Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation

Anonymous Author(s)

ABSTRACT

Unit testing is a critical part of the software development process, ensuring the correctness of basic programming units in a program (e.g., a method). Search-based software testing (SBST) is an automated approach to generating test cases. SBST generates test cases with the genetic algorithms by specifying the coverage criterion (e.g., branch coverage). However, a good test suite must have different properties, which cannot be captured by using an individual coverage criterion alone. Therefore, the state-of-the-art approach combines multiple criteria to generate test cases. As combining multiple coverage criteria brings multiple objectives for optimization, it hurts the test suites' coverage for certain criteria compared with using the single criterion. To cope with this, we propose a novel approach named **smart selection**. Based on the coverage correlations among criteria and the coverage goals' subsumption relationships, smart selection selects a subset of coverage goals to reduce the number of optimization objectives and avoid missing any properties of all criteria. We conduct experiments to evaluate smart selection on 400 Java classes with three state-of-the-art genetic algorithms. On average, smart selection outperforms the original combination (i.e., combining all goals) on 77 Java classes, accounting for 65.1% of the classes having significant differences between the two approaches.

CCS CONCEPTS

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

KEYWORDS

SBST, object oriented, software testing, search-based, testing, test generation

ACM Reference Format:

Anonymous Author(s). 2022. Selectively Combining Multiple Coverage Goals in Search-Based Unit Test Generation. In *Proceedings of IEEE/ACM Automated Software Engineering (ASE 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/XXXXXX.XXXXXXX

1 INTRODUCTION

Unit testing is a common way to ensure software quality. Manually preparing unit tests can be a tedious and error-prone process. Hence, developers and researchers put much effort into automatically generating test cases for programming units over the years.

Search-based software testing (SBST) is considered a promising approach to generating test cases. It generates test cases with the genetic algorithms (e.g. Whole Suite Generation (WS) [15], MOSA [38], DynaMOSA [39]) by specifying the coverage criterion (e.g., branch coverage). The execution of a genetic algorithm relies on fitness functions, which quantify the degree to which a solution (i.e., one or more test cases) achieves its goals (i.e., satisfying a certain coverage criterion). For each coverage criterion, there are a group of fitness functions. Each fitness function describes whether or how far a test case covers a goal (e.g., a branch).

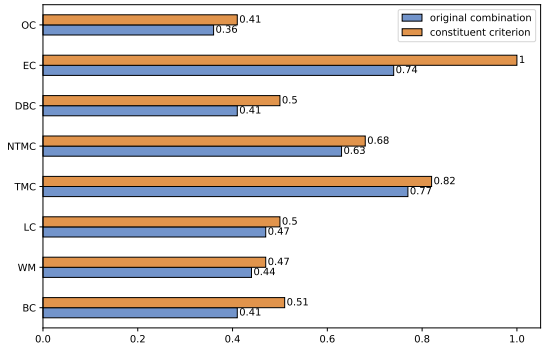

Figure 1: Partial Data of Coverage Gaps

The Problem and Motivation. However, as claimed in [41], a good test suite must have different properties, which cannot easily be captured by any individual coverage criterion alone. Therefore, to generate a good test suite, multiple coverage criteria should be considered in SBST. Hence, the state-of-the-art approach [41] combines multiple coverage criteria to guide genetic algorithms. It involves eight coverage criteria (see Sec. 2.1). We call this method the **original combination** in this paper. However, combining multiple criteria brings more objectives for optimization, potentially affecting the effectiveness of the genetic algorithms [7, 28, 38]. For example, it can increase the probability of being trapped in local optima. As a result, the generated test suite's coverage decreases for certain criteria compared with using a single criterion. Fig. 1 shows the partial results of our experiments (see Sec. 4.2): the average coverage gaps between the original combination and each constituent criterion when applying WS into 85 big Java classes (i.e., with at least 200 branches). The average gap of eight criteria is 8.2%. Branch coverage (BC) decreases 10% and exception coverage (EC) decreases 26%. Note that since we cannot know the total exceptions in a class [41], we normalize the exception coverage values of two approaches (22.08 vs. 29.74) by dividing by the larger one.

Targets. To cope with this, a qualified approach should fulfil the following targets: (1) **T1: GA Effectiveness.** This approach should

ASE 2022, October 10-14, 2022, Ann Arbor, Michigan, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXX.XXXXXXX

select a subset from multiple coverage criteria' coverage goals. This subset improves the effectiveness of guiding genetic algorithms (GAs); and (2) **T2: Property Consistence.** This subset should avoid missing any properties captured by these coverage criteria.

Our Solution. To fulfil these targets, we propose a novel approach named **smart selection** (see Sec. 3). Smart selection also adopts a combined way to generate test cases. In this paper, we also consider the above eight coverage criteria. Instead of directly combining them, in smart selection, we firstly group them into four groups based on coverage correlations (see Sec. 3.2). Next, we select one representative criterion that is more effective to guide the genetic algorithms from each group (T1) (see Sec. 3.3). These selected coverage criteria (SC)' coverage goals are marked as $Goal(SC)$. To keep the property consistency (T2), for each criterion (c) of unselected criteria (USC), we select a subset $Goal(c)_{sub}$ from its coverage goals based on the goals' subsumption relationships (see Sec. 3.4). Finally, we combine $Goal(SC)$ and $\bigcup_{c \in USC} Goal(c)_{sub}$ to guide SBST.

Contribution. In summary, the contribution of this paper includes:

- To the best of our knowledge, this is the *first* paper that uses coverage correlations to address the coverage decrease caused by combining multiple criteria in SBST.
- We implement smart selection atop EvoSuite. It is integrated into three search algorithms, i.e., WS, MOSA, and DynaMOSA.
- We conduct experiments on 400 Java classes to compare smart selection and the original combination. On average of three algorithms (WS/MOSA/DynaMOSA), smart selection outperforms the original combination on 77 (121/78/32) classes, accounting for 65.1% (85.8%/65%/44.4%) of the classes having significant differences between the two approaches. The counterpart data of the 85 big classes is 34 (50/35/16), accounting for 86.1% (98%/87.5%/72.7%). Furthermore, we conduct experiments to compare smart selection with/without the subsumption strategy on 173 classes.

Online Artifact. The online artifact of this paper can be found at: https://doi.org/10.5281/zenodo.6467640.

2 BACKGROUND

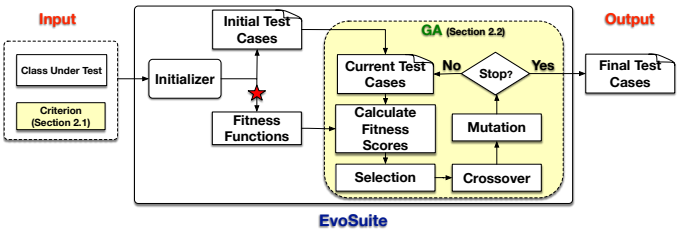

Figure 2: Overview of Unit Tests Generation in EvoSuite

SBST and EvoSuite SBST generates test cases with a genetic algorithm (see Sec. 2.2) by specifying the coverage criterion (see Sec. 2.1). EvoSuite [14] is the state-of-art SBST tool for Java. In this section, we leverage EvoSuite as an example to illustrate the key idea in SBST. Fig. 2 shows the overview of EvoSuite. The red star in this figure will be mentioned in Sec. 3.1.

Input. Evosuite takes two major inputs: (1) the class under test (CUT) and (2) a coverage criterion (see Sec. 2.1).

Test Generation. The test generation process can be divided into two parts: (1) The initializer extracts all the related information needed by the genetic algorithm (e.g., method signatures, including name, parameter type) from the CUT. Based on the information and the criterion, the initializer generates initial test cases and fitness functions. In general, each GA requires one or more specific fitness functions. A fitness function measures how close a test case covers a coverage goal (e.g., a branch); (2) After a specific genetic algorithm is invoked, it selects test cases based on the scores returned by fitness functions. Next, the GA creates new test cases by the crossover and mutation operations [15]. The GA repeats to select, mutate, and crossover test cases until all fitness functions reach the optima or the given budgets are consumed out.

Output. After running the genetic algorithm, EvoSuite outputs the final test cases.

2.1 Coverage Criteria

In this research, we discuss eight criteria as follows. The reason to choose these coverage criteria is that they are EvoSuite's default criteria and are widely used in many previous studies [8, 18, 41].

Branch Coverage (BC) BC checks the number of branches of conditional statements covered by a test suite.

Direct Branch Coverage (DBC) The only difference between DBC and BC is that a test case must directly invoke a public method to cover its branches. DBC treats others in the same way as BC [41].

