An Experimental Assessment of Using Theoretical Defect Predictors to Guide Search-Based Software Testing

Anjana Perera, Aldeida Aleti, Burak Turhan, and Marcel Böhme

Abstract—Automated test generators, such as search-based software testing (SBST) techniques are primarily guided by coverage information. As a result, they are very effective at achieving high code coverage. However, is high code coverage alone sufficient to detect bugs effectively? In this paper, we propose a new SBST technique, predictive many objective sorting algorithm (PreMOSA), which augments coverage information with defect prediction information to decide where to increase the test coverage in the class under test (CUT). Through an experimental evaluation using 420 labelled bugs on the Defects4J benchmark and using theoretical defect predictors, we demonstrate the improved effectiveness and efficiency of PreMOSA in detecting bugs when using any acceptable defect predictor, i.e., a defect predictor with recall and precision $\geq 75\%$, compared to the state-of-the-art dynamic many objective sorting algorithm (DynaMOSA). PreMOSA detects up to 8.3% more labelled bugs on average than DynaMOSA when given a time budget of 2 minutes for test generation per CUT.

Index Terms—Search-Based Software Testing, Automated Test Generation, Defect Prediction

✦

1 INTRODUCTION

SEARCH-BASED software testing (SBST) techniques consider test cases with high code coverage as high quality test cases, and aim at maximising code coverage [1], [2], [3]. As a result, they are very effective at achieving high code coverage [4]. A test suite with high code coverage, however is not sufficient to effectively detect bugs in a program. Previous work shows that SBST techniques have limitations in terms of detecting bugs [5], [6], [7]. For example, DynaMOSA [3], a state-of-the-art SBST technique, could only detect on average 22% of the bugs from the Defects4J dataset, when it is given a 30 seconds time budget per class and using branch coverage as criterion [8]. In this paper, we hypothesise that we can improve the bug detection performance of SBST by augmenting coverage information used by SBST with defect prediction information.

Defect predictors are well-studied techniques for estimating the bug-prone areas in software. The predictions can be coarse-grained like package [9] and file/class [10], [11] levels, or fine-grained like method level [10], [12], [13]. They use various features related to metrics like code size [14], code complexity [15], change history [16] and organisation [17] to predict whether a package, file or method is defective. Defect predictors have been shown to be effective at locating bugs in software [10], [18], [19]. As a result, organisations use defect predictors to help developers in code reviews [20], [21] and to focus their testing efforts on likely buggy parts in code [22]. In addition, defect prediction has been successfully used to inform other automated testing techniques, i.e., Paterson et al. [23] proposed a test case prioritisation strategy, Perera et al. [8] introduced a time budget allocation approach for SBST, and Hershkovich et al. [24] proposed a strategy to select a subset of all the classes in a project to run test generation. In contrast to existing work, ours is the first to adapt defect prediction information to guide the search process of an SBST technique.

We introduce predictive many objective sorting algorithm (PreMOSA) which uses information from a defect predictor and focuses the search for tests in the likely buggy methods to increase the chances of detecting bugs. PreMOSA starts with coverage targets containing likely buggy methods as predicted by a defect predictor that works at method level [12], [13]. Once it deems to have searched enough for test cases that cover the likely buggy targets, it starts finding tests to cover the likely non-buggy targets in the class under test (CUT). It generates more than one test case for all the selected targets, thus increasing the likelihood of detecting bugs. Finally, to ensure the non-trivial targets have an equal chance of being covered, PreMOSA dynamically balances the test coverage among all the targets in the search. To do this, PreMOSA temporarily removes coverage targets from the search in every iteration based on their current test coverage and number of independent paths.

We evaluate how PreMOSA performs in terms of its effectiveness and efficiency in detecting bugs when compared to the state-of-the-art DynaMOSA. We evaluate PreMOSA on 420 labelled bugs from 6 open source java projects in the

- *Anjana Perera, Aldeida Aleti, and Marcel Böhme are with the Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia. E-mail: {Anjana.Perera, Aldeida.Aleti, Marcel.Boehme}@monash.edu.*
- *Burak Turhan is with the Faculty of Information Technology and Electrical Engineering, University of Oulu, 90014 Oulu, Finland. E-mail: Burak.Turhan@oulu.fi.*

Defects4J dataset. We use theoretical (i.e., simulated) defect predictors that can be replaced with any real defect predictor in practice [12], [13]. We intentionally abstract the defect predictor in PreMOSA to avoid potential confounding effects that can be caused by using a single defect predictor. Our experimental evaluation demonstrates that PreMOSA is significantly more effective than DynaMOSA with large effect sizes when using any acceptable defect predictor, i.e., recall and precision ≥ 0.75 [25]. In particular, PreMOSA detects 8.3% and 7.8% more labelled bugs on average than DynaMOSA when using an ideal defect predictor and most conservative and acceptable defect predictor, respectively. Moreover, we find that PreMOSA is significantly more efficient than DynaMOSA with small effect sizes.

In summary, the contribution of this paper is a novel SBST technique that uses defect prediction information along with coverage information to guide the search process in order to improve the bug detection capability of SBST. In addition, we present an empirical evaluation involving 420 bugs from 6 open source java projects (which took roughly 48,800 hours) that demonstrates the effectiveness and efficiency of our proposed approach, PreMOSA with theoretical defect predictors. The source code of PreMOSA, defect predictor simulator, post processing scripts and data are publicly available in the following link: https://doi.org/10.6084/m9.figshare.19027778

2 PROBLEM STATEMENT AND MOTIVATION

2.1 Problem

Coverage is often used to define the fitness function used in search-based software testing (SBST) [2], [3], [26]. During the search process, test cases with high coverage are considered of higher quality, and the aim of the search process is to generate test cases that maximise coverage. Different fitness function formulations exist based on coverage, such as approach level [3] and branch distance [27], [28]. A notable technique that uses coverage is DynaMOSA [3], which is considered the state-of-the-art SBST approach.

High code coverage, however does not necessarily imply effective bug detection by the test suite [8]. Indeed, previous work shows that SBST techniques have limitations in terms of detecting bugs [5], [6]. Even DynaMOSA could only detect on average 22% of the bugs from the Defects4J dataset in a 30 seconds time budget and using branch coverage as guidance [8].

Our hypothesis is that augmenting coverage information used by SBST approaches with defect prediction information improves the performance of SBST in terms of bug detection. Defect predictors can predict the methods in a class that are likely to be buggy. SBST generates tests for a class, and a class usually has only one or few buggy methods. We argue that increasing the coverage in a large number of non-buggy methods is ineffective in terms of detecting bugs. Therefore, we propose to use defect prediction information in the search process along with coverage information to guide the search for test cases towards likely buggy methods in the class.

2.2 Motivating Example

Fig. 1 shows the buggy code snippet and the applied patch for `DateTimeZone` class from Time-8 bug in Defects4J [29].

```
272 272  public static DateTimeZone forOffsetHoursMinutes(int hoursOffset,
                              int minutesOffset) throws IllegalArgumentException {
273 273      if (hoursOffset == 0 && minutesOffset == 0) {
274 274          return DateTimeZone.UTC;
275 275      }
276 276      if (hoursOffset < -23 || hoursOffset > 23) {
277 277          throw new IllegalArgumentException("Hours out of range: " +
                              hoursOffset);
278 278      }
279  -       if (minutesOffset < 0 || minutesOffset > 59) {
    279  +   if (minutesOffset < -59 || minutesOffset > 59) {
280 280          throw new IllegalArgumentException("Minutes out of range: " +
                              minutesOffset);
281 281      }
282  +       if (hoursOffset > 0 && minutesOffset < 0) {
283  +           throw new IllegalArgumentException("Positive hours must not
                              have negative minutes: " + minutesOffset);
284  +       }
282 285      int offset = 0;
            …
295 298  }
```

Fig. 1. Buggy code and patch from time-8 bug.

The buggy method, `forOffsetHoursMinutes`, takes two integer inputs, `hoursOffset` and `minutesOffset`, and returns the `DateTimeZone` object for the offset specified by the two inputs. For example, if the method is called with the inputs `hoursOffset=0` and `minutesOffset=-30`, then it is expected to return a `DateTimeZone` object for the offset $-00:30$. However, such inputs execute the `true` branch of the `if` condition at line 279 and the method throws an `IllegalArgumentException` instead of the expected `DateTimeZone` object. This bug is fixed by modifying the `if` condition at line 279 and adding a new condition at line 282 as shown in the diff in Fig. 1.

To detect this bug, test cases have to execute the `false` branches of the `if` conditions at line 273 and 276; that is `hoursOffset` $\neq 0$ or `minutesOffset` $\neq 0$ and `hoursOffset` $\in [-23, 23]$. They also have to execute the `true` branch at line 279 with an additional constraint; `minutesOffset` $\in [-59, -1]$. Furthermore, the newly added `if` condition at line 282 adds another constraint on the input `hoursOffset`; that is `hoursOffset` ≤ 0. In summary, only the test inputs sampled from the space where `hoursOffset` $\in [-23, 0]$ and `minutesOffset` $\in [-59, -1]$ can detect the bug.

It is evident that just covering the buggy code (i.e., the `true` branch of the `if` condition at line 279) is not sufficient to detect the bug. For example, the inputs `hoursOffset=12` and `minutesOffset=-60` cover the buggy code, however, they do not detect the bug. Also, the space of all possible test inputs that cover the buggy code (i.e., `hoursOffset` $\in [-23, 23]$ and `minutesOffset` $\notin [0, 59]$) is larger than the space of test inputs that can detect the bug. The existing SBST techniques that aim at maximising code coverage, such as DynaMOSA are more likely to sample test inputs from the larger space of inputs that cover the buggy code without detecting the bug, and then terminate without actually detecting the bug.

The existing SBST approaches can be configured to generate many tests for each coverage target in the `DateTimeZone` class, and it will increase the chances of detecting the bug. However, there are 54 methods in the class and only one method is buggy. If we assume the test adequacy criterion to be branch coverage, then there are 201 coverage targets in

total, while only 14 of them actually contain the buggy method. Thus, we find it is ineffective to spend all the critical search resources on covering all the 201 targets, when only a few of them leads to the buggy code. We propose to use buggy methods predictions from a defect predictor to decide where to increase the coverage within the class. Thus, our novel SBST approach concentrates the search for test cases more on the only buggy method in the project, `forOffsetHoursMinutes`.

3 BACKGROUND

3.1 Optimisation Problem

In this paper, our proposed approach, PreMOSA, tackles the test generation problem as a many objective optimisation problem, where each objective represents a coverage target in the CUT. Our approach optimises test cases to meet a given coverage criterion, such as maximising branch coverage, method coverage or a combination of both.

Let $U = \{u_1, \ldots, u_k\}$ be the set of k coverage targets of the CUT. $f_i(t)$, where $i \in [1, k]$ and t is a test case, is the fitness function for the coverage target u_i. For example, assuming the coverage target is a branch $u_i \in U$, $f_i(t)$ is as follows;

$$f_i(t) = al(u_i, t) + d(u_i, t) \quad (1)$$

where $al(u_i, t)$ is the approach level and $d(u_i, t)$ is the normalised branch distance of the branch u_i for the test case t [3]. Approach level is calculated based on the distance (i.e., number of control dependencies) between the branch where the execution diverges from the desired execution path and the branch under consideration. Branch distance [27], [28] is a widely used heuristic in fitness functions to guide the search to find inputs that evaluate the logic in branch predicates as desired (i.e., to true or false).

