Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation

Panichella, Annibale; Kifetew, Fitsum Meshesha; Tonella, Paolo

DOI
10.1007/978-3-319-99241-9_17

Publication date
2018

Document Version
Accepted author manuscript

Published in
Search-Baed Software Engineering - 10th International Symposium, SSBSE 2018 - Proceedings

Citation (APA)
Panichella, A., Kifetew, F. M., & Tonella, P. (2018). Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation. In T. E. Colanzi, & P. McMinn (Eds.), *Search-Baed Software Engineering - 10th International Symposium, SSBSE 2018 - Proceedings* (pp. 309-324). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 11036 LNCS). Springer. https://doi.org/10.1007/978-3-319-99241-9_17

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Incremental Control Dependency Frontier Exploration for Many-Criteria Test Case Generation

Annibale Panichella[1], Fitsum Meshesha Kifetew[2], and Paolo Tonella[3]

[1] Delft University of Technology
[2] Fondazione Bruno Kessler, Trento, Italy
[3] Università della Svizzera Italiana(USI), Switzerland
a.panichella@tudelft.nl, kifetew@fbk.eu, paolo.tonella@gmail.com

Abstract. Several criteria have been proposed over the years for measuring test suite adequacy. Each criterion can be converted into a specific objective function to optimize with search-based techniques in an attempt to generate test suites achieving the highest possible coverage for that criterion. Recent work has tried to optimize for multiple-criteria at once by constructing a single objective function obtained as a weighted sum of the objective functions of the respective criteria. However, this solution suffers the problem of sum scalarization, i.e., differences along the various dimensions being optimized get lost when such dimensions are projected into a single value. Recent advances in SBST formulated coverage as a many-objective optimization problem rather than applying sum scalarization. Starting from this formulation, in this work, we apply many-objective test generation that handles multiple adequacy criteria simultaneously. To scale the approach to the big number of objectives to be optimized at the same time, we adopt an incremental strategy, where only coverage targets in the control dependency frontier are considered until the frontier is expanded by covering a previously uncovered target.

1 Introduction

Various coverage criteria, such as branch or mutation coverage, have been proposed to measure how thoroughly a given test suite exercises the program under test. Correspondingly, automated test case generation has focused on the achievement of these criteria as the objectives of the generation process. While these criteria have been tackled mostly independently from each other, a recent work by Rojas et al. [19] has proposed a test generation approach that optimizes for multiple criteria simultaneously. With this strategy, the individual fitness functions of each criterion are aggregated via weighted sum and optimized using single-objective search algorithms [19]. The resulting test suites are able to detect more faults compared to those generated with a single criterion [11].

While the aforementioned studies showed the benefits of focusing on multiple criteria, the weighted sum suffers from well-known theoretical limitations [7]: (i) it is not able to find optimal solutions for *non-convex* problems; (ii) small changes

in the weights may lead to completely different solutions; (iii) differences along the various criteria being optimized get lost when they are projected into a single value. Moreover, the weighted sum is based on the assumption that the criteria being summed-up are independent of each other. However, this assumption is not applicable in coverage testing, because of the *subsumption* relationships between coverage targets, due to the control dependencies in the program under test. For example, to cover the lines of code in a basic block, the conditional branch leading to it must be covered first. In turn, the branch condition could be nested inside another conditional statement that controls its execution.

Recently, Panichella et al. [16, 17] applied many-objective algorithms to handle single coverage criterion in which each criterion is handled as a different objective to optimize in a many-objective fashion. To cope with a possibly large number of objectives, coverage targets are selected dynamically by the proposed algorithm, `DynaMOSA` [17]. This incremental/dynamic search helps achieve higher coverage than sum scalarization when focusing on a single criterion [17, 4].

In this paper, we extend the idea of many-objective dynamic test generation to multiple heterogeneous criteria being optimized simultaneously. First, we define an *enhanced control dependency graph* (ECDG), a variant of the classical *control dependency graph* (CDG)[4] enriched with the structural dependencies among coverage targets coming from different coverage criteria (e.g., lines of code, mutants, etc.). Second, we introduce a search algorithm, which we named `MC-DynaMOSA`, that performs incremental exploration of the control dependency frontier to achieve multiple criteria coverage. In particular, coverage targets are incrementally selected during the search according to their position in the ECDG, where the covered frontier expands over time. The results of our empirical study show that the incremental exploration implemented in `MC-DynaMOSA` is more effective than (i) using the weighted sum with archiving strategy (`MC-WSA`), and (ii) handling all coverage criteria as fully independent objectives (`MC-MOSA`). Effectiveness is measured as the ability of the generated test suites in both (i) achieving higher coverage scores for seven testing criteria and (ii) detecting more faults. Furthermore, our results confirm that combining multiple criteria leads to test suites with superior fault revealing capability.

2 Background and related work

Several criteria have been proposed for structural coverage over the years. In this work, we focus on branch, line, method, weak mutation, input, output, and exception coverage [19]. In the context of Search-based Software Testing (SBST), each of these coverage criteria is associated with a fitness function that is used to guide the test generation process towards test cases that achieve the maximum possible coverage for that particular criterion.

[4] A control dependency edge between two nodes holds *iff* the latter is not a post-dominator of the former, while it is a post-dominator of all intermediate nodes between the two.