Line Coverage (LC) LC checks the number of lines covered by a test suite.

Weak Mutation (WM) WM checks how many mutants are detected by a test suite [25, 40]. A mutant is a variant of the CUT generated by a mutation operator. For example, RC is an operator that replaces a constant value with different values [16].

Top-Level Method Coverage (TMC) TMC checks the number of methods covered by a test suite with a requirement: A public method is covered only when it is directly invoked by test cases.

No-Exce. Top-Level Method Coverage (NTMC) NTMC is TMC with an extra requirement: A method is not covered if it exits with any exceptions.

Exception Coverage (EC) EC checks the number of exceptions triggered by a test suite.

Output Coverage (OC) OC measures the diversity of the return values of a method. For example, a *boolean* return variable's value can be *true* or *false*. OC's coverage is 100% only if the test suite captures these two return values.

Based on each criterion's definition, EvoSuite extracts a group of coverage goals from the CUT and assigns a fitness function to each coverage goal. For example, EvoSuite extracts all branches for BC, e.g., the true branch of a predicate $x == 10$. A simplified fitness function of this branch can be the branch distance [33], $|x - 10|$, showing how far a test case covers it. More details of these criteria and their fitness functions can be found here [41].

2.2 Genetic Algorithms

In this research, we discuss three GAs as follows. All of them are integrated into EvoSuite and perform well in many SBST competitions [9, 37, 49]. These algorithms share the same inputs and outputs but differ in how to use fitness functions.

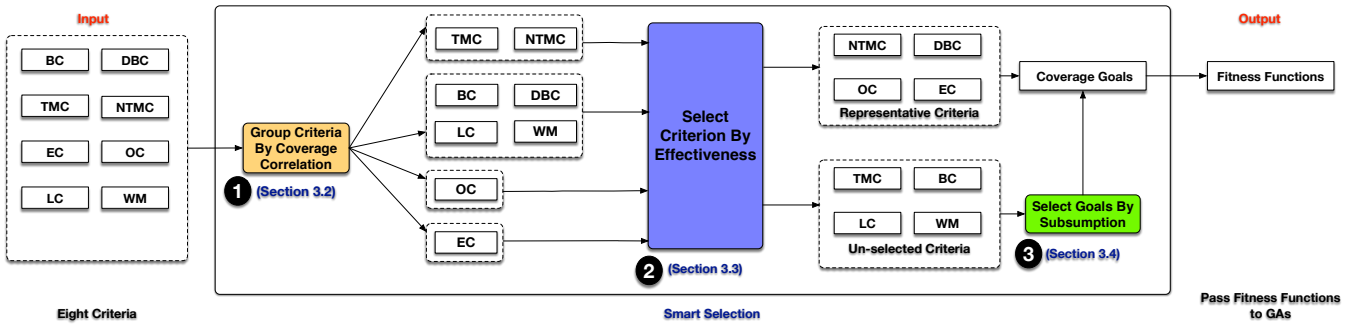

Figure 3: Overview of Smart Selection

WS. WS [15] directly evolves test suites to fit all coverage goals. Consequently, WS can exploit the collateral coverage [5] and not waste time on infeasible goals (e.g., dead code). The collateral coverage means that a test case generated for one goal can implicitly also cover any number of other coverage goals. Hence, WS's fitness function is the sum of all goals' fitness functions.

MOSA. WS sums fitness scores of all coverage goals as a scalar value. This scalar value is less monotonic and continuous than a single goal's fitness score, increasing the probability of being trapped in local optima. To overcome this limit, Panichella et al. [38] formulates SBST as a many-objective optimization problem and propose MOSA, a variant of NSGA-II [44]. In general, MOSA maintains a fitness vector for each test case. An item of the fitness vector is a fitness function value for the test case. Based on Pareto dominance [12], MOSA sorts and selects test cases by the fitness vectors.

DynaMOSA. Based on MOSA, DynaMOSA [39] adopts the control dependency graph to reduce the coverage goals evolved in search. A goal is selected to be in the evolving process only when the branch goals it depends on are covered. Hence, the sizes of DynaMOSA's fitness vectors are often smaller than MOSA's ones. Empirical studies show that DynaMOSA outperforms WS and MOSA [8, 39].

2.3 Combining Coverage Criteria

With a single criterion (e.g., BC) alone, SBST can generate test cases that reach higher code coverage but fail to meet users' expectations [41]. Hence, Rojas et al. [41] proposed to combine multiple criteria to guide SBST to generate a test suite. We leverage replacing BC by combining the eight criteria as an example to show the changes in GAs. Before the combination, the fitness function of WS is $f_{BC} = \sum_{b \in B} f_b$, where B is the set of all branches. The fitness vector of MOSA/DynaMOSA is $[f_{b_1}, ..., f_{b_n}]$. After the combination, the fitness function of WS is $f_{BC} + ... + f_{OC}$. The fitness vector of MOSA/DynaMOSA is $[f_{b_1}, ..., f_{b_n}, ..., f_{o_1}, ..., f_{o_m}]$, where o_i is a output coverage goal.

3 SMART SELECTION

The Problem and Motivation: The main side-effect of combining multiple criteria is that the generated test suite's coverage decreases for certain criteria. It is due to the increase in optimization objectives. Firstly, It brings a *larger* search space, reducing the search weight of each objective. Secondly, it also brings a *harder* search space since some criteria' fitness functions are not monotonic and continuous, such as LC and WM (see Sec. 3.3). Hence, we propose **smart selection**. It aims to relieve the coverage decrease by providing a smaller and easier search space for GAs.

3.1 Overview

Fig. 3 shows the process: ❶ group criteria by coverage correlation (Sec. 3.2), ❷ select representative criteria by effectiveness to guide SBST (Sec. 3.3), and ❸ select representative coverage goals from unselected criteria by subsumption relationships (Sec. 3.4). The red star in Fig. 2 shows the position of smart selection in EvoSuite.

The inputs are the eight criteria (see Sec. 2.1). The output is a subset of fitness functions, i.e., the corresponding fitness functions of our selected coverage goals. This subset is used to guide GAs.

3.2 Group Criteria by Coverage Correlation

The first step is clustering these eight criteria. The standard of whether two criteria can be in one group or not is: whether these two criteria have a coverage correlation. Based on this standard, we can divide these criteria into several groups. For two criteria with coverage correlation, if a test suite achieves high coverage under one of the criteria, then this test suite may also achieve high coverage under the other criterion. Hence, we can select one of these two criteria to guide SBST. Thus, grouping sets the scope for choosing representative criteria.

We determine that two criteria have a coverage correlation by the following rules:

•**Rule 1:** If a previous study shows that two criteria have a coverage correlation, we adopt the conclusion:

① **BC and WM:** Gligoric et al. [20] find that "branch coverage performs as well as or better than all other criteria studied, in terms of ability to predict mutation scores". Their work shows that the average Kendall's τ_b value [27] of coverage between branch coverage and mutation testing is 0.757. Hence, we assume that BC and WM have a coverage correlation.

② **DBC and WM:** Since BC and WM have a coverage correlation, we assume that DBC and WM have a coverage correlation too.

③ **LC and WM:** Gligoric et al. [20] find that statement coverage [20, 41] can be used to predict mutation scores too. Line coverage is an alternative for statement coverage in Java since Java's bytecode

instructions may not directly map to source code statements [41]. Hence, we assume that LC and WM have a coverage correlation.

•**Rule 2:** Two criteria, A and B, have a coverage correlation if they satisfy two conditions: (1) A and B have the same coverage goals; (2) A test covers a goal of A only if it covers the counterpart goal of B and it satisfies A's additional requirements:

④ **DBC and BC:** DBC is BC with an additional requirement (see Sec. 2.1).

⑤ **NTMC and TMC:** NTMC is TMC with an additional requirement (see Sec. 2.1).

