{12}(BBC_{Pr}, \text{DynaMOSA})$ magnitudes with a positive (count > 0) and negative (count < 0) effect and a $p - value < 0.01$

Fig. 3 Branch coverage of the tests generated for the 219 classes under test (out of 30 executions) for different configurations of *BBC*. The square (□) denotes the arithmetic mean, the bold line (—) is the median

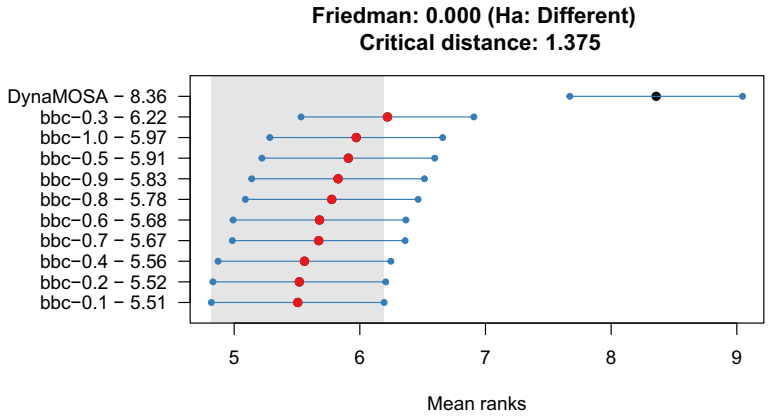


Fig. 4 Non-parametric multiple comparisons of the branch coverage using Friedman's test with Nemenyi's post-hoc procedure

For instance, for the class under test `com.fasterxml.jackson.databind.node.TreeTraversingParser` in `JacksonDatabind-106`, we see that tests generated by *BBC* configurations achieve a higher structural coverage against DynaMOSA. In the majority of runs, the tests generated by *BBC* managed to cover Lines 6 to 11 in method `nextToken()` (Listing 4), while DynaMOSA is not successful in covering these lines. By looking at method `nodeCursor.iterateChildren()` (Listing 5), which is called by `nextToken()` in line 6 of Listing 4, we see that this method may throw an `IllegalStateException` at lines 4 and 12. Since DynaMOSA does not have any information about the branches in the other classes other than the class under test, it cannot guide the search process to execute the method `iterateChildren()` without raising an exception.

Output coverage and implicit exception coverage (RQ 1.2) The improvement of branch coverage also leads to more output diversity, reported in Fig. 5a: from 54.2% for DynaMOSA up to 55.5% for *BBC* 0.8. This improvement is also systematic across all *BBC* configurations according to the effect sizes reported in Fig. 5b. *BBC* 0.6 give the best results

```
1  public JsonToken nextToken(){
2      [...]
3      if (_startContainer) {
4          _startContainer = false;
5          [...]
6          _nodeCursor = _nodeCursor.iterateChildren();
7          _currToken = _nodeCursor.nextToken();
8          if ([...]) {
9              _startContainer = true;
10         }
11             return _currToken;
12     }
13     [...]
14 }
```

Listing 4 method `nextToken()` from `JacksonDatabind-106`

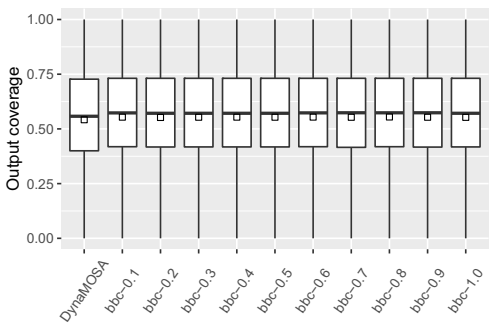
```
public final NodeCursor iterateChildren() {
    [...]
    if (n == null){
        throw new IllegalStateException("No current node");
    }
    if (n.isArray()) {
        [...]
    }
    if (n.isObject()) {
        [...]
    }
    throw new IllegalStateException([...]);
}
```

Listing 5 method `iterateChildren()` in `JacksonDatabind-106`

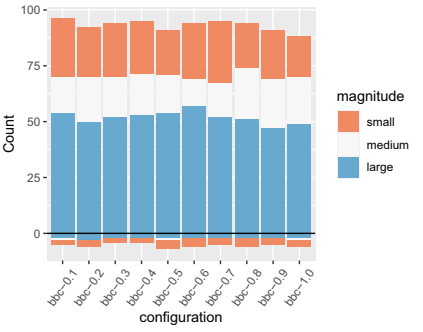
with a *large* positive ($\hat{A}_{12} > 0.5$) effect size for 57 classes under test each (against 2 *large* negative, $\hat{A}_{12} < 0.5$, effect sizes each), followed by *BBC* 0.1 and 0.5 with 54 classes (against 2 classes), and *BBC* 0.4 with 53 classes (against 2 classes).

The two target classes with *large* negative effect sizes on the output coverage are the same classes for the different configurations of *BBC*: *i.e.,* different versions of the class `org.apache.commons.cli.HelpFormatter` in `Cli-31` and `Cli-32`. Interestingly, all *BBC* configurations achieve a statistically significant higher implicit runtime exception coverage (*i.e.,* undeclared runtime exceptions not explicitly thrown by a `throw new` instruction) with a large effect size for the same class on the same buggy versions of `Cli`, indicating that for this particular class, the loss of coverage of output values is compensated by a higher number of methods throwing implicit runtime exceptions.

This could be explained by the fact that *BBC* favors test cases with a higher coverage of basic blocks, but that are not able to reach the return statements of the methods under test (*e.g.,* if the values used by the test cause implicit runtime exceptions). There is however no general correlation between the output coverage and the implicit exception coverage (Spearman's $\rho = -0.008$ with a p-value < 0.001).



(a) Output coverage.



(b) $\hat{A}_{12}(BBC_{Pr}, \text{DynaMOSA})$ magnitudes with a positive (count > 0) and negative (count < 0) effect and a $p - value < 0.01$

Fig. 5 Output coverage of the tests generated for the 219 classes under test (out of 30 executions) for different configurations of *BBC*. The square (□) denotes the arithmetic mean, the bold line (—) is the median

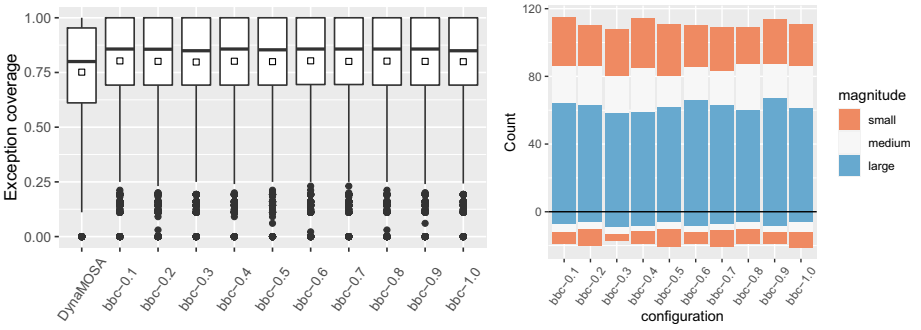
Same as RQ 1.1, we evaluated the correlation between the improvement of *BBC* in terms of output coverage and *BBC* usefulness (presented in RQ 0). This analysis shows a positive correlation between these two metrics (Spearman's $\rho = 0.3$ with a p-value $< 0.1e-5$). As we explained, this observation stems from the correlation between branch coverage and the output coverage achieved by each test: covering more lines and branches increases the chance of seeing more diverse output from CUT. To support this hypothesis, we also checked if there is a correlation between branch coverage and output coverage. Our analysis shows that branch coverage and output coverage are strongly correlated (Spearman's $\rho = 0.6$ with a p-value $< 0.3e-16$).

