On the Impact of Lower Recall and Precision in Defect Prediction for Guiding Search-Based Software Testing

ANJANA PERERA, Faculty of Information Technology, Monash University, Australia and Oracle Labs, Australia
BURAK TURHAN, Faculty of ITEE, University of Oulu, Finland and Monash University, Australia
ALDEIDA ALETI, Faculty of Information Technology, Monash University, Australia
MARCEL BÖHME, Max Planck Institute for Security and Privacy, Germany and Monash University, Australia

Defect predictors, static bug detectors and humans inspecting the code can propose locations in the program that are more likely to be buggy before they are discovered through testing. Automated test generators such as search-based software testing (SBST) techniques can use this information to direct their search for test cases to likely-buggy code, thus speeding up the process of detecting existing bugs in those locations. Often the predictions given by these tools or humans are imprecise, which can misguide the SBST technique and may deteriorate its performance. In this paper, we study the impact of imprecision in defect prediction on the bug detection effectiveness of SBST.

Our study finds that the recall of the defect predictor, i.e., the proportion of correctly identified buggy code, has a significant impact on bug detection effectiveness of SBST with a large effect size. More precisely, the SBST technique detects 7.5 fewer bugs on average (out of 420 bugs) for every 5% decrements of the recall. On the other hand, the effect of precision, a measure for false alarms, is not of meaningful practical significance as indicated by a very small effect size.

In the context of combining defect prediction and SBST, our recommendation is to increase the recall of defect predictors as a primary objective and precision as a secondary objective. In our experiments, we find that 75% precision is as good as 100% precision. To account for the imprecision of defect predictors, in particular low recall values, SBST techniques should be designed to search for test cases that also cover the predicted non-buggy parts of the program, while prioritising the parts that have been predicted as buggy.

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

Additional Key Words and Phrases: search-based software testing, automated test generation, defect prediction

ACM Reference Format:
Anjana Perera, Burak Turhan, Aldeida Aleti, and Marcel Böhme. 2024. On the Impact of Lower Recall and Precision in Defect Prediction for Guiding Search-Based Software Testing. *ACM Trans. Softw. Eng. Methodol.* 1, 1 (March 2024), 27 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Authors' addresses: Anjana Perera, Anjana.Perera@monash.edu, Faculty of Information Technology, and Monash University, Melbourne, Australia and Oracle Labs, Brisbane, Australia; Burak Turhan, Burak.Turhan@oulu.fi, Faculty of ITEE, and University of Oulu, Oulu, Finland and Monash University, Australia; Aldeida Aleti, Aldeida.Aleti@monash.edu, Faculty of Information Technology, and Monash University, Melbourne, Australia; Marcel Böhme, marcel.boehme@acm.org, Max Planck Institute for Security and Privacy, Germany and Monash University, Australia.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Association for Computing Machinery.
1049-331X/2024/3-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Defect predictors [22] and static bug detectors [5] can estimate the locations of the bugs effectively. As a result of their efficacy, both defect predictors and static bug detectors are used in the industry to assist developers in manual code reviews [1, 32, 33, 50]. Defect predictors have also been used to inform search-based software testing (SBST) techniques for unit testing; SBST_{DPG} [46] and BTG [26] are time budget allocation techniques for SBST which allocate a higher time budget to highly-likely-to-be-defective classes, and PreMOSA [47] is an SBST technique which uses defect prediction information along with code coverage to decide where to increase the test coverage in the class under test (CUT).

SBST techniques search for test cases to optimise a given coverage criterion such as branch coverage, method coverage, or a combination of the two. SBST techniques are known to be effective at achieving high code coverage [42, 43]. But, while it is necessary for a test case to cover the buggy code to detect a bug, just covering the buggy code may not be sufficient to detect the bug [46, 53]. In fact, SBST techniques guided only by coverage have been shown to struggle in terms of bug detection [3, 46, 52, 53]. This is because the SBST techniques have no guidance in terms of where the buggy code is likely to be located, and hence spend most of the search effort in non-buggy code which constitutes a greater portion of the code base. To address this, previous works have proposed using defect prediction information to direct the search for tests to likely buggy code [26, 46, 47].

Often, the predictions produced by defect predictors are not perfectly accurate. Defect prediction researchers usually aim at elevating both recall and precision. A lower recall and precision can significantly hamper the benefits of defect predictors for the developers who usually manually inspect or test the predicted buggy code to find bugs. Recall is the proportion of correctly identified buggy code [6]. Poor recall of the defect predictor means that there is a higher rate of false negatives (i.e., labelling buggy code as non-buggy). This can lead the developers to completely miss bugs. Precision measures the proportion of actual buggy code among the code labelled as buggy [6]. Poor precision means there is a higher rate of false positives (i.e., wrongly labelling non-buggy code as buggy). This can lead to a waste of developers' time and loss of trust in the defect predictors [32].

Previous work that uses defect predictors to guide SBST techniques reports on improved bug detection performance of SBST [26, 46, 47]. For instance, PreMOSA [47] can detect more unique bugs, i.e., bugs that are detected only by one approach, than the state-of-the-art DynaMOSA [42], i.e., an SBST technique not guided by defect prediction. The defect predictors used in these approaches have a relatively high performance, e.g., the defect predictor used by Perera et al. [46] had a recall of 85%, and Hershkovich et al. [26] employed a defect predictor which had an area under curve (AUC) of 0.95.

The performance of defect predictors, however can vary, e.g., from as low as 5% to as high as 95% of precision and similarly from 25% to 85% for recall [22]. Given such wavering performance, the question that we address in this paper is *"What is the impact of imprecise predictions on the bug detection performance of SBST?"*. We refer to false negatives and false positives as imprecise predictions. False negatives may result in SBST techniques not generating tests for buggy areas in code because they are not labelled as buggy by the predictor. On the other hand, false positives may not be as important in the context of combining defect prediction and SBST, since searching for tests in false positives may not be a significant burden to the automated test generation techniques in contrast to a developer manually inspecting the false positives. The answer to this question is significant because it helps SBST researchers to understand which types of errors in predictions have to be handled in SBST techniques to maximise effectiveness. In addition, the findings of this paper benefit defect prediction researchers by identifying the significantly impactful errors to inform the design of defect predictors when they are used to guide SBST.

To answer this question, we simulate defect predictors for different value combinations of recall and precision in the range 75% and 100% (Section 2.1). Defect predictors having recall and precision above 75% are considered acceptable defect predictors [60]. We employ the state-of-the-art DynaMOSA [42] as the SBST technique in the EvoSuite [18] tool which is guided by defect predictions (DP) (see Section 2.2), which we refer to as *SBST-guided-by-DP* throughout the paper. We evaluate how the bug detection effectiveness of SBST-guided-by-DP changes with the different levels of imprecision when applied to 420 bugs from the Defects4J dataset [28] (Section 3.1).

The results from our experimental evaluation reveal that the recall of the defect predictor has a significant impact on the bug detection effectiveness of SBST with a large effect size. More specifically, SBST-guided-by-DP detects 7.5 fewer bugs on average (out of 420 bugs) for every 5% decrements of recall. On the other hand, the impact of precision is not of practical significance as indicated by a very small effect size, hence we conclude that the precision of defect predictors has negligible impact on the bug detection effectiveness of SBST. Moreover, the impact of precision on SBST remains the same even when SBST is given very small time budgets like 5, 10, 15 and 30 seconds and also when the test suite size is limited to 10, 20, 40 and 80 test cases per test suite. Further analysis into the results reveals that the impact of recall is greater for the bugs that are isolated in one method than for the bugs that are spread across multiple methods.

In summary, the contributions of this work are as follows;

(1) We perform a comprehensive experimental analysis of the impact of imprecision of defect predictions on bug detection effectiveness of SBST. The experimental evaluation involving 420 bugs from 6 open source Java projects took roughly 180,750 CPU-hours in total. The outcomes of our experimental evaluation reveal the following findings;
   (a) The recall of the defect predictor has a significant impact on the bug detection performance of SBST, while the precision of the defect predictor shows no meaningful practical effect on the bug detection performance of SBST.
   (b) The impact of recall on the bug detection performance of SBST is greater for the bugs that are found within one method than for the bugs that are spread across multiple methods.
   (c) Precision does not have a meaningful practical effect on the bug detection performance of SBST even when the time budget and the test suite size are constrained to smaller amounts.

(2) We find that false negatives, i.e., missed bugs by the defect predictor, have the most significant impact on the effectiveness of SBST. Hence it is important for SBST techniques to handle such cases. Currently, the search for tests exploits the likely buggy targets, however, we recommend that SBST techniques also target the likely non-buggy targets at least with a minimum probability. One possible solution is to prioritise predicted buggy parts of the program, while guiding the search with a certain probability towards locations that are predicted as not buggy.

(3) In the context of combining defect prediction and SBST, increasing recall should be the primary objective and increasing precision above 75% can be the secondary objective. When the predictions are used by SBST, a reasonable amount of false positives is not a significant burden to the automated test generation technique. For SBST, it is important to be informed of most of the buggy targets. We recommend the researchers target higher recall while having a sufficiently high precision, instead of trying to elevate both recall and precision at the same time. In our experimental evaluation, we find that the amount of false positives at 75% precision is not a significant overhead for SBST.

The source code of SBST-guided-by-DP, defect predictor simulator, post processing scripts and data are publicly available in the following link: https://doi.org/10.6084/m9.figshare.16564146

2 METHODOLOGY

Our aim is to understand how the defect prediction imprecision impacts the bug detection performance of SBST. To this end, we design a study that addresses the following research question:

RQ: What is the impact of imprecise defect predictions on the bug detection performance of SBST?

To address this research question, we measure the effectiveness of SBST in terms of detecting bugs when using defect predictors with different levels of imprecision. We use the state-of-the-art DynaMOSA [42] as the SBST technique and EvoSuite as the tool. We incorporate predictions about buggy methods in order to guide the search for test cases towards likely buggy methods (see Section 2.2), which we refer as *SBST-guided-by-DP* throughout the paper. Fine-grained defect prediction at a method level is chosen so that the location of the bug is narrowed down better than coarse-grained defect predictions such as class level. The defect predictors at method level provide additional information to the SBST technique such that it can further narrow down the search for test cases to likely buggy methods. SBST-guided-by-DP fully trusts the defect predictor and focuses the search only in parts that are predicted to be buggy. We explain this in more detail in Section 2.2.

We measure defect predictor imprecision using recall and precision. Recall and precision have been widely used in previous work to report the performance of defect predictors [22, 27]. A defect predictor with either recall or precision less than 75% is considered inadequate, as recommended by Zimmermann et al. [60]. We simulate defect predictors for varying levels of recall and precision in the range 75% to 100% (see Section 2.1) and measure the impact on the bug detection performance of SBST by the prediction imprecision.

In addition, we answer the following two sub-research questions to further analyse the impact of the recall and the precision on the bug detection performance of SBST in different testing situations such as limited time budgets, restricted test suite sizes and distribution of buggy methods in a bug.