For a given test case t, the fitness is a vector of k values ($(\langle f_1, \ldots, f_k \rangle)$, where f_i ($i \in [1, k]$) represents the distance of t from covering the target $u_i \in U$. If a test case t covers a target u_i, then the corresponding fitness f_i of t is zero.

To maximise the coverage of multiple targets, we need to find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ where for each $t_j \in T$, $\exists u_i \in U$ such that $f_i(t_j) = 0$. A set of test cases are said to be non-dominated if each test case in the set is better on at least one coverage target and worse on the remaining targets when compared to other test cases in the set.

To evaluate individual test cases, PreMOSA uses Pareto dominance (Definition 1) and Pareto optimality (Definition 2) of test cases.

Definition 1 Pareto Dominance. *A test case t_i dominates another test case t_j, if, and only if, the values of the fitness vector satisfy the following conditions:*

$$\forall x \in \{1, \ldots, k\} f_x(t_i) \leq f_x(t_j)$$
$$\text{and}$$
$$\exists y \in \{1, \ldots, k\} s.t. f_y(t_i) < f_y(t_j)$$

The definition above states that a test case t_i dominates another test case t_j, if, and only if, t_i is closer to cover at least one coverage target and not worse in terms of covering other targets when compared to t_j.

Definition 2 Pareto Optimality. *A test case is Pareto optimal, if, and only if, it is not dominated by any other test case in the space of all possible test cases.*

The definition above states that a Pareto optimal test case is better on covering one or more targets and can be worse on covering the remaining targets when compared to all possible test cases.

The solution to the many-objective problem is a set of Pareto optimal test cases. Unlike in usual many-objective optimisation problems where there are trade-offs in the objective space, in the context of test generation, the optimal test cases are the ones which cover at least one target, i.e., objective, (i.e., $\exists u_i \in U \ s.t. \ f_i(t) = 0$). Therefore, these test cases that cover at least one target form the final test suite and represent a sub-set of the Pareto optimal test cases.

3.2 Existing Many-Objective Algorithms

Panichella *et al.* [2] first proposed many objective sorting algorithm (MOSA), which formulates the test generation problem as a many objective optimisation problem and produces a set of Pareto optimal test cases. MOSA is based on a genetic algorithm (GA). It starts with a set of randomly generated test cases as the initial population. It generates new population of test cases by applying crossover and mutation operators. Test cases are selected to the next generation using a ranking algorithm called preference sorting algorithm which is based on 'preference criterion' and the non-dominance relation of test cases. According to preference criterion, for each target $u_i \in U$, the test case that is closest to cover u_i is selected to the first non-dominated front. All the test cases in the first non-dominated front are selected to the next generation. MOSA maintains an archive of test cases generated during the evolution process that forms the final test suite. The archive contains the shortest test cases for each covered target.

Dynamic many objective sorting algorithm (DynaMOSA) [3] is the successor of MOSA and stands as the state-of-the-art SBST technique. A main limitation in MOSA is that it tries to cover all the targets from the beginning of the search while most of the targets are not reachable until their control dependent targets are covered. DynaMOSA addresses this problem by introducing a method called dynamic selection of targets.

Fig. 2 shows the control dependency graph (CDG) of the method `forOffsetHoursMinutes` from the motivating example. Nodes denote the predicates and leaves denote the exit points of the program. For example, node 279-1 denotes the `minutesOffset < 0` predicate at line 279 and leaf 280 denote the `return` statement at line 280. Lines between nodes denote the control dependency edges. For example, $b_{5,T}$ is the `true` branch of the `minutesOffset < 0` predicate. For simplicity, we do not include the nodes that are not predicates of the program.

A subset of the targets that are included in the search process cannot be covered until their control dependent targets are covered. Assume a test generation scenario which uses branch coverage as the optimisation criterion. In the beginning of the search, U contains all the branches as shown in the CDG (Fig. 2) as the set of targets to search for test cases. However, branches like $b_{3,T}$ and $b_{3,F}$ cannot be
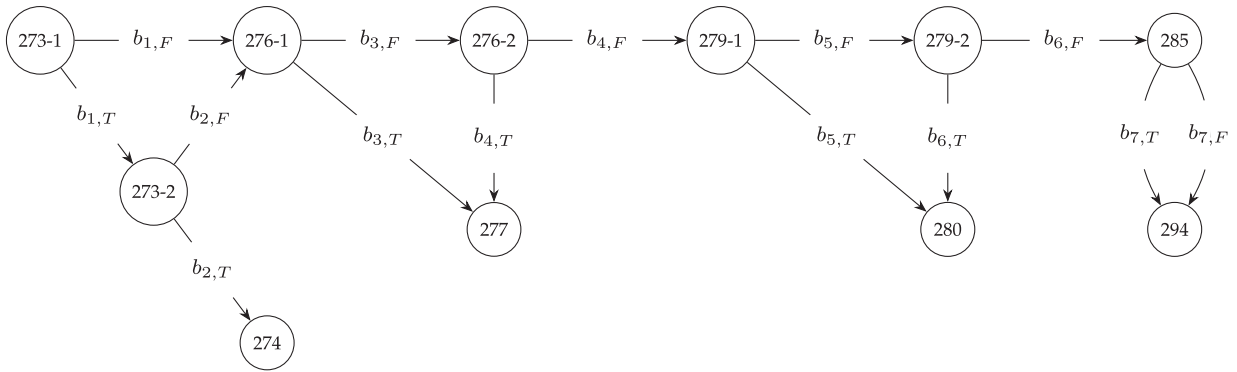
Fig. 2. Control dependency graph.

covered until their control dependent branches $b_{2,T}$ or $b_{1,F}$ are covered. Likewise, $b_{2,F}$ cannot be covered until $b_{1,T}$ is covered. Therefore, it is inefficient to search for tests to cover such targets, e.g., $b_{3,T}$, while their control dependent targets are still uncovered.

To address this, DynaMOSA dynamically selects targets to search for test cases only when their control dependent targets are covered. For example, $b_{3,T}$ and $b_{3,F}$ are selected to the search process only if $b_{1,F}$ or $b_{2,F}$ is covered. At the start of the search, DynaMOSA selects the set of targets $U^* \subseteq U$ that do not have control dependencies. At any given time in the search, DynaMOSA optimises test cases to cover only the targets in U^*. Once a new population is generated, DynaMOSA needs to update U^* with new targets if their control dependent targets are covered. Since DynaMOSA was originally proposed to maximise code coverage, it also removes the covered targets from U^* to allow itself to focus more on uncovered targets.

3.3 Bug Detecting with Search-Based Software Testing Techniques

In order to detect a bug, a test case must satisfy the conditions of the *reachability*, *infection* and *propagation* (RIP) model [30], [31], [32], [33]. In addition, it must also have a test oracle to *reveal* the failure [34]. MOSA, DynaMOSA and Pre-MOSA are implemented in the state-of-the-art SBST tool, EvoSuite. Tests generated by EvoSuite satisfy all three conditions of the RIP model. However, they do not have test oracles, hence are incapable of revealing bugs without test oracle inserted by humans or automated tools [35]. We will explain this more with the motivating example.

Assume a test generation scenario for the buggy version of the `DateTimeZone` class in our motivating example. Fig. 3a shows a sample test case generated by EvoSuite during the search process. The execution of the test case *reaches* the buggy code, i.e., line 279. The execution of the buggy statement causes an incorrect internal program state (*infection*), i.e., a valid `minutesOffset` must not cause the program to throw an `IllegalArgumentException` at line 280. The incorrect internal program state is *propagated* to an incorrect final state (failure) of the program, i.e., at line 4 in the test case, `forOffsetHoursMinutes(0,-30)` call should output a valid `DateTimeZone` object for the offset $-00:30$, instead an `IllegalArgumentException` is thrown. EvoSuite does not have test oracles, hence it generates assertions in the tests assuming the program under test

is correct. For example, Fig. 3b shows the final test case generated by EvoSuite for the test case shown in Fig. 3a, which is not able to *reveal* the bug since the test case does not fail when it is executed against the buggy program.

In the ideal scenario, if oracle automation [35] exists, the generated test cases can reveal the bugs. Without oracle automation, the best EvoSuite can do is to propagate the incorrect state of the program to the output. The scope of this paper is to improve the bug detection capability of the test suites generated by SBST guided by defect prediction and not oracle automation. Therefore, in this paper, we consider that DynaMOSA or PreMOSA detect a bug if they generate a test case that propagates the internal error to the output of the program. Nevertheless, we remind the readers that neither DynaMOSA nor PreMOSA are able to reveal existing bugs in a program without the aid of oracle automation. Finally, this limitation is not only applicable to PreMOSA and DynaMOSA, but also to other SBST techniques [2], [26], [36] in this space as well.

4 PREDICTIVE MANY OBJECTIVE SORTING ALGORITHM

Predictive many objective sorting algorithm (PreMOSA) is a novel search-based software testing approach that incorporates guidance from a defect predictor. PreMOSA receives as input a buggy program with methods labelled as buggy or non-buggy, which are labels that can be obtained using existing defect predictors [12], [13]. PreMOSA is not specific to a certain defect predictor. Hence, we use the most commonly used defect predictor output type in PreMOSA, which is

```
1  ...
2  int int0 = 0;
3  int int1 = (-30);
4  DateTimeZone.forOffsetHoursMinutes(int0, int1);
5  ...
```

(a) Test case generated by EvoSuite during the search

```
1  public void test001()  throws Throwable  {
2    // time taken = 28926
3    try {
4      DateTimeZone.forOffsetHoursMinutes(0, (-30));
5      fail("Expecting exception: IllegalArgumentException");
6    } catch(IllegalArgumentException e) {
7      //
8      // Minutes out of range: -30
9      //
10     verifyException("org.joda.time.DateTimeZone", e);
11   }
12 }
```

(b) Final test case with assertions by EvoSuite

Fig. 3. A sample test generated by EvoSuite for the buggy version of DateTimeZone class from Time-8.

binary classification [37]. PreMOSA uses this information to start searching for test inputs that cover targets that are deemed to contain buggy methods as indicated by the defect prediction information (see Section 4.1). This helps focus the search initially on covering the likely buggy targets rather than the likely non-buggy targets.

Most of the time, defect predictors are not 100% accurate. Hence, there can be actual buggy methods among the methods labelled as non-buggy. Therefore, PreMOSA adds the targets that contain predicted non-buggy methods, once the likely buggy targets coverage does not improve for a pre-defined number of consecutive iterations (see Section 4.1).

PreMOSA also generates more than one test case for all the selected targets, hence, increases the chances of detecting bugs (see Section 4.2).

Finally, to balance the test coverage among all the targets in the search, we introduce a method to dynamically disable coverage targets from the search based on their current test coverage and number of independent paths (see Section 4.3). This ensures that the non-trivial targets have an equal chance of being covered compared to the targets that are easier to cover.

PreMOSA is presented in Algorithm 1. It is based on a genetic algorithm (GA). PreMOSA creates an initial population of randomly generated test cases (line 9 in Algorithm 1). Then, it evolves this initial population through creating new test cases via crossover and mutation (line 13) and selecting test cases to the next generation (line 18), until a termination criterion, such as maximum time budget, is met.