Figure 6a reports the implicit runtime exception coverage of the generated tests. Implicit exceptions are not declared in the method under test and are triggered when implicit branches are executed. Results show that the average exception coverage increases when using *BBC* as a secondary objective: from 75.1% when using DynaMOSA up to 80.3% for *BBC* 0.1 and 0.6. *BBC* 0.9 gives the best results with a *large* positive ($\hat{A}_{12} > 0.5$) effect size for 67 classes under test (against 8 *large* negative, $\hat{A}_{12} < 0.5$, effect size), followed by *BBC* 0.6 with 66 classes (against 8 classes), and *BBC* 0.1 with 64 classes (against 7 classes).

The rankings in Fig. 7 indicate that *BBC* 0.1 to 1.0 perform well, with an average rank much smaller than the baseline, both for output and exception coverage. The configurations' average ranks differences with the average rank of the baseline are larger than the critical distance $CD = 1.375$ determined by Nemenyi's post-hoc procedure.

In contrast with branch coverage and output coverage, Spearman's test does not show any general correlation between *BBC* usefulness and implicit exception coverage (Spearman's $\rho = 0.04$ with a p-value $= 0.5$). This result supports our discussion in Section 3: since *BBC* is only triggered when DynaMOSA compares tests regarding a line or branch coverage search objective, it does not have any negative impact on other search objectives, including the implicit exception coverage of the generated tests. We also analyzed some of the exceptions that are only thrown by the tests generated using *BBC*. The remainder of this section explains one of these instances.

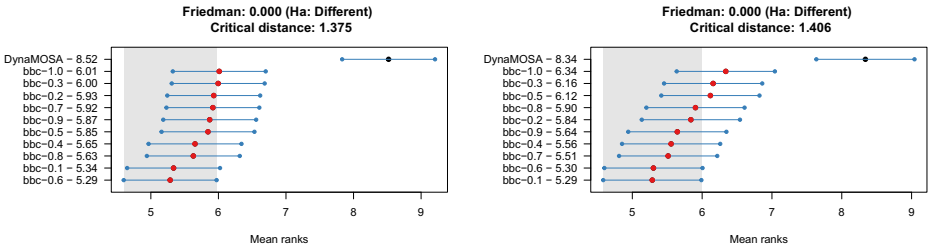
Listing 6 shows an example of an implicit exception that is thrown significantly more often when using *BBC*. DynaMOSA managed to capture this exception in 9 our of 30 runs, while *BBC* 0.5 captured it in 23 out of 30 runs. This exception occurs in line 846



(a) Exception coverage.

(b) $\hat{A}_{12}(BBC_{Pr}, \text{DynaMOSA})$ magnitudes with a positive (count > 0) and negative (count < 0) effect and a $p - value < 0.01$

Fig. 6 Exception coverage of the tests generated for the 219 classes under test (out of 30 executions) for different configurations of *BBC*. The square (□) denotes the arithmetic mean, the bold line (—) is the median



(a) Output coverage. (b) Exception coverage.

Fig. 7 Non-parametric multiple comparisons of the coverage using Friedman's test with Nemenyi's post-hoc procedure

of method `linearCombination` (Listing 7). This exception can be triggered only in one specific case where the input arrays (`a` and `b`) both contain only one element. If these two parameters do not have the same size, this method throws an *explicit* exception at line 838 (*i.e.,* this line is formatted as `throw new [...]`). Since EvoSuite can recognize explicit exception throws in the CUT and convert them to explicit branches while generating the control flow graphs, approach level and branch distance can guide the search process to cover other lines after 839 by prioritizing tests that pass two arrays with the same size to method `linearCombination`.

However, since the explicit branch was the only control-dependent branch for the target line (line 846), the search process does not have any guidance to cover the following lines (including the target line). Assume that test `T1` generates input parameters `a` and `b` with size 0. Then, this method throws `ArrayIndexOutOfBoundsException` in one line before the target line (line 845). This implicit branch will be hidden from the approach level and branch distance heuristics. By adding *BBC*, the search process can differentiate these two tests and help the search process to generate tests that can cover the following lines more often. By having more tests that can cover the target line, the search process has a higher opportunity to execute the target line, and thereby find the exception in this line.

Weak mutation score and real faults (RQ 1.3) As for branch and output coverage, activating *BBC* slightly improves the weak mutation score of the generated tests (reported in Fig. 8a). *BBC* 0.4, 0.6 and 0.8 achieve the higher average mutation score with 74.6%, compared to 73.2% for the baseline (DynaMOSA). That improvement is also systematic across the different configurations of *BBC* according to the effect sizes reported in Fig. 8b. *BBC* 0.5 gives the best results with a *large* positive ($\hat{A}_{12} > 0.5$) effect size for 54 classes under test (against 0 *large* negative, $\hat{A}_{12} < 0.5$, effect size), followed by *BBC* 0.2 with 53 classes (against 0 class), and *BBC* 0.4, 0.6, 0.7 and 0.9 with 51 classes each (against 0 class).

Looking at the ranking reported in Fig. 9, *BBC* 0.1 to 1.0 have an average rank much smaller than the baseline. Those differences are larger than the critical distance $CD = 1.375$ determined by Nemenyi's post-hoc procedure.

```
1 java.lang.ArrayIndexOutOfBoundsException: 1
2     at [...].MathArrays.linearCombination([...]:846)
```

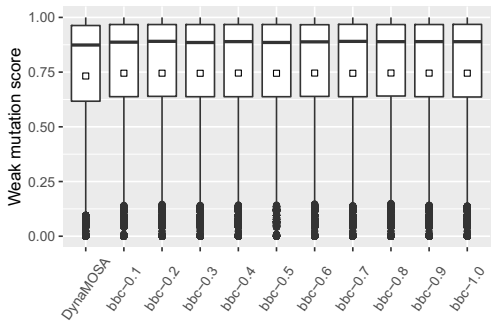
Listing 6 An implicit exception in `MATH-3` which is thrown significantly more often by tests generated by the search process utilizing *BBC* secondary objective

```
834 public static double linearCombination(final double[] a,
        final double[] b)
835 {
836     [...]
837     if (len != b.length) {
838             throw new DimensionMismatchException(len, b.
                length);
839     }
840
841     for (int i = 0; i < len; i++) {
842         [...]
843     }
844
845     final double prodHighCur = prodHigh[0];
846     double prodHighNext = prodHigh[1]; // target line
847     [...]
848 }
```

