Generating Class-Level Integration Tests Using Call Site Information

Derakhshanfar, Pouria; Devroey, Xavier; Panichella, Annibale; Zaidman, Andy; van Deursen, Arie

Published in:
IEEE Transactions on Software Engineering

DOI:
10.1109/TSE.2022.3209625

Publication date:
2023

Document Version
Peer reviewed version

Link to publication

Citation for pulished version (HARVARD):
Derakhshanfar, P, Devroey, X, Panichella, A, Zaidman, A & van Deursen, A 2023, 'Generating Class-Level Integration Tests Using Call Site Information', *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2069-2087. https://doi.org/10.1109/TSE.2022.3209625

Generating Class-Level Integration Tests Using Call Site Information

Pouria Derakhshanfar, *Student Member, IEEE,* Xavier Devroey, *Member, IEEE,* Annibale Panichella, Andy Zaidman *Member, IEEE Computer Society*, and Arie van Deursen *Member, IEEE Computer Society*

Abstract—Search-based approaches have been used in the literature to automate the process of creating unit test cases. However, related work has shown that generated tests with high code coverage could be ineffective, i.e., they may not detect all faults or kill all injected mutants. In this paper, we propose CLING, an integration-level test case generation approach that exploits how a pair of classes, the caller and the callee, interact with each other through method calls. In particular, CLING generates integration-level test cases that maximize the Coupled Branches Criterion (CBC). Coupled branches are pairs of branches containing a branch of the caller and a branch of the callee such that an integration test that exercises the former also exercises the latter. CBC is a novel integration-level coverage criterion, measuring the degree to which a test suite exercises the interactions between a caller and its callee classes. We implemented CLING and evaluated the approach on 140 pairs of classes from five different open-source Java projects. Our results show that (1) CLING generates test suites with high CBC coverage, thanks to the definition of the test suite generation as a many-objectives problem where each couple of branches is an independent objective; (2) such generated suites trigger different class interactions and can kill on average 7.7% (with a maximum of 50%) of mutants that are not detected by tests generated randomly or at the unit level; (3) CLING can detect integration faults coming from wrong assumptions about the usage of the callee class (25 for our subject systems) that remain undetected when using automatically generated random and unit-level test suites.

Index Terms—CLING, Search-based software testing, Class integration testing, Coverage criteria, Test adequacy

1 Introduction

SEARCH-BASED approaches have been applied to a variety of white-box testing activities [1], among which test case and data generation [2]. In white-box testing, most of the existing work has focused on the unit level, where the goal is to generate tests that achieve high structural (e.g., branch) coverage. Prior work has shown that search-based unit test generation can achieve high code coverage [3], [4], [5], detect real-bugs [6], [7], and help developers during debugging activities [8], [9].

Despite these undeniable results, researchers have identified various limitations of the generated unit tests [7], [10], [11]. Prior studies have questioned the effectiveness of the generated unit tests with high code coverage in terms of their capability to detect real faults or to kill mutants when using mutation coverage. For example, Gay *et al.* [10] have highlighted how traditional code coverage could be a poor indicator of test effectiveness (in terms of fault detection rate and mutation score). Shamshiri *et al.* [7] have reported that around 50% of faults remain undetected when relying on generated tests with high coverage. Similar results have also been observed for large industrial systems [3].

Gay *et al.* [10] have observed that traditional unit-level adequacy criteria only measure whether certain code elements are reached, but not *how* each element is covered. The quality of the test data and the paths from the covered element to the assertion play an essential role in better test effectiveness. As such, they have advocated the need for more reliable adequacy criteria for test case generation tools. While these results hold for generated unit tests, other studies on hand-written unit tests have further highlighted the limitation of unit-level code coverage criteria [11], [12].

In this paper, we explore the usage of the integration code between coupled classes as guidance for the test generation process. The idea is that, by exercising the behavior of a class under test E (the calleE) through another class R (the calleR) calling its methods, R will handle the creation of complex parameter values and exercise valid usages of E. In other words, the caller R contains integration code that (1) enables the creation of better test data for the callee E, and (2) allows to better validate the data returned by E.

Integration testing can be approached from many different angles [13], [14]. Among others, *dataflow analysis* seeks to identify possible interactions between the definition and usage (def-use) of a variable. Various coverage criteria based on intra- (for class unit testing) and inter-class (for class integration testing) def-uses have been defined over the years [15], [16], [17], [18], [19], [20], [21]. Dataflow analysis faces several challenges, including the scalability of the algorithms to identify def-use pairs [22] and the number of test objectives that is much higher for dataflow criteria compared to *control flow* ones like branch and branch pair coverage [15], [20].

In our case, we focus on **class integration testing** between a caller and a callee [23]. Class integration testing aims to assess whether two or more classes work together properly by thoroughly testing their interactions [23]. Our

DOI: https://doi.org/10.1109/TSE.2022.3209625

idea is to complement unit test generation for a class under test by looking at its integration with other classes using **control flow analysis**. To that end, we define a novel structural adequacy criterion called the **Coupled Branches Coverage** criterion (CBC), targeting specific integration points between two classes. Coupled branches are pairs of branches $\langle r, e \rangle$, with r a branch of the caller, and e a branch of the callee, such that an integration test that exercises branch r also indirectly exercises branch e.

Furthermore, we implement a search-based approach that generates integration-level test suites leveraging the CBC criterion. We name our approach Cling (for class integration testing). Cling uses a state-of-the-art many-objective solver that generates test suites maximizing the number of covered coupled branches. For the guidance, Cling uses novel search heuristics defined for each pair of coupled branches (the search objectives).

We conducted an empirical study on 140 well-distributed (in terms of complexity and coupling) pairs of caller and callee classes extracted from five open-source Java projects. Our results show that Cling can achieve up to 99% CBC coverage, with an average of 49% across all pairs of classes. We analyzed the benefits of the integration-level test cases generated by Cling compared to unit-level tests generated by EvoSuite [24], the state-of-the-art generator of unit-level tests, and random tests generated by Randoop [25], a random-based test case generator. In particular, we assess whether integration-level tests generated by Cling can kill mutants and detect faults that would remain uncovered when relying on other generated tests given the same generation budget.