4.1 Filtering Targets with Defect Prediction

A defect predictor classifies the methods of the class under test (CUT) as buggy or non-buggy, denoted as c_i, where

$$c_i = \begin{cases} 1 & \text{if } m_i \text{ is predicted as buggy} \\ 0 & \text{otherwise} \end{cases}$$

where m_i denotes method with index i. PreMOSA starts with filtering the likely buggy and likely non-buggy targets, U_B and U_N respectively, from the set of all targets U using the classifications given (line 2 in Algorithm 1). The procedure FILTERTARGETS labels targets as buggy if they belong to a likely buggy method and non-buggy otherwise. Initially, PreMOSA finds tests to cover only the likely buggy targets, hence, only the likely buggy targets are selected to be included in the search process in the beginning (line 5). This way, PreMOSA can extensively search for test cases that cover likely buggy targets, which leads to generating more effective test cases faster than other approaches.

However, defect predictors often are not 100% accurate, and it is likely that buggy methods may be labelled as non-buggy. To address this issue, PreMOSA considers targets that do not contain any methods that are predicted as buggy if it deems to have searched enough for tests that cover the likely buggy targets (line 19). If PreMOSA resorts to searching for tests to cover only the likely buggy targets, then it will miss actual buggy targets that are incorrectly classified. Thus, PreMOSA starts finding tests to cover likely non-buggy targets once the coverage of likely buggy targets does not improve for a predefined number of consecutive iterations (I) in GA (line 28). This way, PreMOSA expects to

account for the errors present in the predictions. Finally, if there are no likely buggy targets, either because the class is not buggy or the defect predictor is inaccurate, PreMOSA considers all targets from the start (line 7).

Algorithm 1. PreMOSA

Input:
$U = \{u_1, \ldots, u_k\}$ ▷ the set of coverage targets of CUT
$G = \langle N, E \rangle$ ▷ control dependency graph of the CUT
$\phi : E \rightarrow U$ ▷ partial map between edges and targets
$C = \{c_1, \ldots, c_m\}$ ▷ the set of defectiveness classifications for methods in the CUT

1: **procedure** PREMOSA
2:   $U_B, U_N \leftarrow$ FILTERTARGETS(U, C)
3:   $L \leftarrow$ INDEPENDENTPATHS(G) ▷ L is a vector of the number of independent paths for each edge
4:   **if** U_B is not empty **then**
5:    $U \leftarrow U_B$
6:   **else**
7:    $U \leftarrow U_N$
8:   $U^* \leftarrow$ targets in U with no control dependencies
9:   $P_0 \leftarrow$ RANDOMPOPULATION(M) ▷ M is the population size
10:   $A \leftarrow$ UPDATEARCHIVE$(P_0, \emptyset)$ ▷ A is the archive
11:   $U^* \leftarrow$ UPDATETARGETS(U^*, G, ϕ)
12:   **for** $r \leftarrow 0$; !terminationCriteria; r++ **do**
13:    $Q_r \leftarrow$ GENERATEOFFSPRING(P_r)
14:    $A \leftarrow$ UPDATEARCHIVE(Q_r, A)
15:    $U^* \leftarrow$ UPDATETARGETS(U^*, G, ϕ)
16:    $R_r \leftarrow P_r \cup Q_r$
17:    $U^* \leftarrow$ SWITCHOFFTARGETS(U^*, A, L, ϕ)
18:    $P_{r+1} \leftarrow$ SELECTPOPULATION(R_r, U^*, M)
19:    $U^* \leftarrow$ ADDNONBUGGYTARGETS$()$
20:   $T \leftarrow A$ ▷ Update the final test suite T
21:   RETURN(T)
22: **procedure** ADDNONBUGGYTARGETS
23:   **if** trigger *not* fired to add non-buggy targets **then**
24:    **if** # covered goals = prev. # covered goals **then**
25:     w++
26:    **else**
27:     $w = 0$
28:    **if** $w = I$ **then**   ▷ I is max. # iterations without coverage improvement
29:     $U \leftarrow U \cup U_N$
30:     $U^* \leftarrow U^* \cup \{u \in U_N | u \text{ has no control dependencies}\}$
31:   RETURN(U^*)

4.2 Updating Targets and Archiving Tests

PreMOSA generates more than one test case for all the selected targets in order to increase the chances of detecting bugs. When updating targets in each iteration (lines 11 and 15), it does not remove covered targets from U^*, allowing it to keep generating more tests to cover those targets as well.

PreMOSA keeps an archive of all the test cases that cover the selected targets $u \in U$ during the search (lines 10 and 14). This archive of test cases form the final test suite. Thus, the final test suite is more likely to detect the bugs as it contains all the generated test cases which cover the potentially buggy targets.

Removing covered targets from U^* and archiving only the shortest test case for each covered target are beneficial for achieving high code coverage with a minimal test suite

size [3]. However, just covering the buggy code is not sufficient to detect the bug. In particular, Perera *et al.* [8] showed that there was an average improvement of up to 79% in terms of detecting bugs when the state-of-the-art Dyna-MOSA was configured to not to remove covered targets from the search and retain all the generated tests. Therefore, we decide to archive all the test cases that cover the selected targets $u \in U$ and not to remove the covered targets from the search in PreMOSA.

4.3 Balanced Test Coverage of Targets

In our running example, assume the branch coverage is used as the optimisation criterion. An SBST technique that does not remove covered targets from the search is more likely to keep on generating test cases which cover more trivial branches like $b_{3,T}$ or $b_{4,T}$ rather than a less trivial branch $b_{5,T}$ (Fig. 2). This is detrimental to the bug detection performance of SBST since it is necessary to find tests that exercise the branch $b_{5,T}$ in order to detect the bug, and also, to increase the chances of detecting the bug, SBST has to find as many tests as possible that cover $b_{5,T}$.

We introduce a method to dynamically remove coverage targets from the search based on their current test coverage and number of independent paths, in order to balance the test coverage among all the targets. A balanced test coverage means that all the targets receive an equitable test coverage. This ensures that, in PreMOSA, the less trivial targets also get a good coverage in the presence of more trivial targets.

We consider a balanced test coverage is achieved when the measure, the number of tests generated per an independent path of a target, is equal for all of the targets. We measure the number of independent paths of a target by assuming the paths start at the control dependent edge of that target (line 3). An independent path is one that traverses one or more new edges in the control dependency graph.

In general, for each target $u \in U^*$, PreMOSA checks the current test coverage (i.e., number of tests in the archive that cover u), and then temporarily removes u from U^* in the current iteration, if the test coverage per an independent path from u is higher than the other targets (line 17).

4.3.1 Independent Paths

We use the measure, the number of independent paths of a target, to determine how much of a test coverage a target should receive compared to other targets, in order to achieve a balanced test coverage for all targets. For a target $u \in U^*$, if there are many independent paths that start from u, then PreMOSA should generate more tests to cover u than the other targets which have few independent paths. In our running example, the target $b_{2,F}$ should receive more test coverage than $b_{2,T}$ because there are more independent paths leading up from $b_{2,F}$ (6) compared to $b_{2,T}$ (1).

In the beginning of the search, PreMOSA finds the number of independent paths of each edge in the control dependency graph G of the program (line 3). The control dependency graph $G = \langle N, E \rangle$ consists of nodes $n \in N$ and edges $e \in E \subseteq N \times N$. The nodes represent statements in the program. The edges represent control dependencies between the statements. For each edge $e \in E$, the procedure

INDEPENDENTPATHS calculates the number of independent paths starting from e using the graph G. The actual executions of the paths start at the root node, however, in the calculation of number of independent paths of e, we assume the paths start at e. All the coverage targets that are directly control dependent by e have the same number of independent paths as that of e.

In the motivating example, the edges $b_{2,T}$, $b_{3,T}$, $b_{4,T}$, $b_{5,T}$, $b_{6,T}$, $b_{7,T}$ and $b_{7,F}$ have only one path each that start from those edges. There are 2 independent paths from the edge $b_{6,F}$, those are $b_{6,F} - b_{7,T}$ and $b_{6,F} - b_{7,F}$. Likewise, there are 7, 6, 6, 5, 4 and 3 independent paths that start from edges $b_{1,T}$, $b_{1,F}$, $b_{2,F}$, $b_{3,F}$, $b_{4,F}$ and $b_{5,F}$, respectively. If the optimisation problem is maximising the branch coverage, then these edges become the coverage targets in the search.

4.3.2 Temporarily Disabling Targets from Search

In many objective optimisation, test cases are optimised simultaneously to satisfy all the coverage targets. Thus, the search resources (e.g., time budget) are not allocated to each coverage target individually. Therefore, to focus the search differently on covering each target, we decide to dynamically switch off targets during the evolution. In every iteration in GA, for each target $u \in U^*$, the procedure SWITCHOFFTARGETS checks the current test coverage of u, and then removes u from U^*, if the test coverage per an independent path of u is higher than that of the other targets. Therefore, after calling the procedure SWITCHOFFTARGETS, only the targets which are having a low test coverage (per an independent path) remain in U^*. Then, the procedure SELECTPOPULATION selects test cases to the next generation considering only these remaining targets in U^*. Hence, this paves way for the search to find more test cases in the next generation that cover these targets, thereby guiding the search to a balanced test coverage for all the targets.

Algorithm 2. Temporarily Removal of Targets to Balance Test Coverage

---

1: **procedure** SWITCHOFFTARGETS(U^*, A, L, ϕ)
2:     $N_P \leftarrow$ NODESWITHPREDICATES(G)
3:     **for** $n \in N_P$ **do**
4:         $\{e_{n,T}, e_{n,F}\} \leftarrow$ outgoing edges in G from node n
5:         $l_{n,T} \leftarrow$ GETINDEPENDENTPATHS($L, e_{n,T}$)
6:         $l_{n,F} \leftarrow$ GETINDEPENDENTPATHS($L, e_{n,F}$)
7:         $u_{n,T} \leftarrow$ RANDOMCHOICE($\{\phi(e_{n,T})\}$)
8:         $u_{n,F} \leftarrow$ RANDOMCHOICE($\{\phi(e_{n,F})\}$)
9:         $A_{n,T} \leftarrow$ GETTESTS($A, u_{n,T}$)
10:         $A_{n,F} \leftarrow$ GETTESTS($A, u_{n,F}$)
11:         **if** $\frac{|A_{n,T}|}{l_{n,T}} > \frac{|A_{n,F}|}{l_{n,F}}$ **then**
12:             $U^* \leftarrow U^* - \{\phi(e_{n,T})\}$
13:         **else if** $\frac{|A_{n,T}|}{l_{n,T}} < \frac{|A_{n,F}|}{l_{n,F}}$ **then**
14:             $U^* \leftarrow U^* - \{\phi(e_{n,F})\}$
15:     RETURN(U^*)

First, the procedure SWITCHOFFTARGETS finds the set of nodes with predicates N_P in G (line 2 in Algorithm 2). Next, for each node $n \in N_P$, it fetches the number of independent paths from the outgoing edges of n (lines 5–6 in Algorithm 2). Then, it randomly selects a control dependent target from each outgoing edge of n (lines 7–8 in Algorithm 2). We

consider all the control dependent targets of an edge receive the same test coverage. Hence, the test coverage of a randomly selected target of an edge is equal to the test coverage of that edge. Finally, it finds the edge which has the largest number of tests in the archive per an independent path, and removes all the control dependent targets of that edge from U^* (lines 9-14 in Algorithm 2).