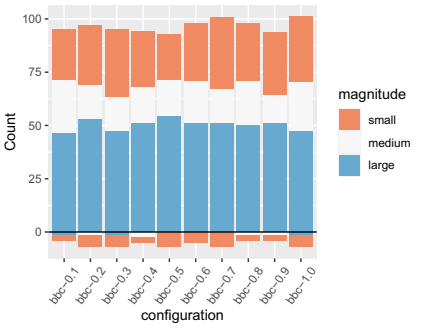
Listing 7 method `linearCombination` from `Apache Commons MATH`

Moreover, we checked if we could find any correlation between the weak mutation score and *BBC* usefulness (presented in RQ 0). This analysis shows a moderate correlation between these two metrics (Spearman's $\rho = 0.37$ with a p-value $< 0.3e-8$). One reason for this correlation could be the strong correlation between weak mutation score and branch coverage (Spearman's $\rho = 0.91$ with a p-value $< 0.3e-16$). Thanks to *BBC* secondary objective, the search-based test generation process can cover more lines and branches, thereby killing the mutants in these newly covered lines.

Finally, we compare the fault revealing capabilities of the generated tests using DEFECTS4J. Table 3 presents the results for the different configurations of *BBC* and the baseline (DYNAMOSA). In general, the tests reveal between 25 and 28 faults at least once in 30 rounds of executions out of the 92 faults considered (the selection procedure is detailed in Section 4.1). For the faults that are revealed in at least one round, the average coverage frequency (for 30 rounds of execution) varies between 22.25% (for *BBC* 0.1 and 1.0) and 23.04% (for *BBC* 0.7). The table also reports the number of faults for which a configuration



(a) Weak mutation score.



(b) $\hat{A}_{12}(BBC_{Pr}, \text{DYNAMOSA})$ magnitudes with a positive (count > 0) and negative (count < 0) effect and a $p - value < 0.01$

Fig. 8 Weak mutation score of the tests generated for the 219 classes under test (out of 30 executions) for different configurations of *BBC*. The square (□) denotes the arithmetic mean, the bold line (—) is the median

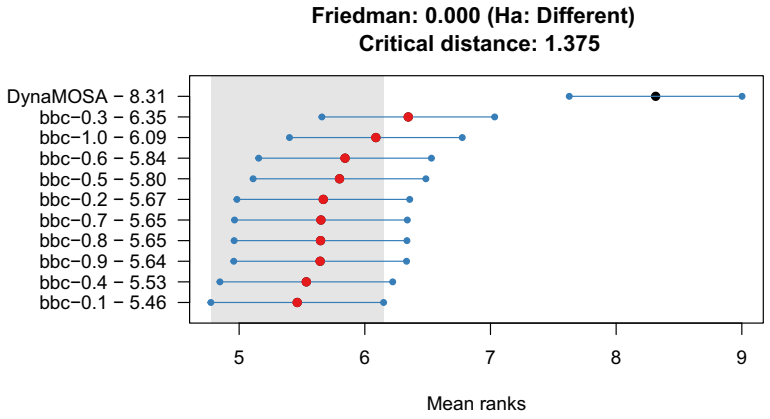


Fig. 9 Non-parametric multiple comparisons of the weak mutation score using Friedman's test with Nemenyi's post-hoc procedure

performed better (odds ratio above 1) or worse (odds ratio below 1) than the DynaMOSA baseline with a significance level of 0.01. The best configuration are *BBC* 0.4, 0.5, 0.6, 0.8, and 1.0 with 3 faults (against 0).

We manually analyzed the three faults that are captured significantly more often by *BBC*. In all of them, we identified potential implicit branches before covering the target line (*i.e.,* the line in which the fault happens) that can prevent the classical and approach level from guiding the search process towards covering these failures.

For instance, Listing 8 presents the stack trace that reveals a fault in `JFreeChart`.[2] When selecting the `XYPlot` class as class under test, *BBC* configurations can throw this exception significantly more often than tests generated by DynaMOSA. This stack trace has five frames that are pointing to a method in the target class (`XYPlot`): Lines 1, 4, 5, 6, and 7 in Listing 8. By analyzing the methods in these frames, we can see that majority of them are simple methods with one line except the first frame in Line 1 of Listing 8, which points to method `getDataRange` that has about 100 lines of codes.

As we can see in Listing 9, the target line, in which the `NullPointerException` occurs (Line 4493), is in an `if` condition which starts at Line 4472. The target line is directly control-dependent on this condition. Hence, when a test fulfills the condition in line 4472, the approach level and branch distance heuristics assume that the generated test eventually will cover the target line (Line 4494), and thereby these two heuristics do not provide any guidance for the test generation search process afterward. However, by taking a closer look, we can see that even after entering the `if` condition, a test needs to, first, call the `combine` method (in one of the Lines 4476, 4479, 4485, or 4488) and also call either `findDomainBounds` (in Lines 4476 or 4479) or `findRangeBounds` (in Lines 4485 or 4488) before it can reach the target line. Each of these methods can throw explicit exceptions. Since these methods are not part of the class under test, the search process is unaware of those exceptions. Also, each of these methods calls multiple methods that can also throw exceptions.

[2]See case `CHART-4` in Defects4J at https://github.com/rjust/defects4j/blob/master/framework/projects/Chart/trigger_tests/4

Table 3 Real faults coverage of the different configurations with the number of faults covered at least once in 30 runs (#) out of 92 faults, the average coverage frequency ($\overline{freq.}$, σ), and the number of time a configuration performed better (> 1) of worse (< 1) than DynaMOSA with a significance level of 0.01

Config.	Faults coverage			Odds ratio		
	#	$\overline{freq.}$	σ	> 1	$= 1$	< 1
bbc-0.1	26	22.25%	38.84%	1	-	-
bbc-0.2	27	22.79%	39.18%	2	-	-
bbc-0.3	26	22.28%	39.02%	2	-	1
bbc-0.4	26	22.28%	38.66%	3	-	-
bbc-0.5	25	22.68%	39.36%	3	-	-
bbc-0.6	27	22.46%	38.86%	3	-	-
bbc-0.7	26	23.04%	39.68%	2	-	-
bbc-0.8	28	22.39%	38.75%	3	-	-
bbc-0.9	25	22.57%	38.96%	2	-	-
bbc-1.0	27	22.25%	38.97%	3	-	-
DynaMOSA	26	21.49%	38.37%	-	-	-

BBC can guide the test generation search process to execute these lines without any exception and cover the target line. By covering the target line, the search process has the opportunity to generate a test that throws a `NullPointerException` in this target line, and thereby captures the fault.