According to our results, on average, Cling kills 7.7% (resp. 13%) of the mutants per class that remain undetected by other tests generated using EvoSuite (resp. Randoop) for both the caller and the callee. The improvements in mutation score are as high as 50% for certain pairs of classes. Our analysis indicates that many of the most frequently killed mutants are produced by integration-level mutation operators. Finally, we have found 25 integration faults (*i.e.,* faults due to wrong assumptions about the usage of the callee class) that were detected only by the integration tests generated with Cling (and not through testing with EvoSuite or Randoop).

The remainder of the paper is organized as follows. Section 2 summarizes the background and related work in the area. Section 3 defines the Coupled Branches Criteria and introduces Cling, our integration-level test case generator. Section 4 describes our empirical study, while Section 5 reports the corresponding empirical results. Section 6 discusses the practical implication of our results. Section 7 discusses the threats to validity. Finally, Section 8 concludes the paper.

2 Background and Related Work

McMinn [2] defined search-based software testing (SBST) as *"using a meta-heuristic optimizing search technique, such as a genetic algorithm, to automate or partially automate a testing task"*. Within this realm, test data generation at different testing levels (such as *unit testing, integration testing, etc.*) has been actively investigated [2]. This section provides an overview of earlier work in this area.

Listing 1
Class `Person`

```
1  class Person{
2      private Car car = new Car();
3      protected boolean lazy = false;
4      public void driveToHome(){
5          if (car.fuelAmount < 100) {
6              addEnergy();
7          } else {
8              car.drive();
9          }
10     }
11
12     protected void addEnergy(){
13         if (this.lazy) {
14             takeBus();
15         } else {
16             car.refuel();
17         }
18     }
19 }
```

2.1 Search-based approaches for unit testing

SBST algorithms have been extensively used for unit test generation. Previous studies confirmed that such generated tests achieve a high code coverage [5], [26], real-bug detection [3], and a debugging cost reduction [9], [27], complementing manually-written tests.

From McMinn's [2] survey about search-based test data generation, we observe that most of the current approaches rely on the control flow graph (CFG) to abstract the source code and represent possible execution flows. The $CFG_m = (N_m, E_m)$ represents a method (or function in procedural programming languages) m as a directed graph of **basic blocks** of code (the nodes N_m), while E_m is the set of the control flow edges. An edge connects a basic block n_1 to another one n_2 if the control may flow from the last statement of n_1 to the first statement of n_2.

Listing 1 presents the source code of `Person`, a class representing a person and her transportation habits. A `Person` can drive home (lines 4-10), or add energy to her car (lines 12-18). Figure 1 presents the CFG of two of Person's methods, with the labels of the nodes representing the line numbers in the code. Since method `driveToHome` calls method `addEnergy`, *node 6* is transformed to two nodes, which are connected to the entry and exit point of the called method. This transformation is explained in the last paragraph of this section.

Many approaches based on CFGs combine two common heuristics to reach a high branch and statement coverage in unit-level testing: the *branch distance* and the *approach level*. The *branch distance* measures (based on a set of rules) the distance to *satisfying* (true branch) and the distance to *not satisfying* (false branch) a particular branching node in the program. For instance, the distance to true for the condition at line 5 in Listing 1 is $100 - car.fuelAmount + 1$, and the distance to false is $car.fuelAmount - 100$. The *approach level* measures the distance between the execution path and a target node in a CFG. For that, it relies on the concepts of **post-dominance** and **control dependency** [28]. As an example, in Figure 1, *node 8* is control dependent on *node 5* and *node 8* post-dominates edge $\langle 5, 8 \rangle$. The *approach level* is the minimum number of control dependencies between a target node and an executed path by a test case.

In this study, we analyze how a class is used/invoked by the other classes within the same system. For this purpose,

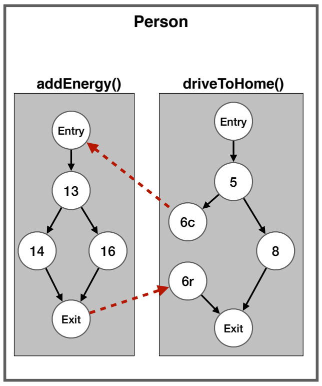


Fig. 1. Class-level CFG for class `Person`

we merge the Class-level Control Flow Graph (CCFG) of target callee and caller classes.

2.2 Search-based approaches for integration testing

Integration testing aims at finding faults that are related to the interaction between components. We discuss existing integration testing criteria and explain the search-based approaches that use these criteria to define fitness functions for automating integration-level testing tasks.

2.2.1 Integration testing criteria

Jin *et al.* [13] categorize the connections between two procedures into four types: *call couplings* (type 1) occur when one procedure calls another procedure; *parameter couplings* (type 2) happen when a procedure passes a parameter to another procedure; *shared data couplings* (type 3) occur when two procedures refer to the same data objects; *external device coupling* (type 4) happens when two procedures access the same storage device. They introduce integration testing criteria according to the data flow graph (containing the definitions and usages of variables at the integration points) of procedure-based software. Their criteria, called *coupling-based testing criteria*, require that the tests' execution paths cover the last definition of a parameter's value in the CFG of a procedure (the *caller procedure*), a node (the *call site*) calling another procedure with that parameter, and the first use of the parameter in the *callee* (and in the caller after the call if the parameter is a call-by-reference).

Harrold *et al.* [16] introduced data flow testing for a single class focusing on method-integration testing. They define three levels of testing: *intra-method testing*, which tests an individual method (*i.e.,* the smallest possible unit to test); *inter-method testing*, in which a public method is tested that (in)directly calls other methods of the same class, and *intra-class testing*, in which the various sequences of public methods in a class are tested. For data flow testing of inter-method and intra-class testing, they defined a *Class-level Control Flow Graph* (CCFG). The CCFG of class C is a directed graph $CCFG_C = (N_{Cm}, E_{Cm})$ which is a composition of the control flow graphs of methods in C; the CFGs are connected through their call sites to methods in the same class [16]. This graph demonstrates all paths that might be crossed within the class by calling its methods or constructors.

Let us consider again the class $Person$ in Listing 1. The CCFG of class $Person$ is created by merging the CFGs of its method, as demonstrated in Figure 3. For example, in the CFG of the method `Person.driveToHome()`, the *node 6c* is a call site to `Person.addEnergy()`. In the approach introduced by Harrold *et al.* [16], they detect the def-use paths in the constructed CCFGs and try to cover those paths.