In the running example, if we consider the node 276-2, the outgoing edges are $b_{4,T}$ and $b_{4,F}$, and the number of independent paths from these edges are 1 and 4, respectively. Assume the coverage criterion is maximise branch coverage, hence $b_{4,T}$ and $b_{4,F}$ are also targets in the search, and there are currently 30 and 20 tests in the archive covering $b_{4,T}$ and $b_{4,F}$, respectively. Thus, $b_{4,T}$ has 30 (= 30/1) tests in the archive per an independent path, while $b_{4,F}$ has only 5 (= 20/4) tests per a path. Hence SWITCHOFFTARGETS temporarily removes the target $b_{4,T}$ from U^*, and paves way for the search to find more test cases that cover $b_{4,F}$. Overall, this encourages the search to have a balanced test coverage for all the targets rather an excessive coverage of more trivial targets like $b_{3,T}$ and $b_{4,T}$. As a result, a less trivial target like $b_{5,T}$, which contains the buggy code, receives a good coverage in the presence of other more trivial targets.

5 DESIGN OF EXPERIMENTS

We design a set of experiments to evaluate PreMOSA in terms of its effectiveness and efficiency in detecting bugs compared to the state-of-the-art DynaMOSA. Through these experiments, we aim to investigate if augmenting coverage information with defect prediction information in the search process of SBST indeed helps to improve the bug detection performance of the generated test suites. Our first research question is:

RQ1: Is PreMOSA more effective in detecting bugs compared to the state-of-the-art DynaMOSA?

To answer this research question, we compare the number of bugs detected by PreMOSA against DynaMOSA, which we discuss in Section 5.4. We run test generation on Defects4J bugs [29] (discussed in Section 5.3) using both PreMOSA and the baseline. To account for randomness in PreMOSA and DynaMOSA, we repeat the test generation for 25 runs for each bug and testing approach. Once test cases are generated and evaluated for bug detection, we report the bug detection results as means and medians over 25 runs. To check if PreMOSA significantly detects more bugs than DynaMOSA and the effect size of the difference, we employ one-tailed non-parametric Mann-Whitney U-Test with the significance level (α) 0.05 [38] and Vargha and Delaney's $\hat{A}_{12}$ statistic [39].

To analyse the efficiency of PreMOSA, we seek to answer the following research question:

RQ2: Is PreMOSA more efficient at generating test cases that can detect bugs compared to the state-of-the-art DynaMOSA?

To answer this research question, we measure the time to generate the first test case that can detect a bug by the two approaches over 25 runs. As we described in Section 3.3, a test case detects a bug if it satisfies all the three conditions of RIP model, and we call such test cases *bug detecting tests* throughout the paper. For each bug that is detected by both approaches, we calculate the difference of the mean time to generate the first test case that detects the particular bug by

the two approaches. If the difference is positive, that means PreMOSA generates a test case to detect the bug in a shorter time. A negative difference means otherwise. To check if PreMOSA generates a bug detecting test in a significantly shorter time, we employ one-tailed Wilcoxon signed-rank test [38] and its effect size, r [40]. We remind the readers that the time taken to generate the first bug detecting test is not equal to time taken to reveal the bug. The latter happens only after the test generation is completed.

Zimmermann *et al.* [25] argues that a defect predictor is strong if, and only if, all recall, precision and accuracy are greater than 75%. Therefore, we consider defect predictors having both recall and precision in the range 75% to 100% as acceptable defect predictors. In *RQ1* and *RQ2*, we simulate defect predictor outcomes for two levels of performance for PreMOSA; i) most conservative and acceptable defect predictor (recall=precision=75%) and ii) ideal defect predictor (recall=precision=100%). We will discuss this more in Section 5.1. We expect PreMOSA to perform best with the latter defect prediction simulation, and with the former simulation, we can see the most conservative performance of PreMOSA when using acceptable defect predictors.

5.1 Defect Prediction Simulation

We simulate defect predictor outcomes at two levels of recall and precision, which correspond to the theoretical upper bound and lower bound performance of an acceptable defect predictor. This would not be possible with real defect predictors since their performance cannot be controlled. Using a real defect predictor would have demonstrated the viability of PreMOSA in practice. However, it would then have the disadvantage of limiting the findings of our study to one single defect predictor, e.g., a specific defect predictor built with one learner and one set of metrics. Therefore, we abstract the defect predictor component in the experimental evaluation.

Recall is the probability that the defect predictor correctly labels a buggy method. It can be calculated as in Equation. (2), where tp is the number of true positives, i.e., number of buggy methods that are correctly classified, and fn is the number of false negatives, i.e., number of buggy methods that are incorrectly classified. Higher recall means PreMOSA is informed of more buggy methods, hence, it is expected to increase the chances of detecting bugs.

Precision measures what percentage of methods that are labelled as buggy by the defect predictor are actually buggy. It can be calculated as in Equation. (3), where fp is the number of false positives, i.e., number of non-buggy methods that are incorrectly classified as buggy methods. For example, if precision is 50%, that means half of the methods that are labelled as buggy are not actual buggy methods. Higher precision means that PreMOSA does not concentrate more on covering non-buggy code, which otherwise will likely be ineffective in terms of detecting bugs.

$$\text{recall} = \frac{tp}{tp + fn} \tag{2}$$

$$\text{precision} = \frac{tp}{tp + fp} \tag{3}$$

Algorithm 3 illustrates the steps of simulating the defect predictor outputs for a given recall and precision. The procedure SIMULATEDEFECTPREDICTOR receives the set of methods in the project with the ground truth labels for their defectiveness, $M = \{m_1, \ldots, m_k\}$. $m_i = 1$ if the ith method is buggy and $m_i = 0$ if the ith method is not buggy. First, it calculates the number of buggy (d) and non-buggy methods (nd) in the project (lines 2–3 in Algorithm 3). Next, it finds the set of indices of all the buggy (M_b) and non-buggy methods (M_n) in the project (lines 4–5). Then, it calculates the number of true positives (tp) and false positives (fp) for the given recall (r) and precision (p) (lines 6–7). The RANDOMCHOICE(S, n) procedure returns a set of randomly picked n items from the set S. Likewise, at line 8, C_b is assigned a set of randomly picked tp number of buggy and fp number of non-buggy method indices. C_b denotes the set of buggy method indices as classified by the simulated defect predictor. Finally, at line 9, the output $C = \{c_1, \ldots, c_k\}$ is formed where $c_i = 1$ if the ith method is labelled as buggy and $c_i = 0$ if the ith method is labelled as not buggy by the defect predictor.

Algorithm 3. Defect Predictor Simulation

Input:

r	▷ recall
p	▷ precision
$M = \{m_1, \ldots, m_k\}$	▷ ground truth

1: **procedure** SIMULATEDEFECTPREDICTOR
2:     $d \leftarrow$ COUNT(m_i) for $m_i \in M$ s.t. $m_i = 1$
3:     $nd \leftarrow |M| - d$
4:     $M_b \leftarrow \{i \,|\, \forall i \in [1, k] \wedge m_i = 1\}$
5:     $M_n \leftarrow \{i \,|\, \forall i \in [1, k] \wedge m_i = 0\}$
6:     $tp \leftarrow d * r$
7:     $fp \leftarrow tp * (1 - p)/p$
8:     $C_b \leftarrow$ RANDOMCHOICE(M_b, tp) $\cup$ RANDOMCHOICE(M_n, fp)
9:     $C \leftarrow \{c_i = 1 \,|\, \forall i \in [1, k] \wedge i \in C_b, c_i = 0 \,|\, \forall i \in [1, k] \wedge i \notin C_b\}$
10:    RETURN(C)

5.2 Time Budget

We set 2 minutes as the time budget for test generation per class. In practice, the time reserved for test generation for a project depends on the project size and resource availability in the organisation. If the project is small and has a few number of classes, then it takes a very short time to run test generation on all the classes. Perhaps, the test generation can be done on developer machines outside of the working hours. If the project is large, which is usually the case, then it may not be possible to run SBST on developer machines as it may run for a longer duration and also slow down them [41].

For example, running SBST for 2 minutes per class on a project having several hundreds of classes could take up to 30 hours to finish test generation for the whole project. In a situation like this, SBST tools can be setup in the continuous integration (CI) system [42] of the organisation. Although, if the organisation wants to run an SBST tool in their CI system, then it should use as little resources as possible, such that it will not cause any impact (e.g., idling other jobs) on the existing processes in the system (e.g., regression testing, code quality checks, project

builds etc.) [8]. Therefore, we decide that 2 minutes per class is a reasonable time budget in a usual resource constrained environment.

5.3 Experimental Subjects

We use the Defects4J dataset (version 1.5.0) [29], [43] as our benchmark. It contains 438 real bugs from 6 real-world open source Java projects. We remove 4 deprecated bugs, 12 bugs that do not have buggy methods, and 2 bugs for which SBST generated uncompilable tests (e.g., method signature is changed in the bug fix). This results in the following 18 bugs that are not part of the experiments: Lang-2, 23, 25, 30, 56, 63, Math-12, 104, Time-11, 21, Chart-23, Closure-15, 28, 63, 83, 93, 111 and Mockito-26 are removed. Thus, we evaluate PreMOSA against the baseline on a total of 420 bugs[1]. The 420 bugs are from the following projects; JFreeChart (25 bugs), Closure Compiler (170 bugs), Apache commons-lang (59 bugs), Apache commons-math (104 bugs), Mockito (37 bugs), and Joda-Time (25 bugs).

For each bug, the Defects4J benchmark gives a buggy version and a fixed version of the program. The difference between these two versions of the program is the applied patch to fix the bug, which indicates the location of the bug. We label all the methods that are either modified or removed in the bug fix as buggy methods [44].

Defects4J is widely used for research on automated unit test generation [5], [8], [45], automated program repair [46], fault localisation [47], test case prioritisation [23] etc. This makes Defects4J a suitable benchmark for evaluating our approach, as it allows us to compare our results to existing work.

5.4 Baseline

We use the current state-of-the-art SBST technique, Dyna-MOSA [3], as the baseline. It is more effective at achieving high branch, statement and strong mutation coverage than previously proposed SBST techniques ([1], [2], [26]) [3]. DynaMOSA is implemented in the state-of-the-art SBST tool, EvoSuite [36], and won the unit testing tool competition at SBST 2019 [48].

We configure DynaMOSA to not remove the covered targets from the search, retain all the test cases generated, and continue the search until the full time budget is consumed in our experimental evaluation. DynaMOSA primarily focuses on achieving high code coverage with a minimal test suite size. Hence, it aims at generating only one short test case to cover each target in the program. However, just covering the buggy code is not sufficient to detect the bug. Perera *et al.* [8] showed that DynaMOSA detects 79% more bugs on average when it is configured to not remove covered targets from the search, use the full time budget, and retain all the generated tests in the final test suite (i.e., disable test suite minimisation).

---

1. While there may be more unlabelled bugs in the six projects, we use only the labelled bugs in the Defects4J dataset. The experimental results and the conclusions are based on these bugs.