Branch coverage efficiency (RQ 1.4). Figure 10a presents the tendency of the branch coverage over time using the smoothed conditional means. Overall, *BBC* 0.5 tends to achieve a higher branch coverage. This is confirmed by the number of classes for which we observe a significant difference (with $\alpha = 0.01$) in the coverage achieved, reported in Fig. 10b and grouped by effect size ($\hat{A}_{12}$) magnitude. Counts above (resp. below) 0 denote the number of classes for which we observe a positive (resp. negative) effect. After three minutes, *BBC* 0.4 achieves a large (resp. medium) positive effect size for 34 (resp. 18) classes under

```
0 java.lang.NullPointerException
1     at org.jfree.chart.plot.XYPlot.getDataRange(XYPlot.java
          :4493)
2     at org.jfree.chart.axis.NumberAxis.autoAdjustRange(
          NumberAxis.java:434)
3     at org.jfree.chart.axis.NumberAxis.configure(NumberAxis.
          java:417)
4     at org.jfree.chart.plot.XYPlot.configureDomainAxes(XYPlot
          .java:972)
5     at org.jfree.chart.plot.XYPlot.setRenderer(XYPlot.java
          :1644)
6     at org.jfree.chart.plot.XYPlot.setRenderer(XYPlot.java
          :1620)
7     at org.jfree.chart.plot.XYPlot.setRenderer(XYPlot.java
          :1607)
```

Listing 8 The fault in `CHART-4` which is captured significantly more often by tests generated by the search process utilizing *BBC* secondary objective

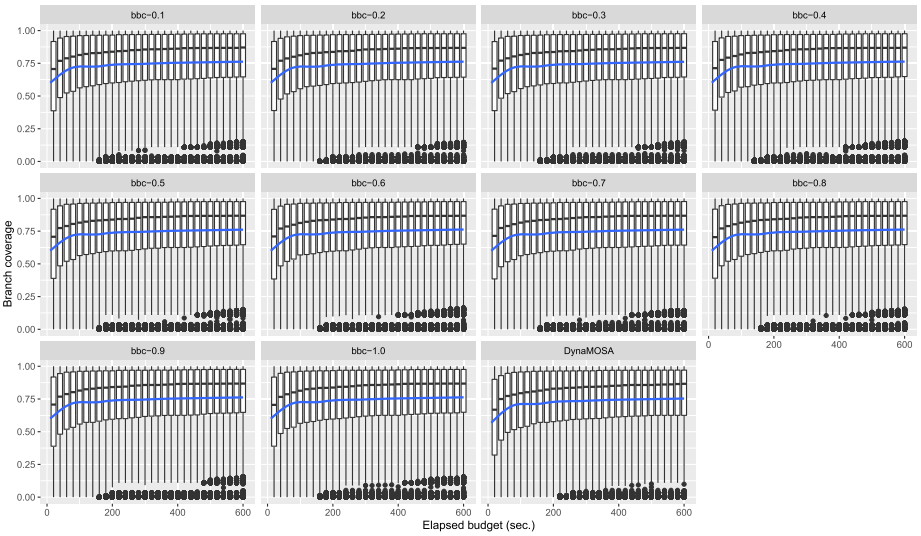
```
public Range getDataRange(ValueAxis axis) {
    [...]
    // iterate through the datasets that map to the axis and
        get the union
    // of the ranges.
    Iterator iterator = mappedDatasets.iterator();
    while (iterator.hasNext())
    {
        XYDataset d = (XYDataset) iterator.next();
        if (d != null) {
            XYItemRenderer r = getRendererForDataset(d);
            if (isDomainAxis) {
                if (r != null) {
                    result = Range.combine(result, r.
                        findDomainBounds(d));
                }
                else {
                    result = Range.combine(result,
                            DatasetUtilities.findDomainBounds
                                (d));
                }
            }
            else {
                if (r != null) {
                    result = Range.combine(result, r.
                        findRangeBounds(d));
                }
                else {
                    result = Range.combine(result,
                            DatasetUtilities.findRangeBounds(
                                d));
                }
            }

            Collection c = r.getAnnotations(); // target line
            [...]
        }
    }
```
```
}
```

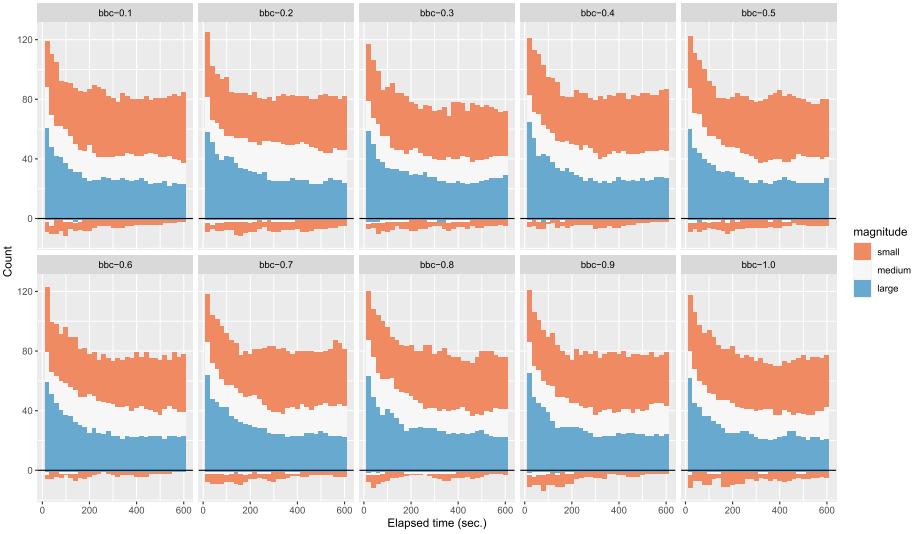
Listing 9 method `getDataRange` from `JFreeChart`

test against 1 (resp. 0) large (resp. medium) negative effect sizes. Those numbers slightly decrease over time with 27 (resp. 18) classes under test with a large (resp. medium) effect size after exhaustion of the ten minutes search budget, for 1 (resp. 0) large (resp. medium) classes with a negative effect size.

Summary (RQ 1) We see an improvement of the branch coverage of the generated tests when activating *BBC* as a secondary objective in DynaMOSA. This improvement in branch coverage also leads to an increase of the output and exception coverage, and of the diversity of runtime states (denoted by an increase of the weak mutation score). Among the different configurations, *BBC* 0.5 gives the best results and those results remains stable over time. It also leads to the coverage of three additional faults in Defects4J without any loss compared to the baseline. Giving our results, we can recommend using *BBC* 0.5 as a secondary objective for unit test generation.



(a) Data distribution and smoothed conditional means.

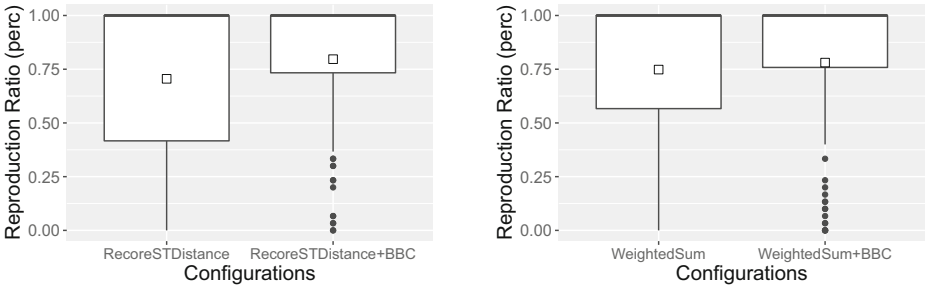


(b) $\hat{A}_{12}(BBC_{Pr}, \text{DynaMOSA})$ magnitudes evolution with a positive (count > 0) and negative (count < 0) effect and a $p - value < 0.01$.