Branch coverage (BC): is the most widely adopted coverage criterion. The fitness function of a test case t with respect to a branch b is computed by considering the sum of the *approach level (al)* and the normalized *branch distance (bd)* [15]: $f(t,b) = al(t,b) + norm(bd(t,b))$, where $norm$ is a function that normalizes values into the range $[0,1]$.

Line coverage (LC): is the simplest and most straightforward coverage criterion, which measures coverage of non-comment lines of code in the System Under Test (SUT). The associated fitness function is computed by minimizing the distance to the closest branch on which the line is control dependent.

Weak mutation coverage (WMC): is a coverage criterion based on mutation where a mutant is considered *weakly killed* if for a given test case t, the execution of t on the mutant results in a different internal state than the original program. Differently, from strong mutation coverage, the internal state difference (aka *infection* [21]) may not necessarily propagate to any externally visible difference (e.g., to a return value). Given a mutant μ and a test t, the fitness function for calculating WMC is defined based on a heuristic *infection distance (id)* as follows: $f(t,\mu) = al(t,\mu) + norm(bd(t,\mu)) + norm(id(t,\mu))$, where approach level and branch distance refer to the branch which holds a control dependency on μ, while $id(t,\mu)$ estimates the distance to infecting the mutant state. If the mutation is executed, the minimal state infection distance depends on the mutation operator that was applied and is estimated as the numerical distance from a value that would make the states of mutant and original program differ. If the mutation is not executed, the normalized infection distance is equal to 1 [20].

Input coverage (IC): captures the diversity in the inputs to the SUT used by the test cases. It measures how spread the values are in the SUT input space.

Output coverage (OC): captures the diversity of the values output by methods in the SUT. Ultimately, it measures the uniqueness of the output values produced as a result of executing a test on the SUT.

Exception coverage (EC): measures the number of exceptions triggered by the execution of a test. The more exceptions a test triggers, the higher EC.

Method coverage (MC): requires that every method of the SUT be called, either directly or indirectly, by at least one test case.

When used as components of SBST, not all fitness functions of the various criteria mentioned above provide the same degree of guidance to the search. In fact, in our experience, IC, OC, EC, and MC contribute little or no guidance to the search during test generation. On the other hand, criteria such as BC, LC, and WMC provide strong guidance to the search. The reason for such stronger guidance is that all the mentioned criteria are, directly or indirectly, associated with some underlying branches that must be necessarily covered because they hold a control dependency on the coverage targets.

Multiple criteria coverage. The first attempt to combine multiple coverage criteria was proposed by Rojas et al. [19]. The authors aggregated the various coverage criteria using a weighted sum with uniform weights (equal to 1), and have left further investigation of different weight assignments to future work. The approach was implemented in EvoSuite [10], and experimental results on sub-

jects sampled from the SF110 corpus [10] showed that adding a second criterion besides line coverage resulted in 14% increase in test suite size and 20% increase in coverage. On the other hand, using all coverage criteria increased test suite size by 70%, while the coverage of the individual criteria was reduced on average by just 0.4%. Overall, the work provides encouraging evidence that combining multiple coverage criteria during test generation is feasible and beneficial.

After this initial work by Rojas et al., recent work explored, beyond feasibility, fine-grained analysis of combinations of multiple criteria. In particular, Gay [11] explored the ability of test suites, generated via multiple criteria, of exposing known, real-world faults, considering the Defects4J benchmark [12]. Results show that combining multiple criteria could improve fault detection up to 31%.

A recent trend in automated test case generation consists in recasting it as a many-objective optimization problem [2, 16, 17]. However, none of the existing work on multi-criteria coverage [19, 11] takes advantage of the recent, advanced many-objective test generation algorithms. In fact, they aggregate all fitness functions associated with multiple criteria into a single fitness function by means of sum scalarization [6]. This paper presents the first attempt to apply many-objective test generation to multiple coverage criteria, rather than just to the multiple targets that can be found for a single criterion.

Recently, Panichella et al. [16] proposed `MOSA` (Many-Objective Sorting Algorithm), a many-objective genetic algorithm that considers each coverage target as an independent objective to be optimized. It employs a specialized preference criterion to favor promising individuals in the search. Such a preference criterion helps `MOSA` focus the search on the most promising individuals, whereas traditional dominance-based ranking would have resulted in a larger number of non-dominated individuals, which are not necessarily useful for covering new targets. Empirical results show that `MOSA` is indeed superior to state-of-the-art single objective approaches [16]. `MOSA` was later improved by its successor `DynaMOSA` [17] with the objective of increasing the efficiency of the test generation process. Indeed, in the presence of a high number of coverage targets, `MOSA` could suffer from the algorithmic overhead for computing the Pareto fronts. `DynaMOSA` introduces a smarter approach for dealing with this issue, by dynamically adding new targets to be covered each time a previously uncovered target is reached. `DynaMOSA` starts with branches that represent method entries, and every time a branch is covered, all targets dependent on the covered branch are added to the set of targets to be covered.

A recent study by Campos et al. [4] empirically explored the performance of various test generation algorithms. They compared variants of traditional `Evolutionary Algorithms` (EAs), `MOSA`, `DynaMOSA`, and `Random Search` in terms of various coverage metrics. Results showed that EAs, supported by test archives, perform better than random search. Furthermore, many-objective algorithms (`MOSA`, `DynaMOSA`) achieve superior performance on branch coverage.