5.5 Prototype

We implement PreMOSA in the state-of-the-art SBST tool, EvoSuite [36]. EvoSuite is an automated test generation framework that generates JUnit test suites for java programs [49], [50]. To date, EvoSuite is actively maintained and evaluated for its effectiveness on both industrial and open source projects in terms of code coverage [2], [3], [4], [26], [51] and bug detection capability [5], [6], [8], [45]. We implement PreMOSA within EvoSuite version 1.0.7, forked from the GitHub repository [49] on June 18th, 2019. The prototype is available to download from here: https://github.com/premosa-sbst

5.6 Parameter Settings

There are several parameters that need to be configured in PreMOSA. Parameter tuning of search algorithms is a long and expensive process [52]. Arcuri and Fraser [52] showed that the default parameter values in EvoSuite give reasonable results when compared to tuned parameters. Moreover, Panichella *et al.* [3] also used these default values in the state-of-the-art DynaMOSA. Therefore, we decide to use the default parameter values used in EvoSuite [1] and DynaMOSA [3] except for the following parameters.

Coverage criteria: We use branch coverage as coverage criterion in PreMOSA inline with the prior studies which investigated bug detection effectiveness of EvoSuite [5], [6]. EvoSuite with branch coverage was shown to be the most effective coverage criterion in terms of detecting bugs when compared with other criteria like line, output and weak mutation coverage [7], [45].

Termination criteria: We use the maximum time budget as the termination criterion. We find that stopping the search after it covers all the coverage targets is detrimental to bug detection. Just covering the buggy code is not sufficient to detect the bug. Thus, the search needs to utilise the full time budget to generate more tests in order to increase the chances of detecting bugs. Therefore, we run PreMOSA until it consumes the allocated time budget.

Test suite minimisation: We disable test suite minimisation since all the test cases in the archive form the final test suite (see Section 4.2).

Assertion strategy: Mutation-based assertion filtering can be computationally expensive and can lead to timeouts. Therefore, following a similar approach to previous work [5], [8], we choose all possible assertions as the assertion strategy.

Similar to PreMOSA, we configure the baseline technique, DynaMOSA, as described above.

Finally, following the results of our pilot runs, we use 50 consecutive iterations for the parameter *maximum number of iterations without coverage improvement* (*I*) in PreMOSA. Furthermore, we configure PreMOSA to add non-buggy targets to the search if it cannot cover any buggy target in the first 25 iterations in the search. For some of the classes, PreMOSA cannot find a test that covers the buggy targets until the trigger is fired to add non-buggy targets to the search. Thus, all the search resources spent until this point are ineffective in terms of detecting bugs. Our preliminary results suggest that for a significant number of classes, PreMOSA covers the first buggy target within the first 25 iterations. Therefore, we decide to add non-buggy targets to the search if PreMOSA fails to cover at least one target after 25 iterations.

5.7 Experimental Protocol

We run experiments with PreMOSA using 2 instances of simulated defect predictors and DynaMOSA on 420 bugs. For each bug in the Defects4J dataset, we take the buggy version of the project and collect the ground truth labels for the buggy and non-buggy methods. Next, for each of the six projects in Defects4J, we combine all the ground truth labels from the bugs of those projects. For example, for Apache commons-lang project, we combine the labels from all the 59 bugs. Then, we simulate the defect predictor outcomes using the Algorithm 3 for each of the six projects in separate.

Our intended application scenario is generating tests to detect bugs that already exist in the system. Hence, we run test generation on the buggy version of the projects. Since we are measuring the bug detection capability of both approaches only on the Defects4J bugs, we do not run test generation on the non-buggy classes, i.e., classes that are not modified in the bug fixes of the Defects4J bugs.

To take the randomness of SBST into account, we repeat each test generation run 25 times. Due to the randomness of the Defect Prediction Simulation Algorithm, we repeat the simulation runs 5 times for the recall=precision=75% experiments. For each of these simulated defect predictor instances, we repeat test generation runs 5 times. Consequently, we have to run a total of 3 (approaches) $*$ 25 (repetitions) $*$ 482 (buggy classes) = 36,150 test generations.

We evaluate if the 36,150 generated test suites detect the selected Defects4J bugs by using the interfaces provided by Defects4J [29]. First, the flaky test cases are removed from the test suites using the 'fix test suite' interface in Defects4J [29] as described in [5]. We use the fixed versions of the programs as the test oracles [53]. If a test suite running against the buggy version of a program produces a different output compared to what it produces when it is run against the fixed version, then it means the test suite detects the bug. The 'run bug detection' interface uses the fixed version as the test oracle and determines if a test suite detects a bug by comparing the test execution results between the two versions. EvoSuite generates assertions assuming the program under test is correct, therefore the generated tests should always pass when they are run against the buggy version. A test suite is considered broken, if it is not compilable or fails when it is run against the buggy version. The test suite is considered it has missed detecting the bug, if the test execution results are same when it is run against the buggy and fixed versions of the program, if the results are different, then it is considered as it has detected the bug.

The 'run bug detection' interface logs the test cases that produce different test execution results when run against the buggy and fixed versions. We configure both PreMOSA and DynaMOSA to log the time taken to generate each test case since the start of the search (in milliseconds). Fig. 4 shows a sample test case with the time taken to generate it logged as a comment. Hence, we can find the time taken to generate test cases that detect the bugs by each approach, which will be used to evaluate the efficiency of the two approaches (*RQ2*).

6 RESULTS

We present the results for our research questions following the method described in Section 5. Our main aim is to

```
@Test(timeout = 4000)
public void test006()  throws Throwable  {
    // time taken = 20971
    try {
        NumberUtils.createNumber("0x");
        fail("Expecting exception: NumberFormatException");

    } catch(NumberFormatException e) {
        //
        // For input string: \"\"
        //
        verifyException("java.lang.NumberFormatException", e);
    }
}
```

Fig. 4. A sample test case with the time taken to generate.

evaluate if PreMOSA is more effective and efficient than the state-of-the-art DynaMOSA.

RQ1: Is PreMOSA Effective in Detecting Bugs?

As we described in Section 5, we perform 25 runs of Pre-MOSA using defect predictions at recall=precision=75% (PreMOSA-75) and recall=precision=100% (PreMOSA-100), and DynaMOSA against each buggy program in Defects4J (Section 5.3) and report the bug detection results as boxplots in Fig. 5. As we can see, both PreMOSA-100 and PreMOSA-75 detect more bugs than DynaMOSA.

We report the means and medians of the number of bugs detected and the results from the statistical analysis in Table 1. DynaMOSA detects 197.16 bugs on average in 2 minutes. PreMOSA-100 and PreMOSA-75 outperform DynaMOSA, and detect 213.56 and 212.6 bugs on average, which are average improvements of 16.4 (+8.3%) and 15.44 (+7.8%) more bugs than DynaMOSA, respectively. The differences of the number of bugs detected by PreMOSA-100/ PreMOSA-75 and DynaMOSA are statistically significant according to the Mann-Whitney U-Test (p-value <0.0001) with large effect sizes ($\widehat{A}_{12} \geq 0.98$). Thus, we conclude that PreMOSA is significantly more effective than DynaMOSA
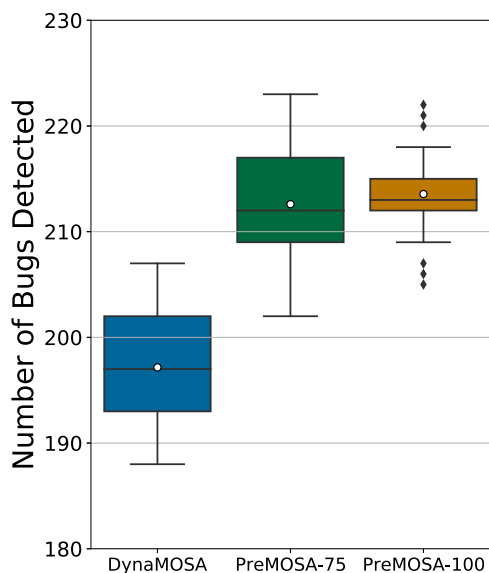

Fig. 5. The number of bugs detected by PreMOSA and DynaMOSA in 2 minutes time budget.

TABLE 1
Mean and Median Number of Bugs Detected by PreMOSA and DynaMOSA in 2 Minutes Time Budget

	Mean	Median		p-value	$\widehat{A}_{12}$
PreMOSA-100	**213.56**	**213**	PreMOSA-100 versus DynaMOSA	<**0.0001**	**0.99**
PreMOSA-75	**212.6**	**212**	PreMOSA-75 versus DynaMOSA	<**0.0001**	**0.98**
DynaMOSA	197.16	198	PreMOSA-100 versus PreMOSA-75	0.5512	0.53

when using any acceptable defect predictor (i.e., recall, precision $\geq$ 75%).

PreMOSA-75 detects only 0.96 (-0.4%) less bugs on average than PreMOSA-100. According to the one-tailed Mann-Whitney U-Test, this difference is not statistically significant (p-value = 0.5512), and the $\widehat{A}_{12}$ statistic indicates a negligible effect size of 0.53. Therefore, we can confirm PreMOSA successfully accounts for errors in the predictions of defect predictors in the acceptable range.

Certain bugs are harder to detect than others. We identify a bug as a unique bug if it is only detected by one approach, i.e., PreMOSA or DynaMOSA. The number of unique bugs detected by an approach is an indication of the ability of that approach to detect the bugs that are not detected otherwise in the given time budget, which is an important strength given how hard it is to detect a bug [54].

Table 2 shows a summary of the bug detection results of PreMOSA and DynaMOSA. PreMOSA-100 and PreMOSA-75 detect 287 and 292 bugs altogether, which is 68.3% and 69.5% of the total bugs respectively, whereas DynaMOSA detects only 280 (66.7%) bugs. PreMOSA-100 detects 17 unique bugs that DynaMOSA cannot detect in any of the runs, whereas DynaMOSA only detects 10 such unique bugs. Similarly, PreMOSA-75 detects 22 unique bugs that are not detected by DynaMOSA, whereas DynaMOSA only detects 10 unique bugs that PreMOSA-75 cannot detect in any of the runs. This shows that PreMOSA is capable of detecting more bugs that are not detected by DynaMOSA.

We find that PreMOSA-100 detects less bugs in total and less unique bugs than PreMOSA-75 when the bugs are isolated in buggy methods with private access modifier (i.e., private buggy methods). For example, PreMOSA-75 detects Closure-25, 50, 55, 57, 67, 68, 143, 154 bugs, which all have only private buggy methods, while PreMOSA-100 detects none of them. PreMOSA-100 starts the search for test cases to cover only the buggy targets. When all the buggy targets are in private methods, PreMOSA-100 has only limited guidance to cover these targets since it cannot directly call the private buggy methods. PreMOSA-100 will get further

TABLE 2
Summary of the Bug Detection Results at 2 Minutes

	Bugs detected	Bugs detected in every run	Unique bugs
PreMOSA-100	**287**	**140**	**17**
PreMOSA-75	**292**	**127**	**22**
DynaMOSA	280	114	10

TABLE 3
Mean and Median Difference of Time Taken to Generate Bug
Detecting Tests by PreMOSA and DynaMOSA

	Mean (s)	Median (s)	p-value	r
PreMOSA-100 versus DynaMOSA	**2.59**	**0.22**	**0.0016**	0.18
PreMOSA-75 versus DynaMOSA	**2.02**	**0.05**	**0.0347**	0.11

guidance to cover these targets only after the non-buggy targets are added to the search (line 19 in Algorithm 1). It will be able to indirectly call the buggy targets in private methods through non-buggy methods with non-private access modifier. In contrast, PreMOSA-75 is more likely to start with non-buggy targets incorrectly predicted as buggy or all likely non-buggy targets (line 7 in Algorithm 1). This means PreMOSA-75 has a better chance of having more guidance to cover buggy targets in private methods from the beginning of the search compared to PreMOSA-100, and as a result, it is able to detect more bugs in total and more unique bugs than PreMOSA-100.