Fig. 10 Evolution of the branch coverage of the tests generated for the 219 classes under test (out of 30 executions) for different configurations of *BBC*

5.3 Search-based crash reproduction (RQ 2)

Crash reproduction effectiveness (RQ 2.1) Figure 11 presents the crash reproduction ratio of the search processes guided by *STDistance* (Fig. 11a) and *WeightedSum* (Fig. 11b), with and without *BBC* as a secondary objective. This figure shows that, on average, the

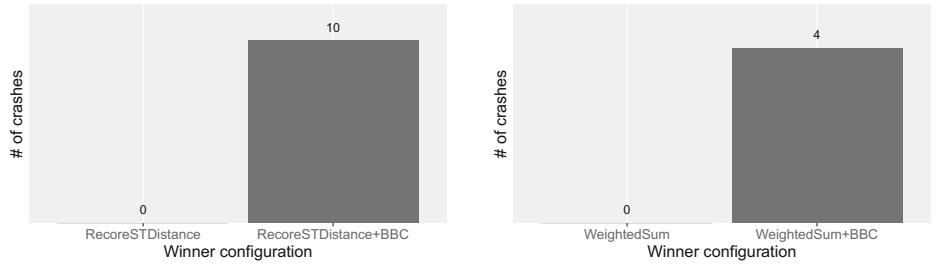


(a) *STDistance* (b) *WeightedSum*

Fig. 11 Crash reproduction ratio (out of 30 executions) of fitness functions with and without *BBC*. The square (□) denotes the arithmetic mean and the bold line (—) is the median

crash reproduction ratio of *WeightedSum* improves 3.3% when using *BBC*. This improvement is higher for crash reproduction using *STDistance*. On average, the crash reproduction ratio achieved by *STDistance* + *BBC* is 9.2% higher than *STDistance* without *BBC*. Higher improvement in *STDistance* was expected as this fitness function relies more on the approach level and branch distance heuristics for covering each of the frames in the given stack trace. Also, in both of the fitness functions, the lower quartile of crash reproduction ratio has been improved by utilizing *BBC*. These improvements for *WeightedSum* and *STDistance* are 19.1% and 31.7%, respectively.

Figure 12 depicts the number of crashes, for which *BBC* has a significant impact on the effectiveness of crash reproduction guided by *STDistance* (Fig. 12a) and *WeightedSum* (Fig. 12b). *BBC* significantly improves the crash reproduction ratio in 10 and 4 crashes for fitness functions *STDistance* and *WeightedSum*, respectively. Notably, the application of this secondary objective does not have any significant adverse effect on crash reproduction. Tables 4 and 5 present the odds ratio and p-value in cases that *BBC* leads to a significant improvement in crash reproduction ratios of *WeightedSum* and *STDistance*, respectively. As we can see in these tables, the odds ratio values in all cases are lower or equal to 0.2, indicating the high impact of *BBC*. Finally, we observed that *BBC* helps each of the *STDistance* and *WeightedSum* to reproduce 3 new crashes that could not be reproduced without this secondary objective.



(a) *STDistance* (b) *WeightedSum*

Fig. 12 Pairwise comparison of impact of *BBC* on each fitness function in terms of crash reproduction ratio with a statistical significance < 0.01

Table 4 Comparing the crash reproduction ratio between crash reproduction using *WS* and *WS + BBC*, for cases where one of the configurations has a significantly higher crash reproduction ratio (p-value < 0.01)

Crash	Reproduction ratio		OR	p-value
	WeightedSum	WeightedSum+BBC		
LANG-54b	19	29	0.1	2.4659e-03
XCOMMONS-1057	17	27	0.2	7.4098e-03
XWIKI-12889	17	27	0.2	7.4098e-03
XWIKI-14556	11	24	0.2	1.4306e-03

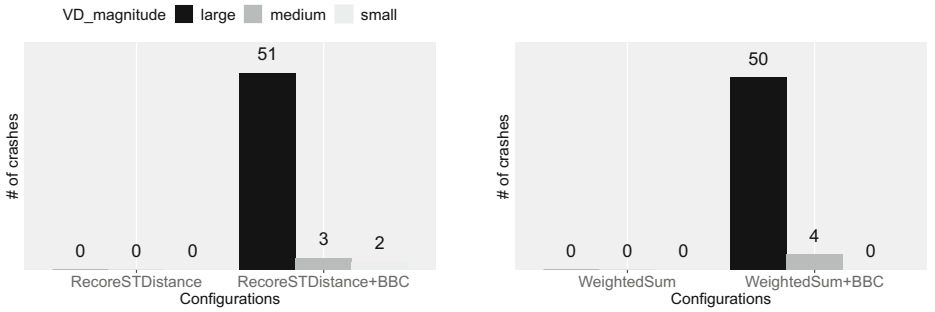
Crash reproduction efficiency (RQ 2.2) Figure 13 illustrates the number of crashes, in which *BBC* significantly affects the time consumed by the crash reproduction search process. As Fig. 13b shows, *BBC* significantly improves the speed of crash reproduction guided by *WeightedSum* in 54 crashes (43.5% of cases), while it does not lose efficiency in the reproduction of any crash.

Similarly, Fig. 13a shows that *BBC* has a higher positive impact on the efficiency of the search process guided by *STDistance*. It significantly reduces the time consumed by the search process in 56 crashes (45.1% of cases), while it had no adverse impact on the reproduction efficiency of any crash.

Figure 14 depicts the average improvements in the efficiency and effect sizes for crashes where the difference in the consumed budget when using *BBC* or not was significant. According to the right-side plot in Fig. 14a, *BBC* reduces the time consumed by the search process guided by *STDistance* up to 98% (being 71.7% on average). Also, the left-side plot indicates that the average effect size of differences between *STDistance* and *STDistance+BBC* (calculated by Vargha-Delaney) is 0.102 (lower than 0.5 indicates that *BBC* improved the efficiency). Figure 14b shows that the average improvement (right-side plot)

Table 5 Comparing the crash reproduction ratio between crash reproduction using *STD* and *STD + BBC*, for cases where one of the configurations has a significantly higher crash reproduction ratio (p-value < 0.01)

Crash	Reproduction ratio		OR	p-value
	RecoreSTDistance	RecoreSTDistance+BBC		
LANG-54b	20	29	0.1	5.5791e-03
MATH-78b	10	21	0.2	9.2060e-03
TIME-7b	1	12	0.1	1.0508e-03
XWIKI-12667	16	30	0.0	1.6767e-05
XWIKI-13141	13	27	0.1	2.5073e-04
XWIKI-13196	19	30	0.0	3.1881e-04
XWIKI-13316	17	29	0.0	4.3102e-04
XWIKI-13916	19	30	0.0	3.1881e-04
XWIKI-14152	3	18	0.1	9.4143e-05
XWIKI-14556	0	24	0.0	3.2940e-11