This paper shares with `DynaMOSA` [17, 4] the idea of dynamically updating the coverage targets to be addressed by many-objective optimization. However, `DynaMOSA` cannot be applied directly to multiple, heterogeneous criteria. In this

paper, we extend the idea of dynamic target update to take into account targets of heterogeneous nature (mutants, branches, diversity, etc.). Our intuition is that the benefits brought by considering multiple coverage targets at the same time could be even larger when not only multiple targets but also multiple criteria, which in turn include multiple targets, are considered at the same time.

3 Approach

Our approach relies on control dependency analysis and branch/dependency coverage as the guiding criterion, and exploits this guidance to effectively explore the search space with respect to all the other criteria. Moreover, our approach optimizes for multiple criteria via many-objective optimization, rather than summing up several different fitness functions into a single-objective function.

Problem formulation. Given a SUT, the multiple criteria test generation problem can be formulated as follows: *Let* $B = \alpha \cup \beta \cup \ldots \cup \omega$ *be the set of all coverage targets representing different adequacy criteria* $\alpha, \beta, \ldots, \omega$ *and corresponding fitness functions* $f_\alpha, f_\beta, \ldots, f_\omega$. *Find a set of test cases* $T = \{t_1, \ldots, t_n\}$ *that minimize the fitness functions for all targets* $\tau_i \in B$. This formulation gives rise to the many-objective optimization problem for minimizing the following $k_\alpha + k_\beta + \ldots + k_\omega$ objectives:

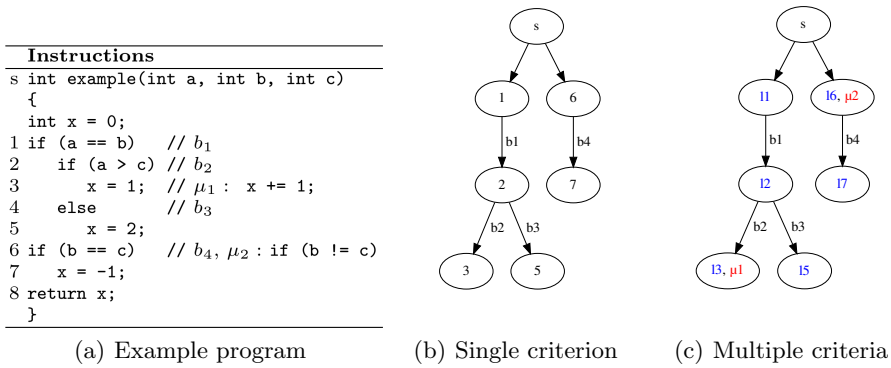
$$\begin{cases} \min O_{\alpha,1}(t) = f_\alpha(\tau_{\alpha,1}, t), \ldots, & \min O_{\alpha,k_\alpha}(t) = f_\alpha(\tau_{\alpha,k_\alpha}, t) \\ \min O_{\beta,1}(t) = f_\beta(\tau_{\beta,1}, t), \ldots, & \min O_{\beta,k_\beta}(t) = f_\beta(\tau_{\beta,k_\beta}, t) \\ \vdots & \\ \min O_{\omega,1}(t) = f_\omega(\tau_{\omega,1}, t), \ldots, & \min O_{\omega,k_\omega}(t) = f_\omega(\tau_{\omega,k_\omega}, t) \end{cases} \tag{1}$$

where $f_\alpha, f_\beta, \ldots, f_\omega$ represent the fitness functions of adequacy criteria $\alpha, \beta, \ldots, \omega$.

Example. To explain our approach, we present a simple example whose code is shown in Figure 1(a). In the example, three types of coverage targets are indicated: (i) branches $\alpha = \{b_1, b_2, b_3, b_4\}$, (ii) lines $\beta = \{l1, \ldots, l8\}$, and (iii) mutants $\gamma = \{\mu_1, \mu_2\}$. The final set of coverage targets would be: $B = \alpha \cup \beta \cup \gamma = \{l_1, \ldots, l_8, b_1, b_2, b_3, b_4, \mu_1, \mu_2\}$, and the problem consists of finding a set of test cases that achieve full coverage of all targets in B. The control dependency graph (CDG) of the example program is shown in Figure 1(b). We can see from the CDG that the branches in the sample program are interdependent, with some branches being control dependent on others. For example, branch b_2 can only be executed after branch b_1 has been executed.

3.1 `MC-DynaMOSA`: Many-Criteria Dynamic Many-objective Optimization with Incremental Frontier Exploration

Our approach, `MC-DynaMOSA` (Many-Criteria Dynamic Many-objective Sorting Algorithm) hereafter, optimizes for multiple criteria simultaneously by representing the various coverage criteria into an *Enhanced CDG* (ECDG), in such a



(a) Example program (b) Single criterion (c) Multiple criteria

Fig. 1. Code (left), CDG (middle), and ECDG (right) of an example program

Algorithm 1: MC-DynaMOSA

Input:
$B = \{\tau_1, \ldots, \tau_m\}$ the set of coverage targets of a program.
$CDG = \langle N, E, s \rangle$: control dependency graph of the program
Result: A test suite T