Denaro *et al.* [22] revisited previous work on data flow analysis for object-oriented programs [16], [17] to define an efficient approach to compute *contextual def-use* coverage [17] for class integration testing. The approach relies on *contextual data flow analysis* to take state-dependent behavior of classes that aggregate other classes into account. Compared to def-use paths, contextual def-use include the chain of method calls leading to the definition or the use.

A special case is represented by the polymorphic interactions that need to be tested. Alexander *et al.* [18], [19] used the data flow graph to define testing criteria for integrations between classes in the same hierarchy tree.

All of the mentioned approaches are using data-flow analysis to define integration testing criteria. However, generating data-flow graphs covering the def-uses involved in between classes is expensive and not scalable in complex cases [15]. Vivanti *et al.* [20] shows that the average number of def-use paths in a single class in isolation is three times more than the number of branches. By adding def-use paths between the non-trivial classes, this number grows exponentially.

In search-based approaches, the number of search objectives matters, as too many objectives lead to search process misguidance. Compared to previous work, our approach does not try to cover def-use paths. Instead, we use a *control flow analysis* to identify from a CCFG a restricted number of pairs of branches (in a caller and a callee) that are not trivially executed together. For instance, the couple of branches $\langle 13, 16\rangle$ and $\langle b8, b9\rangle$ in Figure 3 are used to define the search objectives of our test case generator. Section 3 details the analysis of the CCFG to identify such pairs of branches, including for special cases of interaction (namely inheritance and polymorphism), and the definition of the objectives and search algorithm.

CCFGs have previously been used in other approaches. For instance, Wang *et al.* [29] merge the CFGs of methods of classes in the dependencies of the software under test to identify dependency conflicts.

2.2.2 Search-based approaches

Search-based approaches are widely used for test ordering [30], [31], [32], [33], [33], [34], [35], [36], [37], [38], [39], [40], [41], typically with the aim of executing those tests with the highest likelihood of failing earlier on. However, search-based approaches have rarely been used for generating class integration tests. Ali Khan *et al.* [42] have proposed a high-level evolutionary approach that detects the coupling paths in the data-flow graphs of classes and have used it to define the fitness function for the genetic algorithm. They also proposed another approach for the same goal relying on Particle Swarm Optimization [43]. Since objectives are defined according to the def-use paths between classes, the number of search objectives can grow exponentially, thus severely limiting the scalability of the approach (as we explained in Section 2.2.1).

Most related to our approach is the work on *dynamic data flow testing* (DYNAFLOW) from Denaro *et al.* [21]. Dynamic data flow testing is a two steps *test amplification* pipeline [44] where: (i) a set of existing test cases are executed to collect execution traces, compute new data flow information, and subsequently derive new test objectives; and (ii) the new test objectives are fed to a test case generation tool. The pipeline is repeated until no new test objectives are found.

In this study, we propose a novel approach for class integration test generation. Instead of using the data flow graph, which is more expensive to construct than a class call graph, or incrementally amplify the existing test suite, requiring several executions of a test case generation tool, we use the information available in the class call graph of the classes to calculate the fitness of the generated tests. We do note that we could not find any available implementation of data flow-based approaches.

2.3 Evolutionary approaches for other testing levels

Arcuri [45] proposed EvoMaster, an evolutionary-based white-box approach for system-level test generation for RESTful APIs. A test for a RESTful web service is a sequence of HTTP requests. EvoMaster tries to cover three types of targets: (i) the statements in the System Under Test (SUT); (ii) the branches in the SUT; and (iii) different returned HTTP status codes. Although EvoMaster tests different classes in the SUT, it does not systematically target different integration scenarios between classes.

In contrast to EvoMaster, other approaches perform fuzzing [46], *"an automated technique providing random data as input to a software system in the hope to expose a vulnerability."* Fuzzing uses information like grammar specifications [46], [47], [48], [49] or feedback from the program during the execution of tests [50] to steer the test generation process. These approaches are black-box and do not rely on any knowledge about classes in the SUT. Hence, their search processes are not guided by the integration of classes.

Our approach performs white-box testing. It monitors the interaction between the target classes and strives to cover different integration scenarios between them.

3 CLASS INTEGRATION TESTING

The main idea of our class integration testing (hereinafter referred to as CLING) is to test the integration of two classes by leveraging the usage of one class by another class. More specifically, we focus on the calls between the former, the callee (E), and the latter, the caller (R). By doing so, we benefit from the additional context setup by R before calling E (*e.g.,* initializing a complex input parameter), and the additional post-processing after E returns (*e.g.,* using the return value later on in R), thus (implicitly) making assumptions on the behavior of E.

Figure 2 presents the general overview of CLING. CLING takes as input a pair of caller-callee $\langle R, E\rangle$ classes with at least one call (denoted *call site* hereafter) from R to E. Since the goal of CLING is to generate test cases covering E by calling methods in R, the first step (①) statically collects the list of *covering methods* in R that, when called, may directly or indirectly cover statements in E. This list is later used during the generation process to ensure that test cases contain calls to covering methods. The second step (②) statically analyzes the CCFGs of R and E to identify the coupled branches between R and E used later on to guide the search. The CCFGs are statically built from the CFGs of the methods (including inherited ones) in R and E. Finally, the generation of the test cases (③) uses a genetic algorithm with two additional *repair* steps, ensuring that the crossover and mutation only produce test cases able to cover lines in E. The result is a test suite for E, whose test cases invoke methods in R that cover the interactions between R and E.