(a) $STDistance$ (b) $WeightedSum$

Fig. 13 Pairwise comparison of impact of *BBC* on each fitness function in terms of efficiency with a small, medium, and large effect size $\hat{A}_{12} < 0.5$ and a statistical significance < 0.01

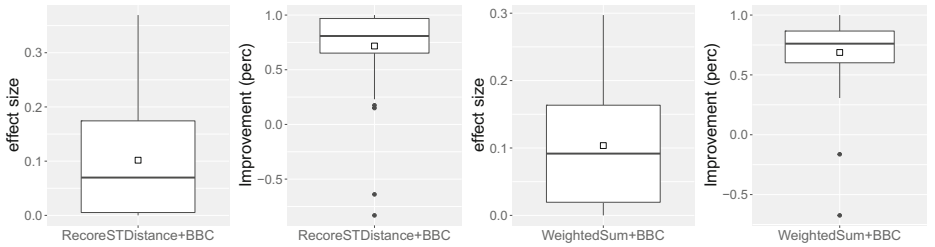
achieved by using *BBC* as the second objective of *WeightedSum* is 68.7%, and the average effect size (left-side plot), in terms of the crash reproduction efficiency, is 0.104.

Summary (RQ 2) *BBC* improves the crash reproduction ratio for both of the *WeightedSum* and *STDistance* fitness functions. This imnprovement is higher for *STDistance* as this fitness function is more relied on approach level and branch distance. Moreover, *BBC* improves the efficiency of the search process with both of the crash reproduction fitness functions.

6 Discussion

6.1 *BBC* for unit test generation

Increase in program state and return value diversity Using *BBC* as a secondary objective leads to a better branch coverage. Although small on average, the improvement is systematic, as demonstrated by the effect sizes. More interestingly, *BBC* also leads to a better output and implicit exception coverage. This is particularly interesting in a unit testing context because it allows to capture more diverse returned values (including implicit



(a) $STDistance$ (b) $WeightedSum$

Fig. 14 The effect size and the average improvement achieved by *BBC* on each of the fitness functions in cases that *BBC* makes a significant difference in terms of efficiency

exceptions) from the methods under test. We observe the same trends for weak mutation, denoting more diverse program states. Although the evaluation of the quality of the generated tests is outside of the scope of this study, we believe that diverse return values and program states can have a positive impact on the quality of the generated assertions, which is one of the known current limitations preventing the large industrial adoption of search-based unit test generation (Almasi et al. 2017).

Adaptive secondary objectives As explained in Section 3.3, applying *BBC* can be expensive ($\mathcal{O}(N \times E \times log\, V)$), compared to classical secondary objectives (linear time). Therefore, *BBC* should be activated only when it can effectively contribute to decide between two test cases with the same fitness value. As shown by our preliminary analysis, this is especially relevant in the context of unit test generation, where each branch should be covered, which could trigger a high number of *BBC* evaluations. In our implementation of *BBC* for unit testing (described in Section 3.3.2), we limit the number of activations of *BBC*, based on the activation time of an objective (SLEEP TIME) and a user-defined probability (USAGE RATE). This approach might however not be optimal. For instance, for classes under test with a high number of implicit branches, activating *BBC* sooner and more often might improve the search process. In our future work, we will explore how the secondary objective can be dynamically adapted during the search, for instance, based on the evolution of the fitness values of the different objectives in DYNAMOSA.

6.2 *BBC* for crash reproduction

Generally, using *BBC* as secondary objective leads to a better crash reproduction ratio and higher efficiency in search-based crash reproduction. This improvement is achieved thanks to the additional ability to guide the search process when facing implicit branches during the search. Combining *BBC* with *STDistance* shows an important improvement compared to the combination of *BBC* with *WeightedSum*. This result was expected, since only one (out of three) component in *WeightedSum* is allocated to line coverage, and thereby most parts of the fitness function do not use the approach level and branch distance heuristics. In contrast, *STDistance* uses the approach level and branch distance to cover each of the frames in the given stack trace incrementally.

Our results show that *BBC* helps the crash reproduction process to reproduce new crashes. For instance, the crash that we used in this study (XWIKI-13377) can be reproduced only by *STDistance* + *BBC*.

6.3 *BBC* and testability transformations

In this study, we tried to evaluate TT in DYNAMOSA. However, EVOSUITE failed before starting the search process for all the different classes under test. After a deeper investigation, we found out that TT is not compatible with DYNAMOSA, which is the default search algorithm in EVOSUITE. Moreover, as discussed in Section 2.2, TT faces extra challenges while it needs extra bytecode instrumentation.

In theory, giving the nature of TT and *BBC*, these two techniques can be applied simultaneously to the search process. Hence, these two approaches can complement each other to achieve high structural coverage and detect more faults. Studying the impact of using both TT and *BBC* on search-based test generation calls for further implementation and efforts, and thereby, it is part of our future research agenda.

7 Threats to Validity

Internal validity We cannot guarantee that our implementation of *BBC* in EvoSuite and Botsing is bug-free. However, we mitigated this threat by testing our implementations and manually examining some samples of the results. Moreover, following the guidelines of the related literature (Arcuri and Briand 2014), we executed each configuration 30 times to take the randomness of the search process into account.

External validity We cannot ensure that our results are generalizable to all cases. However, for both of our experiments for unit test generation and crash reproduction, we have used two earlier established benchmarks: JCrashPack (Soltani et al. 2020), which is a benchmark for crash reproduction containing 124 hard-to-reproduce crashes provoked by real bugs in a variety of open-source applications, and Defects4J (Just et al. 2014), a collection of real-world Java projects failures containing 835 bugs.

To increase the external validity while maintaining a good balance between the statistical power and the overall execution, analysis, and reporting time, we choose to consider only the ten most recent bugs from the 17 projects available in Defects4J. After filtering out classes that cannot be handled by EvoSuite, we ran our evaluation on 219 classes. Among those 219 classes, 44 come from different versions of the same projects. Although involved in different bugs, those classes might be similar and influence our results. To mitigate this threat, we performed a qualitative analysis to confirm the effect of *BBC*.

Construct validity For unit test generation (**RQ 1**), we left the parameters of DynaMOSA to their default values used by EvoSuite. Those values are commonly used in the literature and it has been empirically shown that they give good results (Panichella et al. 2018a, b; Arcuri and Fraser 2013; Fraser and Arcuri 2014). We can, however, not guarantee that these default values are the best when used with *BBC*. Nevertheless, our results show that *BBC* can improve search-based unit test generation when using the default parameter values.