•**Rule 3:** Two criteria, A and B, have a coverage correlation if, for an arbitrary test suite, the relationship between these two criteria' coverage (i.e., C_A and C_B) can be formulated as:

$$C_B = \Theta C_A, \tag{1}$$

where Θ is a nonnegative random variable and $E\Theta \approx 1$:

⑥ **BC and LC:** Intuitively, when a branch is covered, then all lines in that branch are covered. But this is not always true. When a line exits abnormally (e.g., it throws an exception.), the subsequent lines are not covered either. First, we discuss the coverage correlation of branch and line coverage in the absence of abnormal exiting. Let B be the set of branches of the CUT, L be the set of lines, and T be a test suite. For any $b \in B$, let L_b be the set of lines **only** in the branch b (i.e., we don't count the lines in its nested branches). Consequently, $L = \bigcup_{b \in B} L_b$. Let $B^{'}$ be the set of covered branches. Let $L^{'}$ be the set of covered lines. The coverage values measured by branch and line coverage are:

$$C_{Branch} = \frac{|B^{'}|}{|B|}, C_{Line} = \frac{|L^{'}|}{|L|} = \frac{\sum\limits_{b \in B^{'}} |L_b|}{\sum\limits_{b \in B} |L_b|}. \tag{2}$$

Hence, the relationship of C_{Branch} and C_{Line} is:

$$\frac{C_{Line}}{C_{Branch}} = \frac{\sum\limits_{b \in B^{'}} |L_b|}{\sum\limits_{b \in B} |L_b|} \div \frac{|B^{'}|}{|B|} = \frac{\sum\limits_{b \in B^{'}} |L_b|}{|B^{'}|} \div \frac{\sum\limits_{b \in B} |L_b|}{|B|}. \tag{3}$$

Suppose we treat branches with different numbers of lines equally in generating T. Then we have:

$$\frac{\sum\limits_{b \in B^{'}} |L_b|}{|B^{'}|} \approx \frac{\sum\limits_{b \in B} |L_b|}{|B|}, \tag{4}$$

i.e.,

$$\frac{C_{Line}}{C_{Branch}} \approx 1. \tag{5}$$

As a result, branch coverage and line coverage have a coverage correlation in the absence of abnormal exiting. With abnormal exiting, the coverage measured by line coverage decreases. Assuming that any line can exit abnormally, we can formulate the coverage relationship as:

$$C_{Line} = \Theta C_{Branch}, \tag{6}$$

where Θ is a random variable. In this research, instead of analyzing Θ precisely, we only need to check whether $E\Theta \approx 1$. Previous work [41] shows that, on average, when 78% of branches are covered, test suites can only find 1.75 exceptions. Hence, we assume that $E\Theta \approx 1$, i.e., BC and LC have a coverage correlation.

⑦ **DBC and LC:** We assume that DBC and LC have a coverage correlation since BC and LC have a coverage correlation.

Output. We cluster the eight criteria into four groups: (1) BC, DBC, LC, and WM; (2) TMC and NTMC; (3) EC; and (4) OC.

3.3 Select Representative Criterion by Effectiveness to guide SBST

In this step, among the criteria in each group, we select a criterion to represent the others. The criteria within a group differ in the ability to guide SBST. If we only select one criterion with the best effectiveness to guide SBST, SBST will be more efficient in generating unit tests. To select the best criterion to guide SBST in each group, we need to compare the criteria' effectiveness in guiding SBST. A criterion's effectiveness in guiding SBST largely depends on the continuity of monotonicity of its fitness functions [30, 31]. Hence, we need to analyze and compare the criteria' fitness functions.

Group1: BC, DBC, LC and WM. We use branch coverage as the baseline and divide them into three pairs for discussion. The reason to use branch coverage as the baseline is that branch coverage has been widely used to guide unit test generation [15, 38, 39] due to the monotonic continuity of its fitness functions. For a branch goal b and a test case t, its fitness function is below [15]:

$$f_{bc}(b,t) = \begin{cases} 0 & \text{if the branch has been covered,} \\ v(d(b,t)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise,} \end{cases} \tag{7}$$

where $v(x)$ is a normalizing function in $[0,1]$ (e.g., $v(x) = x/(x+1)$). $d(b,t)$ is a function to provide a branch distance to describe how far a test case covers this goal [33]. To avoid an oscillate situation of a predicate [15], $f_{bc}(b,t)$ uses $v(d(b,t))$ only when a predicate is executed at least twice.

WS uses the sum of all fitness functions as one fitness function (see Sec. 2.2). Hence, for WS, branch coverage's fitness function is:

$$d_1(b,T) = \min\{f_{bc}(b,t) | t \in T\}, \tag{8}$$

$$f_{BC}(T) = \sum_{b \in B} d_1(b,T), \tag{9}$$

where B denotes all branches of the CUT.

•**BC vs. LC.** Based on line coverage's definition (see Sec. 2.1), a line l's fitness function can be:

$$f_{lc}(l,t) = \begin{cases} 0 & \text{if the line has been covered,} \\ 1 & \text{otherwise.} \end{cases} \tag{10}$$

For WS, line coverage's fitness function is:

$$f_{LC}(T) = v(|L| - |CL|), \tag{11}$$

where L is the set of all lines and CL is the set of covered lines.

These two fitness functions are not continuous and monotonic since they only tell whether the lines are covered. To overcome this limit, EvoSuite uses branch coverage's fitness functions to augment

line coverage's fitness functions [41]. A line l's fitness function is:

$$f_{lc}(l,B,t) = \begin{cases} 0 & \text{if the line has been covered,} \\ 1 + \min\{f_{bc}(b,t) | b \in B\} & \text{otherwise,} \end{cases} \tag{12}$$

where B is the set of branches that l depends on [39]. For WS, line coverage's fitness function is:

$$f_{LC}(T) = v(|L| - |CL|) + f_{BC}(T). \tag{13}$$

We call Equation 10 and 11 *def-based* (definition-based) fitness functions and call Equation 12 and 13 *augmented* fitness functions.

Firstly, we compare branch coverage's fitness functions with line coverage's *def-based* fitness functions. Line coverage's *def-based* fitness functions are not continuous and monotonic since they only tell whether the lines are covered. Hence, branch coverage is better than line coverage in the effectiveness to guide SBST when we use line coverage's *def-based* fitness functions. After the augmentation, line coverage's *def-based* fitness functions disturb the continuity and monotonicity of branch coverage's fitness functions, undermining branch coverage's effectiveness to guide SBST. As a result, BC is better than LC in the effectiveness to guide SBST.

•**BC vs. WM.** Based on weak mutation's definition (see Sec. 2.1), a mutant's fitness function is:

$$f_{wm}(\mu,t) = \begin{cases} 1 & \text{if mutant } \mu \text{ was not reached,} \\ v(id(\mu,t)) & \text{if mutant } \mu \text{ was reached,} \end{cases} \tag{14}$$

where $id(\mu,t)$ is the infection distance function. It describes how distantly a test case triggers a mutant's different state from the source code. Different mutation operators have different infection distance functions [16]. A mutant's fitness function is always 1 unless a test case reaches it (i.e., the mutated line is covered). Hence, like line coverage, EvoSuite uses the same way to augment weak mutation's fitness functions [16, 39]. As the conclusion of comparing BC and LC, BC is better than WM in the effectiveness to guide SBST.

•**BC vs. DBC.** Direct branch coverage is branch coverage with an extra requirement: A test case must directly invoke a public method to cover its branches. Based on branch coverage's fitness function, we can get direct branch coverage's one: For a branch in a public method, when the method is not invoked directly, the fitness function always returns 1. Otherwise, the fitness function is the same as branch coverage's one. It is easy for SBST to generate a test case that invokes a public method directly. Hence, we regard that BC is nearly equal to DBC in guiding SBST.

Order of Group1. Above all, we get a rough order of this group: (1)BC and DBC; (2) LC and WM. Since we only need one representative, the rough order satisfies our need.

The Representative Criterion of Group1. We choose DBC to represent this group instead of BC. The reason is: When a test case covers a goal of DBC, the test case covers the counterpart of BC. As a result, DBC can fully represent BC. The opposite may not hold.

Group2: TMC and NTMC. Like the relationship between branch coverage and direct branch coverage, no-exce. top-level method coverage is top-level method coverage with an extra requirement: A method must be invoked without triggering exceptions.

The Representative Criterion of Group2. We choose NTMC to represent this group. The reason is the same as why we choose DBC to represent group 1: NTMC can fully represent TMC. The opposite does not hold.

Group3: EC and Group 4: OC. Since group 3 only contains EC, we choose EC to represent group 3. Similarly, we choose OC to represent group 4.

Output. The representative criteria are DBC, NTMC, EC, and OC.

3.4 Select Representative Coverage Goals by Subsumption Relationships

After selecting the representative criteria in the previous step, there are four unselected criteria: LC, WM, BC, and TMC. To keep property consistency for each unselected criterion, we select a subset from its coverage goals. This subset can represent all properties required by this criterion. We have another requirement for these subsets: they should be as small as possible. These unselected criteria' fitness functions are less continuous and monotonic than the ones of the representative criteria (see Sec. 3.3). Hence, to minimize the negative effects on guiding SBST, these subsets should be as small as possible.