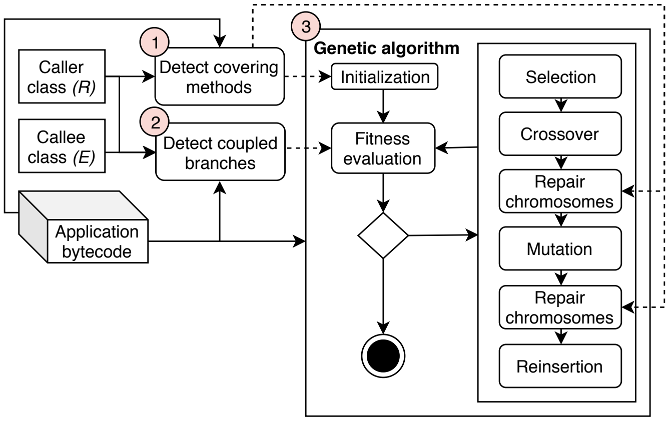


Fig. 2. General overview of CLING

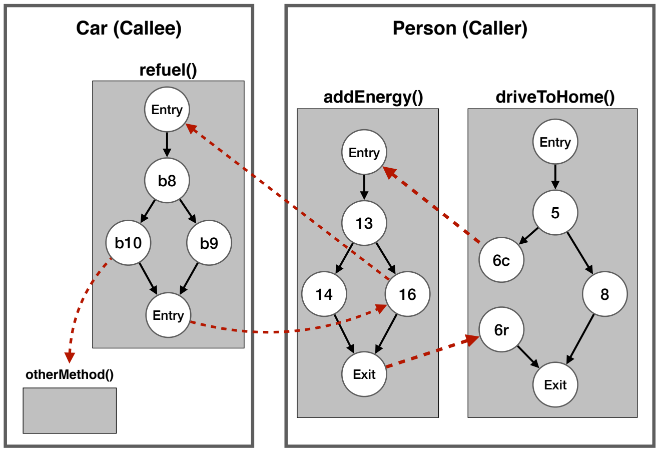


Fig. 3. Merging CCFGs of two classes: `Person` (caller) and `Car` (callee)

The remainder of this section describes our novel underlying Coupled Branches Criterion, the corresponding search-heuristics, and test case generation in CLING.

3.1 Coupled Branch testing criterion

To test the integration between two classes E and R, we need to define a coverage criterion that helps us to measure how thoroughly a test suite T exercises the interaction calls between the two classes (E and R). One possible coverage criterion would consist of testing all possible paths (*inter-class path coverage*) that start from the entry node of the caller R, execute the integration calls to E and terminate in one of the exit points of R. However, such a criterion will be affected by the *path explosion problem* [51]: the number of paths increases exponentially with the cyclomatic complexity of E and R, and thus the number of interaction calls between the two classes.

To avoid the *path explosion problem*, we define an integration-level coverage criterion, namely the Coupled Branch Criterion (CBC), where the number of coverage targets remains polynomial to the cyclomatic complexity of E and R. More precisely, CBC focuses on call coupling between caller and callee classes. Intuitively, let $s \in R$ be a call site, *i.e.,* a call statement to a method of the class E. Our criterion requires to cover all pairs of branches (b_r, b_e), where b_r is a branch in R that leads to s (the method call), and b_e is a branch of the callee E that is not trivially covered by every execution of E. So, in the worst case, the number of coverage targets is quadratic in number of branches in the caller and callee classes.

3.1.1 Target caller branches

Among all branches in the caller class, we are interested in covering the branches that are not trivially (always) executed, and that always lead to the integration call site (*i.e.,* calling the callee class) when covered. We refer to these branches as *target branches* for the caller.

Definition 3.1 (Target branches for the caller)**.** For a call site s in R, the set of target branches $B_R(s)$ for the caller R contains the branches having the following characteristics: (i) the branches are outgoing edges for the node on which s is control dependent (*i.e.,* nodes for which s post-dominates one of its outgoing branches but does not post-dominate the node itself); and (ii) the branches are post-dominated by s, *i.e.,* branches for which all the paths through the branch to the exit point pass through s.

To understand how we determine the target branches in the caller, let us consider the example of the caller and the callee in Figure 3. The code for the class `Person` is reported in Listing 1. The class `Person` contains two methods, `addEnergy()` and `driveToHome()`, with the latter invoking the former (line 6 in Listing 1). The method `Person.addEnergy()` invokes the method `refuel()` of the class `Car` (line 16 in Listing 1). The method `Person.driveToHome()` invokes the method `Car.drive()` (line 8 in Listing 1). Therefore, the class *Person* is the caller, while `Car` is the callee.

Figure 3 shows an excerpt of the Class-level Control Flow Graphs (CCFGs) for the two classes. In the figure, the names of the nodes are labelled with the line number of the corresponding statements in the code of Listing 1. Node 16 in `Person.addEnergy()` is a call site to `Car.refuel()`; it is also control dependent on nodes 5 (`Person.driveToHome()`) and 13 (`Person.addEnergy()`). Furthermore, node 16 only post-dominates branch $\langle 13, 16\rangle$. Instead, the branch $\langle 5, 6c\rangle$ is not post-dominated by node 16 as covering $\langle 5, 6c\rangle$ does not always imply covering node 16 as well. Therefore, the branches in the caller `Person.addEnergy()` that always lead to the callee are $B_{\texttt{Person}}(\texttt{Car.refuel()}) = \{\langle 13, 16\rangle\}$. Hence, among all branches in the caller class (`Person` in our example), we are interested in covering the branches that, when executed, always lead to the integration call site (i.e., calling the callee class). We refer to these branches as *target branches* for the caller.

3.1.2 Target callee branches

Like the target branches of the caller, the target branches of the callee are branches that are not trivially (always) executed each time the method is called.

Definition 3.2 (Target branches for the callee)**.** The set of target branches $B_E(s)$ for the callee E contains branches satisfying the following properties: (i) the branches are among the outgoing branches of branching nodes (*i.e.,* the nodes having more than one outgoing edge); and (ii) the branches are accessible from the entry node of the method called in s.

Let us consider the example of Figure 3 again. This time, let us look at the branches in the callee (`Car`) that are directly related to the integration call. In the example, executing the method call `Car.refuel()` (node 16 of the method `Person.addEnergy()`) leads to the execution of the branching node $b8$ of the class `Car`. Hence, the set of branches affected by the interaction calls is $B_{\texttt{Car}}(\texttt{Car.refuel()}) = \{\langle b8, b9\rangle;\ \langle b8, b10\rangle\}$. In the following, we refer to these branches as target branches for the callee. Note that, for a call site s in R calling E, the set of target branches for the callee also includes branches that are trivially executed by any execution of s.

3.1.3 Coupled branches

Given the sets of target branches for both the caller and callee, an integration test case should exercise at least one target branch for the caller (branch affecting the integration call) and one target branch for the callee (i.e., the integration call should lead to covering branches in the callee). In the following, we define pairs of target branches $(b_r \in B_R(s), b_e \in B_E(s))$ as *coupled branches* because covering b_r can lead to covering b_e as well.