1 **begin**
2 $\phi \longleftarrow$ EXTEND-CDG(CDG, B)
3 $B^* \longleftarrow$ ENTRY-POINTS (CDG, ϕ, B) //targets without control dependencies
4 $P_t \longleftarrow$ RANDOM-POPULATION(M) // followed by fitness evaluation
5 archive $\longleftarrow$ UPDATE-ARCHIVE(P_t, $\emptyset$) // collect tests covering new targets
6 $B^* \longleftarrow$UPDATE-TARGETS(B^*, CDG, ϕ)
7 **while** *not (search_budget_consumed) AND* ($B^* \neq \emptyset$) **do**
8 $P_{t+1} \longleftarrow$ EVOLVE(P_t) // crossover, mutation, evaluation, selection
9 archive $\longleftarrow$ UPDATE-ARCHIVE(P_{t+1}, archive)
10 $B^* \longleftarrow$UPDATE-TARGETS(B^*, CDG, ϕ)
11 $T \longleftarrow$ archive

way that the control dependency based frontier exploration can be performed on multiple criteria using the ECDG as guidance. The high-level algorithm used to build the ECDG is outlined in Algorithm 2: the original CDG is enriched with the coverage targets that are control dependent on each branch. For example, the ECDG for the program in Figure 1(a) is depicted in Figure 1(c): the coverage targets related to line, branch, and weak mutation are all represented in the ECDG, as either node or edge labels, because the associated fitness functions can be computed only when executing the frontier node whose outgoing control dependency edge leads to the target. For method, input and output coverage, the corresponding targets are associated with the root branch of the ECDG: if the root branch is covered it implies that the method has been called/covered and therefore it is possible to measure the diversity of both its input and output.

The high-level algorithm of `MC-DynaMOSA` is shown in Algorithm 1. As outlined in Algorithm 1, `MC-DynaMOSA` starts (line 2) by building the enhanced CDG using Algorithm 2, which essentially extends the CDG by attaching coverage targets to the edges of the CDG which hold a control dependency over the targets. Once the ECDG is built (i.e., function ϕ has been determined), `MC-DynaMOSA` computes the initial set of coverage targets from the ECDG by collecting the

Algorithm 2: EXTEND-CDG

```
Input:
G = ⟨N, E, s⟩: control dependency graph of the program
B: set of all coverage targets
Result:
φ : E → P(B): partial map between edges and targets
1 begin
2     ∀e ∈ E : φ(e) ⟵ ∅
3     for τ ∈ B do
4         e ⟵ getImmediateControlDependency(τ)
5         φ(e) ⟵ φ(e) ∪ {τ}
```

Algorithm 3: UPDATE-TARGETS

```
Input:
CDG = ⟨N, E, s⟩: control dependency graph
B* ⊆ B: current set of targets
φ : E → B: partial map between edges and targets
Result:
B*: updated set of current targets
1 begin
2     for τ ∈ B* do
3         if τ is covered then
4             B* ⟵ B* − {τ}
5             e_τ ⟵ φ^{-1}(τ)
6             B* ⟵ADD-NEXT-TARGETS(B*, e_τ)
```

Algorithm 4: ADD-NEXT-TARGETS

```
Input:
CDG = ⟨N, E, s⟩: control dependency graph
B* ⊆ B: current set of targets
Result:
B*: updated set of current targets
1 begin
2     for each e_n ∈ E immediately following e in CDG do
3         for each τ ∈ φ(e_n) do
4             if τ is not covered then
5                 B* ⟵ B* ∪ {τ}
6     return B*
```

targets in the initial frontier, i.e., those targets which are not under any control dependency (line 3). It then generates the initial population of individuals (test cases) and evaluates them (line 4). Subsequently, it collects individuals that cover one or more previously uncovered targets (line 5). It then enters an evolutionary loop in which it evolves the individuals by applying genetic operators (crossover, mutation, fitness evaluation, and selection), resulting in the next generation of individuals (line 8). It then collects individuals covering new targets (line 9), and finally updates the current set of targets by removing those covered and adding the targets which have the covered targets as their control dependencies (line 10). This process continues until either the search budget is finished or all targets are covered. Finally, the archive of test cases collected throughout the process is returned as the final solution, i.e., the test suite (line 11).

We now illustrate the `MC-DynaMOSA` algorithm on the example in Figure 1(a) whose ECDG is shown in Figure 1(c). A possible execution trace of `MC-DynaMOSA` is shown in Table 1. In the beginning, the set of current targets are those with no control dependency (second row in Table 1). When target $b1$ is covered (which also means $l1, l2$ are covered), the set of current targets is updated by removing $b1, l1, l2$ and adding other targets over which $b1$ holds a control dependency

Table 1. A simulation of `MC-DynaMOSA` on the example

	current targets	covered targets
init	{l1, l6, μ2, b1, b4}	{}
b1 covered	{b2, b3, μ2, b4}	{l1, b1, l2, l6}
b4 covered	{b2, b3, μ2}	{l1, b1, l2, l6, b4, l7}
b2 covered	{b3, μ2, μ1}	{l1, b1, l2, l6, b4, l7, b2, l3}
b3 covered	{μ2, μ1}	{l1, b1, l2, l6, b4, l7, b2, l3, b3, l5}
μ2 covered	{μ1}	{l1, b1, l2, l6, b4, l7, b2, l3, b3, l5, μ2}

according to the ECDG, i.e., $b2, b3$ (third row in Table 1). Similarly, as new branches are covered, new targets are added to the set of current targets incrementally following the ECDG. Finally, `MC-DynaMOSA` may be able to cover all targets or may fail to cover some, as in this case where target $\mu 1$ remains uncovered at the end of the process.

4 Empirical Evaluation

To evaluate the performance of `MC-DynaMOSA`, we conducted an empirical study with a set of non-trivial Java classes selected from different open-source projects.