RQ1: What is the impact of the recall of the defect predictor when the bugs are spread across multiple methods compared to bugs located in a single method?

False negatives in the predictions could mean that the SBST technique misses generating tests for some buggy areas in code because they are not labelled as buggy. This may lead to poorer bug detection performance. In our study, the unit of prediction is a method. If a bug is found only in one method, then it is more likely to be missed by a defect predictor with imperfect recall (i.e., recall < 100%) than a bug spread across multiple methods. This research question analyses the bug detection performance of SBST-guided-by-DP by dividing the bug dataset into two subsets; bugs located in one method and bugs spread across multiple methods.

RQ2: What is the impact of the precision of the defect predictor when the time budget and the test suite size are restricted?

False positives add an additional overhead to the test generation since the SBST technique gives them the same importance as to true positives. The effects of this overhead can be more prominent when the time budget allocated for test generation is limited (e.g., 5, 10, 15 and 30 seconds) and the test suite size, i.e., number of test cases is restricted (e.g., 10, 20, 40 and 80 test cases). In this research question, we analyse how the impact of the precision changes when time budget and test suite size are constrained to smaller amounts.

2.1 Defect Prediction Simulation

To measure the bug detection performance of SBST against the imprecision of defect predictions, we simulate defect predictor outcomes at various levels of performance in the range 75% and 100% for both precision and recall. We do not use real defect predictors in our study because their

performance cannot be controlled to systematically investigate the impact of imprecision of defect prediction. Recall is the proportion of the buggy methods identified by the defect predictor [6]. It is calculated as in Equation. (1), 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.

$$\text{recall} = \frac{tp}{tp + fn} \tag{1}$$

Precision is the proportion of the correctly labelled buggy methods by the defect predictor [6]. It can be calculated as in Equation (2), where fp is the number of false positives, i.e., number of non-buggy methods that are incorrectly classified as buggy methods.

$$\text{precision} = \frac{tp}{tp + fp} \tag{2}$$

We simulate defect predictions from 75% to 100% recall in 5% steps, with 75% and 100% precision. Thus, there are altogether 12 defect predictor configurations, with the following values of (precision, recall): (75%, 75%), (75%, 80%), (75%, 85%), (75%, 90%), (75%, 95%), (75, 100%), (100%, 75%), (100%, 80%), (100%, 85%), (100%, 90%), (100%, 95%), (100, 100%). Our preliminary experiments suggest that the bug detection performance of SBST-guided-by-DP changes by a small margin when the precision is changed from 100% to 75%, while keeping the recall unchanged at 100% and 75%. On the other hand, the bug detection performance of SBST-guided-by-DP changes by a large margin when only the recall is changed from 100% to 75%, while keeping the precision unchanged at 100% and 75%. Hence, we decide to consider only the values of 75% and 100% for precision, while recall is sampled at 5% steps.

The output of the simulated defect predictor is binary, i.e., method is buggy or not buggy, similar to most of the existing defect predictors. Some of the existing defect predictors output the likelihood of the components being buggy or the ranking of the components according to their likelihood of being buggy. Since we employ a theoretical defect predictor and not a specific one, we resort to the generic defect predictor, which is the one that gives a binary classification.

Algorithm 1 Defect Predictor Simulation

Input: r, p ▷ recall and precision
$M = \{m_1, \ldots, m_k\}$ ▷ ground truth
1: **procedure** SimulateDefectPredictor
2: $d \leftarrow \text{Count}(m_i)$ for $m_i \in M$ s.t. $m_i = 1$
3: $M_b \leftarrow \{i \mid \forall i \in [1, k] \wedge m_i = 1\}$
4: $M_n \leftarrow \{i \mid \forall i \in [1, k] \wedge m_i = 0\}$
5: $tp \leftarrow d * r$
6: $fp \leftarrow tp * (1 - p)/p$
7: $C_b \leftarrow \text{RandomChoice}(M_b, tp) \cup \text{RandomChoice}(M_n, fp)$
8: $C \leftarrow \{c_i = 1 \mid \forall i \in [1, k] \wedge i \in C_b, c_i = 0 \mid \forall i \in [1, k] \wedge i \notin C_b\}$
9: Return(C)

Algorithm 1 illustrates the steps of simulating the defect predictor outputs for a given recall and precision combination. The procedure SimulateDefectPredictor receives the set of methods in

the project with the ground truth labels for their defectiveness, $M = \{m_1, \ldots, m_k\}$, where

$$m_i = \begin{cases} 1 & \text{if method with index } i \text{ is buggy} \\ 0 & \text{otherwise} \end{cases}$$

and outputs a set of labels for each method in the project, $C = \{c_1, \ldots, c_k\}$, to the required level of recall and precision where

$$c_i = \begin{cases} 1 & \text{if method with index } i \text{ is predicted buggy} \\ 0 & \text{otherwise} \end{cases}$$

In the ground truth labels, a method is considered buggy ($m_i = 1$) if there is a known bug in that method. While other methods may have bugs that are not found yet, we consider those methods as non-buggy ($m_i = 0$) in the ground truth label set for defect prediction simulation. The experimental results and the conclusions are also based on these known bugs.

First, the procedure SimulateDefectPredictor calculates the number of buggy methods (d) in the project (line 2 in Algorithm 1). Next, it finds the set of indices of all the buggy (M_b) and non-buggy methods (M_n) in the project (lines 3-4). The desired number of true positives (tp) and false positives (fp) are then calculated for the given recall (r) and precision (p) (lines 5-6). The RandomChoice(M_x, n) procedure returns n number of randomly selected methods from the set M_x, where $x \in \{b, n\}$. C_b is assigned a set of randomly picked tp number of buggy and fp number of non-buggy method indices (line 7). C_b is the set of buggy method indices as classified by the simulated defect predictor. The indices ($\in [1, k]$) that are not in C_b form the non-buggy method indices as classified by the simulated defect predictor. The number of indices classified as non-buggy methods is equal to the sum of required number of false negatives, i.e., $(d - tp)$, and true negatives, i.e., $(|M| - d - fp)$. The output is the set $C = \{c_1, \ldots, c_k\}$, where $c_i = 1$ if the method with index i is labelled as buggy and $c_i = 0$ if the method with index i is labelled as not buggy (line 8).

2.2 Search-Based Software Testing Guided By Defect Prediction

We incorporate buggy method predictions in DynaMOSA [42], the state-of-the-art SBST technique, to guide the search for test cases towards likely buggy methods. DynaMOSA tackles the test generation problem as a many-objective optimisation problem, where each coverage target in the program, e.g., branch and statement, is an objective to optimise. It aims at finding a set of non-dominated test cases that minimise the fitness functions for all the coverage targets. DynaMOSA is more effective at achieving high branch, statement and strong mutation (i.e., variants of the original program that mimic real faults [42]) coverage than the previously proposed SBST techniques using single objective [19, 48] and many objective [41] optimisation [42]. For a test case, covering (i.e., reaching) the buggy code is necessary to detect a bug according to the reachability condition in reachability, infection and propagation (RIP) principle [13, 38–40]. Previous work indicates that mutation coverage significantly correlates with the bug detection of the test suites [30]. Therefore, DynaMOSA is a good candidate for our task given its good performance in terms of code and mutation coverage.

The DynaMOSA approach guided by the defect predictor is referred as *SBST-guided-by-DP* and presented in Algorithm 2. It shares similar search steps and genetic operators as DynaMOSA, except for the updated steps shown in blue colour in Algorithm 2. In this paper, we describe the updated steps in Algorithm 2 in detail.

In addition to the inputs DynaMOSA already receives, SBST-guided-by-DP receives as input a class with methods labelled as buggy or non-buggy, which are labels that can be obtained using

existing defect predictors [21, 23]. In our study, SBST-guided-by-DP receives these labels from defect predictor simulations for given recall and precision (Section 2.1).

SBST-guided-by-DP is designed not to handle the potential errors in the predictions to allow us to assess the impact of imprecise predictions on the bug detection performance of SBST. It devotes all the search resources to find tests that cover likely buggy methods, thereby increasing the chances of detecting bugs. Initially, SBST-guided-by-DP filters out the coverage targets that are deemed to not contain buggy methods as indicated by the defect prediction information, and keeps only targets that contain likely buggy methods (as shown in line 2 of Algorithm 2 and described in Section 2.2.1).

SBST-guided-by-DP generates more than one test case for all the selected buggy targets, hence, further increases the chances of detecting bugs (lines 6, 7, 10 and 11 and described in Section 2.2.2) [46].

To generate more than one test case for all the likely buggy targets, SBST-guided-by-DP does not remove a target once it is covered during the search. Such a behaviour would be likely to cause SBST-guided-by-DP to miss nontrivial targets in the search and keep on generating tests to cover more trivial targets [48]. To address this, we use a method called balanced test coverage proposed by Perera et al. [47] to dynamically disable targets from the search based on their current test coverage and number of independent paths (lines 3 and 13). At the start of the search, the procedure IndependentPaths finds the number of independent paths starting from each edge $e \in E$ in the control dependency graph G of the program (line 3) [47]. In each iteration in the genetic algorithm, the procedure SwitchOffTargets checks the test coverage for each target $u \in U^*$ (i.e., number of tests in the archive A that cover u) and temporarily removes u from U^*, if the test coverage per an independent path from u is higher than the other targets (line 13) [47]. The number of independent paths from a target u is computed using the partial map between edges and targets ϕ and the vector of the number of independent paths for each edge L. The balanced test coverage method paves way for the search to find more tests for targets that have low test coverage in the next iteration. This ensures that the nontrivial targets have an equal chance of being covered compared to the targets that are easier to cover.

Like DynaMOSA, SBST-guided-by-DP randomly generates a set of test cases that forms the initial population P_0 (line 5). Then, it evolves this initial population through creating new test cases via crossover and mutation (line 9), and selecting test cases to the next generation P_{r+1} (line 14), until a termination criterion, such as maximum time budget, is met (line 8). To select test cases to the next generation, the SelectPopulation procedure uses the preference sorting algorithm used in DynaMOSA. For each target $u \in U^*$, the preference sorting algorithm selects the test case from R_r that is closest to cover u according to its fitness function to the next generation.

2.2.1 *Filtering Targets with Defect Prediction.*

A defect predictor classifies the methods of the class under test (CUT) as buggy or non-buggy. The procedure FilterTargets filters out the likely non-buggy targets from the set of all targets U using the classifications C given by the defect predictor (line 2). Spending the limited search resources on covering non-buggy targets is likely to be ineffective when it comes to detecting bugs. Filtering out targets that are unlikely to be buggy allows the search to focus on test cases that cover the likely buggy targets (i.e., $\forall u \in U_B$), hence, generating more effective test cases faster than other approaches which search for tests in all the targets in the CUT.