Definition 3.3 (Coupled branches)**.** Let $B_R(s)$ be the set of target branches in the caller class R; let $B_E(s)$ be the set of target branches in the callee class E; and let s be the call site in R to the methods of E. The set of coupled branches $CB_{R,E}(s)$ is the cartesian product of $B_R(s)$ and $B_E(s)$:

$$CB_{R,E}(s) = CB_{R,E}(s) = B_R(s) \times B_E(s) \tag{1}$$

In our example of Figure 3, we have two coupled branches: the branches $(\langle 13, 16\rangle, \langle b8, b9\rangle)$ and the branches $(\langle 13, 16\rangle, \langle b8, b10\rangle)$.

Definition 3.4 (Set of coupled branches)**.** Let $S = (s_1, \ldots, s_k)$ be the list of call sites from a caller R to a callee E, the set of coupled branches for R and E is the union of the coupled branches for the different call sites S:

$$CB_{R,E} = \cup_{s \in S} CB_{R,E}(s)$$

Listing 2
Class GreenPerson

```
1  class GreenPerson extends Person{
2      private HybridCar car = new HybridCar();
3      @override
4      public void addEnergy(){
5          if(this.lazy){
6              takeBus();
7          }else if (chargerAvailable()){
8              car.recharge()
9          }else{
10             car.refuel();
11         }
12     }
13
14     private void chargerAvailable(){
15         if(ChargingStation.takeavailableStations().size >
              0){
16             return true;
17         }
18         return false;
19     }
20 }
```

3.1.4 The Coupled Branches Coverage criterion (CBC)

Based on the definition above, the CBC criterion requires that for all the call sites S from a caller R to a callee E, a given test suite T covers all the coupled branches:

$$CBC_{R,E} = \frac{|\{(r_i, e_i) \in CB_{R,E} | \exists t \in T : t \ covers \ r_i \ and \ e_i\}|}{|CB_{R,E}|}$$

We do note that this formula is only relevant if there are indeed call interactions between caller and callee. As for classical branch and branch-pair coverage, $CB_{R,E}$ may contain incompatible branch-pairs (*e.g.*, when the conditions are mutually exclusive). However, detecting and filtering such pairs is an undecidable problem. Hence, in this study, we target all coupled branches.

3.1.5 Inheritance and polymorphism

In the special case where the caller and callee classes are in the same inheritance tree, we use a different procedure to build the CCFG of the super-class and find the call sites S. The CCFG of the super-class is built by merging the CFGs of the methods that are not overridden by the sub-class. As previously, the CCFG of the sub-class is built by merging the CFGs of the methods defined in this class, including the inherited methods overridden by the sub-class (other non-overridden inherited methods are not part of the CCFG of the sub-class).

For instance, the class `GreenPerson` in Listing 2, representing owners of hybrid cars, extends class `Person` from Listing 1. For adding energy, a green person can either refuel or recharge her car (lines 7 to 11). `GreenPerson` overrides the method `Person.addEnergy()` and defines an additional method `GreenPerson.chargerAvailable()` indicating whether the charging station is available. Only those two methods are used in the CCFG of the class `GreenPerson` presented in Figure 4, inherited methods are not included in the CCFG; the CCFG of the super-class `Person` does not contain the method `Person.addEnergy()`, redefined by the sub-class `GreenPerson`.

The call sites S are identified according to the CCFGs, depending on the caller and the callee. If the caller R is the super-class, S will contain all the calls in R to methods that have been redefined by the sub-class. For instance, nodes 6 and 13 in Figure 3 with `Person` as caller. If the caller R is the sub-class, S will contain all the calls in R to methods that have been inherited but not redefined by R. For instance, node 7c in Figure 4 with `GreenPerson` as caller.

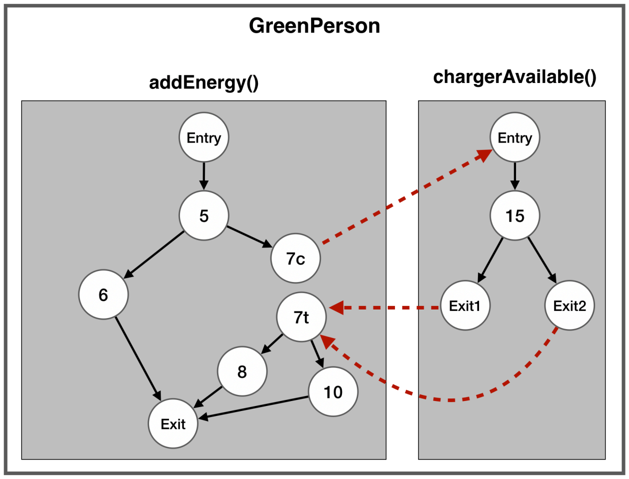


Fig. 4. CCFG of $GreenPerson$ as subclass

3.2 Cling

Cling is the tool that we have developed to generate integration-level test suites that maximize the proposed CBC adequacy criterion. The inputs of Cling are the (1) *application's bytecode*, (2) a *caller class* R, and (3) a *callee class* E. As presented in Figure 2, Cling first detects the covering methods (step ①) and identifies the coupled branches $CB_{R,E}(s)$ for the different call sites (step ②), before starting the search-based test case generation process (detailed in the following subsections). Cling produces a test suite that maximizes the CBC criterion for R and E.

Satisfying the CBC criterion is essentially a many-objective problem where integration-level test cases have to cover pairs of coupled branches separately. In other words, each pair of coupled branches corresponds to a search objective to optimize. The next subsection describes our search objectives.