For search-based crash reproduction (**RQ 2**), we used *BBC* with two different fitness functions and left other parameters to their default values, used in previous studies (Soltani et al. 2018; Derakhshanfar et al. 2020). Those studies do not investigate the sensitivity of search-based crash reproduction to these values, and tuning these parameters should be undertaken as future work. However, as for unit test generation, our results show that *BBC* can improve search-based crash reproduction with the default parameter values.

Conclusion validity We based our conclusion on standard statistical analysis for significance (Arcuri and Briand 2014) with $\alpha = 0.01$. Effects of multiple comparisons are mitigated by adjusting $p - values$ via Nemenyi's post-hoc procedure (Japkowicz and Shah 2011; Panichella 2021). Furthermore, we complemented our quantitative analysis with qualitative investigations to confirm the observed effects.

Verifiability Finally, we openly provide all our implementations: Botsing [3], as an open-source crash reproduction tool, and implementation of *BBC* on EvoSuite [4]. Also, the data and the processing scripts used to present the results are available as two replication packages on Zenodo (Derakhshanfar and Devroey 2020; 2021).

[3] https://github.com/STAMP-project/botsing
[4] https://github.com/pderakhshanfar/evosuite

8 Conclusion and Future Work

Approach level and branch distance are two well-known heuristics, widely used by search-based test generation approaches to guide the search process towards covering target statements and branches. These heuristics measure the distance of a generated test from covering the target using the coverage of control dependencies. However, these two heuristics do not consider implicit branches. For instance, if a test throws an exception during the execution of a non-branch statement, approach level and branch distance cannot guide the search process to tackle this exception. In this paper, we extended our previous work on Basic Block Coverage (*BBC*), a secondary objective addressing this issue. We complemented our previous study into *BBC* on search-based crash reproduction with an investigation of *BBC* for unit test generation.

Our results show that *BBC* improves the branch coverage for unit test generated using DynaMOSA. Although small (∼1%), this improvement in the branch coverage is systematic and leads to an increase of the output and implicit runtime exception coverage, and of the diversity of runtime states. *BBC* also helps *STDistance* and *WeightedSum* to reproduce 6 and 1 new crashes, respectively. Finally, *BBC* significantly improves the efficiency in 26.6% and 13.7% of the crashes using *STDistance* and *WeightedSum*, respectively.

An important implication of our work for future research is that we need to investigate secondary search objectives that can be *dynamically* activated depending on the software under test. In this work, we applied the activation mechanism for secondary search objectives (*BBC*) based on user-provided (static) meta-parameters. We have seen indications that such a mechanism can both improve the search process and at the same time reduce the computational cost, yet it can be counter-productive in some cases. We envision that *BBC* and other secondary objectives would benefit from an adaptive activation, depending on the runtime behavior (*e.g.,* if the number of implicit runtime exceptions increases) or structure (*e.g.,* high coupling or deep inheritance hierarchy) of the classes under test.

In our future work, we will investigate the application of *BBC* for other search-based test generation techniques (such as testability transformations, and system and integration testing), as well as the implications of an increase of the diversity of program states in the generated unit tests (*e.g.,* for assertions generation). We will also investigate how *BBC* can be dynamically activated using an adaptive secondary objectives approach to reduce the computational overload on the search process.

Acknowledgements This research was partially funded by the EU Horizon 2020 ICT-10-2016-RIA "STAMP" project (No.731529), the EU Horizon 2020 H2020-ICT-2020-1-RIA "COSMOS" project (No.957254), and the Vici "TestShift" project (No. VI.C.182.032) from the Dutch Science Foundation NWO.

References

Allen FE (1970) Control flow analysis. ACM SIGPLAN Notices 5(7):1–19. https://doi.org/10.1145/390013.808479

Almasi MM, Hemmati H, Fraser G, Arcuri A, Benefelds J (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, pp 263–272

Alshahwan N, Harman M (2014) Coverage and fault detection of the output-uniqueness test selection criteria. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014. ACM Press, pp 181–192

Arcuri A (2019) RESTful API automated test case generation with evomaster. ACM Trans Softw Eng Methodol (TOSEM) 28(1):1–37

Arcuri A, Briand L (2014) A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw Test Verif Reliab 24(3):219–250. https://doi.org/10.1002/stvr.1486

Arcuri A, Fraser G (2013) Parameter tuning or default values? An empirical investigation in search-based software engineering. Empir Softw Eng 18(3):594–623. https://doi.org/10.1007/s10664-013-9249-9

Le T-DB, Lo D, Le Goues C, Grunske L (2016) A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp 177–188

Borba P, Cavalcanti A, Sampaio A, Woodcook J (2010) Testing techniques in software engineering: Second pernambuco summer school on software engineering, psse 2007, revised lectures, vol 6153. Springer, Recife

Campos J, Ge Y, Albunian N, Fraser G, Eler M, Arcuri A (2018) An empirical evaluation of evolutionary algorithms for unit test suite generation. Inf Softw Technol 104:207–235. https://doi.org/10.1016/j.infsof.2018.08.010

Chen N, Kim S (2015) STAR: Stack trace based automatic crash reproduction via symbolic execution. IEEE Trans Softw Eng 41(2):198–220. https://doi.org/10.1109/TSE.2014.2363469

Derakhshanfar P, Devroey X (2020) Replication package of Basic Block Coverage for Search-Based Crash Reproduction. https://doi.org/10.5281/zenodo.3953519

Derakhshanfar P, Devroey X (2021) pderakhshanfar/EMSE-BBC-experiment: Replication package for EMSE journal extension. https://doi.org/10.5281/zenodo.4665874

Derakhshanfar P, Devroey X, Panichella A, Zaidman A, van Deursen A (2020) Towards integration-level test case generation using call site information. arXiv:2001.04221

Derakhshanfar P, Devroey X, Perrouin G, Zaidman A, Deursen A (2020) Search-based crash reproduction using behavioural model seeding. STVR 30(3):e1733. https://doi.org/10.1002/stvr.1733

Derakhshanfar P, Devroey X, Zaidman A (2020) It Is Not Only About Control Dependent Nodes: Basic Block Coverage for Search-Based Crash Reproduction. In: Aleti A, Panichella A (eds) Search-Based Software Engineering - 12th International Symposium, SSBSE 2020. Springer, pp 42–57

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: 35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20). IEEE / ACM, pp 211–223

Devroey X, Panichella S, Gambi A (2020) Java Unit Testing Tool Competition - Eighth Round. In: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ACM, pp 545–548

Fraser G, Arcuri A (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, ESEC/FSE '11. ACM, New York, pp 416–419

Fraser G, Arcuri A (2013) Evosuite: On the challenges of test case generation in the real world. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. IEEE, pp 362–369

Fraser G, Arcuri A (2013) Whole test suite generation. IEEE Trans Softw Eng 39(2):276–291. https://doi.org/10.1109/TSE.2012.14

Fraser G, Arcuri A (2014) A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans Softw Eng Methodol 24(2):1–42. https://doi.org/10.1145/2685612

Fraser G, Arcuri A (2015) 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. Empir Softw Eng 20(3):611–639

Fraser G, Arcuri A (2015) Achieving scalable mutation-based generation of whole test suites. Empir Softw Eng 20(3):783–812. https://doi.org/10.1007/s10664-013-9299-z