Research Questions. Our first RQ aims at assessing the benefits (if any) of using incremental many-objective search compared to the weighted-sum:

> **RQ1**: *What is the effectiveness of unified many-objective/multi-criteria coverage compared to the weighted-sum approach?*

Furthermore, we want to investigate how `MC-DynaMOSA` performs compared with a simple many-objective search that considers all coverage targets related to different coverage criteria as independent search objectives:

> **RQ2**: *What are the benefits of incremental control dependency frontier exploration?*

A key observation in the previous section is that existing coverage criteria can be directly related to branch-coverage (via ECDG) and that control dependency branches can be used to guide multi-criteria coverage. Hence, the next question is whether combining multiple criteria and performing an incremental many-objective search provides any benefits over optimizing just branch coverage:

> **RQ3**: *Is it enough to target all branches in order to achieve high coverage of all the other criteria?*

Moreover, we question whether multi-criteria coverage is associated with higher fault detection than branch coverage alone:

> **RQ4**: *What is the fault detection capability of the final test suites obtained by many-criteria coverage vs branch coverage only?*

In the following, we refer to `MC-WSA` (Multi-criteria Weighted Sum with Archives) and `MC-MOSA` (Multi-criteria Many-Objective) as the baselines for RQ1 and RQ2, respectively. For RQ3, we refer to `SC-DynaMOSA` (Single-Criteria Many-Objective) for the many-objective algorithm (`DynaMOSA`) that optimizes branch coverage alone.

Benchmark. The benchmark of our study is a set of 180 non-trivial Java classes randomly sampled from the SF110 dataset [10], which contains 110 open-source projects from the `SourceForge.net` repository. This dataset has been used in recent studies [20, 9, 10, 17] to assess both the efficiency and the effectiveness of test case generation tools.

To form our benchmark, we applied the same selection procedure used in prior studies [17, 18], which first measures the McCabe's cyclomatic complexity [14]

(CC) to avoid sampling trivial classes. Specifically, we first removed classes from SF110 containing exclusively methods with a CC lower than five [17]. Then, we randomly sampled 180 classes from the resulting pruned SF110 dataset: two classes from the largest projects in SF110 and one class from the remaining projects. The number of coverage targets ranges between 61 (for class `SapdbTableList` from project `db-everywhere`) and 4,252 (for class `JVCParserTokenManager` from project `javaviewcontrol`); the median number of coverage targets per class is 405. These numbers include all coverage targets from the seven coverage criteria described in Section 2, namely *branch*, *line*, *weak mutation*, *input*, *output*, *method* and *exception coverage*. The complete list of classes under test (CUTs) in our benchmark is publicly available in `FigShare` at the following link: `https://figshare.com/s/c74652d1fcb79fa853dd`.

Implementation. The four test generation strategies —i.e., `MC-DynaMOSA`, `MC-MOSA`, `MC-WSA`, `SC-DynaMOSA`— were implemented in EvoSuite [9, 8]. `MC-WSA` corresponds to the default strategy in EvoSuite, which evolves test suites using a monotonic genetic algorithm [4] guided by one single fitness function that combines all coverage criteria using a weighted sum. `MC-MOSA` corresponds to the `MOSA` algorithm [16], which evolves test cases rather than test suites, targeting all coverage criteria simultaneously. Each coverage criterion corresponds to a different set of search objectives, one objective for each coverage target. Therefore, the set of objectives in `MC-MOSA` is the union of the sets of objectives from all seven coverage criteria. `SC-DynaMOSA` considers only branch coverage as testing criterion, but it dynamically updates the set of objectives based on the structural dependencies among branches, i.e., it corresponds to original `DynaMOSA` [17]. Finally, we implemented `MC-DynaMOSA` in EvoSuite as described in Section 3.1. The implementation is publicly available from `FigShare` at the following link: `https://figshare.com/s/ecdef1f88cb00f1ad31f`.

All testing strategies are implemented in the same version of EvoSuite, downloaded from GitHub on October 1st, 2017. Furthermore, all strategies use an *archive* [17, 20], to take accidental coverage into account: whenever a test case (or test suite) T satisfies a previously uncovered target, T is stored in the *archive* while the target is removed from the set of objectives [1] or from computation of the weighted sum [20, 4]. Therefore, in all testing strategies, the search is focused on the uncovered targets only.

Methodology. For each CUT in our benchmark, we ran each testing strategy 30 times and collected the number and the type of targets covered in each run. This setting led to 30 EvoSuite runs $\times$ 4 strategies $\times$ 180 CUTs = 21,600 executions in total. In each run, we measured the percentage of covered targets for each coverage criterion as: $Cov(C,T) = \frac{\#Covered(C,T)}{\#Total(C)}$ where $\#Total$ denotes the total number of targets for a given criterion C, while $\#Covered(C,T)$ is the number of targets covered by the generated test suite T. Coverage scores are computed after EvoSuite's post-processing, which minimizes the generated suite T and adds candidate assertions using a mutation-based strategy [10].

Then, we compare each pair of testing strategies by considering the average (arithmetic mean) of each coverage criterion over 30 independent repetitions.

Differences (if any) are shown and discussed in terms *percentage points* (*pp*), i.e., the absolute difference between the coverage scores of the test suites generated by the two testing strategies being compared. To assess the statistical significance of such differences, we applied the Wilcoxon rank sum test [5] for each CUTs and for each pair of testing strategies, adopting a significance level $\alpha = 0.05$. The obtained p-values are then adjusted with the Holm-Bonferroni procedure [5] as required when comparing more than two treatments.