3.2.1 Search objectives

In our approach, each objective function measures the distance of a generated test from covering one of the coupled branch pairs. The value ranges between $[0, +\infty)$ (zero denoting that the objective is satisfied). Assume that $CB_{R,E} = \{c_1, c_2, \ldots, c_n\}$ is the set of coupled branches $\langle r_i, e_i \rangle$ between R and E. Then, the fitness for a test case t is defined by the following distinct objectives:

$$Objectives = \begin{cases} d(c_1, t) \ = \ D(r_1, t) \oplus D(e_1, t) \\ \ldots \\ d(c_n, t) = D(r_n, t) \oplus D(e_n, t) \end{cases} \qquad (2)$$

where $D(b, t) = al(b, t) + bd(b, t)$ computes the distance between the test t to the branch b using the classical approach level $al(b, t)$ (*i.e.*, the minimum number of control dependencies between b and the execution path of t) and normalized branch distance $bd(b,t)$ (*i.e.,* the distance, computed based on a set of rules, to the branch leading to b in the closest node on the execution path of t) [2]; and $D(r_i,t) \oplus D(e_i,t)$ is defined as $D(r_i,t)+1$ if $D(r_i,t) > 0$ (*i.e.,* the caller branch is not covered) and $D(e_i,t)$ otherwise (*i.e.,* the caller branch is covered).

For example, assume that we want to measure the fitness of a test case t', generated during the search process while targeting coupled branches from the classes `Person` (caller class) and `Car` (callee class). This test case covers the following path in the CCFG depicted by Figure 3: $Entry \rightarrow 13 \rightarrow 16 \rightarrow b8 \rightarrow b10 \rightarrow Exit$. As explained in Section 3.1.3, this pair of classes contains two coupled branches: $(\langle 13,16\rangle, \langle b8,b10\rangle)$ and $(\langle 13,16\rangle, \langle b8,b9\rangle)$, each corresponding to a search objective. Since t' covers both of the branches in the first couple, the objective corresponding to that couple is fulfilled and its fitness value is 0. In contrast, t' only covers the first branch of the second couple (*i.e.,* $\langle b8,b9\rangle$ is not covered). In this case, $D(r,t')$ equals zero, but $D(e,t')$ is calculated using the approach level and branch distance heuristics. Since t' covers all of the control dependent branches the approach level, $al(b,t')$, equals zero. The branch distance, $bd(b,t') \in [0,1]$, is calculated according to the concrete values used in the branching condition in the last covered control dependent node (here, $b8$) where the execution path of t' changed away from reaching to the second target branch $\langle b8,b9\rangle$.

3.2.2 Search algorithm

To solve such a many-objective problem, we tailored the Many-Objective Sorting Algorithm (MOSA) [52] to generate test cases through class integration. MOSA has been introduced and assessed in the context of unit test generation [52] and security testing [53]. Additionally, previous studies [26], [52] have shown that MOSA is very competitive compared with alternative algorithms when handling hundreds and thousands of testing objectives. Interested readers can find more details about the original MOSA algorithm in Panichella *et al.* [52]. Although a more efficient variant of MOSA has recently been proposed [54], such a variant (DynaMOSA) requires to have a hierarchy of dependencies between coverage targets that exists only at the unit level. Since targets in unit testing are all available in the same control flow graph, the dependencies between objectives can be calculated (*i.e.,* the control dependencies). In contrast, CLING's objective is covering combinations of targets in different control flow graphs. Since covering one combination does not depend on the coverage of another combination, DynaMOSA is not applicable to this problem.

Therefore, in CLING, we tailored MOSA to work at the integration level, targeting pairs of coupled branches rather than unit-level coverage targets (e.g., statements). In the following, we describe the main modifications we applied to MOSA to generate integration-level test cases.

3.2.3 Initial population

The search process starts by generating an initial population of test cases. A random test case is a sequence of statements (*object instantiations*, *primitive statements*, *method calls*, and *constructor calls to the class under test*) of variable lengths. More precisely, the random test cases include *method calls* and *constructors* for the caller R, which directly or indirectly invoke methods of the callee E (*covering methods*). Although CLING generates these test cases randomly, it extends the initialization procedure used for search-based crash reproduction [27]. In particular, the initialization procedure in CLING gives a higher priority to methods in the caller class R that invoke methods of the callee class E. While calls to other methods of R are also inserted, their insertion has a lower probability. This prioritization ensures to generate tests covering call sites to the callee class. In the original MOSA algorithm, all methods of the class under test are inserted in each random test case with the same probability without any prioritization. The execution time of the initialisation procedure is part of the search budget.

3.2.4 Mutation and crossover

CLING uses the traditional single-point crossover and mutation operators [24] (adding, changing and removing statements) with an additional procedure to repair broken chromosomes. The initial test cases are guaranteed to contain at least one *covering method* (a method of R that directly or indirectly invokes methods of E). However, mutation and crossover can lead to generating *offspring* tests that do not include any *covering method*. We refer to these chromosomes as *broken chromosomes*. To fix the broken chromosomes, the *repair procedure* works in two different ways, depending on whether the broken chromosome is created by the crossover or by the mutation.

If the broken chromosome is the result of the mutation operator, then the repair procedure works as follows: let t be the broken chromosome and let M be the list of covering methods; then, CLING applies the mutation operator to t in an attempt to insert one of the covering methods in M. If the insertion is not successful, then the mutation operator is invoked again within a loop. The loop terminates when either a covering method is successfully injected in t or when the number of unsuccessful attempts is greater than a threshold (50 by default). In the latter case, t is not inserted in the new population for the next generation.

If the broken chromosome is generated by the crossover operator, then the broken child is replaced by one of its parents.

3.2.5 Polymorphism

If the caller and callee are in the same hierarchy and the caller is the super-class, CLING cannot generate tests for the caller class that will cover the callee class (since the methods to cover are not defined in the super-class). This is the case for instance if the super-class (caller) calls abstract methods defined in the sub-class (callee). In this particular case, CLING generates tests for the callee class. However, it selects the covering methods only from the inherited methods which are not overridden by the callee (sub-class). A covering method should be able to cover calls to the methods that have been redefined by the sub-class. With this slight change, CLING can improve the CBC coverage, as described in Section 3.1.5.

3.3 Implementation

We implemented CLING as an open-source tool written in Java.[1] The tool relies on the EVOSUITE [24] library as an external dependency. It implements the code instrumentation for pairs of classes, builds the CCFGs at the byte-code level, and derives the coverage targets (pairs of branches) according to the CBC criterion introduced in Section 3.1.4. The tool also implements the search heuristics, which are applied to compute the objective scores as described in Section 3. Besides, it implements the repair procedure described in Section 3.2.4, which extends the interface of the genetic operators in EVOSUITE. Moreover, we customized the many-objective MOSA algorithm [54], which is implemented in EVOSUITE, for our test case generation problem in CLING.