García S, Molina D, Lozano M, Herrera F (2009) 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. J Heurist 15(6):617–644. https://doi.org/10.1007/s10732-008-9080-4

Howden WWE (1982) Weak Mutation Testing and Completeness of Test Sets. IEEE Trans Softw Eng SE-8(4):371–379. https://doi.org/10.1109/TSE.1982.235571

Japkowicz N, Shah M (2011) Evaluating learning algorithms: a classification perspective. Cambridge University Press

Just R, Jalali D, Ernst MD (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, pp 437–440

Kifetew F, Devroey X, Rueda U (2019) Java Unit Testing Tool Competition - Seventh Round. In: 2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST). IEEE, pp 15–20

Li Y, Fraser G (2011) Bytecode testability transformation. In: International Symposium on Search Based Software Engineering. Springer, pp 237–251

Lu Y, Lou Y, Cheng S, Zhang L, Hao D, Zhou Y, Zhang L (2016) How does regression test prioritization perform in real-world software evolution? In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). IEEE, pp 535–546

Ma L, Artho C, Zhang C, Sato H, Gmeiner J, Ramler R (2015) Grt: Program-analysis-guided random testing (t). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 212–223

Martinez M, Monperrus M (2016) ASTOR: a program repair library for Java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp 441–444

McMinn P (2004) Search-based software test data generation: A survey. Softw Test Verif Reliab 14(2):105–156. https://doi.org/10.1002/stvr.294

McMinn P (2011) Search-based software testing: Past, present and future. In: Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW '11. IEEE Computer Society, Washington, pp 153–163

Molina UR, Kifetew F, Panichella A (2018) Java Unit Testing Tool Competition - Sixth Round. In: Proceedings of the 11th International Workshop on Search-Based Software Testing - SBST '18. ACM, pp 22–29

Nayrolles M, Hamou-Lhadj A, Tahar S, Larsson A (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, pp 101–110

Noor TB, Hemmati H (2015) A similarity-based approach for test case prioritization using historical failure data. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, pp 58–68

Offutt AJ, Lee SD (1994) An empirical evaluation of weak mutation. IEEE Trans Softw Eng 20(5):337–344. https://doi.org/10.1109/32.286422

Panichella A (2021) A Systematic Comparison of search-Based approaches for LDA hyperparameter tuning. Inf Softw Technol 130:106411. https://doi.org/10.1016/j.infsof.2020.106411

Panichella A, Kifetew FM, Tonella P (2018a) A large scale empirical comparison of state-of-the-art search-based test case generators. Inf Softw Technol 104:236–256. https://doi.org/10.1016/j.infsof.2018.08.009

Panichella A, Kifetew FM, Tonella P (2018b) Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans Softw Eng 44(2):122–158. https://doi.org/10.1109/TSE.2017.2663435

Papadakis M, Malevris N (2011) Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Softw Qual J 19(4):691–723. https://doi.org/10.1007/s11219-011-9142-y

Pearson S, Campos J, Just R, Fraser G, Abreu R, Ernst MD, Pang D, Keller B (2017) Evaluating and improving fault localization. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press, pp 609–620

Rojas JM, Campos J, Vivanti M, Fraser G, Arcuri A (2015) Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: Search-Based Software Engineering (SSBSE 2015), LNCS, vol 9275. Springer, pp 93–108

Rößler J, Zeller A, Fraser G, Zamfir C, Candea G (2013) Reconstructing core dumps. In: Proceedings of International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp 114–123

Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A (2015) Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges. In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp 201–211

Smith EK, Barr ET, Le Goues C, Brun Y (2015) Is the cure worse than the disease? overfitting in automated program repair. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering. ACM, pp 532–543

Soltani M, Derakhshanfar P, Devroey X, van Deursen A (2020) A benchmark-based evaluation of search-based crash reproduction. Empir Softw Eng 25(1):96–138. https://doi.org/10.1007/s10664-019-09762-1

Soltani M, Derakhshanfar P, Panichella A, Devroey X, Zaidman A, van Deursen A (2018) Single-objective Versus Multi-objectivized Optimization for Evolutionary Crash Reproduction. In: Colanzi TE, McMinn P (eds) Symposium on Search-Based Software Engineering. SSBSE 2018, LNCS, vol 11036. Springer, Montpellier, pp 325–340

Soltani M, Panichella A, Van Deursen A (2018) Search-based crash reproduction and its impact on debugging. IEEE Trans Softw Eng. https://doi.org/10.1109/TSE.2018.2877664

Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. J Educ Behav Stat 25(2):101–132

Xiao X, Xie T, Tillmann N, De Halleux J (2011) Precise identification of problems for structural test generation. In: Software Engineering (ICSE), 2011 33rd International Conference on. ACM, Waikiki, pp 611–620

Xuan J, Xie X, Monperrus M (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, pp 910–913

Zeller A (2009) Why programs fail, second edition: A guide to systematic debugging, 2nd edn. Morgan Kaufmann Publishers Inc., San Francisco

Pouria Derakhshanfar is a postdoctoral researcher in the Software Engineering Research Group (SERG) of the Delft University of Technology. He received his PhD in Computer Software Engineering from the Delft University of Technology in 2021. His main research interests include search-based software engineering, software test generation, and software engineering for Cyber-physical systems. He is part of the H2020 funded project called DevOps for Complex Cyber-physical Systems (COSMOS), where he studies the approaches to improve the software quality in Cyber-physical Systems. Earlier, he was involved in another EU H2020 funded project, called Software Testing AMPlification (STAMP), where he developed new search-based approaches for crash replication and test amplification in a DevOps context.

Xavier Devroey is an assistant professor of software engineering at the Namur Digital Institute of the University of Namur, Belgium. His main research interests include search-based and model-based software testing, test suite augmentation, and variability-intensive systems. He received his PhD in Computer Science from the University of Namur in 2017. He worked as a postdoctoral researcher in the software engineering research group of the Delft University of Technology from 2017 to 2021.

Andy Zaidman is a full professor in software engineering at Delft University of Technology, The Netherlands. He received the MSc and PhD degrees in Computer Science from the University of Antwerp, Belgium, in 2002 and 2006, respectively. His main research interests include software evolution, program comprehension, mining software repositories, software quality, and software testing. He is an active member of the research community and involved in the organization of numerous conferences (WCRE'08, WCRE'09, VISSOFT'14 and MSR'18). In 2013 he was the laureate of a prestigious Vidi mid-career grant, while in 2019 he received the most prestigious Vici career grant from the Dutch science foundation NWO.

Affiliations

Pouria Derakhshanfar[1] · Xavier Devroey[2] · Andy Zaidman[1]

Xavier Devroey
xavier.devroey@unamur.be

Andy Zaidman
a.e.zaidman@tudelft.nl

[1] Delft University of Technology, Postbus 5, 2600 AA, Delft, The Netherlands

[2] Namur Digital Institute, University of Namur, rue de Bruxelles 61, 5000 Namur, Belgium