We also wanted to assess the ability of the generated test suites to detect faults. To this aim, we applied strong mutation coverage as a proxy for the actual fault detection capability. This is a common practice when assessing software testing tools and approaches since previous studies [13] showed that mutants can be regarded as valid substitutes of real-world faults to assess fault detection rates. In this study, we used the mutation testing engine available in EvoSuite. Strong mutation coverage is computed as the percentage of mutants that are strongly killed by a generated test suite, i.e., the test suite contains a test case that fails when comparing the output of the mutant to the output of the original program. Note that strong mutation is not used in this study as guidance to the search (i.e., as part of a fitness function or of some objectives), so it was possible to use it to assess the fault-detection capability of the suite generated by each testing strategy. It is also worth to remark that strong mutation has never been used in previous studies in combination with other coverage criteria due to its large overhead [4, 11, 19].

Parameter Setting. Previous studies have shown that default parameters provide acceptable results compared to fine-tuning of the evolutionary parameters [3]. Hence, we adopted default parameter values in EvoSuite [20], as done in previous studies targeting the SF110 dataset [20, 17, 19]: for all testing strategies, (single and many-objective), genetic algorithms are configured with a population size of 50 test cases/suites; *single-point crossover* with probability p_c=0.75; *mutation* with probability $p_m = 1/n$, where n is the number of statements in a test case (or number of test cases in the test suite for `MC-WS`); the selection operator is *tournament selection*, the default in EvoSuite. For the search budget, we set the same maximum execution time of three minutes for all testing strategies. The search stopped earlier only when 100% coverage was obtained for all coverage criteria before reaching the search timeout.

5 Experimental Results

Due to the space limits, we report only the average results obtained by each strategy across all CUTs and the number of classes with a statistically significant difference. The detailed results for each class are available at the following link: `https://figshare.com/s/b06984aa36bfe2e9d934`.

Table 2 summarizes the results of the pairwise comparison for the three multi-criteria testing strategies, i.e., `MC-DynaMOSA`, `MC-WSA`, and `MC-MOSA`. For each strategy, the table reports (i) the mean coverage score obtained for each coverage criterion across all 180 CUTs and (ii) the number of classes in our

Table 2. Comparison between MC-DynaMOSA, MC-MOSA, and MC-WSA on all considered coverage criteria and on strong mutation

Cov. Criterion	Average Coverage			MC-DynaMOSA vs. MC-MOSA			MC-DynaMOSA vs. MC-WSA		
	MC-DynaMOSA	MC-MOSA	MC-WSA	#Better	#Worse	#No Diff.	#Better	#Worse	#No Diff.
Branch	**0.62**	0.60	0.59	71	5	104	82	9	72
Line	**0.67**	0.65	0.64	62	4	114	85	10	85
Weak Mutation	**0.64**	0.63	0.62	57	6	117	81	12	87
Method	**0.97**	0.96	0.96	16	4	160	12	6	162
Input	**0.95**	0.94	0.94	30	4	146	29	14	137
Output	**0.60**	0.59	0.58	39	4	137	39	9	132
Exception	**1.00**	0.99	0.99	13	0	167	16	0	164
Strong mutation	**0.29**	0.27	0.23	55	25	100	97	4	79

benchmark in which MC-DynaMOSA is statistically better, worse or equivalent to MC-WSA and MC-MOSA according to the Wilcoxon test. Finally, the last row of the table shows the results (average scores and number of significant data points) for strong mutation.

Results of MC-DynaMOSA vs. MC-WSA (RQ1). For branch coverage, MC-DynaMOSA achieved on average $+3pp$ across all CUTs in our benchmark. The former was statistically significantly better than the latter in 82 classes out of 180, while the latter statistically outperformed the former in only 9 classes. Similar results can be observed for *line coverage* and *weak mutation*: in 85 classes and in 82 classes out of 180, MC-DynaMOSA achieved statistically significantly higher line and weak mutation coverage than MC-WSA, respectively. For these cases, the largest difference of $27.94pp$ in line coverage is observed for class TableMeta (project schemaspy); the largest difference in weak mutation is equal to $33.21pp$ and was observed for class Shift (project jiggler). Only in 10 classes (for line coverage) and 12 classes (for weak mutation) out of 180, MC-WSA outperformed MC-DynaMOSA with an average difference of $3.61pp$ and $5.11pp$ for line and weak mutation, respectively. For the remaining criteria —i.e., method, input, output, and exception— MC-DynaMOSA still achieves higher coverage scores than MC-WSA. However, the number of CUTs with statistically significant difference decreases compared to the previously discussed criteria. In the very large majority of the classes, the two testing strategies turned out to be statistically equivalent. In 6%-21% of the classes, the winner of the comparison is MC-DynaMOSA, while in 3%-7% the winner is MC-WSA. Remarkably, in none of the 180 classes, the test suites generated by MC-WSA could trigger/cover more exceptions than the suites generated by MC-DynaMOSA. In summary, we observed a much larger number of significantly improved cases for branch, line and weak mutation coverage, as compared to the other criteria, when MC-DynaMOSA is used. A possible explanation for this finding is that branch, line and weak-mutation coverage provide stronger guidance, as their associated fitness functions are crafted based on carefully defined, fine-grained heuristics, i.e., approach level [15], branch distance [15] and infection distance [19].