Two coverage goals, G_1 and G_2, having the subsumption relationship denotes that if a test suite covers one coverage goal, it must cover another goal. Specifically, G_1 subsuming G_2 represents that if a test suite covers G_1, it must cover G_2. According to this definition, for a criterion, if the coverage goals not subsumed by others are covered, all coverage goals are covered. Hence, These coverage goals form the desired subset.

LC. For the lines in a basic block, the last line subsumes others. Hence, these last lines of all basic blocks form the desired subset. Since Sec. 3.2 shows that BC/DBC and LC have a strong coverage correlation and DBC is the representative criterion, we do a tradeoff to shrink this subset: We add an integer parameter *lineThreshold*. If a basic block's lines are less than *lineThreshold*, we skip it. In this research, we set *lineThreshold* as 8 (Sec. 5.1 discusses it).

WM. The process to extract the subset from weak mutation's all mutants can be divided into three parts: ① We select the key operators from EvoSuite's all implemented mutation operators; ② From the key operators we filter out the **equal-to-line** operators; ③ For the remaining operators, we select the subsuming mutants by following the previous work [19].

① **Select Key Operators.** Offutt et al. [36] find that five key operators achieve 99.5% mutation score. They are UOI, AOR, ROR, ABS, and LCR. EvoSuite does not implement LCR (an operator that replaces the logical connectors) and ABS (an operator that inserts absolute values) [16]. Hence, we select three operators: UOI, AOR, and ROR (see Table 1).

Table 1: Evosuite's Partial Mutation Operators

Operator	Usage
UOI	Insert unary operator
AOR	Replace an arithmetic operator
ROR	Replace a comparison operator

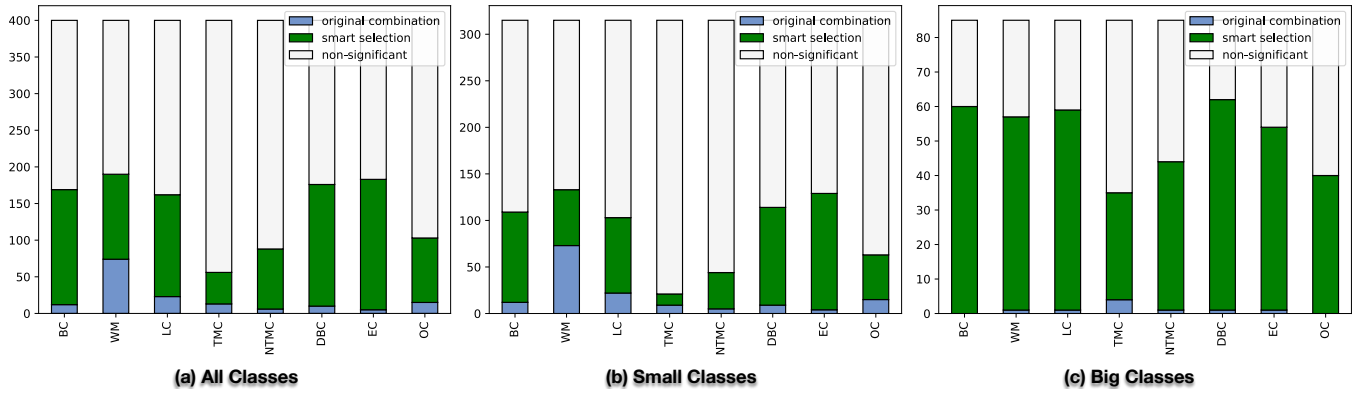

Figure 4: Significant case summary of smart selection and the original combination with WS

② **Filter out Equal-To-Line Operators.** For each mutation operator, EvoSuite designs an infection distance function to describe how far a mutant's different state from the source code is triggered [16]. Some infection distance functions always return 0. For example, UOI only adds 1 to, subtracts 1 to, or negates a numerical value, so the infection distance is always 0. Hence, if a test case covers the line mutated by UOI, the mutant is killed. We call this kind of operator an **equal-to-line** operator. Among three key operators, only UOI is the equal-to-line operator [16]. Since line coverage has been dealt with, we filter out it.

③ **Select Subsuming Mutants.** The remains are AOR and ROR. We choose one of the existing approaches [19, 22, 24, 26, 35] to select subsuming mutants for them. These approaches can be divided into three categories: (1) Manual analysis: Just et al. [26] build the subsumption relationships for ROR and LCR by analyzing all possible outputs of their mutants. This approach can not be applied to non-logical operators [22]; (2) Dynamic analysis: By running an exhaustive set of tests, Guimarães et al. [22] build the subsumption relationships. This approach needs many tests, which we can not provide; (3) Static analysis: Gheyi and Souza et al. [19, 46] encode a theory of subsumption relations in the Z3 theorem prover to identify the subsumption relationships. We adopt this approach because (i) This approach can be applied to both AOR and ROR; (ii) Using the Z3 prover to identify the subsumption relationships is a once-for-all job. We can hardcode their results into EvoSuite.

BC and TMC. For a coverage goal of branch coverage, there is a subsuming goal from direct branch coverage (see Sec. 3.2). As a result, the subset for branch coverage is empty since we select direct branch coverage as the representative (see Sec. 3.3). Similarly, the subset for top-level method coverage is empty too.

Output. For four unselected criteria, we select four subsets of their coverage goals. Two of them are empty.

Finally, smart selection joins these subsets with the representative criteria to get their fitness functions for guiding GAs.

4 EVALUATION

4.1 Experiment Setting

The evaluation focuses on the performance of smart selection. Our evaluation aims to answer the following research questions:

- **RQ1:** How does smart selection perform with WS?
- **RQ2:** How does smart selection perform with MOSA?
- **RQ3:** How does smart selection perform with DynaMOSA?
- **RQ4:** How does the subsumption strategy affect the performance of smart selection?

Environment. All experiments are conducted on two machines with Intel(R) Core(TM) i9-10900 CPU @ 2.80GHz and 128 GB RAM.

Subjects. We randomly select Java classes from 2 sources: the benchmark of DynaMOSA [39] and Hadoop [1]. Following the previous work [38], the only restriction of randomly selecting classes is that the class must contain at least 50 branches, aiming to filter out the trivial classes. As a result, we select 400 classes: 158 from the benchmark of DynaMOSA and 242 from Hadoop.

Baseline for RQ1-3. We have two baselines: (1) the original combination, used to be compared with smart selection on each Java class; (2) a single constituent criterion, used to show the data of coverage decrease caused by the above two combination approaches. A single constituent criterion means that we only use each criterion of these eight criteria (see Sec. 2.1) to guide GAs. There is one exception: when the constituent criterion is exception or output coverage, we combine this criterion and branch coverage to guide GAs. The reason is that only exception or output coverage is weak in the effectiveness of guiding the GAs [18, 41]. Branch coverage can guide the GAs to reach more source lines of the CUT, increasing the possibility of triggering exceptions or covering output goals.

Configuration for RQ1-3. EvoSuite provides many parameters to run the algorithms, such as crossover probability and population size [15]. We choose its default parameter values to run smart selection and other baselines. Smart selection introduces a new parameter *lineThreshold* (see Sec. 3.4). It controls smart selection to skip basic blocks with less than *lineThreshold* lines. We set *lineThreshold* as 8. The discussion on this value is in Sec. 5.1. For each Java class, we run EvoSuite with ten approaches: (1) smart selection, (2) the original combination, and (3) each constituent criterion of all eight criteria. We run each approach for 30 rounds per Java class, and each run's search budget is 2 minutes.

4.2 RQ1: How does smart selection perform with WS?

Motivation. In this RQ, we evaluate smart selection (**SS**) with WS. First, we compare the performance of SS and the original combination (**OC**). Next, we use the coverage of each constituent criterion (**CC**) to show the coverage decrease caused by SS and OC. Furthermore, we show these approaches' differences in the resulted suite size (i.e., the number of tests in a test suite).

Methodology. EvoSuite records the coverage for generated unit tests. For each class, we obtain 10 coverage data sets: One data set records the coverage of the eight criteria when using SS; One data set records the coverage of the eight criteria when using OC; The rest data sets record the coverage when using each CC.

For each Java class, we follow previous research work [41] to use Mann-Whitney U Test to measure the statistical difference between SS and OC. Then, we use the Vargha-Delaney $\hat{A}_{ab}$ [48] to evaluate whether a particular approach a outperforms another approach b ($A_{ab} > 0.5$ and the significant value p is smaller than 0.05).