2.2.2 *Dynamic Selection of Targets and Archiving Tests.*

There are structural dependencies of targets that should be considered when selecting objectives, i.e., targets, to optimise. For instance, some of the targets can be covered only if their control dependent targets are covered. To better understand this, let us consider the following example in Figure 1. Assume the test generation scenario is to optimise branch coverage and b_1, b_2, b_3, b_4, b_5 and b_6 are the branches to be covered. Branch b_1

Algorithm 2 SBST Guided By Defect Prediction

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** SBST
2: $U_B \leftarrow$ FilterTargets(U, C)
3: $L \leftarrow$ IndependentPaths(G) ▷ L is a vector of the number of independent paths for each edge
4: $U^* \leftarrow$ targets in U_B with no control dependencies
5: $P_0 \leftarrow$ RandomPopulation(M) ▷ M is the population size
6: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ, U_B)
7: $A \leftarrow$ UpdateArchive($P_0, \emptyset, U_B$) ▷ A is the archive
8: **for** $r \leftarrow 0$; !terminationCriteria; r++ **do**
9: $Q_r \leftarrow$ GenerateOffspring(P_r)
10: $U^* \leftarrow$ UpdateTargets(U^*, G, ϕ, U_B)
11: $A \leftarrow$ UpdateArchive(Q_r, A, U_B)
12: $R_r \leftarrow P_r \cup Q_r$
13: $U^* \leftarrow$ SwitchOffTargets(U^*, A, L, ϕ)
14: $P_{r+1} \leftarrow$ SelectPopulation(R_r, U^*, M)
15: $T \leftarrow A$ ▷ Update the final test suite T
16: Return(T)

holds a control dependency link to b_3 and b_4, which means that they can be covered only if b_1 is covered by a test case. If an SBST technique optimises test cases to cover b_3 and b_4, while b_1 is uncovered, this will unnecessarily increase the computational complexity of the algorithm because of the added objectives, i.e., b_3 and b_4, to the search without any added benefit. To address this, DynaMOSA dynamically selects targets to the search only when their control dependent targets are covered [42]. In our example, b_3 and b_4 are added to the search only when b_1 is covered.

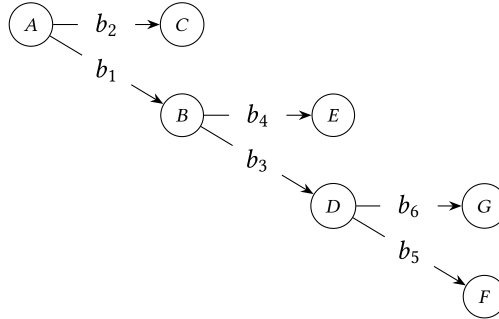


Fig. 1. Control Dependency Graph

At the start of the search, SBST-guided-by-DP selects the set of targets $U^* \subseteq U_B$ that do not have control dependencies (line 4). These are the targets SBST-guided-by-DP can cover without

requiring to cover any other targets in the program. At any given time in the search, it searches for test cases to cover only the targets in U^*.

Once the procedures RandomPopulation (line 5) and GenerateOffspring (line 9) generate new populations of test cases, the procedure UpdateTargets is executed to update U^* by adding new targets to the search. The procedure UpdateTargets adds a target $u \in U_B$ to U^* only if the control dependent targets of u are covered, as explained with the example above. The UpdateTargets procedure obtains the control dependent targets of a target u using the control dependency graph G of the program and the partial map between edges and targets ϕ.

Control dependency graph is calculated at method level. Since we label all the targets in a likely buggy method as likely buggy, all the nodes in the corresponding control dependency graph of a likely buggy method are considered likely buggy. If the actual buggy targets are deeply nested in a method, SBST-guided-by-DP still has guidance to cover them because all the control dependent targets of that method (including control dependent targets of the buggy targets) are considered likely buggy and they are added to U* and kept throughout the search as described above.

SBST-guided-by-DP maintains an archive of test cases found during the search which cover the selected targets. Once the search finishes, this archive forms the final test suite. Unlike in DynaMOSA, we configure the UpdateTargets procedure to not remove a covered target from U^* and the UpdateArchive procedure (lines 7 and 11) to archive all the test cases that cover the selected targets $u \in U_B$. This way, SBST-guided-by-DP can generate more than one test case for each target $u \in U_B$, hence increasing the bug detection capability of the generated test suites [46]. Perera et al. [46] showed that DynaMOSA detects up to 79% more bugs when it was configured to not remove covered targets from the search and retain all the generated tests.

3 DESIGN OF EXPERIMENTS

We design a set of experiments to evaluate the effectiveness of SBST-guided-by-DP in terms of detecting bugs when using defect predictors with 12 different levels of imprecision as described in Section 2.1 (*RQ*). We use the bugs from the Defects4J dataset as the experimental objects [28] (see Section 3.1).

To account for the randomness of the defect prediction simulation algorithm (Algorithm 1), we repeat the simulation runs 5 times for each defect predictor configuration (i.e., recall and precision pair). For each of these simulation runs, we repeat the test generation runs 5 times, to account for the randomness in SBST-guided-by-DP. Altogether, we run test generation 25 times for each defect predictor configuration.

Once tests are generated and evaluated for bug detection, we conduct two-way ANOVA test to statistically analyse the effects of recall and precision of the defect predictor on the bug detection effectiveness of SBST-guided-by-DP.

3.1 Experimental Objects

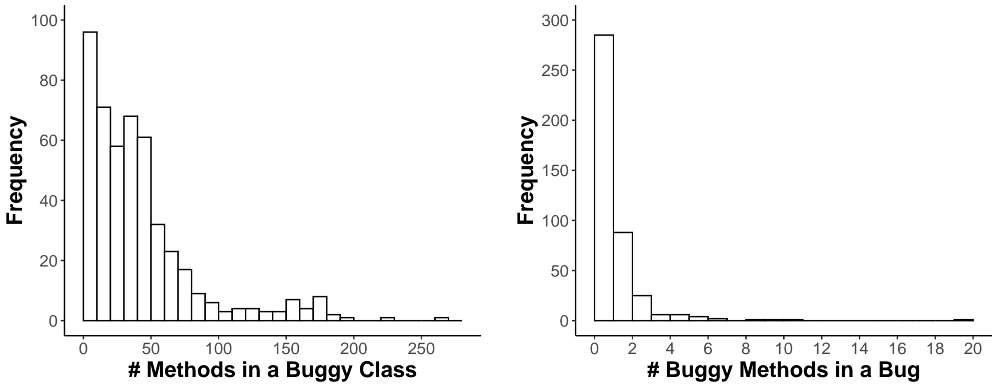
We use the Defects4J dataset (version 1.5.0) [28, 29] as our benchmark. It contains 438 bugs that are from manually validated bug fixes from 6 real-world open source Java projects. In our experiments, we remove 18 bugs altogether from the dataset; 4 deprecated bugs (i.e., not reproducible under Java 8, which is required by EvoSuite), 12 bugs that do not have buggy methods, and 2 bugs for which SBST-guided-by-DP generated uncompilable tests (e.g., method signature is changed in the bug fix). For the 12 bugs that do not have buggy methods, their bug fixes (patches) did not modify or remove existing methods in the code, instead the patches only added new methods (e.g., Lang-23), modified only static blocks (e.g., Time-11) or added/modified class and instance variables (e.g., Math-12 and Closure-111). Thus, we evaluate SBST-guided-by-DP on a total of 420 bugs. The bugs are drawn 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).

We calculate the adequate sample size [15] for two-way ANOVA test with power=0.80, alpha=0.05 and medium effect size (f=0.25). The required sample size with these parameters is 212, which is well below our sample size of 420 bugs.

The Defects4J benchmark gives a buggy version and a fixed version of the program for each bug in the dataset. The fixed version is different to the buggy version by 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 [54]. Figure 2a shows the distribution of the number of methods in the buggy classes in the chosen set of bugs. There are 42.4 methods in a buggy class on average. Figure 2b shows the distribution of the number of buggy methods in the bugs. There are 1.6 buggy methods in a bug on average.

Defects4J is widely used for research on automated unit test generation [20, 46, 53], automated program repair [2], fault localisation [45], test case prioritisation [44], etc. This makes Defects4J a suitable benchmark for evaluating SBST-guided-by-DP, as it allows us to compare our results to existing work.



(a) Distribution of the number of methods in the buggy classes. Total buggy classes = 482.

(b) Distribution of the number of buggy methods in the bugs. Total bugs = 420.

Fig. 2. Distribution of the number of methods and buggy methods in the chosen set of bugs in the Defects4J benchmark.

3.2 Prototype

DynaMOSA is implemented in the state-of-the-art SBST tool, EvoSuite [18]. EvoSuite is an automated test generation framework that generates JUnit test suites for java programs [14, 17]. EvoSuite is actively maintained and evaluated for its effectiveness in terms of bug detecting on both industrial and open source projects [3, 20, 46, 53]. For the experimental evaluation, we implement the changes described in Section 2.2 for SBST-guided-by-DP. The changes are implemented within EvoSuite version 1.0.7, forked from the GitHub repository [14] on June 18th, 2019. We also implement the defect predictor simulator as described in Section 2.1. The prototypes are available to download from here: https://doi.org/10.6084/m9.figshare.16564146

3.3 Parameter Settings

We use the default parameter settings of EvoSuite [19] and DynaMOSA [42] except for the parameters mentioned in the next paragraphs. Parameter tuning of SBST techniques is a long and expensive process [4]. According to Arcuri and Fraser [4], EvoSuite with default parameter values performs on par compared to EvoSuite with tuned parameters.

Time Budget: We set 2 minutes as time budget per CUT for test generation. In practice, the time budget allocated for SBST tools depends on the size of the project, frequency of test generation runs and availability of computational resources in the organisation.

Real world projects are usually very large and can have thousands of classes [7]. If an SBST tool runs test generation for 2 minutes per class, then it will take at least 33 hours to finish the task for the whole project.

To address this issue, practitioners can adapt the SBST tools in their continuous integration (CI) systems [16]. However, the introduction of new SBST tools to the CI system should not make the existing processes in the system idle [46].

Thus, given the limited computational resources available in practice [8] and the expectation of faster feedback cycles from testing in agile development prompt the necessity of frequent test generation runs with limited testing budget. Therefore, we decide that 2 minutes per class is a reasonable time budget in a usual resource constrained environment.

Coverage criteria: We use branch coverage as coverage criterion in line with the prior studies which investigated bug detection effectiveness of EvoSuite [3, 46, 47, 53]. 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 [20, 52].

Termination criteria: We use only the maximum time budget as the termination criterion. Stopping the search after it covers all the targets is detrimental to bug detection [46]. The search needs to utilise the full time budget to generate as many tests for each target in the CUT in order to increase the chances of detecting bugs. Therefore, we terminate the search for test cases only when the allocated time budget runs out.

Test suite minimisation: We disable test suite minimisation since all the test cases in the archive form the final test suite (see Section 2.2.2).

Assertion strategy: We choose all possible assertions as the assertion strategy because the mutation-based assertion filtering can be computationally expensive and can lead to timeouts [46, 53].