4 EMPIRICAL EVALUATION

Our evaluation aims to answer three research questions. The first research question analyzes the levels of CBC coverage achieved by CLING. For this research question, we first analyze the coupled branches covered by CLING in each of the cases:

RQ1.1 *What is the CBC coverage achieved by* CLING*?*

As explained in Section 2.2.2, to the best of our knowledge, there is no class-integration test case generator available for comparison. We thus compare CLING to the state-of-the-art unit test generators in terms of CBC coverage:

RQ1.2 *How does the CBC coverage achieved by* CLING *compare to automatically generated unit-level and random tests?*

Since the test cases generated by CLING aim to cover coupled branches between two classes, we need to determine the effectiveness of this kind of coverage compared to test suites generated for high branch coverage in unit testing:

RQ2.1 *What is the effectiveness of the integration-level tests compared to unit-level and random tests?*

Additionally, as the integration code of the caller can help to create better test data for the callee and validate its returned data, we investigate the complementarity between CLING and unit testing *w.r.t.* fault detection in the callee:

RQ2.2 *How complementary are the integration-level tests to the unit-level and random tests w.r.t. fault detection?*

Finally, we want to see whether the tests generated by CLING can make any difference in practice. Hence, we analyzed the integration faults captured by these tests:

RQ3 *What integration faults does* CLING *detect?*

4.1 Baseline Selection

The goal of this evaluation is to explore the impact and complementarity of the tests generated by CLING on the results of the search-based unit testing in various aspects. To achieve this purpose, we run our tool against EVOSUITE, which is currently the best tool in terms of achieving branch coverage [55], [56], [57], [58], [59]. Additionally, we compare CLING against randomly generated tests using RANDOOP [25], a feedback-directed random test case generator. In contrast to EVOSUITE, RANDOOP can randomly generate tests for multiple classes.

4.2 Subjects Selection

The subjects of our studies are five Java projects listed in Table 1, namely *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, and *Joda-Time*. Our primary reason to use these projects is that they have been used in prior studies to assess the coverage and the effectiveness of unit-level test case generation [7], [54], [60], [61], program repair [62], [63], fault localization [64], [65], and regression testing [66]. A consequence of this selection is that the source code under analysis is relatively old, making it hard to interact with developers to get confirmation about potential faults. Thus, the route that we take instead is to use *future commits* (after the commits under analysis) to explore whether the bugs we identify were addressed (possibly after failures in production), as explained in the next section.

To sample the classes under test, we first extract pairs of `caller` and `callee` classes (*i.e.,* pairs with interaction calls) in each project. Then, we remove pairs that contain trivial classes, *i.e.,* classes where the caller and callee methods have no decision point (*i.e.,* with cyclomatic complexity equal to one). This is because methods with no decision points can be covered with single method calls at the unit testing level. Note that similar filtering based on code complexity has been used and recommended in the related literature [4], [54], [57]. From the remaining pairs, we sampled 140 distinct pairs of classes from the five projects in total, which offers a good balance between generalization (*i.e.,* the number of pairs to consider) and statistical power (*i.e.,* the number of executions of each tool against each class or pair of classes). We performed the sampling to have classes with a broad range of complexity and coupling. In our sampling procedure, each selected class pair includes either the classes with the highest cyclomatic complexity or the mosts coupled classes. The numbers of pairs selected from each project are reported in Table 1 (column #), as well as the average cyclomatic complexity ($\overline{cc}$) of the caller and the callee, the average number ($\overline{count}$) of calls from the caller to the callee, and the minimum (min), average ($\overline{count}$), and maximum (max) number of coupled branches. Each pair of caller and callee classes represents a target for CLING.

As reported in Table 1, CLING did not identify any coupled-branches for three pairs of classes (one in `mockito` and two in `time`). This is due to the absence of target branches in either the caller or the callee, resulting in no couple of branches to cover. Those three pairs have been excluded from the results.

Our replication package [67] contains the list of class pairs sampled for our study, their detailed statistics (*i.e.,* cyclomatic complexity and the number of interaction calls), and the project versions.

4.3 Configurations

To answer the research questions, we run CLING on each of the selected class pairs. For each class pair targeted with CLING, we run EVOSUITE with the caller and the callee classes as target classes under test (*i.e.,* each class is targeted independently) to compare the class integration test suite with unit level test suites for the individual classes. We configure EVOSUITE to use *DynaMOSA*

[1] Available at https://github.com/STAMP-project/botsing/tree/master/cling

TABLE 1
Projects in our empirical study. `#` indicates the number of caller-callee pairs. `CC` indicates the cyclomatic complexity of the caller and callee classes. `Calls` indicates the number of calls from the caller to the callee. `Coupled branches` indicates the number of coupled branches.

Project	**#**	**Caller**		**Callee**		**Calls**		**Coupled branches**			
		$\overline{cc}$	σ	$\overline{cc}$	σ	$\overline{count}$	σ	*min*	$\overline{count}$	σ	*max*
closure	26	1,221.3	1,723.0	377.2	472.5	70.3	101.0	4	10,542	17,080	60,754
mockito	20	115.3	114.4	127.8	113.2	39.5	64.9	0	1,185	1,974	6,929
time	51	68.7	84.0	87.2	92.3	23.9	50.5	0	494	1,093	5,457
lang	18	145.0	177.8	235.3	242.7	12.4	14.6	2	409	598	1,826
math	25	79.2	88.4	57.5	64.4	18.8	34.5	2	294	613	2,682
All	140	301.1	859.5	160.6	257.7	32.4	62.8	0	2,412	8,294	60,754

(`-Dalgorithm=DynaMOSA`), which has the best outcome in structural and mutation coverage [54] and branch coverage (`-Dcriterion=BRANCH`).