Result. ►**All Classes.**

Table 2: Average coverage results for each approach with WS

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	55%	53%	57%	NTMC	71%	70%	71%
WM	59%	57%	59%	DBC	55%	53%	56%
LC	60%	58%	60%	EC	15.92	14.52	16.52
TMC	84%	83%	84%	OC	44%	43%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	58%	57%	59%	NTMC	73%	72%	72%
WM	62%	61%	62%	DBC	57%	56%	58%
LC	62%	61%	62%	EC	13.29	12.48	12.95
TMC	85%	85%	85%	OC	46%	45%	46%

(c) Big Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	45%	41%	51%	NTMC	67%	63%	68%
WM	48%	44%	47%	DBC	45%	41%	50%
LC	50%	47%	50%	EC	25.69	22.08	29.74
TMC	79%	77%	82%	OC	39%	36%	41%

Table 3: Average test suite size of each approach with WS

approach	SS	OC	CC (Average)
size (All Classes)	51.35	47.77	31.59
size (Small Classes)	37.27	36.39	19.43
size (Big Classes)	103.53	89.95	76.64

Significant Cases. Fig. 4 (a) shows the comparison results of SS and OC on all 400 Java classes. SS outperforms OC on 121 (30.3%) classes (a.k.a., SS-outperforming classes) on average for each coverage. OC outperforms SS on 20 (4.9%) classes (a.k.a., OC-outperforming classes). These two approaches have no significant difference on 259 (64.8%) classes (a.k.a., No-significant classes) on average.

Average Coverage. Table 2 (a) shows the average coverage of all classes with three approaches. For exception coverage, the table shows the average number of the triggered exceptions since we can not know the total number of exceptions in a class [41]. The green number represents the highest coverage at a given criterion. SS outperforms OC for eight criteria' coverage. Among three approaches, SS reaches the highest coverage for four criteria. CC reaches the highest coverage for all criteria.

Average Suite Size. The first row of Table 3 shows the test suites' average sizes of all classes. Compared to CC (average suite size of all constituent criteria), the size of OC increases by 51.2% ((47.77 − 31.59)/31.59). Compared to OC, the size of SS increases by 7.4% ((51.35 − 47.77)/47.77).

►**Small Classes.(< 200 branches)**

Significant Cases. Fig. 4 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 71 (22.5%). OC-outperforming classes are 19 (5.9%). No-significant classes are 226 (71.6%).

Average Coverage. Table 2 (b) shows the average coverage of small classes. SS outperforms OC for seven criteria' coverage. SS reaches the highest coverage for five criteria.

Average Suite Size. The second row of Table 3 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 87.3%. Compared to OC, the size of SS increases by 2.4%.

►**Big Classes. (≥ 200 branches)**

Significant Cases. Fig. 4 (c) shows the comparison results of SS and OC on 85 big Java classes. For each criterion, on average, SS-outperforming classes are 50 (59.1%). The number of OC-outperforming classes is 1 (1.3%). No-significant classes are 34 (39.6%).

Average Coverage. Table 2 (c) shows the average coverage of big classes. SS outperforms OC for eight criteria' coverage. SS reaches the highest coverage for two criteria.

Average Suite Size. The third row of Table 3 shows the average suite sizes of big classes. Compared to CC, the size of OC increases by 17.4%. Compared to OC, the size of SS increases by 15.1%.

Analysis. SS outperforms OC statistically, especially on the big classes. There is one exception: On the small classes, the number (73) of OC-outperforming classes in weak mutation is more than the counterpart number (60) (see Fig. 4 (b)). On average, each of those 73 classes has 82 branches and 321 mutants, while each of those 60 classes has 115 branches and 381 mutants. It also supports that SS outperforms OC on the big classes. Furthermore, in most cases, the average coverage of CC is higher than the one of OC. SS narrows the coverage gap between them. For example, the biggest gap is 10%, happening in the big classes' branch coverage. SS narrows the gap by 4% (see Table 2 (c)). These facts confirm that combining criteria offers more objectives for optimization, affecting the efficacy of GAs. The bigger classes bring more objectives, leading to a higher impact. The suite size increase brought by OC/SS to CC is significant, confirming the experimental results of the work proposing OC [41]. The main reason is that the GA (not only WS but also MOSA/DynaMOSA) needs more tests for more goals. With the coverage increase, the suite size of SS is also greater than OC. Compared with the extent of the increasing extent of OC to CC, we regard that it is reasonable.

Answer to RQ1

With WS, SS outperforms OC statistically, especially on the big classes (i.e., the classes with no less than 200 branches).

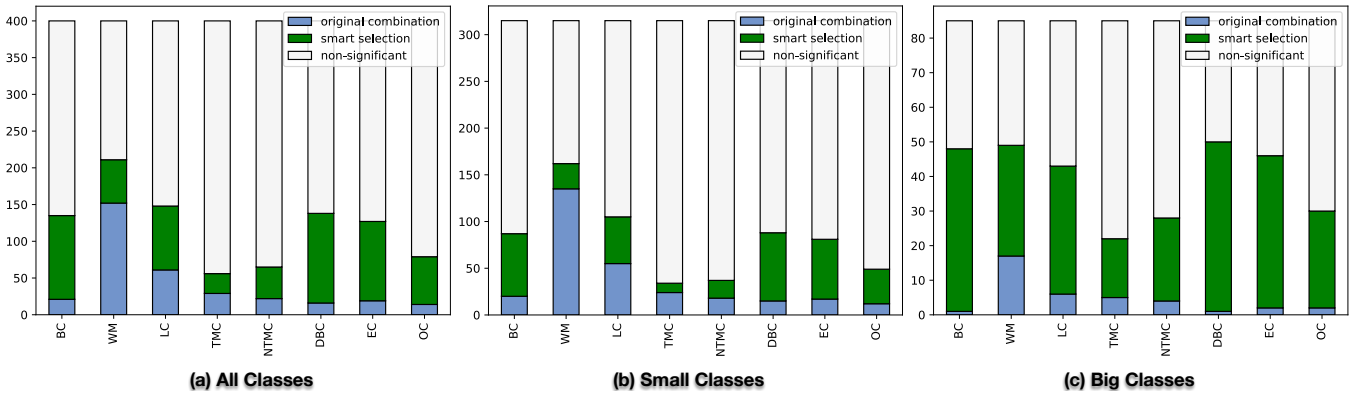

Figure 5: Significant case summary of smart selection and the original combination with MOSA

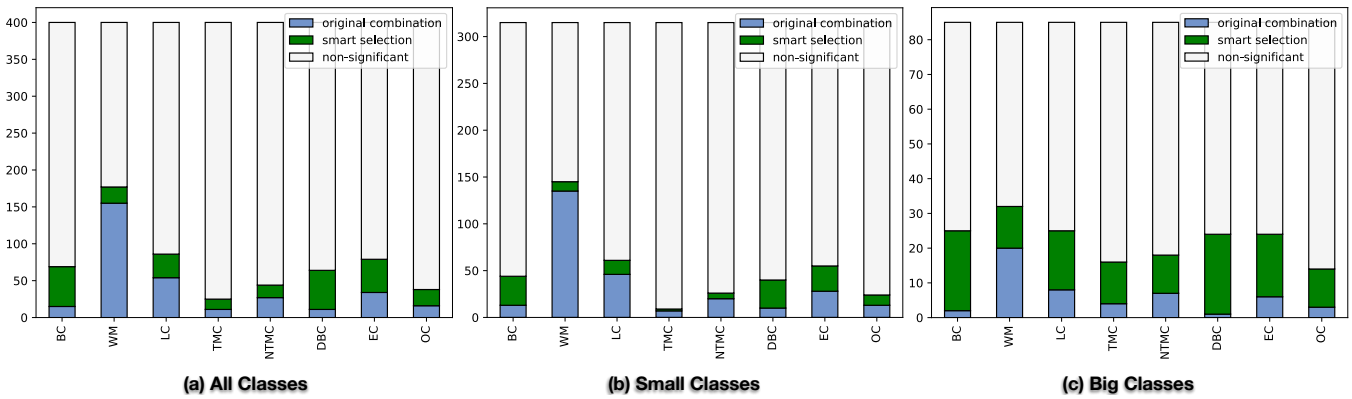

Figure 6: Significant case summary of smart selection and the original combination with DynaMOSA

4.3 RQ2: How does smart selection perform with MOSA?

Motivation. In this RQ, we evaluate smart selection with MOSA.
Methodology. The methodology is the same as RQ1's.
Result. ►**All Classes.**