3.4 Experimental Protocol

As shown in Figure 3, the experimental setup is divided into 4 steps. Step 1 is ground truth label collection. For each bug in the Defects4J dataset, we check out the buggy versions of the respective Defects4J project (e.g., Lang, Math, Chart, Time, Closure or Mockito). Next, we collect the ground truth labels for the buggy and non-buggy methods. If a method is either modified or removed in the bug fix, we label that method as a buggy method, and non-buggy otherwise [54]. We simulate defect predictions per Defects4J project. Therefore, we combine the ground truth labels from all the bugs from the respective Defects4J project. For example, Figure 3 shows combining the labels from all the 59 bugs from Apache commons-lang project. These ground truth labels are then sent to the defect prediction simulator in step 2.

In step 2, we simulate defect prediction outcomes for each project using the defect prediction algorithm described in Section 2.1. We run experiments with SBST-guided-by-DP using defect predictors with 12 different levels of imprecision (recall and precision pairs) as described in Section 2.1.

We assume an application scenario of generating tests to detect bugs not only limited to regressions, but also the bugs introduced to the code in various times in development. Therefore, we run

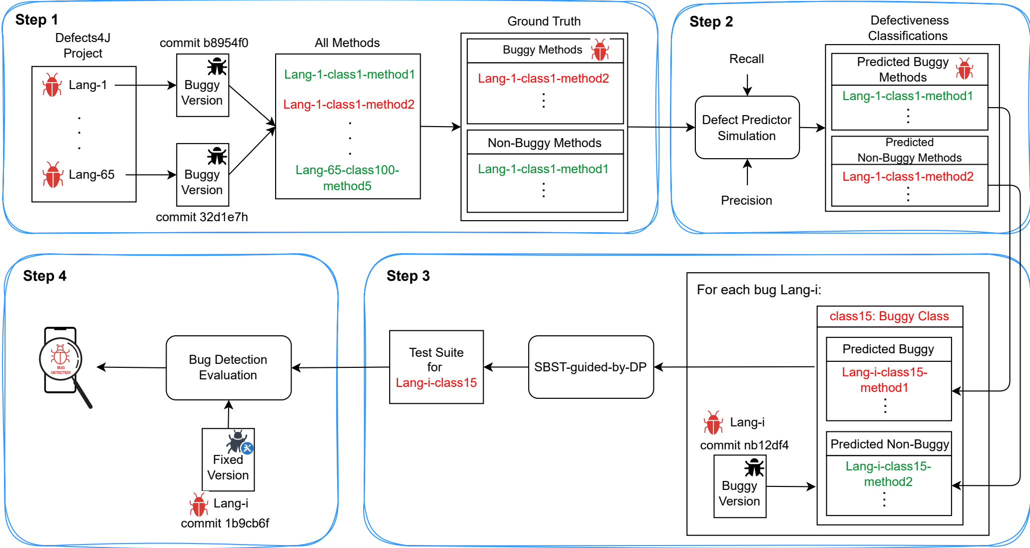


Fig. 3. Experimental Design. Actual buggy methods and classes (corresponding to Defects4J bugs) are shown in red and actual non-buggy methods are shown in green. Figure shows ground truth label collection (step 1), defect prediction simulation (step 2), test generation (step 3), and bug detection evaluation (step 4) for Lang project in Defects4J. Experiments are conducted for 6 Defects4J projects, i.e., Lang, Math, Chart, Time, Closure and Mockito. Recall can take any value from {75%, 80%, 85%, 90%, 95%, 100%} and precision is either 75% or 100%.

test generation on the buggy versions of the projects for each bug in step 3. We measure the bug detecting effectiveness of SBST-guided-by-DP only on the Defects4J bugs. Thus, we only run test generation for buggy classes, i.e., classes that are modified in the bug fixes, in the projects.

For each level of defect predictor imprecision, we run test generation with SBST-guided-by-DP 25 times for each bug in the dataset. Consequently, we have to run a total of 12 (levels of defect prediction imprecision) $*$ 25 (repetitions) $*$ 482 (buggy classes) = 144,600 test generations.

Defects4J [28] allows us to evaluate if the 144,600 generated test suites in the experiments detect the bugs (step 4). First, we remove the flaky test cases in test suites using the 'fix test suite' interface [28] in Defects4J as described in [53]. We use the 'run bug detection' interface [28], which executes a test suite against the buggy and fixed versions of a program and determines if the test suite detects the bug by checking if the test execution results are different between the two versions. We use the fixed versions of the programs as the test oracles [10]. 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 run against the buggy version of the program. The test suite is considered to have failed to detect the bug if it produces the same execution results when run against the buggy and fixed versions of the program, and it is considered to have detected the bug if the test results are different.

4 RESULTS

We present the results for our research question following the method described in Section 3. Our aim is to evaluate the effectiveness of bug detecting performance of SBST-guided-by-DP when using imprecise defect predictors.

RQ. What is the impact of the imprecision of defect prediction on bug detection performance of SBST?

Figure 4 shows the distributions of the number of bugs detected by SBST-guided-by-DP as violin plots and the profile plot of the mean number of bugs detected by SBST-guided-by-DP for each combination of the factors of six recalls and two precisions. The two lines in the profile plot do not cross each other at any point. This means that there is no observable interaction effect between recall and precision.

The two lines descent steeply from recall 100% to 75%. This shows that recall has an effect on number of bugs detected by SBST-guided-by-DP. In particular, bug detection effectiveness decreases as recall decreases.

The precision=75% line closely follows the precision=100% line while staying slightly above the latter, except at recall=85%, where there is a considerable gap between the two. We check if this difference is significant using the two-way ANOVA test results.

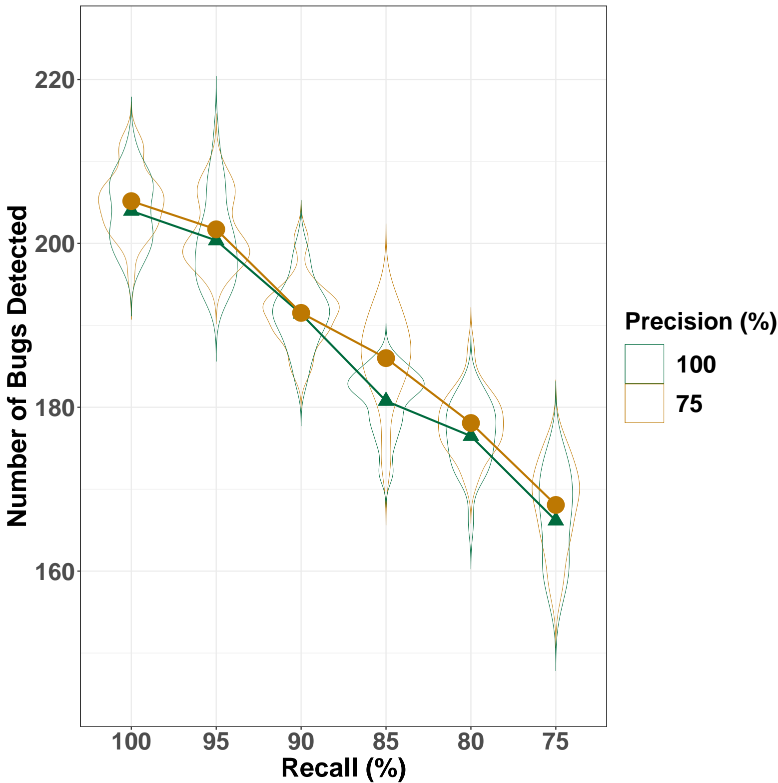


Fig. 4. Distributions of the number of bugs detected by SBST-guided-by-DP in 25 runs as violin plots together with the profile plot of mean number of bugs detected by SBST-guided-by-DP for each combination of the groups of recall and precision. The x-axis represents the recall of the defect predictor configuration. Green violins correspond to defect predictors with 100% precision and gold violins correspond to 75% precision. In the profile plot, bullets (●) and triangles (▲) represent the mean number of bugs detected by SBST-guided-by-DP. Green line and triangles correspond to 100% precision and gold line and bullets correspond to 75% precision.

To statistically test the effect of each of the metrics, recall and precision, and their interaction on the number of bugs detected by SBST-guided-by-DP, we conduct the two-way ANOVA test.

Prior to conducting the two-way ANOVA test, we make sure that our data holds the following assumptions of the test.

(1) The dependent variable should approximately follow a normal distribution for all the combinations of groups of the two independent variables.
(2) Homogeneity of variances exists for all the combinations of groups of the two independent variables.

To check the first assumption, we conduct the Kolmogorov-Smirnov test [36] for normality of the distributions ($\alpha = 0.05$) of the number of bugs detected for each combination of the groups of recall and precision. Based on the results of the tests, we cannot reject our null hypothesis (p-values ≥ 0.131), i.e., H_0 = the number of bugs detected is normally distributed, hence we assume all the samples come from a normal distribution (i.e., H_0 is true).

To check the second assumption, we conduct the Bartlett's test for homogeneity of variances ($\alpha = 0.05$) in each combination of the groups of recall and precision. Based on the results of the test, we cannot reject our null hypothesis (p-value = 0.305), i.e., H_0 = variances of the number of bugs detected are equal across all combinations of the groups, hence we assume the variances are equal across all samples (i.e., H_0 is true).

	Df	Sum Sq	Mean Sq	F value	p-value
Recall	5	51341	10268	497.42	<0.001
Precision	1	273	273	13.21	<0.001
Recall:Precision	5	190	38	1.84	0.105
Residuals	288	5945	21		

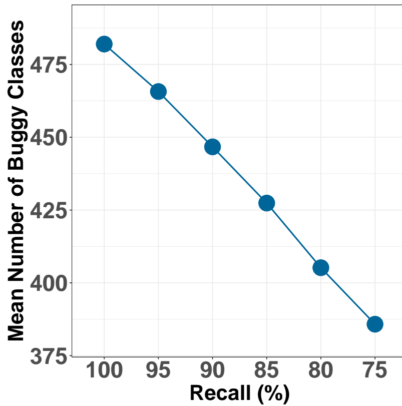
Table 1. Summary of the two-way ANOVA test results. Df = degrees of freedom, Sum Sq = sum of squares and Mean sq = mean sum of squares.

Table 1 shows the summary of the two-way ANOVA test results. According to the two-way ANOVA test, recall and precision in the defect predictor explain a significant amount of variation in number of bugs detected by SBST-guided-by-DP (p-values < 0.001). The test also indicates that we cannot reject the null hypothesis that there is no interaction effect between recall and precision on number of bugs detected (p-value = 0.105). That means we can assume the effect of recall on number of bugs detected does not depend on the effect of precision, and vice versa.