In terms of strong mutation (last row in Table 2), MC-DynaMOSA detects a significantly larger number of faults (strongly killed mutants) than MC-WSA in 97

CUTs out of 180. The opposite is true in only 4 classes. MC-DynaMOSA improved the strong mutation score by $+6pp$ on average. The largest improvement ($+53pp$) is observed for class TableMeta from schemaspy (as for branch coverage).

The test suites generated by MC-DynaMOSA achieve higher coverage scores and are able to detect more faults than the suites produced by MC-WS.

Results of MC-DynaMOSA vs. MC-MOSA (RQ2). As indicated in Table 2, MC-DynaMOSA yielded on average $+2pp$ over MC-MOSA for branch and line coverage, and $+1pp$ for the remaining coverage criteria. In 71 CUTs out of 180, MC-DynaMOSA achieved a significantly higher branch coverage; the opposite is true in only 5 classes. The largest difference ($+23.68pp$) is achieved for class JMCAAnalyzer from project jmca. Instead, in the very few cases where MC-MOSA achieves significantly higher branch coverage, the difference ranges between $0.50pp$ (class ServerGameModel from hft-bomberman) and $6.49pp$ (class SimpleComboBox from caloriecount). The results for the other coverage criteria are in line with those observed for branch coverage. In 62 classes for line coverage and in 57 classes for weak mutation out of 180, MC-DynaMOSA outperformed MC-MOSA. The differences range between $0.10pp$ (class FBProcedureCall from firebird) and $24.15pp$ (class JMCAAnalyzer from jmca) for line coverage and between $0.10pp$ (class AntPathMatcher from jsecurity) and $32.36pp$ (the same class of line and branch coverage) for weak mutation. MC-MOSA achieved higher scores than MC-DynaMOSA in only 4 and 6 classes out of 180, for line coverage ($3.05pp$ on average) and weak mutation ($1.37pp$ on average), respectively. The number of CUTs with statistically significant difference decreases when analyzing method, input, output and exception coverage compared to the other three criteria. Nevertheless, there are many more CUTs where better coverage scores are obtained when running MC-DynaMOSA (6%-21% of the benchmark) than cases where the winner of the comparison is MC-MOSA (2%-3% of the benchmark). In none of the 180 classes, the test suites generated by MC-MOSA covered more exceptions than the tests generated with MC-DynaMOSA.

The values reported in the last row in Table 2 indicate that MC-DynaMOSA detected a significantly larger number of faults (strongly killed mutants) than MC-MOSA in 55 CUTs out of 180. On these cases, the average improvement in strong mutation score is $+8.79pp$, with the maximum of $+39.46pp$ for class HostMonitoringService (project quickserver). On the other hand, MC-MOSA achieved a better mutation score in 25 classes out of 180. However, in these cases the magnitude of the difference is small, being $4.76pp$ on average.

The incremental exploration of the control dependency frontier implemented in MC-DynaMOSA leads to larger coverage scores and to a better fault-detection capability than simply targeting all coverage targets as done by MC-MOSA.

Results of MC-DynaMOSA vs. SC-DynaMOSA (RQ3, RQ4). Table 3 summarizes the results of the comparison of many-objective search with an incremental exploration of the control dependency frontier when handling multiple

Table 3. Comparison between MC-DynaMOSA and SC-DynaMOSA in terms of coverage and strong mutation scores

Coverage Criterion	Average Coverage		MC-DynaMOSA vs. SC-DynaMOSA		
	MC-DynaMOSA	SC-DynaMOSA	#Better	#Worse	#No Diff.
Branch	0.62	**0.63**	36	53	119
Line	**0.67**	0.65	88	28	64
Weak Mutation	**0.64**	0.62	107	19	54
Method	**0.97**	0.90	89	2	89
Input	**0.95**	0.57	146	1	33
Output	**0.60**	0.46	115	5	60
Exception	**1.00**	0.45	137	0	43
Strong mutation	**0.29**	0.26	89	21	70

criteria (MC-DynaMOSA) compared to branch coverage only (SC-DynaMOSA). In 53 classes out of 180, SC-DynaMOSA achieved significantly higher branch coverage than MC-DynaMOSA; on the other hand, the latter outperformed the former in 36 classes. This finding clearly indicates that optimizing many coverage criteria at the same time may lead to lower coverage scores compared to the optimization of each criterion, taken separately from the others (as for branch coverage in this case). For example, branch coverage decreases by 1.65*pp* on average, with a minimum decrement of 1.00*pp* for class `jgaapGUI` (project `jgaap`) and a maximum one of 24.66*pp* for class `JMCAAnalyzer` (project `jmca`). On the CUTs where MC-DynaMOSA won the comparison, the differences range between 1.40*pp* (class `Profile` from project `jiprof`) and 24.71*pp* (class `JSJshop` from project `shop`), being 5.63*pp* on average. While we observe that targeting only branches leads to higher branch coverage in around 30% of CUTs, the results are quite different when looking at the other coverage criteria. For example, MC-DynaMOSA statistically outperforms SC-DynaMOSA in 88 CUTs and 107 CUTs for line coverage and weak mutation, respectively. This means that the additional branches covered by SC-DynaMOSA and not by MC-DynaMOSA are associated to basic blocks in the control flow graph with no statements (other than the branch itself) or with no (or very few) weakly killed mutants. Although branches represent the main backbone to build the multi-criteria control dependency graph (and to incrementally explore the frontier), branch coverage is not equivalent to the other criteria.