If we consider a bug as detected only if all the 25 runs by an approach detect that bug (i.e., success rate = 1.0), then the number of bugs detected by PreMOSA-100 and PreMOSA-75 becomes 140 and 127 respectively, whereas it is only 114 bugs for DynaMOSA. We further find that PreMOSA-100 detects 108 bugs more times than DynaMOSA, while for DynaMOSA, this is only 62 bugs. Similarly, PreMOSA-75 detects 124 bugs more times than DynaMOSA, whereas it is only 61 bugs for DynaMOSA. Altogether, this demonstrates that PreMOSA is also more robust in detecting bugs when compared to DynaMOSA.

> In summary, PreMOSA is significantly more effective than the state-of-the-art DynaMOSA with large effect sizes when using any acceptable defect predictor. The superior performance of PreMOSA is supported by both its capability to detect new bugs that are not detected by DynaMOSA and the robustness of the approach.

RQ2: Is PreMOSA Efficient in Generating Test Cases That can Detect Bugs?

As described in Section 5, for each approach, we calculate the mean time to generate the first test case that detects each bug. In the case of a bug that is detected by both PreMOSA-100 and DynaMOSA, we then calculate the difference of the mean times to generate the first bug detecting test by the two approaches, i.e., mean time to generate the first bug detecting test by DynaMOSA - mean time to generate the first bug detecting test by PreMOSA-100. We repeat the same procedure for PreMOSA-75 and DynaMOSA as well. If the difference is positive, that means PreMOSA generates a bug detecting test in a shorter time on average. A negative difference means PreMOSA has a worst performance.

We report the means and medians of the differences of the time taken to generate bug detecting tests and the results from the statistical analysis in Table 3. The average difference of mean time to generate bug detecting tests between PreMOSA-100 and DynaMOSA is 2.59 seconds, and it is 2.02 seconds between PreMOSA-75 and DynaMOSA.
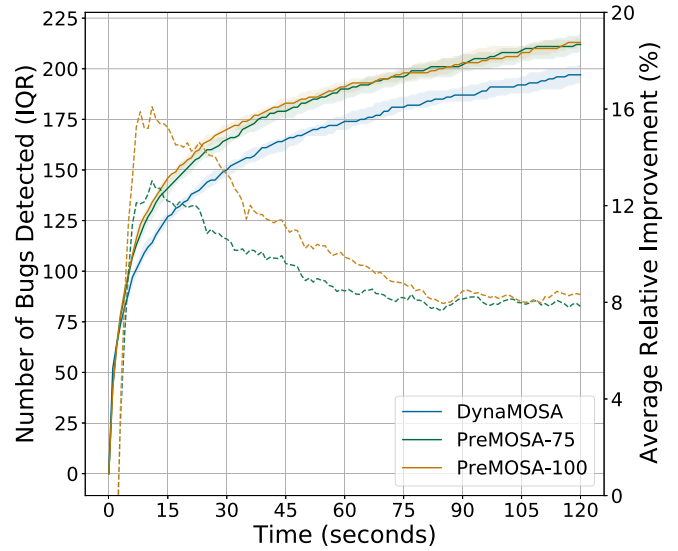

Fig. 6. The number of bugs detected by PreMOSA and DynaMOSA over the time budget spent.

According to the one-tailed Wilcoxon signed-rank test, these differences are statistically significant with p-values <0.05. However, we find that the effect sizes (i.e., r) estimated using the Wilcoxon signed-rank test are small. The effect size of the difference of mean time to generate bug detecting tests between PreMOSA-100 and DynaMOSA is 0.18, which translates to approximately 60% probability of PreMOSA-100 generating a bug detecting test faster than DynaMOSA [40]. The effect size of 0.11 between PreMOSA-75 and DynaMOSA suggests that PreMOSA-75 generates a bug detecting test faster than DynaMOSA approximately 56% of the time. Therefore, we can conclude PreMOSA is significantly faster than DynaMOSA to generate a bug detecting test when using any acceptable defect predictor.

The above analysis is carried out with respect to the time to generate bug detecting test for each bug that is detected by all the approaches in the comparison. In addition, we also analyse the efficiency of PreMOSA and DynaMOSA with respect to the number of bugs detected over the time budget spent, which includes all the bugs in the dataset.

Fig. 6 shows the median number of bugs detected by each approach over the time budget spent. The number of bugs detected by an approach x ($x \in$ {PreMOSA, DynaMOSA}) at time t ($t \in [0, 120]$) is equal to the number of bugs that can be detected by the tests generated by x after t seconds of test generation. The shaded area around the curves depicts the interquartile range. The dashed lines depict the average improvements of PreMOSA-100 and PreMOSA-75 relative to the baseline DynaMOSA.

In the first 2 seconds, DynaMOSA has a head start, due to the slight additional overhead of PreMOSA in filtering targets and calculating number of independent paths (Sections 4.1 and 4.3). However, both PreMOSA-100 and PreMOSA-75 outperform DynaMOSA after 2 seconds.

According to the Mann-Whitney U-Test ($\alpha = 0.05$), PreMOSA-100 and PreMOSA-75 detect significantly more bugs than DynaMOSA with large effect sizes ($\widehat{A}_{12} \geq 0.87$) at any time after 3 seconds. This confirms that PreMOSA not only detects more bugs than DynaMOSA at the end of 120

seconds, but also is ahead of DynaMOSA from the very beginning of the search (i.e., after 2 seconds).

The relative improvement by PreMOSA using any acceptable defect predictor is much higher when it is given a tight time budget. In particular, the average relative improvements of PreMOSA-100 and PreMOSA-75 reach maximums of 16.1% and 13.0% at 11 seconds respectively. We also find that both PreMOSA-100 and PreMOSA-75 have an average improvement more than 10% in the interval of 6 and 38 seconds. This further demonstrates the increased efficiency of PreMOSA compared to DynaMOSA, such that the large improvements of PreMOSA occur when it is given tight time budgets like in a usual resource constrained scenario.

In summary, PreMOSA is significantly more efficient than the state-of-the-art DynaMOSA with small effect sizes when using any acceptable defect predictor. Overall, PreMOSA not only detects more bugs than DynaMOSA when they are given a reasonably large time budget, but also when they are given tight time budgets like in a resource constrained environment.

---

7 DISCUSSION

The execution time of PreMOSA is comprised of the time taken by the defect predictor and the execution time of the search process. With simulated defect predictors, it is not possible to know the execution time of an actual defect predictor. Also, the run-time of an actual defect predictor changes from one model to another model depending on several factors like the classifier used in the model etc. Therefore, in the experimental evaluation, we do not account for the time taken by the defect predictor, and allocate the full time budget of 2 minutes to the search process. However, we find that PreMOSA with an acceptable defect predictor reaches the final number of bugs detected by DynaMOSA in 79.2 seconds on average. This suggests that even if the defect predictor takes 40.8 seconds to run on average per CUT, PreMOSA will still perform on par with DynaMOSA. Furthermore, Perera *et al.* [8] reported the defect predictor used in their study spent 0.68 seconds per class on average (with a standard deviation of 0.4 seconds). Therefore, the execution time of an actual defect predictor is not expected to affect the conclusions of this study.

PreMOSA is guided by coverage and defect prediction information. It first attempts to cover the likely buggy targets and starts finding tests to cover likely non-buggy targets once it deems to have searched enough in likely buggy targets. On the other hand, DynaMOSA is only guided by coverage and aims at maximising code coverage. In our experiments, PreMOSA-100, PreMOSA-75 and DynaMOSA achieved 57.89%, 59.14% and 62.94% branch coverage of the classes under test on average, respectively.

In the experimental evaluation, we do not consider additional cost factors such as the effort required to insert test oracles manually or automatically and the execution time of test suites. PreMOSA generates more than one test case for each target in the CUT and retains all these test cases. DynaMOSA is also configured to do the same as described in

Section 5.4. In our experiments, PreMOSA-100, PreMOSA-75 and DynaMOSA generate 12548, 13004 and 14344 test cases on average per test suite, respectively. Both PreMOSA and DynaMOSA are implemented in EvoSuite, which generates assertions in the tests assuming the program under test is correct. EvoSuite uses a mutation-based assertion filtering strategy to minimise the number of assertions in the generated test suites. However, we disable this in our experiments since it can be computationally expensive and can lead to timeouts. Therefore, in the experiments, there are 1,416,817, 1,462,391 and 1,277,024 assertions generated on average per test suite by PreMOSA-100, PreMOSA-75 and DynaMOSA, respectively. In practice, these assertions need to be updated manually or automatically for generated tests to reveal the bugs, which can be problematic when the test suites become large. Appropriate test suite minimisation techniques can be applied to the test suites generated by PreMOSA to mitigate this problem.

For completeness, we report the accuracy and Matthews correlation coefficient (MCC) [55] of the defect predictors used in PreMOSA. For recall=precision=100%, the accuracy of the defect predictor is 100%, and for recall=precision=75%, the accuracy is on average 99.97%. A high accuracy is observed for the defect predictor with recall=precision=75% because of the highly imbalanced nature of the Defects4J dataset, which we discuss in threats to construct validity (Section 8). MCC of the recall=precision=100% and recall=precision=75% predictors are 1.0 and 0.75 on average, respectively.

The baseline method, DynaMOSA, does not use a defect predictor and aims to cover all the targets in the CUT equally. This means that in the eyes of DynaMOSA, all the methods in a class are likely buggy, which translates to a 100% recall and precision per project as follows; Lang - 0.06%, Math - 0.03%, Time - 0.05%, Chart - 0.02%, Closure - 0.02% and Mockito - 0.15%.

8 THREATS TO VALIDITY

Construct Validity. The defect prediction simulator assumes a uniform distribution of predictions. This means each method has an equal chance of being labelled as buggy or non-buggy by the simulator. However, the prediction distributions of real defect predictors are likely to deviate from a uniform distribution depending on the underlying characteristics and nature of the prediction problem. This could impact the realism of our defect prediction simulations. Nevertheless, in the absence of prior knowledge about defect prediction distributions, the reasonable choice is to assume a uniform distribution of predictions.

In the experimental evaluation, we simulate defect predictors for a given recall and precision, and consider an acceptable defect predictor with respect to these metrics (Section 5.1). Recall and precision have been widely used in previous work to report the performance of defect predictors [37], [56]. However, recall and precision can be biased in the case of highly imbalanced datasets, which is usually a commonplace situation for defect datasets as there are only a few number of buggy methods compared to non-buggy ones [57]. Thus, future works need to be done with defining and simulating acceptable defect predictors with unbiased

performance metrics like Matthews correlation coefficient (MCC) [55]. Even though our experiments are designed to be driven by recall and precision, the MCC corresponds to 0.75 on average for the most conservative defect predictor (i.e., recall=precision=75%), and 1.0 for the ideal defect predictor.