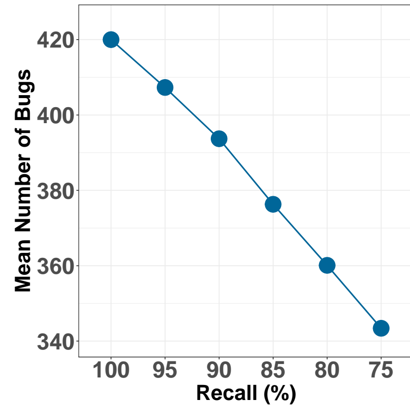
To check if the observed differences among the groups are of practical significance, we measure the epsilon squared effect size ($\widehat{\epsilon^2}$) [58] of the variations in number of bugs detected with respect to recall and precision. We find that the effect of recall on bug detection effectiveness is large with an effect size of 0.89, while the effect of precision is very small ($\widehat{\epsilon^2} = 0.004$) [11], which can be seen from the overlapping distributions in the violin plots in Figure 4 as well.

To further analyse which groups are significantly different from each other, we conduct the Tukey's Honestly-Significant-Difference test [55]. The Tukey post-hoc test shows that the number bugs detected by SBST-guided-by-DP is significantly different between each of the six levels of recall (p-values < 0.002). The Cohen's d effect sizes of the differences between the groups of recall range from medium ($d = 0.77$ for recall 95% and 100%) to large ($d \geq 1.33$ for all other pairs of groups).

Figure 5b shows the mean number of bugs for which tests are generated by SBST-guided-by-DP at each recall. SBST-guided-by-DP detects 7.5 fewer bugs and misses test generation for 15 bugs on average (out of 420) when the recall decreases by 5% in the experiments. SBST-guided-by-DP



(a) Mean number of buggy classes for which tests are generated by SBST-guided-by-DP. Total buggy classes = 482.



(b) Mean number of bugs for which tests are generated by SBST-guided-by-DP. Total bugs = 420.

Fig. 5. Means plots of number of buggy classes and bugs for which tests are generated by SBST-guided-by-DP for the groups of recall.

completely trusts the defect predictor and only generates tests for classes having at least one method predicted as buggy (e.g., true positive). The number of true positives by the defect predictor decreases when the recall decreases. This results in SBST-guided-by-DP generating tests for a fewer number of classes as the recall decreases (as shown in Figure 5a), hence detecting less number of bugs when recall drops from 100% to 75%.

Change of precision from 100% to 75% means that there are false positives in the defect prediction results. In the experiments, when recall is 100% and precision is 75%, there are altogether 224 false positives while the number of true positives is 678, i.e., there are 0.5 non-buggy methods predicted as buggy on average for every bug in the dataset while the number of correctly labelled buggy methods is 1.6 on average per bug. Usually SBST techniques such as DynaMOSA generate tests for classes with higher number of methods than this. For example, there are altogether 46.9 methods on average per bug in the dataset we used. The experimental results indicate that the amount of false positives at 75% precision is not a significant burden to the search for tests of the SBST technique to cause a meaningful practical difference in the number of bugs detected.

Summary: False negatives of the defect predictor have a significant impact on the bug detection performance of SBST. When the recall of the defect predictor decreases, the bug detection effectiveness significantly decreases with a large effect size. On the other hand, we conclude that there is no meaningful practical effect of precision on the bug detection performance of SBST, as indicated by a very small effect size.

RQ1: What is the impact of the recall of the defect predictor when the bugs are spread across multiple methods compared to bugs located in a single method?

Further analysis of the results indicates that SBST-guided-by-DP only misses test generation for 4.5% of the bugs that are spread across multiple methods on average, while it misses 24.7% of the bugs that are located in a single method on average, when recall decreases from 100% to 75%. This

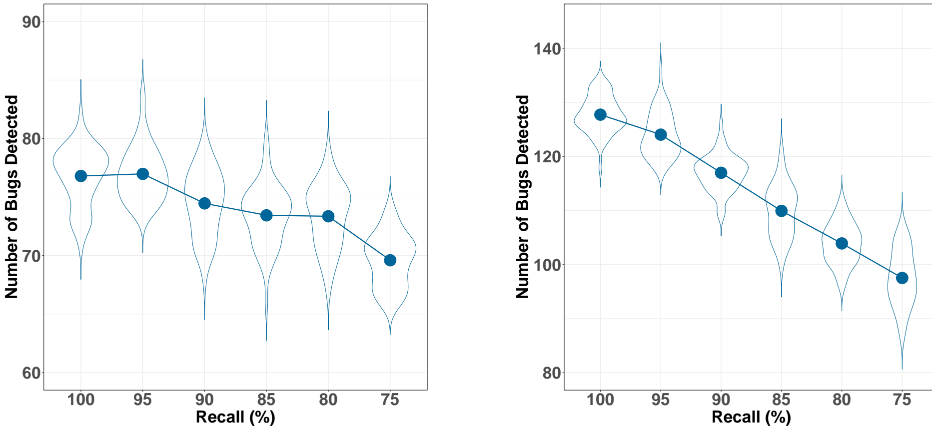
suggests that the bugs that are found within only one method are more prone to the impact of recall compared to bugs that are spread across multiple methods.

To characterise the effects of recall on detecting bugs which are found within a single method and spread across multiple methods, we conduct Welch ANOVA test [34] separately for the two subsets of our dataset, i.e., bugs having only one buggy method and bugs having more than one buggy method. The reason for carrying out Welch ANOVA test is because our data fails the assumption of homogeneity of variances for each combination of the groups of recall for bugs having only one buggy method.

	Num Df	Denom Df	F value	p-value
# buggy methods > 1	5.00	137.06	67.24	<0.001
# buggy methods = 1	5.00	136.68	395.91	<0.001

Table 2. Summary of the Welch ANOVA test results. Num Df = degrees of freedom of the numerator and Denom Df = degrees of freedom of the denominator.

The results of the Welch ANOVA test are shown in Table 2. There are 135 bugs which have more than one buggy method. The results for these bugs show that overall recall has a significant effect on the number of bugs detected by SBST-guided-by-DP (p-value <0.001) with a large effect size ($\widehat{\epsilon}^2 = 0.53$) [9]. However, the Games-Howell post-hoc test reveals that the bug detection effectiveness is not significantly different between recall 80%-85%, 80%-90%, 85%-90%, and 95%-100%. This can be seen in the violin plots in Figure 6a as well.



(a) Bugs that have more than one buggy method. Total number of bugs = 135.

(b) Bugs that have one buggy method. Total number of bugs = 285.

Fig. 6. Distributions of the number of bugs detected by SBST-guided-by-DP as violin plots together with the means plot of number of bugs detected by SBST-guided-by-DP for the groups of recall.

There are 285 bugs which have only one buggy method. The results of Welch ANOVA test for these bugs show that recall has a significant effect on number of bugs detected by SBST-guided-by-DP (p-value <0.001) with a large effect size ($\widehat{\epsilon}^2 = 0.87$). The Games-Howell post-hoc test confirms that the number of bugs detected by SBST-guided-by-DP is significantly different between each group of recall (p-values <0.001) with large effect sizes ($d \geq 0.98$) as can be seen in Figure 6b.

Summary: Recall has a significant effect on bug detection effectiveness of SBST-guided-by-DP regardless of whether the bugs are found within one method or spread across multiple methods. However, for the bugs that are spread across multiple methods, the effect size of recall effect is smaller when compared to bugs that are found within one method ($0.53 < 0.87$). In contrast to bugs that are found within one method, the effect of recall is not significant between the groups of recall 80%, 85% and 90%, and 95% and 100% for the bugs that are spread across multiple methods.

RQ2: What is the impact of the precision of the defect predictor when the time budget and the test suite size are restricted?

4.1 Impact of Precision at Small Time Budgets

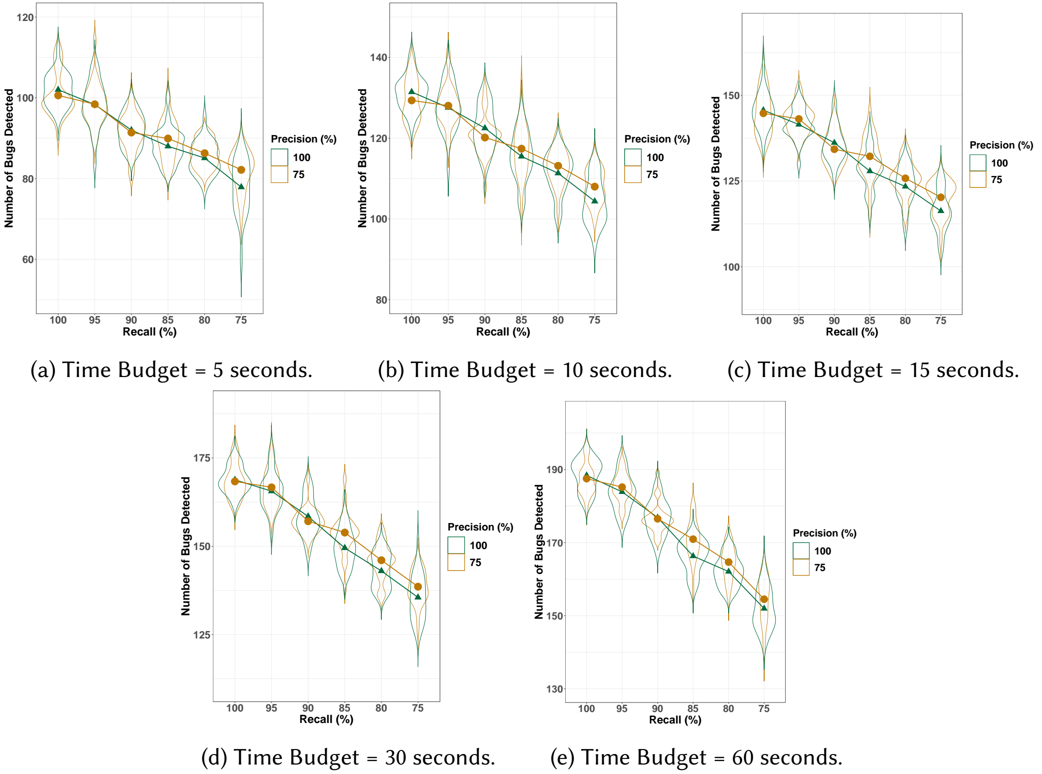
The impact of precision on SBST may be different when SBST-guided-by-DP is given a smaller time budget. In particular, SBST-guided-by-DP with a sufficient time budget like 120 seconds may have enough time to search for tests in actual buggy methods (i.e., true positives) despite searching for tests in false positives. Whereas the search for tests in actual buggy methods may be greatly impacted by false positives when SBST-guided-by-DP is given smaller time budgets like 15 or 30 seconds. Hence, we further investigate the impact of the time budget on the conclusion about sensitivity to the defect prediction precision. To do that, we conduct two-way ANOVA test at time budgets 5, 10, 15, 30 and 60 seconds.

We find that there is no meaningful practical effect on the bug detection performance of SBST when precision is changed from 100% to 75% at all the time budgets we considered. This is evident from the overlapping distributions in the violin plots at each time budget in Figure 7 as well. In particular, according to the two-way ANOVA tests, we cannot reject the null hypothesis that there is no effect of precision on the number of bugs detected by SBST-guided-by-DP at 5 and 10 seconds time budgets (p-values ≥ 0.104). At 15, 30 and 60 seconds time budgets, the tests indicate that precision has a significant effect on number of bugs detected by SBST-guided-by-DP (p-values ≤ 0.010), however the effects are not of practical significance as indicated by very small effect sizes ($\widehat{\epsilon}^2 \leq 0.005$). This shows that the additional overhead caused by the false positives at 75% precision is not a significant burden to the search process of the SBST technique even at smaller time budgets.