Table 4: Average coverage results for each approach with MOSA

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	57%	56%	58%	NTMC	71%	71%	69%
WM	60%	60%	60%	DBC	57%	55%	57%
LC	61%	60%	61%	EC	16.95	16.15	16.41
TMC	84%	83%	82%	OC	44%	44%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	59%	58%	60%	NTMC	73%	72%	71%
WM	62%	62%	62%	DBC	58%	58%	59%
LC	63%	63%	63%	EC	13.28	13.07	12.73
TMC	85%	85%	84%	OC	45%	45%	46%

(c) Big Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	49%	46%	51%	NTMC	66%	64%	64%
WM	52%	49%	51%	DBC	50%	46%	51%
LC	54%	52%	53%	EC	30.54	27.53	30.03
TMC	79%	78%	77%	OC	40%	38%	42%

Table 5: Average test suite size of each approach with MOSA

approach	SS	OC	CC (Average)
size (All Classes)	57.03	54.46	31.47
size (Small Classes)	38.27	38.83	19.85
size (Big Classes)	126.56	112.38	74.53

Significant Cases. Fig. 5 (a) shows the comparison results of SS and OC on all 400 Java classes. For each criterion, on average, SS-outperforming classes are 78 (19.5%). OC-outperforming classes are 42 (10.4%). No-significant classes are 280 (70.1%).
Average Coverage. Table 4 (a) shows the average coverage of all classes. SS outperforms OC for five criteria' coverage. Among three approaches, SS reaches five criteria' highest coverage.
Average Suite Size. The first row of Table 5 shows the average suite sizes of all classes. Compared to CC, the size of OC increases by 73.1%. Compared to OC, the size of SS increases by 4.7%.
►**Small Classes.**
Significant Cases. Fig. 5 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 43 (13.8%). OC-outperforming classes are 37 (11.7%). No-significant classes are 235 (74.5%).

Average Coverage. Table 4 (b) shows the average coverage of small classes. SS outperforms OC for three criteria' coverage. SS reaches three criteria' highest coverage.
Average Suite Size. The second row of Table 5 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 95.6%. OC is nearly equal to SS.
►**Big Classes.**
Significant Cases. Fig. 5 (c) shows the comparison results of SS and OC on 85 big Java classes. For each criterion, on average, SS-outperforming classes are 35 (40.9%). OC-outperforming classes are 5 (5.6%). No-significant classes are 46 (53.5%).
Average Coverage. Table 4 (c) shows the average coverage of big classes. SS outperforms OC for eight criteria' coverage. SS reaches five criteria' highest coverage.
Average Suite Size. The third row of Table 5 shows the average suite sizes of big classes. Compared to CC, the size of OC increases by 50.7%. Compared to OC, the size of SS increases by 12.6%.
Analysis. SS outperforms OC on the big classes like WS. But the advantage of SS is unnoticeable on the small classes. The coverage gap between CC and OC is not significant as the gap in WS. SS nearly bridges this gap. The biggest gap is 5%, happening in branch coverage of the big classes. SS narrows this gap by 3%. The suite size gap between SS and OC is smaller than on WS, which is consistent with the fact that SS and OC have a smaller coverage gap on MOSA. These facts show the advantage of multi-objective approaches (e.g., MOSA) over single-objective approaches (e.g., WS) [7, 28, 38]. However, the advantage of SS on the big classes indicates that too many objectives also affect the multi-objective algorithms.

Answer to RQ2

With MOSA, SS outperforms OC statistically on the big classes. Smart selection has only a slight advantage on the small classes.

4.4 RQ3: How does smart selection perform with DynaMOSA?

Motivation. In this RQ, we evaluate smart selection with DynaMOSA.
Methodology. The methodology is the same as RQ1's.
Result. ►**All Classes.**
Significant Cases. Fig. 6 (a) shows the comparison results of SS and OC on all 400 Java classes. For each criterion, on average, SS-outperforming classes are 32 (8.1%). OC-outperforming classes are 40 (10.1%). No-significant classes are 328 (81.8%).
Average Coverage. Table 6 (a) shows the average coverage of all classes with three approaches. SS outperforms OC for one criterion's coverage, i.e., exception coverage. Among three approaches, SS reaches the highest coverage for three criteria.
Average Suite Size. The first row of Table 7 shows the average suite sizes of all classes. Compared to CC, the size of OC increases by 54.7%. OC is nearly equal to SS.
►**Small Classes.**
Significant Cases. Fig. 6 (b) shows the comparison results of SS and OC on 315 small Java classes. For each criterion, on average, SS-outperforming classes are 17 (5.2%). OC-outperforming classes are 34 (10.8%). No-significant classes are 265 (84%).

Table 6: Average coverage results for each approach with DynaMOSA

(a) All Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	58%	58%	58%	NTMC	71%	71%	70%
WM	60%	61%	62%	DBC	57%	57%	58%
LC	61%	62%	62%	EC	17.15	17.14	16.64
TMC	83%	83%	81%	OC	45%	45%	45%

(b) Small Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	60%	59%	60%	NTMC	72%	73%	72%
WM	63%	63%	64%	DBC	59%	59%	59%
LC	63%	64%	64%	EC	13.42	13.42	12.81
TMC	84%	85%	82%	OC	46%	46%	46%

(c) Big Classes

approach	SS	OC	CC	approach	SS	OC	CC
BC	51%	51%	53%	NTMC	66%	66%	65%
WM	53%	52%	54%	DBC	51%	50%	53%
LC	54%	54%	55%	EC	30.98	30.93	30.81
TMC	79%	79%	78%	OC	41%	41%	42%

Table 7: Average test suite size of each approach with DynaMOSA

approach	SS	OC	CC (Average)
size (All Classes)	61.13	60.59	39.2
size (Small Classes)	38.9	39.59	23.71
size (Big Classes)	143.51	138.44	96.58

Average Coverage. Table 6 (b) shows the average coverage of small classes. SS outperforms OC for one criterion's coverage (branch coverage). SS reaches two criteria' highest coverage.
Average Suite Size. The second row of Table 7 shows the average suite sizes of small classes. Compared to CC, the size of OC increases by 66.9%. OC is nearly equal to SS.
►**Big Classes.**
Significant Cases. Fig. 6 (c) shows the comparison results of SS and OC on 85 big Java classes. For each criterion, on average, SS-outperforming classes are 16 (18.7%). OC-outperforming classes are 6 (7.5%). No-significant classes are 63 (73.8%).
Average Coverage. Table 6 (c) shows the average coverage of big classes. SS outperforms OC for three criteria' coverage. SS reaches three criteria' highest coverage.
Average Suite Size. The third row of Table 7 shows the average suite sizes of big classes. Compared to CC, the size of OC increases by 43.3%. Compared to OC, the size of SS increases by 3.7%.
Analysis. SS still outperforms OC on the big classes, but not as obvious as WS and MOSA. In addition, SS is almost the same or slightly worse than OC on the small classes. Furthermore, the coverage gaps among the three approaches are not significant. The gap in the suite size between SS and OC is slight as in the coverage. The reason is that DynaMOSA selects the uncovered goals into the iteration process only when its branch dependencies are covered (see Sec. 2.2). Hence, the number of optimization objectives is reduced. Therefore, an increase in the goals has a much smaller impact on DynaMOSA's coverage performance than on WS and MOSA.

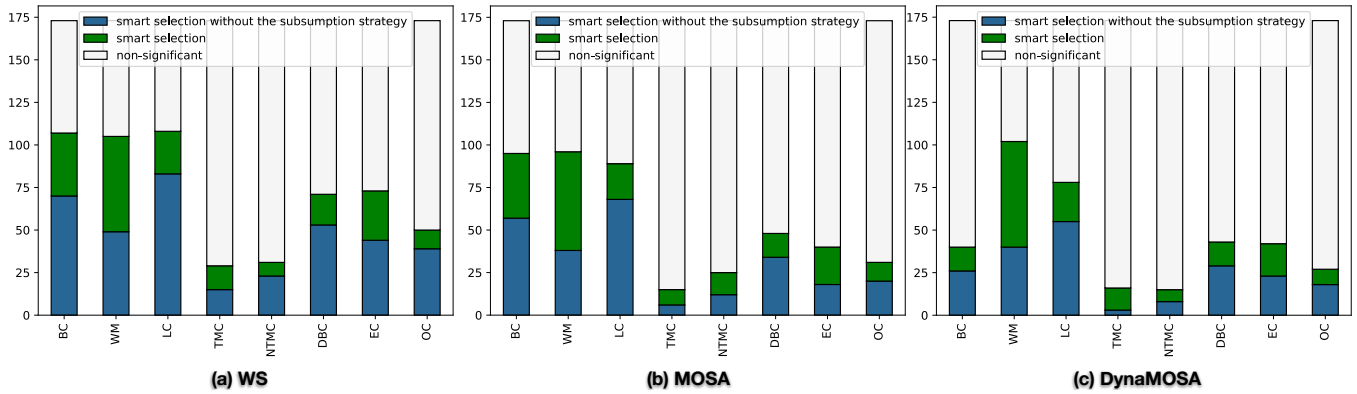

Figure 7: Significant case summary of smart selection and smart selection with subsumption strategy

Answer to RQ3

With DynaMOSA, smart selection slightly outperforms the original combination on the big classes. But, the original combination slightly outperforms smart selection on the small classes.