We only consider the labelled bugs in the Defects4J dataset in our experimental evaluation, which is likely smaller than the set of actual bugs in the dataset. In order to check how many actual bugs are detected by PreMOSA and DynaMOSA, we have to manually validate all the 36,150 generated test suites, which is not a feasible task. Therefore, in line with previous work [7], [45], we choose to conduct the experimental evaluation considering only the labelled bugs in Defects4J dataset.

Internal Validity. To account for the randomness of the simulated defect predictor, we repeat the simulations for 5 times for recall=precision=75% configuration. Then, for each simulation, we repeat the test generation for 5 times to account for the non-deterministic behaviour of PreMOSA. For PreMOSA-100 and DynaMOSA, we repeat the test generation runs for 25 times to account for the non-deterministic behaviour of the two techniques. To derive conclusions from the results of our experiments, we conduct sound statistical tests; one-tailed non-parametric Mann-Whitney U-Test, Vargha and Delaney's $\widehat{A}_{12}$ statistic, one-tailed Wilcoxon signed-rank test, and its effect size, r.

The parameter *maximum number of iterations without coverage improvement* (I) in PreMOSA is configured based on the results of our pilot runs. We expect the performance of PreMOSA can be further improved by fine-tuning the parameter.

A threat to internal validity exists from the use of the term *experiment* in our study. According to the hallmarks characterised by Ayala *et al.* [58], our study corresponds to an *experiment with limited control*. This is because we use a retrospective repository (i.e., Defects4J) as the dataset, hence our experimental design does not fully cover the *control* hallmark [58].

External Validity. Our experimental evaluation is done using 420 real bugs from Defects4J dataset. These bugs are drawn from 6 open source projects. At the time of writing this paper, 401 more bugs were added to the Defects4J benchmark from additional 11 projects. Nevertheless, these open source projects do not represent all program characteristics, especially industrial projects. However, Defects4J has been widely used in related literature as a benchmark [5], [7], [8], [23], [47], [59]. We expect that future work needs to be done on evaluating the performance of PreMOSA on other bug datasets.

We implement PreMOSA in the state-of-the-art SBST tool, EvoSuite, that generates JUnit test suites for Java programs. Therefore, we may not be able to generalise our findings to other programming languages. However, the concept behind PreMOSA is not language dependent and can be applied to other object oriented programming languages.

Our findings may not be generalised to the defect predictors which have recall or precision less than 75%. We experimentally assess the bug detection performance of PreMOSA when using theoretically most conservative and acceptable defect predictors (recall=precision=75%) and ideal defect predictor (recall=precision=100%). The experimental results demonstrate the improved performance of PreMOSA when using either of the defect predictors, which suggest PreMOSA is significantly better at detecting bugs than DynaMOSA when using defect predictors having recall and precision greater than 75%. We choose 75% recall and precision as the lower bound for an acceptable defect predictor with the justification that Zimmermann *et al.* [25] recommended only the defect predictors having recall and precision more than 75% as acceptable defect predictors.

9 RELATED WORK

9.1 Search-Based Software Testing

Search-based software testing (SBST) techniques use meta heuristics search algorithms like genetic algorithms (GA) to search for high quality test cases for a specific criteria (e.g., maximise branch coverage). Mainly, this test generation problem can be formulated in two ways; i) single objective formulation [26], [36], and ii) many-objective formulation [2], [3]. In the latter one, the test generation problem is approached as a many objective optimisation problem whereas the single objective formulation is used in the whole test suite approaches [26], [36]. In particular, Panichella *et al.* [2] proposed many objective sorting algorithm (MOSA) which simultaneously optimises test cases to satisfy hundreds of coverage targets (e.g., branch coverage goals). Previous work shows that these many objective sorting algorithms [2], [3] are more effective and efficient than whole test suite approaches in terms of code coverage [2], [3], [4]. In this paper, we formulate the test generation problem as a many objective optimisation problem and develop PreMOSA as a many objective solver.

DynaMOSA [3] is the successor of the many objective solver, MOSA, and stands as the state-of-the-art SBST technique. It considers all the coverage targets in the class under test (CUT) as equally important to cover. Hence, it simultaneously optimises test cases to cover all of the targets in the CUT. However, only one or few methods in a class are buggy, hence, it is likely to be ineffective to search for tests to cover targets that contain non-buggy methods. Contrary to DynaMOSA, PreMOSA initially searches for tests in likely buggy methods, and introduces targets containing likely non-buggy methods to the search only when it deems to have searched enough for tests to cover likely buggy targets.

Results from previous work show that SBST techniques have limitations in terms of bug detection [5], [6], [7]. We argue that aiming at maximising code coverage alone is not sufficient to maximise the number of bugs detected. In particular, Salahirad *et al.* [7] showed that EvoSuite [36] - a state-of-the-art SBST tool - is more effective when it is using fitness functions based on maximising branch coverage compared to other coverage criteria. However, EvoSuite only detected an average of 25% bugs from the Defects4J dataset in a 10 minutes time budget even when using branch coverage in the fitness function, suggesting that code coverage alone is not enough to effectively detect bugs.

9.2 Defect Prediction

Zimmermann *et al.* [25] argued a defect predictor with recall, precision and accuracy greater than 75% is a strong

defect predictor, and vice versa. Results from previous work suggest that 75% recall and precision is an achievable level of performance for a defect predictor [13], [17], [19]. In particular, Giger et al. [13] reported the prediction models built with change metrics can locate buggy methods with 88% and 84% of recall and precision, respectively. In our study, we consider a defect predictor is acceptable if its recall and precision are greater than 75%.

Previous work on integrating defect prediction and software testing have used single instances of defect predictors in their experimental evaluations [8], [23], [24]. The purpose of our study is not to find the best defect predictor to be used in PreMOSA. In our experimental evaluation, we intentionally abstract the defect predictor in PreMOSA to avoid unfair evaluation and bias from using a single defect predictor. We simulate the defect predictor outcomes for different levels of performance. In particular, we simulate defect predictor outcomes at i) the most conservative performance of an acceptable defect predictor, i.e., recall=precision=75% and ii) the ideal performance, i.e., recall=precision=100%. The conclusions derived from our results cover the full spectrum of acceptable defect predictors.

There is plethora of defect predictors working at coarse-grained level such as file and class levels [20], [22], [60]. Hata et al. [12] hypothesise that the effort required to find bugs using coarse-grained predictions is higher than using fine-grained predictions such as method level. Their method level prediction model built with historical metrics was shown to outperform package and file level predictors in terms of effort required to find bugs. Similar results were also observed by Caglayan et al. [18] when using pre-release bugs related metrics to build defect prediction models. Any of these defect predictors are suitable to be used in PreMOSA.

9.3 Defect Prediction in Software Testing

Defect predictors are shown to be effective at locating bugs in software [10], [18], [19]. As a result, they have been used in the industry to support developers in code reviews [20], [21] and in testing [22]. While the main assumption of defect prediction is to provide useful information to developers [20], prediction outcomes have also been used successfully in automated testing techniques [8], [23], [24], [44], [53].

Perera et al. [8] and Hershkovich et al. [24] introduced defect prediction guided time budget allocation approaches for SBST. Our work is the first to use defect prediction information in the search process of SBST to effectively guide the search for test cases to likely buggy methods. Our proposed SBST technique, PreMOSA, is orthogonal to the aforementioned budget allocation approaches [8], [24], and can be used together by simply replacing the SBST components in each of the approaches with PreMOSA to further improve the performance of them.

G-clef [23] is a test prioritisation strategy that uses a defect predictor based on change history related metrics and prioritises test cases in terms of their likelihood of finding bugs. It was shown to be effective at reducing the number of test cases required to find bugs. FaRM [53] is a mutant selection technique that selects and ranks fault revealing mutants using prediction models based on source code metrics. It was shown to outperform the state-of-the-art mutant selection and mutant prioritisation methods in

terms of revealing faults. FLUCCS [44] is a fault localisation approach that ranks methods according to their likelihood of being faulty using pre-trained models based on source code and change metrics. It was shown to outperform the state-of-the-art spectrum based fault localisation (SBFL) techniques. All these approaches are applied after the test generation step, i.e., G-clef and FaRM can be used to prioritise and select test cases, and FLUCCS can be applied to localise the fault once a bug is detected through test generation. In contrast, PreMOSA uses defect prediction in the test generation phase.

Perera et al. [61] incorporated buggy method predictions outside of the search process in DynaMOSA and investigated the impact of defect prediction imprecision on the bug detection performance of DynaMOSA guided by defect predictions. The bug detection effectiveness of DynaMOSA significantly decreased as the recall of the defect predictor decreased, while the effect of precision was not practically significant. In contrast, PreMOSA uses buggy method predictions inside the search process, i.e., balancing between likely buggy and non-buggy targets, and the experimental results indicate that its bug detection performance is not significantly impacted by the change of defect predictor performance from recall=precision=100% to recall=precision=75%. This shows that PreMOSA is able to successfully handle the potential errors in the predictions, while DynaMOSA with simply filtering out the likely non-buggy targets suffers from loss of recall of the predictor.

10 CONCLUSION

We hypothesise that augmenting coverage information with defect prediction information in the search process of SBST improves the bug detection performance of the generated test suites. We develop a many-objective solver for test generation called predictive many objective sorting algorithm (PreMOSA) that uses buggy methods predictions to decide where to increase the test coverage in the CUT. We experimentally assess the performance of PreMOSA when using defect predictors having the theoretical upper and lower bound performance of acceptable defect predictors. We validate our technique against 420 labelled bugs from Defects4J dataset. Our experimental evaluation demonstrates that PreMOSA is significantly more effective than the state-of-the-art DynaMOSA with large effect sizes when using any acceptable defect predictor. In particular, it detects 8.3% and 7.8% more labelled bugs on average than DynaMOSA when using an ideal defect predictor and most conservative and acceptable defect predictor, respectively. We also find PreMOSA is significantly more efficient than DynaMOSA.

The performance of PreMOSA does not decrease significantly when replacing the ideal defect predictor (i.e., recall=precision=100%) with most conservative defect predictor in the acceptable range (recall=precision=75%). On the other hand, if defect predictions with errors, i.e., false positives and false negatives, are directly used by developers, e.g., in code reviews and manual testing, it can lead to waste of developer time, miss important bugs, etc [20]. Our results show that PreMOSA successfully accounts for errors in the predictions of defect predictors that are considered acceptable [15].

We find that after 60 seconds of time budget, there is no significant difference in the performances of PreMOSA with an ideal defect predictor and with the most conservative defect predictor. Therefore, in the context of combining defect prediction and SBST, we recommend practitioners to not focus on improving the defect predictor performance beyond 75% recall and precision if their testing resources allow reasonably large time budget for test generation. On the other hand, if there is a tight time budget for test generation, then improving the defect predictor performance would further improve the bug detection performance of PreMOSA.

We identify the following directions as future works to extend this study; i) integrate PreMOSA in a continuous integration environment, ii) adapt an appropriate test suite minimisation technique to address the generation of large test suites, iii) define and simulate acceptable defect predictors with respect to unbiased performance metrics like MCC, and iv) validate PreMOSA against other bug datasets [62], [63], [64].