4.2 Impact of Precision When Test Suite Size is Limited

The impact of precision on SBST may be different when the final test suite size, i.e., number of test cases, is restricted. SBST-guided-by-DP generates multiple test cases for each target in the search (Section 2.2.2) and retains all these tests in the final test suite. When the number of test cases in the final test suite is not controlled, the false positives in the predictions are not likely to diminish the bug detection performance of the test suite. Instead, they are likely to create redundant test cases in the test suite in terms of detecting bugs and increase the test suite size. When the number of test cases is controlled, bug detection is likely to be impacted by the presence of redundant test cases created because of the false positives.

We further investigate the impact of test suite size on the conclusions about the sensitivity to the defect prediction precision. We apply the additional branch coverage prioritisation technique [49] to prioritise the test cases in the final test suite produced by SBST-guided-by-DP at 120 seconds time budget and conduct two-way ANOVA test after controlling the test suite size for 10, 20, 40, 80 and 160 test cases. The top n test cases of a prioritised test suite are used to create a test suite with size n. The controlled test suite sizes are chosen to be in line with the sizes of the test suites generated by the original implementation of DynaMOSA, which produces minimised test suites.



(a) Time Budget = 5 seconds. (b) Time Budget = 10 seconds. (c) Time Budget = 15 seconds.

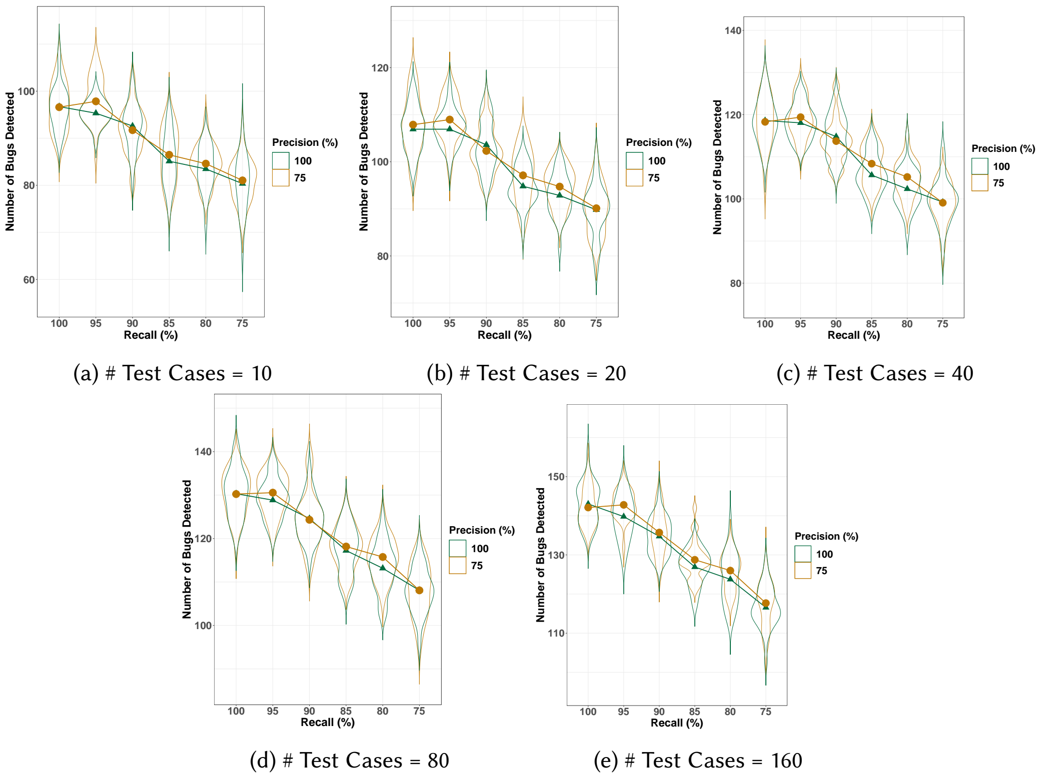
(d) Time Budget = 30 seconds. (e) Time Budget = 60 seconds.

Fig. 7. Distributions of the number of bugs detected by SBST-guided-by-DP as violin plots together with the profile plot of mean number of bugs detected by SBST-guided-by-DP for each combination of the groups of recall and precision at different time budgets.

We find that precision has no meaningful practical impact on the bug detection performance of SBST for all the controlled test suite sizes considered. This can be seen in the violin plots in Figure 8 as well, where the plots at 75% precision are overlapped with the respective plots at 100% precision. In particular, according to the two-way ANOVA tests, we cannot reject the null hypothesis that there is no effect of precision on the number of bugs detected by SBST-guided-by-DP when the test suite sizes are controlled for 10, 20, 40 and 80 test cases (p-values ≥ 0.067). When the test suite size is 160 test cases, the two-way ANOVA test indicates that precision has a significant effect on the number of bugs detected (p-value = 0.020), however it is not of practical significance as indicated by a very small effect size ($\widehat{\epsilon^2} = 0.004$).

To understand why there is no significant effect of precision when the test suite size is restricted, we further analyse the test suites generated by SBST-guided-by-DP for the runs where the predictions contained at least one false positive. We categorise each test case in a size-controlled test suite (number of test cases = 10, 20, 40, 80 and 160) into the following categories; i) covers only false positive branches, ii) covers both false and true positive branches, and iii) covers only true positive branches. Test cases that cover only the false positive branches are redundant test cases and they do not detect the bugs. SBST-guided-by-DP generates test cases with collateral coverage and some of the test cases cover both false and true positive branches. Even though the test cases that cover both false and true positive branches have redundant coverage, they are capable of detecting bugs.

We find that only 18% of the test suites on average cover solely the false positives, whereas 61% and 21% of the test suites on average cover both true and false positives and solely the true positives, respectively. This shows that when the test generation runs are affected by the predictions containing false positives at 75% precision, the test suites generated by SBST-guided-by-DP have higher amount of test cases covering the true positives (82% on average) compared to the test cases covering only the false positives, i.e., redundant test cases (18% on average). This is because SBST-guided-by-DP generates test cases with collateral coverage, which ensures true positive branches are also covered in the presence of a reasonable amount of false positive branches in the search for test cases. As a result, the bug detection effectiveness of the reduced test suites is not significantly impacted when the precision is changed from 100% to 75%.



(a) # Test Cases = 10

(b) # Test Cases = 20

(c) # Test Cases = 40

(d) # Test Cases = 80

(e) # Test Cases = 160

Fig. 8. Distributions of the number of bugs detected by SBST-guided-by-DP as violin plots together with the profile plot of mean number of bugs detected by SBST-guided-by-DP for each combination of the groups of recall and precision for different controlled test suite sizes.

Summary: Precision does not have a meaningful practical impact on the bug detection performance of SBST when the time budget and the test suite size are constrained to smaller amounts.

5 DISCUSSION

We observe that completely trusting the defect predictor can be detrimental to the SBST technique when the defect predictor misses labelling buggy code. To mitigate this, SBST techniques have to

take potential false negatives in the predictions into account. SBST-guided-by-DP fully exploits the buggy methods predicted by the defect predictor. We recommend that SBST techniques require exploration of the likely non-buggy methods while prioritising the exploitation of likely buggy methods. One way to do this is to always generate tests for methods that are predicted buggy, while also generating tests for predicted non-buggy methods at least with a minimum probability. This way the SBST technique gets a chance to search for tests in incorrectly classified buggy methods (when recall <100%), while also giving higher priority to methods that are predicted buggy by the defect predictor.

Defect predictors have mainly been used to provide a list of likely defective parts of a program (e.g., classes and methods) to programmers, who then manually inspect or test the likely defective parts to find the bugs [12, 32]. In this context, the precision of the defect predictor is very important [56]. Poor precision of the defect predictor means there are more false positives. A higher number of false positives can waste developers' time and lead to losing their trust on the prediction results [32]. However, when the defect predictions are consumed by another automated testing technique such as SBST, this may not be the case. In the context of SBST, our study reveals contrasting findings. We find that the recall of the defect predictor is more important than precision when predictions are used by SBST.

The primary actionable conclusion from this paper for the research community is to define a recall-at-precision measure, for example 75%, for defect predictors in the context of combining defect prediction and SBST. Currently the defect prediction community aims to increase both precision and recall at the same time [59]. We recommend that researchers target higher recall while having a sufficiently high precision, e.g., 75%. This approach is widely used in training machine learning classifiers, where there is a particular precision level required to avoid false positives, such that any classifier that fails to meet this criterion is considered unacceptable. When the criterion for minimum precision level is met, increasing recall at the minimum precision or above becomes the goal. In our experiments, we operate with the assumption that practical defect predictors should have at least 75% precision and recall, as suggested by Zimmerman et al. [60], in order to investigate the impact of defect prediction performance on bug detection of SBST. Finding a lower bound for precision to define the max-recall-at-precision measure is a topic for future work.

SBST-guided-by-DP generates multiple test cases for each coverage target, e.g., branch, in the CUT. While SBST-guided-by-DP is designed for the purpose of this theoretical investigation, if a similar approach to be adapted in practice, an appropriate test suite prioritisation or minimisation technique can be used as a post test generation step to address the generation of large test suites. Otherwise it can be a significant overhead for developers to manually review a large number of test cases. In Section 4.2, we limit the test suite size by applying additional branch coverage prioritisation technique and then selecting the top n test cases of the prioritised test suite. A potential future work could be to use defect prediction to guide test suite prioritisation to increase the bug detection of the prioritised test suites.

We choose EvoSuite in our study as it is considered state of the art, and extend the best performing technique in EvoSuite, i.e., DynaMOSA. There are other SBST techniques such as whole test suite generation (WS) [18] and archive-based WS (WSA) [48], which can also be used in our experiments. For instance, both WS and WSA use single objective optimisation, and their fitness functions are composed of branch distances for all the coverage targets (assuming a branch coverage scenario). They can be integrated into our experimental design by filtering branch distances corresponding to buggy targets from the fitness function according to the defect predictions. However, we choose DynaMOSA in our study as it was shown to perform better than other SBST techniques, including WS and WSA, at achieving high code and mutation coverage, which are indicators of good bug

detection performance (see Section 2.2). Moreover, DynaMOSA (or an extension of it) was also used in previous work that combined defect prediction and SBST [46, 47].