4.5 RQ4: How does the subsumption strategy affect the performance of smart selection?

Motivation. We select the representative goals from line coverage and weak mutation by the subsumption relationships (see Sec 3.4). We need to test how it affects the performance of smart selection.
Subjects. From 400 classes, we select those classes that satisfy this condition: The subsumption strategy can find at least one line coverage goal and one mutant. As a result, 173 classes are selected.
Configuration. We take smart selection without the subsumption strategy (SSWS) as a new approach. Then we keep other configuration settings the same as RQ1-3's.
Methodology. To compare SS and SSWS, we follow RQ1's methodology.
Result. ►**WS.**

Table 8: Average coverage and size results for smart selection and smart selection without the subsumption strategy

(a) WS (Suite Size: SS (45.18), SSWS (48.08))

approach	SS	SSWS	approach	SS	SSWS
BC	47%	48%	NTMC	63%	63%
WM	52%	53%	DBC	46%	48%
LC	51%	53%	EC	16.43	17.46
TMC	79%	78%	OC	38%	39%

(b) MOSA (Suite Size: SS (52.08), SSWS (51.66))

approach	SS	SSWS	approach	SS	SSWS
BC	49%	49%	NTMC	63%	63%
WM	53%	53%	DBC	49%	49%
LC	53%	54%	EC	18.19	18.32
TMC	79%	78%	OC	38%	39%

(c) DynaMOSA (Suite Size: SS (57.1), SSWS (54.79))

approach	SS	SSWS	approach	SS	SSWS
BC	50%	50%	NTMC	63%	63%
WM	54%	54%	DBC	49%	49%
LC	53%	54%	EC	18.26	18.68
TMC	79%	78%	OC	39%	39%

Significant Cases. Fig. 7 (a) shows the comparison results of SS and SSWS on 173 classes with WS. For each criterion, on average, SS-outperforming classes are 25 (14.5%). SSWS-outperforming classes are 47 (27.2%). No-significant classes are 101 (58.3%).
Average Coverage. Table 8 (a) shows the average coverage for WS. SS outperforms SSWS for one criterion's coverage (top-level method coverage). SSWS outperforms SS for six criteria.
►**MOSA.**
Significant Cases. Fig. 7 (b) shows the comparison results of SS and SSWS on 173 classes with MOSA. For each criterion, on average, SS-outperforming classes are 23 (13.3%). SSWS-outperforming classes are 32 (18.5%). No-significant classes are 118 (68.2%).
Average Coverage. Table 8 (b) shows the average coverage for MOSA. SS outperforms SSWS for one criterion's coverage (top-level method coverage). SSWS outperforms SS for three criteria.
►**DynaMOSA.**
Significant Cases. Fig. 7 (c) shows the comparison results of SS and SSWS on 173 Java classes with DynaMOSA. On average, SS-outperforming classes are 20 (11.6%). SSWS-outperforming classes are 25 (14.5%). No-significant classes are 128 (73.9%).
Average Coverage. Table 8 (c) shows the average coverage for DynaMOSA. SS outperforms SSWS for one criterion's coverage (top-level method coverage). SSWS outperforms SS for two criteria.
Analysis. SSWS outperforms slightly SS for most criteria on WS. It confirms that an increase in the objectives has a much bigger impact on WS than on MOSA and DynaMOSA. Furthermore, the results are different on line coverage and weak mutation for which SS adds subsets. For three algorithms, SSWS is better in line coverage in terms of the outperforming classes and average coverage. Contrarily, SS is better in weak mutation in terms of the outperforming classes. It indicates that the coverage correlation between (direct) branch coverage and line coverage is stronger than the one between (direct) branch coverage and weak mutation. As for the suite size, Table 8 shows that SS and SSWS are similar in all three algorithms.

Unexpectedly, SS outperforms SSWS on top-level method coverage. We analyze some classes qualitatively. For example, there is a public method named *compare* in an inner class of the class *org.apache.hadoop.mapred* [2]. The results show that 88 out of 90 test suites generated by SS cover this top-level method goal, while

only 1 out of 90 test suites generated by SSWS covers this goal. We find that this method contains 8 lines, 2 branches, and 3 output goals. EvoSuite skips the branches and output goals in the inner class (lines, methods, and mutants are kept). This class has 350 branches, 48 methods, and 10 output goals. SS selects 14 lines and 216 mutants for this class (0 lines and 10 mutants for this method). As a result, if a test directly invokes this method, under SS, at most (1 + 10) out of 638 (2%) goals are closer to being covered; under SSWS, the number is 1 out of 408 (0.2%). It explains why all algorithms with SSWS tend to ignore this method goal since the gain is tiny. We find that this scenario is common in big classes containing short and branch-less methods.

Answer to RQ4

Smart selection without the subsumption strategy outperforms slightly smart selection in most criteria on WS (except for WM and TMC). Smart selection outperforms slightly smart selection without the subsumption strategy in WM and TMC on three algorithms.

5 DISCUSSION

5.1 Parameter Tuning.

Smart selection introduces a new parameter: *lineThreshold* (see Sec. 3.4). In handling line coverage, smart selection skips those basic blocks (BBs) with lines less than *lineThreshold*. The larger the value of this parameter, the more BBs we skip. Without considering the dead code, (direct) branch coverage fails to capture the following lines only when a certain line in a basic block exits abnormally. Previous work [41] shows that, on average, when 78% of branches are covered, test suites can only find 1.75 exceptions. It indicates that (direct) branch coverage can capture most properties of line coverage. Therefore, to minimize the impacts of line coverage goals on SBST, we prefer a larger *lineThreshold*. After statistics on the benchmarks used in DynaMOSA [39], we find that 50% of the BBs have less than 8 lines. Therefore, we set *lineThreshold* to 8.

5.2 Threats to Validity

The threat to external validity comes from the experimental subjects. We choose 158 Java classes from the benchmark of DynaMOSA [39]. [39] was published in 2018. Many classes have already become obsolete. Some projects even are no longer maintained [30]. To reduce this risk, we choose 242 classes at random from Hadoop [1], thereby increasing the diversity of the dataset. The threat to internal validity comes from the randomness of the genetic algorithms. To reduce the risk, we repeat each approach 30 times for every class.

6 RELATED WORK

In this section, we introduce related studies on (1) SBST and (2) coverage criteria combination in SBST.

SBST. SBST formulates test cases generation as an optimization problem. Miller et al. [34] proposed the first SBST technique to generate test data for functions with inputs of float type. SBST techniques have been widely used in various objects under test [3, 10, 13, 14, 17, 23, 32, 51], and types of software testing [29, 45, 50]. Most researchers focus on (1) search algorithms: Tonella [47] proposed to iterate to generate one test case for each branch. Fraser et al. [15] proposed to generate a test suite for all branches. Panichella et al. [38, 39] introduced many-objective optimization algorithms. Grano et al. [21] proposed a variant of DynaMOSA [39] to reduce the computation costs; (2) fitness gradients recovery: Lin et al. [31] proposed an approach to address the inter-procedural flag problem. Lin et al. [30] proposed a test seed synthesis approach to create complex test inputs. Arcuri et al. [4] integrated testability transformations into API tests. Braione et al. [6] combined symbolic execution and SBST for programs with complex inputs; (3) readability of generated tests: Daka et al. [11] proposed to assign names for tests by summarizing covered coverage goals. Roy et al. [42] introduced deep learning approaches to generate test names; (4) fitness function design: Xu et al. [52] proposed an adaptive fitness function for improving SBST. Rojas et al. [41] proposed to combine multiple criteria to satisfy users' requirements.