Even though MC-DynaMOSA may lead to lower branch coverage than SC-DynaMOSA, it achieves higher coverage on all other criteria. Therefore, it is not enough to target all branches in order to achieve high coverage of all the other criteria.

Despite leading to lower branch coverage, MC-DynaMOSA achieved a higher strong mutation score than SC-DynaMOSA in 89 CUTs out of 180. The increment in strong mutation score ranges between 1.11*pp* (class `ExportHook` from project `freemind`) and 35.80*pp* (class `QuickServerConfig` from project `quickserver`), being 9.62*pp* on average. On the other hand, SC-DynaMOSA outperformed MC-DynaMOSA in just 21 CUTs, with an average difference of only 2.93*pp*. This finding is particularly remarkable as it shows that a statistically higher branch coverage does not necessarily lead the generated test suites to reveal more faults.

Handling many criteria with `MC-DynaMOSA` *increases the fault detection capability of the generated test suites compared to targeting branch coverage alone.*

Threats to validity. *Construct validity.* All algorithms are implemented in the same tool, minimizing the risk of confounding factors. *Internal validity.* We did 30 independent runs and drew conclusions following statistical significance. We used default parameter values and those used in the respective algorithms. The comparison was based on metrics with respect to the considered criteria, and mutation scores. *External validity.* Enlarging the benchmark (beyond 180 CUTs) in future experiments could increase confidence of the results.

6 Conclusion

Coverage of multiple criteria has been the subject of recent research effort. While targeting multiple criteria simultaneously offers various advantages, it also poses difficulties to the search algorithm as the number of targets to be considered increases. In this paper, we have presented an approach, `MC-DynaMOSA`, based on incremental frontier exploration for multiple criteria test generation. In particular, we exploit inherent inter-dependencies among the various criteria to establish an enhanced control dependency graph, based on which we explore the coverage targets incrementally. Experimental results on 180 classes showed that `MC-DynaMOSA` outperforms the state-of-the-art approach for multiple criteria coverage, which is based on sum scalarization, in terms of coverage of the various criteria as well as strong mutation scores. Furthermore, results also showed that covering all branches is not sufficient to achieve higher coverage of the other criteria, even though control dependency branches provide the principal guidance to the search.

Acknowledgement

This work is partially supported by the Italian Ministry of Education, University, and Research (MIUR) with the PRIN project GAUSS (grant n. 2015KWREMX).

References

1. Abreu, R., Zoeteweij, P., Van Gemund, A.J.: An observation-based model for fault localization (2008)
2. Arcuri, A.: Many Independent Objective (MIO) Algorithm for Test Suite Generation, pp. 3–17. Springer International Publishing, Cham (2017)
3. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3), 594–623 (2013)
4. Campos, J., Ge, Y., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for test suite generation. In: International Symposium on Search Based Software Engineering. pp. 33–48. Springer (2017)

5. Conover, W.J.: Practical Nonparametric Statistics. Wiley, 3rd edition edn. (1998)
6. Deb, K.: Multi-objective optimization. In: Search Methodologies, pp. 403–449. Springer US (2014)
7. Fonseca, C.M., Fleming, P.J.: An overview of evolutionary algorithms in multiobjective optimization. Evolutionary computation 3(1), 1–16 (1995)
8. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering. pp. 416–419. ESEC/FSE '11 (2011)
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Software Eng. 39(2), 276–291 (2013)
10. Fraser, G., Arcuri, A.: A large-scale evaluation of automated unit test generation using evosuite. ACM Trans. Softw. Eng. Methodol. 24(2), 8:1–8:42 (2014), http://doi.acm.org/10.1145/2685612
11. Gay, G.: Generating effective test suites by combining coverage criteria. In: Search Based Software Engineering - 9th International Symposium, SSBSE 2017, Paderborn, Germany, September 9-11, 2017, Proceedings. pp. 65–82 (2017), https://doi.org/10.1007/978-3-319-66299-2_5
12. Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014. pp. 437–440 (2014)
13. Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: 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. pp. 654–665. FSE 2014, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2635868.2635929
14. McCabe, T.J.: A complexity measure. IEEE Transactions on software Engineering (4), 308–320 (1976)
15. McMinn, P.: Search-based software test data generation: a survey. Softw. Test. Verif. Reliab. 14(2), 105–156 (2004)
16. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: 8th IEEE International Conference on Software Testing, Verification and Validation, ICST. pp. 1–10 (2015)
17. Panichella, A., Kifetew, F.M., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans. Software Eng. 44(2), 122–158 (2018), https://doi.org/10.1109/TSE.2017.2663435
18. Panichella, A., Molina, U.R.: Java unit testing tool competition: fifth round. In: Proceedings of the 10th International Workshop on Search-Based Software Testing. pp. 32–38. IEEE Press (2017)
19. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining multiple coverage criteria in search-based unit test generation. In: Search-Based Software Engineering, pp. 93–108. Springer (2015)
20. Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering 22(2), 852–893 (2017), https://doi.org/10.1007/s10664-015-9424-2
21. Voas, J.M.: Pie: A dynamic failure-based technique. IEEE Trans. Softw. Eng. 18(8), 717–727 (Aug 1992), http://dx.doi.org/10.1109/32.153381