We look into the difference observed in Figure 4 for the bug detection performance of SBST-guided-by-DP with 75% and 100% precision at 85% recall. The analysis attributes this variation to inherent randomness in SBST techniques which we discuss under internal threats to validity in Section 6. Moreover, we find this variation observed at 85% recall does not impact the key findings of the paper. Initially, we examine the false positives at 75% precision and 85% recall to assess their impact on bug detection performance. The findings suggest that false positives did not influence the disparity, as bugs with false positives at 75% precision were detected similarly at both precision levels. Subsequently, we scrutinise the defect prediction simulation runs at 75% and 100% precision with 85% recall to check for disproportionate allocation of positive labels (i.e., likely buggy methods). The analysis indicates fair treatment in the allocation of positive labels by the defect prediction simulator in the two precision settings at 85% recall. For this analysis, we use 'detectability' of the bugs allocated in each simulation run as a measure to assess fair treatment by the simulator. Detectability of a bug is the success rate of detecting that bug by SBST-guided-by-DP, i.e., proportion of test generation runs the bug was detected, when using a defect predictor with 100% recall and precision.

6 THREATS TO VALIDITY

Construct Validity. To systematically investigate the impact of defect prediction imprecision, we simulate the predictions by assuming a uniform distribution of defect prediction errors which is similar to previous work [24, 47]. This means in our simulations, every method has an equal chance of being labelled incorrectly independent of each other. However, real defect predictors may have different distributions of their predictions depending on the underlying characteristics and nature of the prediction problem, which may impact the realism of a simulated defect predictor. Nevertheless, in the absence of prior knowledge about empirical or theoretical defect prediction distributions, it is reasonable to assume a uniform distribution of predictions in the defect prediction simulation.

We consider buggy methods only from the labelled bugs in the Defects4J dataset. There can be other bugs in the dataset that are not yet discovered, which correspond to other buggy methods. This may impact the measured recall, precision and bug detection performance. The only way to find out the other bugs that have not been detected yet is through manual validation of the generated test suites. However, this is not a feasible task given there are 144,600 test suites. Nonetheless, since our evaluation considers the bug detection performance only against the labelled bugs in the dataset and the defect prediction simulation uses only the labelled bugs, our findings, i.e., impact of defect prediction imprecision on SBST, are not affected by this.

Internal Validity. To account for the randomness in the defect prediction simulation, we repeat the simulations 5 times for each combination of the groups of recall and precision. For each simulation, we repeat the test generation 5 times to account for the non-deterministic behaviour of SBST-guided-by-DP. In total, we conduct 25 test generation runs for each bug and for each level of defect prediction imprecision.

Conclusion Validity. To account for any threats to the conclusion validity, we derive conclusions from the experimental results after conducting sound statistical tests; two-way ANOVA test, epsilon squared effect size, Tukey's Honestly-Significant-Difference test, Cohen's d effect size, Welch ANOVA test and Games-Howell post-hoc test.

External Validity. We use 420 real bugs from Defects4J dataset as the experimental objects. They are drawn from 6 open source projects. At the time of writing this paper, another 401 bugs from 11 projects were added to the Defects4J dataset. However, we understand that these projects do

not represent all program characteristics, especially in industrial projects. Nevertheless, Defects4J dataset has been widely used in previous work as a benchmark [2, 44, 46, 53]. Future work needs to be done on investigating the impact of imprecision of defect prediction on SBST with respect to other bug datasets.

SBST-guided-by-DP uses defect prediction information at method level. Our findings may not be generalised to previous work [26, 46] which use defect prediction at a different level of granularity (class level). Nevertheless, the findings from our study will help to further explore the opportunities of combining defect predictions and SBST.

SBST-guided-by-DP completely trusts the defect predictor and does not handle the potential prediction errors. The findings of our study may vary if the SBST technique handles these errors, like in PreMOSA [47]. In particular, PreMOSA was shown to not be significantly affected when both recall and precision changed from 100% to 75%. By not handling potential defect prediction errors in SBST-guided-by-DP, we are able to observe the direct impact of defect prediction imprecision on guiding SBST. Otherwise the effects of the prediction errors can be masked by the error handling techniques.

We investigate the impact of defect prediction imprecision only in the range of 75% to 100% for recall and precision. Therefore, our findings may not be generalised to the defect predictors which have recall or precision less than 75%. While this choice of performance sampling in our simulation is a threat to external validity, it is also a threat to construct validity for lack of characterising all possible defect predictors. However, we opted to use this range with the justification that this is the range for an acceptable performance for a defect predictor as recommended by Zimmermann et al. [60].

7 RELATED WORK

7.1 Defect Prediction in Software Testing

Defect prediction was originally proposed to provide a list of likely defective parts of a program to assist developers in code reviews [32, 33], manual testing [12], etc. More recently, defect predictors have been used to inform automated testing techniques as well. G-clef [44] is a test prioritisation strategy that uses the likelihood of the defectiveness of classes to prioritise test cases and it was shown to be effective at reducing the number of test cases required to find bugs. FLUCCS [54] is a fault localisation approach that leverages the likelihood of methods being defective and it was shown to significantly outperform the state-of-the-art spectrum based fault localisation (SBFL) techniques.

Perera et al. [46] and Hershkovich et al. [26] used defect predictions at class level to determine the time budget allocated to classes in a project to run test generation with SBST techniques. A highly likely to be defective class according to the defect predictor has more chance of being selected to run test generation [26] or allocated a higher time budget [46]. Despite showing the improved bug detection performance of the proposed SBST techniques, we find that the defect predictors used in these two works have relatively high performance, e.g., 85% recall in [46] and 0.95 AUC in [26], which can be difficult to achieve for a defect predictor sometimes.

For example, Zimmermann et al. [60] found that only 21 out of 622 cross-project defect predictor combinations to have recall, precision and accuracy greater than 75%. In their systematic literature review, Hall et al. [22] reported defect predictor performances in the ranges of 5%-95% and 25%-85% for precision and recall, respectively. This leads to the question of how does the variation in defect prediction performance affect the bug detection effectiveness of SBST techniques that incorporate defect prediction information. To address this gap, we study the impact of imprecision in defect predictions on the bug detection performance of SBST.

Perera et al. [47] developed an SBST technique called PreMOSA that uses defect prediction information along with code coverage to guide the search for test cases to likely buggy targets. PreMOSA accounts for potential errors in the defect predictions, in particular, the false negatives, and it was shown not to be significantly affected by the prediction errors. Therefore, it does not allow us to use PreMOSA in our study as the SBST technique to investigate the impact of defect prediction imprecision on SBST. To do that, we use an SBST technique, i.e., DynaMOSA, that does not handle the potential prediction errors. DynaMOSA is incorporated in SBST-guided-by-DP and completely trusts the defect predictor, and the variation of its performance for different defect prediction imprecision levels reflects the impact of prediction errors on guiding SBST. To the best of our knowledge, this is the first study that assesses the impact of defect prediction imprecision on guiding SBST.

7.2 Imprecision in Defect Predictors

There is a plethora of defect predictors which have been proposed over the past 40 years [56]. Measures such as recall, precision, f-measure, AUC, Matthews correlation coefficient (MCC) [57], etc. have been used to measure the predictive power of the defect predictors [22]. Out of these measures, recall and precision have been widely used in previous work [22, 27] and are often preferred by practitioners [56]. Existing defect predictors have wavering performance. For example, Hall et al. [22] reported defect predictor performances from as low as 5% and 25% to as high as 95% and 85% for precision and recall, respectively. Hosseini et al. [27] also reported similar findings in their systematic literature review of cross-project defect predictors. It is thus important to study the impact of the wavering defect prediction performance on the bug detection performance of SBST. In our study, we consider the recall and the precision should be greater than 75% to be considered acceptable as recommended by Zimmermann et al. [60], and simulate defect predictions in the range from 75% to 100% for recall and precision.

Previous work reports on the developers' opinions about the defect predictor performance [12, 32, 56], showing that false positives cause developers to waste their precious time on inspecting non-buggy code, which eventually leads to loosing trust on the defect predictor [12, 32]. In the eyes of the developers, higher precision is more important compared to higher recall in a defect predictor, because higher precision means low false positives [56]. There is a trade-off between recall and precision of defect predictors [31, 37]. In some instances, higher recall is more important than higher precision [37]; for example, when the cost of missing a bug is prohibitively expensive or the cost of inspecting false positives is negligible [37]. Our study reveals similar findings to this where in the context of using defect prediction to guide SBST, the impact of recall on the bug detection performance of SBST is more important than the impact of precision.

7.3 Search-Based Software Testing

Search-based software testing techniques use search algorithms like genetic algorithms to search for test cases to meet a given criteria like branch coverage [18]. The test generation problem can be formulated in two ways; i) single objective formulation [18, 48] and ii) many objective formulation [41, 42]. In many objective optimisation, such as MOSA [41] and DynaMOSA [42], SBST techniques aim to find a set of non-dominated test cases that minimise the fitness functions for all the test targets, e.g., branches. In single objective optimisation, SBST techniques optimise whole test suites to minimise a single fitness function which is created by aggregating all the individual test target distances. A target distance measures how far away the test suite is from covering that target [18]. Whole test suite generation (WS) [18] and archive-based WS (WSA) [48] are two examples for techniques that use single objective optimisation. Previous work showed that DynaMOSA, a state-of-the-art many objective optimisation technique, is better than single

objective optimisation techniques in terms of achieving high code coverage [42]. In this paper, we study the effect of defect prediction imprecision on bug detection performance of an SBST technique that uses many objective optimisation.

8 CONCLUSION

We study the impact of imprecision in defect prediction on the bug detection performance of SBST. We use simulated defect predictors to systematically sample defect predictors in the range of 75% to 100% for recall and precision. We use the state-of-the-art SBST technique, DynaMOSA, and incorporate predictions about buggy methods as given by the simulated defect predictor to guide the search for test cases towards likely buggy methods. Through a comprehensive experimental evaluation on 420 bugs from the Defects4J dataset, we find that the recall of the defect predictor has a significant impact on the bug detection effectiveness of SBST with a large effect size. On the other hand, the impact of precision is not of meaningful practical significance as indicated by a very small effect size. The impact of precision on SBST remains the same even when SBST is given small time budgets and the test suite size is limited. Further analysis of the results shows that the impact of the recall for the bugs that are spread across multiple methods is smaller compared to the bugs that are found within only one method.

Based on the results of our study, we make the following recommendations:

(1) SBST techniques must take the potential false negatives in the predictions into account. One way to do this is to prioritise the likely buggy parts of the program, while guiding the search towards the likely non-buggy parts with at least a minimum probability.
(2) In the context of combining defect prediction and SBST, the primary objective should be to increase recall of the defect predictor and the secondary objective can be increasing precision. Our study demonstrates that a reasonable amount of false positives is not a significant burden for SBST when searching for tests. False negatives, however, can deteriorate the bug detection of SBST significantly. Researchers should target higher recall while maintaining a sufficiently high precision, instead of trying to increase both of them at the same time.

We identify the following directions as future work to extend this study; i) find a lower bound for precision to define a max-recall-at-precision measure for defect predictors in the context of combining defect prediction and SBST, ii) validate the findings against other bug datasets [25, 35, 51], and iii) explore the options for using the likelihood of defectiveness of methods to guide SBST techniques.