Coverage Criteria Combination in SBST. Rojas et al. [41] proposed to combine multiple criteria to guide SBST. Gregory Gay [18] experimented with different combinations of coverage criteria. His experiment compares the effectiveness of multi-criteria suites in detecting complex, real-world faults. Omur et al. [43] introduced the Artificial Bee Colony algorithm as a substitute for the genetic algorithms used in WS [15]. Our work aims to increase the coverage decrease caused by combing multiple criteria [41] and is orthogonal to the latter two studies [18, 43].

7 CONCLUSION

We propose smart selection to address the coverage decrease caused by combining multiple criteria in SBST. We compare smart selection with the original combination on 400 Java classes. The experiment results confirm that with WS and MOSA, smart selection outperforms the original combination, especially for the Java classes with no less than 200 branches. But with DynaMOSA, the differences between smart selection and the original combination are slight.

REFERENCES

[1] Apache Software Foundation. 2006-. *Hadoop.* https://hadoop.apache.org
[2] Apache Software Foundation. 2006-. *Hadoop's compare method.* https://tinyurl.com/mrxvxusy
[3] Andrea Arcuri. 2018. Evomaster: Evolutionary multi-context automated system test generation. In *Proceedings of ICST*. 394–397.
[4] Andrea Arcuri and Juan P Galeotti. 2021. Enhancing search-based testing with testability transformations for existing APIs. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 1 (2021), 1–34.
[5] Andrea Arcuri, Muhammad Zohaib Iqbal, and Lionel Briand. 2011. Random testing: Theoretical results and practical implications. *IEEE transactions on Software Engineering* 38, 2 (2011), 258–277.
[6] Pietro Braione, Giovanni Denaro, Andrea Mattavelli, and Mauro Pezzè. 2017. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proceedings of ISSTA*. 90–101.
[7] Dimo Brockhoff, Tobias Friedrich, Nils Hebbinghaus, Christian Klein, Frank Neumann, and Eckart Zitzler. 2007. Do additional objectives make a problem harder?. In *Proceedings of GECCO*. 765–772.
[8] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology* 104 (2018), 207–235.
[9] José Campos, Annibale Panichella, and Gordon Fraser. 2019. EvoSuiTE at the SBST 2019 tool competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*. 29–32.
[10] Jeroen Castelein, Maurício Aniche, Mozhan Soltani, Annibale Panichella, and Arie van Deursen. 2018. Search-based test data generation for SQL queries. In *Proceedings of ICSE*. 1220–1230.
[11] Ermira Daka, José Miguel Rojas, and Gordon Fraser. 2017. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2?. In *Proceedings of ISSTA*. 57–67.

[12] Kalyanmoy Deb. 2014. *Multi-objective optimization*. 403–449.

[13] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. 2020. Time-travel testing of android apps. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. 481–492.

[14] Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proceedings of ESEC/FSE*. 416–419.

[15] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

[16] Gordon Fraser and Andrea Arcuri. 2015. Achieving Scalable Mutation-Based Generation of Whole Test Suites. *Empirical Softw. Engg.* 20, 3 (2015), 783–812.

[17] Alessio Gambi, Marc Mueller, and Gordon Fraser. 2019. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of ISSTA*. 318–328.

[18] Gregory Gay. 2017. Generating Effective Test Suites by Combining Coverage Criteria. In *Search Based Software Engineering*. 65–82.

[19] Rohit Gheyi, Márcio Ribeiro, Beatriz Souza, Marcio Guimarães, Leo Fernandes, Marcelo d'Amorim, Vander Alves, Leopoldo Teixeira, and Baldoino Fonseca. 2021. Identifying method-level mutation subsumption relations using Z3. *Information and Software Technology* 132 (2021), 106496.

[20] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. 2015. Guidelines for coverage-based comparisons of non-adequate test suites. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 4 (2015), 1–33.

[21] Giovanni Grano, Christoph Laaber, Annibale Panichella, and Sebastiano Panichella. 2019. Testing with fewer resources: An adaptive approach to performance-aware test case generation. *IEEE Transactions on Software Engineering* 47, 11 (2019), 2332–2347.

[22] Marcio Augusto Guimarães, Leo Fernandes, Márcio Ribeiro, Marcelo d'Amorim, and Rohit Gheyi. 2020. Optimizing mutation testing by discovering dynamic mutant subsumption relations. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. 198–208.

[23] Fitash Ul Haq, Donghwan Shin, Lionel C Briand, Thomas Stifter, and Jun Wang. 2021. Automatic test suite generation for key-points detection DNNs using many-objective search (experience paper). In *Proceedings of ISSTA*. 91–102.

[24] Yue Jia and Mark Harman. 2008. Constructing Subtle Faults Using Higher Order Mutation Testing. *Proceedings of SCAM*, 249 – 258.

[25] Yue Jia and Mark Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (2011), 649–678.

[26] René Just, Gregory M Kapfhammer, and Franz Schweiggert. 2012. Do redundant mutants affect the effectiveness and efficiency of mutation analysis?. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*. 720–725.

[27] Maurice G Kendall. 1938. A new measure of rank correlation. *Biometrika* 30, 1/2 (1938), 81–93.

[28] Joshua D Knowles, Richard A Watson, and David W Corne. 2001. Reducing local optima in single-objective problems by multi-objectivization. In *International conference on evolutionary multi-criterion optimization*. 269–283.

[29] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007), 225–237.

[30] Yun Lin, You Sheng Ong, Jun Sun, Gordon Fraser, and Jin Song Dong. 2021. Graph-Based Seed Object Synthesis for Search-Based Unit Testing. In *Proceedings of ESEC/FSE*. 1068–1080.

[31] Yun Lin, Jun Sun, Gordon Fraser, Ziheng Xiu, Ting Liu, and Jin Song Dong. 2020. Recovering fitness gradients for interprocedural Boolean flags in search-based testing. In *Proceedings of ISSTA*. 440–451.

[32] Alberto Martin-Lopez, Sergio Segura, and Antonio Ruiz-Cortés. 2021. RESTest: automated black-box testing of RESTful web APIs. In *Proceedings of ISSTA*. 682–685.

[33] Phil McMinn. 2004. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

[34] Webb Miller and David L. Spooner. 1976. Automatic generation of floating-point test data. *IEEE Transactions on Software Engineering* 3 (1976), 223–226.

[35] Elfurjani S Mresa and Leonardo Bottaci. 1999. Efficiency of mutation operators and selective mutation strategies: An empirical study. *Software Testing, Verification and Reliability* 9, 4 (1999), 205–232.

[36] A Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H Untch, and Christian Zapf. 1996. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5, 2 (1996), 99–118.

[37] Annibale Panichella, José Campos, and Gordon Fraser. 2020. EvoSuite at the SBST 2020 Tool Competition. In *Proceedings of ICSEW*. 549–552.

[38] 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)*. 1–10.

[39] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Transactions on Software Engineering* 44 (2018), 122–158.

[40] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 910–921.

[41] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *Search-Based Software Engineering*, Márcio Barros and Yvan Labiche (Eds.). 93–108.

[42] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. 2020. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proceedings of ASE*. 287–298.

[43] Omur Sahin, Bahriye Akay, and Dervis Karaboga. 2021. Archive-based multi-criteria Artificial Bee Colony algorithm for whole test suite generation. *Engineering Science and Technology* 24, 3 (2021), 806–817.

[44] Aravind Seshadri. 2006. A fast elitist multiobjective genetic algorithm: NSGA-II. *MATLAB Central* 182 (2006).

[45] Rodolfo Adamshuk Silva, Simone do Rocio Senger de Souza, and Paulo Sérgio Lopes de Souza. 2017. A systematic review on search based mutation testing. *Information and Software Technology* 81 (2017), 19–35.

[46] Beatriz Souza. 2020. Identifying mutation subsumption relations. In *Proceedings of ASE*. 1388–1390.

[47] Paolo Tonella. 2004. Evolutionary Testing of Classes. In *Proceedings of ISSTA*. 119–128.

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

[49] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. 2021. EVOSUITE at the SBST 2021 Tool Competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. 28–29.

[50] Kristen R Walcott, Mary Lou Soffa, Gregory M Kapfhammer, and Robert S Roos. 2006. Time-Aware Test Suite Prioritization. In *Proceedings of ISSTA*. 1–12.

[51] Song Wang, Nishtha Shrestha, Abarna Kucheri Subburaman, Junjie Wang, Moshi Wei, and Nachiappan Nagappan. 2021. Automatic Unit Test Generation for Machine Learning Libraries: How Far Are We?. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 1548–1560.

[52] Xiong Xu, Ziming Zhu, and Li Jiao. 2017. An adaptive fitness function based on branch hardness for search based testing. In *Proceedings of GECCO*. 1335–1342.