REFERENCES

[1] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.
[2] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating branch coverage as a many-objective optimization problem," in *Proc. IEEE 8th Int. Conf. Softw. Testing, Verification Validation*, 2015, pp. 1–10.
[3] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.
[4] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," *Inf. Softw. Technol.*, vol. 104, pp. 236–256, 2018.
[5] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges (T)," in *Proc. 30th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2015, pp. 201–211.
[6] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *Proc. 39th Int. Conf. Softw. Eng.: Softw. Eng. Pract. Track*, 2017, pp. 263–272.
[7] A. Salahirad, H. Almulla, and G. Gay, "Choosing the fitness function for the job: Automated generation of test suites that detect real faults," *Softw. Testing, Verification Rel.*, vol. 29, no. 4–5, 2019, Art. no. e1701.
[8] A. Perera, A. Aleti, M. Böhme, and B. Turhan, "Defect prediction guided search-based software testing," in *Proc. 35th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2020, pp. 448–460.
[9] A. Schröter, T. Zimmermann, and A. Zeller, "Predicting component failures at design time," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng.*, 2006, pp. 18–27.
[10] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.
[11] P. A. F. de Freitas, "Software repository mining analytics to estimate software component reliability," Faculty of Engineering, University of Porto, Tech. Rep., 2015.
[12] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 200–210.
[13] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *Proc. ACM-IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2012, pp. 171–180.
[14] T. Menzies, J. Greenwald, and A. Frank, "Data mining static code attributes to learn defect predictors," *IEEE Trans. Softw. Eng.*, vol. 33, no. 1, pp. 2–13, Jan. 2007.
[15] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *Proc. 3rd Int. Workshop Predictor Models Softw. Eng.*, 2007, Art. no. 9.
[16] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
[17] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 521–530.
[18] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini, "Merits of organizational metrics in defect prediction: An industrial replication," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 89–98.
[19] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy, "Change bursts as defect predictors," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 309–318.
[20] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr, "Does bug prediction support human developers? Findings from a Google case study," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 372–381.
[21] C. Lewis and R. Ou, "Bug prediction at Google," 2011, Accessed: Sep. 16, 2019. [Online]. Available: http://google-engtools.blogspot.com
[22] H. K. Dam et al., "Lessons learned from using a deep tree-based model for software defect prediction in practice," in *Proc. 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 46–57.
[23] D. Paterson, J. Campos, R. Abreu, G. M. Kapfhammer, G. Fraser, and P. McMinn, "An empirical study on the use of defect prediction for test case prioritization," in *Proc. 12th IEEE Conf. Softw. Testing, Validation Verification*, 2019, pp. 346–357.
[24] E. Hershkovich, R. Stern, R. Abreu, and A. Elmishali, "Prediction-guided software test generation," in *Proc. 30th Int. Workshop Princ. Diagnosis*, 2019, Accessed: Feb. 08, 2022. [Online]. Available: https://dx-workshop.org/2019/
[25] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data versus domain versus process," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp.*, 2009, pp. 91–100.
[26] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Softw. Eng.*, vol. 22, no. 2, pp. 852–893, 2017.
[27] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 1990.
[28] P. McMinn, "Search-based software testing: Past, present and future," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshops*, 2011, pp. 153–163.
[29] R. Just, "Defects4J - A database of real faults and an experimental infrastructure to enable controlled experiments in software engineering research," 2019, Accessed: Oct. 02, 2019. [Online]. Available: https://github.com/rjust/defects4j
[30] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.
[31] L. J. Morell, "A theory of fault-based testing," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 844–857, Aug. 1990.
[32] L. J. Morell, "A theory of error-based testing," Dept. Comput. Sci., Maryland Univ. College Park, MD, USA, Tech. Rep. TR-1395, 1984.
[33] A. Offutt, "Automatic test data generation," Georgia Institute of Technology, Tech. Rep., 1989.
[34] N. Li and J. Offutt, "Test Oracle strategies for model-based testing," *IEEE Trans. Softw. Eng.*, vol. 43, no. 4, pp. 372–395, Apr. 2017.
[35] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle problem in software testing: A survey," *IEEE Trans. Softw. Eng.*, vol. 41, no. 5, pp. 507–525, May 2015.
[36] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *Proc. 11th Int. Conf. Qual. Softw.*, 2011, pp. 31–40.
[37] S. Hosseini, B. Turhan, and D. Gunarathna, "A systematic literature review and meta-analysis on cross project defect prediction," *IEEE Trans. Softw. Eng.*, vol. 45, no. 2, pp. 111–147, Feb. 2019.
[38] A. Arcuri and L. Briand, "A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Testing, Verification Rel.*, vol. 24, no. 3, pp. 219–250, 2014.
[39] A. Vargha and H. D. Delaney, "A critique and improvement of the "CL" common language effect size statistics of McGraw and Wong," *J. Educ. Behav. Statist.*, vol. 25, no. 2, pp. 101–132, 2000.
[40] C. O. Fritz, P. E. Morris, and J. J. Richler, "Effect size estimates: Current use, calculations, and interpretation." *J. Exp. Psychol.: Gen.*, vol. 141, no. 1, pp. 2–18, 2012.

[41] J. Campos, A. Arcuri, G. Fraser, and R. Abreu, "Continuous test generation: Enhancing continuous integration with automated test generation," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 55–66.

[42] M. Fowler and M. Foemmel, "Continuous integration," 2006, Accessed: Feb. 10, 2022. [Online]. Available: https://www.martinfowler.com/articles/continuousIntegration.html

[43] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.

[44] J. Sohn and S. Yoo, "Empirical evaluation of fault localisation using code and change metrics," *IEEE Trans. Softw. Eng.*, 2019, vol. 47, no. 8, pp. 1605–1625, Aug. 2021.

[45] G. Gay, "The fitness function for the job: Search-based generation of test suites that detect real faults," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation.*, 2017, pp. 345–355.

[46] A. Aleti and M. Martinez, "E-APR: Mapping the effectiveness of automated program repair," *Empirical Softw. Eng.*, vol. 26, no. 5, pp. 1–30, 2021.

[47] S. Pearson et al., "Evaluating and improving fault localization," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 609–620.

[48] J. Campos, A. Panichella, and G. Fraser, "EvoSuite at the SBST 2019 tool competition," in *Proc. 12th Int. Workshop Search-Based Softw. Testing*, 2019, pp. 29–32.

[49] EvoSuite, "EvoSuite - Automated generation of Junit test suites for Java classes," 2019, Accessed: Nov. 29, 2019. [Online]. Available: https://github.com/EvoSuite/evosuite

[50] G. Fraser, "Evosuite - Automatic test suite generation for Java," Accessed: Sep. 19, 2019. [Online]. Available: http://www.evosuite.org/

[51] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using EvoSuite," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 2, pp. 1–42, 2014.

[52] A. Arcuri and G. Fraser, "Parameter tuning or default values? An empirical investigation in search-based software engineering," *Empirical Softw. Eng.*, vol. 18, no. 3, pp. 594–623, 2013.

[53] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Softw. Eng.*, vol. 25, no. 1, pp. 434–487, 2020.

[54] A. Habib and M. Pradel, "How many of all bugs do we find? A study of static bug detectors," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 317–328.

[55] J. Yao and M. Shepperd, "Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters," in *Proc. Eval. Assessment Softw. Eng.*, 2020, pp. 120–129.

[56] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.

[57] M. Shepperd, D. Bowes, and T. Hall, "Researcher bias: The use of machine learning in software defect prediction," *IEEE Trans. Softw. Eng.*, vol. 40, no. 6, pp. 603–616, Jun. 2014.

[58] C. Ayala, B. Turhan, X. Franch, and N. Juristo, "Use and misuse of the term experiment in mining software repositories research," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2021.3113558.

[59] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol. Reeng.*, 2016, pp. 213–224.

[60] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Trans. Softw. Eng.*, vol. 22, no. 10, pp. 751–761, Oct. 1996.

[61] A. Perera, B. Turhan, A. Aleti, and M. Böhme, "How good does a defect predictor need to be to guide search-based software testing?," 2021, *arXiv:2110.02682*.

[62] S. Herbold et al., "Large-scale manual validation of bug fixing commits: A fine-grained analysis of tangling," *CoRR*, vol. abs/2011.06244, 2020. [Online]. Available: https://arxiv.org/abs/2011.06244

[63] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: An extensible Java bug benchmark for automatic program repair studies," in *Proc. 26th IEEE Int. Conf. Softw. Anal., Evol. Reeng.*, 2019, pp. 468–478.

[64] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world Java bugs," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 10–13.

Anjana Perera received the BSc degree (Hons.) in electronics and telecommunication engineering from the University of Moratuwa, Sri Lanka, in 2017. He is currently working toward the PhD degree in software engineering with Monash University, Australia. The goal of his PhD is to use defect prediction to improve the bug detection capability of search-based software testing. He was a software engineer, developing a latency critical, highly scalable and reliable electronic trading system of London Stock Exchange Group, LSEG Technology (formerly known as Millennium IT), until 2018. His research interests include search-based software engineering, automated test generation, and fairness testing of AI systems. He was a PC member of SSBSE 2020 and an artefact evaluation PC member of ASE 2021 and ICSME 2021.

Aldeida Aleti received the PhD degree from the Swinburne University of Technology in 2012. She is currently an associate professor and the associate dean of engagement and impact with the Faculty of Information Technology, Monash University. She has authored or coauthored more than 50 papers in top AI, optimisation and software engineering venues. Her research interests include search-based software engineering (SBSE), with a particular focus on the methodological aspects of how to assess the effectiveness of these techniques. This includes fitness landscape characterisation to analyse how hard SBSE problems are for search techniques, and algorithm selection, which is about identifying which SBSE technique works in what scenario. She was a PC member and an organising committee member of both SE and optimisation conferences, including ASE 2020, 21, ICSE 2019, GECCO 2017, SSBSE 2018, and SSBSE 2019. She has attracted more than $2.500,000 in competitive funding. She was the recipient of the prestigious Discovery Early Career Researcher (DECRA) Award from the Australian Research Council.

Burak Turhan (Senior Member, IEEE) received the PhD degree from Bo gaziçi University. He is currently a professor of software engineering with the University of Oulu and an adjunct professor of research with the Faculty of IT, Monash University, Australia. His research interests include empirical software engineering, software analytics, quality assurance and testing, human factors, and (agile) development processes. He is a senior associate editor for the *Journal of Systems and Software*, an associate editor for *ACM Transactions on Software Engineering and Methodology and Automated Software Engineering*, and an editorial board member of *Empirical Software Engineering*, *Information and Software Technology*, and *the Software Quality Journal*. He is a senior member of ACM.

Marcel Böhme received the PhD degree from the National University of Singapore in 2014. He is currently a lecturer with Monash University, Australia. From 2014 to 2015, he was a research fellow with CISPA, Saarland University, Germany. His research interests include automated vulnerability detection, analysis, testing, debugging, and repair of large software systems. His tools discovered more than 100 bugs in widely used software systems, more than 40 of which are security-critical vulnerabilities registered as CVEs with the U.S. National Vulnerability Database. He is a fellow of ARC DECRA 2019.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.