ACKNOWLEDGMENTS

This research was supported by the Australian Research Council under grants DP210100041 and by the Faculty Postgraduate Publications Award from the Faculty of Information Technology of Monash University.

REFERENCES

[1] Edward Aftandilian, Raluca Sauciuc, Siddharth Priya, and Sundaresan Krishnan. 2012. Building useful program analysis tools using an extensible Java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 14–23.
[2] Aldeida Aleti and Matias Martinez. 2020. E-APR: Mapping the Effectiveness of Automated Program Repair. *arXiv preprint arXiv:2002.03968* (2020).
[3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Jānis Benefelds. 2017. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press, 263–272.
[4] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[5] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008), 22–29.

[6] David Bowes, Tracy Hall, and Jean Petrić. 2018. Software defect prediction: do different classifiers find the same defects? *Software Quality Journal* 26 (2018), 525–552.

[7] Manfred Broy, Ingolf H Kruger, Alexander Pretschner, and Christian Salzmann. 2007. Engineering automotive software. *Proc. IEEE* 95, 2 (2007), 356–373.

[8] José Campos, Andrea Arcuri, Gordon Fraser, and Rui Abreu. 2014. Continuous test generation: enhancing continuous integration with automated test generation. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 55–66.

[9] Robert M Carroll and Lena A Nordholm. 1975. Sampling Characteristics of Kelley's ε and Hays' ω. *Educational and Psychological Measurement* 35, 3 (1975), 541–554.

[10] Thierry Titcheu Chekam, Mike Papadakis, Tegawendé F Bissyandé, Yves Le Traon, and Koushik Sen. 2020. Selecting fault revealing mutants. *Empirical Software Engineering* 25, 1 (2020), 434–487.

[11] Jacob Cohen. 1992. A power primer. *Psychological bulletin* 112, 1 (1992), 155.

[12] Hoa Khanh Dam, Trang Pham, Shien Wee Ng, Truyen Tran, John Grundy, Aditya Ghose, Taeksu Kim, and Chul-Joo Kim. 2019. Lessons learned from using a deep tree-based model for software defect prediction in practice. In *Proceedings of the 16th International Conference on Mining Software Repositories*. IEEE Press, 46–57.

[13] Richard A DeMillo, A Jefferson Offutt, et al. 1991. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering* 17, 9 (1991), 900–910.

[14] EvoSuite. 2019. EvoSuite - automated generation of JUnit test suites for Java classes. https://github.com/EvoSuite/evosuite Last accessed on: 29/11/2019.

[15] Franz Faul, Edgar Erdfelder, Albert-Georg Lang, and Axel Buchner. 2007. G* Power 3: A flexible statistical power analysis program for the social, behavioral, and biomedical sciences. *Behavior research methods* 39, 2 (2007), 175–191.

[16] Martin Fowler and Matthew Foemmel. 2006. Continuous integration.

[17] Gordon Fraser. 2018. EvoSuite - Automatic Test Suite Generation for Java. http://www.evosuite.org/ Last accessed on: 19/09/2019.

[18] Gordon Fraser and Andrea Arcuri. 2011. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, 31–40.

[19] Gordon Fraser and Andrea Arcuri. 2012. Whole test suite generation. *IEEE Transactions on Software Engineering* 39, 2 (2012), 276–291.

[20] Gregory Gay. 2017. The fitness function for the job: Search-based generation of test suites that detect real faults. In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 345–355.

[21] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C Gall. 2012. Method-level bug prediction. In *Proceedings of the 2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 171–180.

[22] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2011. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2011), 1276–1304.

[23] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. 2012. Bug prediction based on fine-grained module histories. In *2012 34th international conference on software engineering (ICSE)*. IEEE, 200–210.

[24] Steffen Herbold. 2021. On the Costs and Profit of Software Defect Prediction. *IEEE Transactions on Software Engineering* 47, 11 (2021), 2617–2631. https://doi.org/10.1109/TSE.2019.2957794

[25] Steffen Herbold, Alexander Trautsch, Benjamin Ledel, Alireza Aghamohammadi, Taher Ahmed Ghaleb, Kuljit Kaur Chahal, Tim Bossenmaier, Bhaveet Nagaria, Philip Makedonski, Matin Nili Ahmadabadi, Kristóf Szabados, Helge Spieker, Matej Madeja, Nathaniel Hoy, Valentina Lenarduzzi, Shangwen Wang, Gema Rodríguez-Pérez, Ricardo Colomo Palacios, Roberto Verdecchia, Paramvir Singh, Yihao Qin, Debasish Chakroborti, Willard Davis, Vijay Walunj, Hongjun Wu, Diego Marcilio, Omar Alam, Abdullah Aldaeej, Idan Amit, Burak Turhan, Simon Eismann, Anna-Katharina Wickert, Ivano Malavolta, Matús Sulír, Fatemeh Fard, Austin Z. Henley, Stratos Kourtzanidis, Eray Tuzun, Christoph Treude, Simin Maleki Shamasbi, Ivan Pashchenko, Marvin Wyrich, James Davis, Alexander Serebrenik, Ella Albrecht, Ethem Utku Aktas, Daniel Strüber, and Johannes Erbel. 2020. Large-Scale Manual Validation of Bug Fixing Commits: A Fine-grained Analysis of Tangling. *CoRR* abs/2011.06244 (2020). arXiv:2011.06244 https://arxiv.org/abs/2011.06244

[26] Eran Hershkovich, Roni Stern, Rui Abreu, and Amir Elmishali. 2019. Prediction-Guided Software Test Generation. In *Proceedings of the 30th International Workshop on Principles of Diagnosis DX'19*.

[27] Seyedrebvar Hosseini, Burak Turhan, and Dimuthu Gunarathna. 2017. A systematic literature review and meta-analysis on cross project defect prediction. *IEEE Transactions on Software Engineering* 45, 2 (2017), 111–147.

[28] Rene Just. 2019. Defects4J - A Database of Real Faults and an Experimental Infrastructure to Enable Controlled Experiments in Software Engineering Research. https://github.com/rjust/defects4j Last accessed on: 02/10/2019.

[29] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*.

ACM, 437–440.

[30] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. 2014. Are mutants a valid substitute for real faults in software testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 654–665.

[31] A Gunes Koru and Hongfang Liu. 2005. Building effective defect-prediction models in practice. *IEEE software* 22, 6 (2005), 23–29.

[32] Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E James Whitehead Jr. 2013. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 372–381.

[33] Chris Lewis and Rong Ou. 2011. Bug Prediction at Google. http://google-engtools.blogspot.com/2011/12/ Last accessed on: 16/09/2019.

[34] Hangcheng Liu. 2015. *Comparing Welch ANOVA, a Kruskal-Wallis test, and traditional ANOVA in case of heterogeneity of variance*. Virginia Commonwealth University.

[35] Fernanda Madeiral, Simon Urli, Marcelo Maia, and Martin Monperrus. 2019. Bears: An Extensible Java Bug Benchmark for Automatic Program Repair Studies. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER '19)*. https://arxiv.org/abs/1901.06024

[36] Frank J Massey Jr. 1951. The Kolmogorov-Smirnov test for goodness of fit. *Journal of the American statistical Association* 46, 253 (1951), 68–78.

[37] Tim Menzies, Alex Dekhtyar, Justin Distefano, and Jeremy Greenwald. 2007. Problems with Precision: A Response to "Comments on 'Data Mining Static Code Attributes to Learn Defect Predictors'". *IEEE Transactions on Software Engineering* 33, 9 (2007), 637–640. https://doi.org/10.1109/TSE.2007.70721

[38] Larry Joe Morell. 1984. *A Theory of Error-Based Testing.* Technical Report. MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE.

[39] Larry J. Morell. 1990. A theory of fault-based testing. *IEEE Transactions on Software Engineering* 16, 8 (1990), 844–857.

[40] AJV Offutt. 1989. Automatic test data generation. (1989).

[41] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*. IEEE, 1–10.

[42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering* 44, 2 (2017), 122–158.

[43] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology* 104 (2018), 236–256.

[44] David Paterson, Jose Campos, Rui Abreu, Gregory M Kapfhammer, Gordon Fraser, and Phil McMinn. 2019. An Empirical Study on the Use of Defect Prediction for Test Case Prioritization. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 346–357.

[45] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 609–620.

[46] Anjana Perera, Aldeida Aleti, Marcel Böhme, and Burak Turhan. 2020. Defect Prediction Guided Search-Based Software Testing. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM. https://doi.org/10.1145/3324884.3416612

[47] Anjana Perera, Aldeida Aleti, Burak Turhan, and Marcel Boehme. 2022. An Experimental Assessment of Using Theoretical Defect Predictors to Guide Search-Based Software Testing. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3147008

[48] José Miguel Rojas, Mattia Vivanti, Andrea Arcuri, and Gordon Fraser. 2017. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering* 22, 2 (2017), 852–893.

[49] Gregg Rothermel, Roland H Untch, Chengyun Chu, and Mary Jean Harrold. 1999. Test case prioritization: An empirical study. In *Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)*. IEEE, 179–188.

[50] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 598–608.

[51] Ripon K Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul R Prasad. 2018. Bugs. jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th International Conference on Mining Software Repositories*. 10–13.

[52] Alireza Salahirad, Hussein Almulla, and Gregory Gay. 2019. Choosing the fitness function for the job: Automated generation of test suites that detect real faults. *Software Testing, Verification and Reliability* 29, 4-5 (2019), e1701.
[53] Sina Shamshiri, Rene Just, Jose Miguel Rojas, Gordon Fraser, Phil McMinn, and Andrea Arcuri. 2015. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 201–211.
[54] Joengju Sohn and Shin Yoo. 2019. Empirical evaluation of fault localisation using code and change metrics. *IEEE Transactions on Software Engineering* (2019).
[55] John W Tukey. 1949. Comparing individual means in the analysis of variance. *Biometrics* (1949), 99–114.
[56] Zhiyuan Wan, Xin Xia, Ahmed E Hassan, David Lo, Jianwei Yin, and Xiaohu Yang. 2018. Perceptions, expectations, and challenges in defect prediction. *IEEE Transactions on Software Engineering* 46, 11 (2018), 1241–1266.
[57] Jingxiu Yao and Martin Shepperd. 2020. Assessing software defection prediction performance: Why using the Matthews correlation coefficient matters. In *Proceedings of the Evaluation and Assessment in Software Engineering*. 120–129.
[58] Soner Yigit and Mehmet Mendes. 2018. Which effect size measure is appropriate for one-way and two-way ANOVA models? A Monte Carlo simulation study. *Revstat Statistical Journal* 16, 3 (2018), 295–313.
[59] Hongyu Zhang and Xiuzhen Zhang. 2007. Comments on "Data Mining Static Code Attributes to Learn Defect Predictors". *IEEE Transactions on Software Engineering* 33, 9 (2007), 635–637. https://doi.org/10.1109/TSE.2007.70706
[60] Thomas Zimmermann, Nachiappan Nagappan, Harald Gall, Emanuel Giger, and Brendan Murphy. 2009. Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. 91–100.