Randoop does not include any dynamic dependency analysis and requires the user to manually specify the list of classes whose methods, constructors, and fields may appear in a test. Following the guidelines provided in the Randoop manual[2], we use the Java dependencies analysis utility (`jdeps`) to identify direct and indirect dependencies of the caller and callee classes. As the first step, we recursively collected all of the dependencies for caller and callee classes (as suggested by the Randoop manual). However, after running the first round of the experiment with all of the dependencies, we noticed that using all the indirect dependencies resulted in a large number of Randoop executions not terminating due to infinite test case executions (Randoop did not terminate in 128/140 of cases used in this experiment). As mentioned in the Randoop manual, this scenario occurs when one of the tests generated by Randoop traps in an infinite loop and drives the whole test generation process to get stuck in an infinite loop. Hence, we followed another suggestion mentioned in the Randoop manual and limited the depth to 2: *i.e.,* for each caller and callee, we provided a list of classes to use in the generated tests containing the caller and the callee, their direct dependencies, and the direct dependencies of those dependencies. Additionally, we specify (using option `--require-covered-classes=`) to keep only test cases in which the caller or callee class are directly or indirectly used. As mentioned by the manual, this option only works if Randoop is executed using the `covered-class javaagent` to instrument the classes. So, we also used this javaagent for Randoop executions in our experiment.

This results in having the following configurations, each one corresponding to a test suite generated by one independent execution of Cling, Randoop or EvoSuite:

1) T_{CLING}, the integration-level test suite generated by Cling (`-target_classes <Caller>, <Callee>`);
2) T_{Ran}, the random test suite generated by Randoop for the caller and callee (`--classlist=<Caller>, <Callee>, <level 1 dependencies>, <level 2 dependencies>`);
3) T_{EvoR}, the unit-level test suite generated by EvoSuite for the caller (`-class <Caller>`);
4) T_{EvoE}, the unit-level test suite generated by EvoSuite for the callee (`-class <Callee>`).

All other parameters were left to their default values.

4.4 Evaluation Procedure

To address the random nature of the three tools, we repeat each run 20 times (140 pairs of classes × 4 executions × 20 repetitions = 11,200 executions). Moreover, each Cling run is configured with a search budget of five minutes, including two minutes of search initialization timeout. To allow a fair comparison, we run EvoSuite for five minutes on each caller and callee class, and Randoop for ten minutes as it generates tests for both the caller and the callee, including default initialization timeout. This represents a total of ~48.6 days execution time for test case generation.

For **RQ1**, we analyze the CBC coverage achieved by T_{Cling}. As the CBC coverage of T_{EvoE} is equal to 0.0 by construction, we compare T_{Cling} with T_{Ran} and T_{EvoR} across the 20 independent runs.

For **RQ2**, we measure the effectiveness of the generated test suite using both *line coverage* and *mutation analysis* on the callee classes E (considered as the class under test in our approach). Mutation analysis is a high-end coverage criterion, and mutants are often used as substitutes for real faults since previous studies highlighted its significant correlation with fault-detection capability [68], [69]. Besides, mutation analysis provides a better measure of the test effectiveness compared to more traditional coverage criteria [12] (*e.g.,* branch coverage).

We compute the line coverage and mutation scores achieved by T_{CLING} for the callee class in each target class pair. Then, we compare them to the line coverage and mutation scores achieved by T_{Ran}, and the unit-level test suites T_{EvoR} and T_{EvoE}) for the callee class. Moreover, we analyse the orthogonality of the sets of mutants in the callee that are strongly killed by T_{CLING}, and those killed by the random and unit-level tests individually. In other words, we look at whether T_{CLING} allows killing mutants that are not killed at unit-level or by random tests (strong mutation). Also, we analyze the type of the mutants which are only killed by T_{CLING}.

For line coverage and mutation analysis, we use Pit [70], which is a state-of-the-art mutation testing tool for Java code, to mutate the callee classes. Pit also collects and reports the line coverage of the test suite on the original class before mutation. Pit has been used in literature to assess the effectiveness of test case generation tools [56], [57], [58], [59], [60], [71], and it has also been applied in industry[3]. In

[2] https://randoop.github.io/randoop/manual/

[3] http://pitest.org/sky_experience/

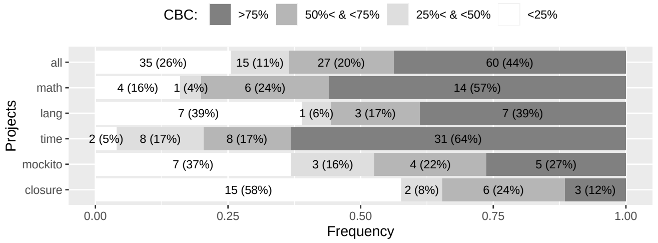


Fig. 5. Distribution of CLING's CBC coverage for the different class pairs.

our study, we use PIT v.1.4.9 with all mutation operators activated (*i.e.,* the `ALL` mutators group).

For **RQ3**, we analyze the exceptions triggered by both integration, random, and unit-level test suites. In particular, we extract unexpected exceptions causing crashes, *i.e.,* exceptions that are triggered by the test suites but that are (i) not declared in the signature of the caller and callee methods using `throws` clauses, (ii) not caught by a `try-catch` blocks, and (iii) not documented in the `Javadoc` of the caller or callee classes. Then, we manually analyze unexpected exceptions that are triggered by the integration-level test cases (*i.e.,* by CLING), but not by the random and unit-level tests. Since our subjects are selected from DEFECTS4J, and thereby the projects used as subjects in this study are not the latest versions, the faults that we find for this research question may be fixed in the subsequent commits. Hence, the three first authors performed a code history analysis by looking at the modifications made to the source code of the classes involved in a fault. Based on this analysis, the authors could examine whether the faults found by CLING were later identified, approved, and fixed by the developers.

The test suites generated by CLING, EVOSUITE, and RANDOOP may contain **flaky tests**, *i.e.,* test cases that exhibit intermittent failures if executed with the same configuration. To detect and remove flaky tests, we ran each generated test suite five times. Hence, the test suites used to answer our three research questions likely do not contain flaky tests. In this process, we identified 8%, 3.5%, and 4.7% of the tests generated by CLING, EVOSUITE, and RANDOOP, respectively, as flaky. For 20 runs, we detected a total of 1,410,320 flaky tests out of 29,785,260 generated test cases.

To keep the execution time (which includes test generation, flaky test detection, and mutation and coverage analysis) manageable, we used a cluster (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) to parallelize the execution for our evaluation (50 simultaneous executions). With this parallelization, the automated execution of the whole evaluation took about five days (one day for test generation and four days for flaky test detection and mutation and line coverage measurement).

5 EVALUATION RESULTS

This section presents the results of the evaluation and answers the research questions.

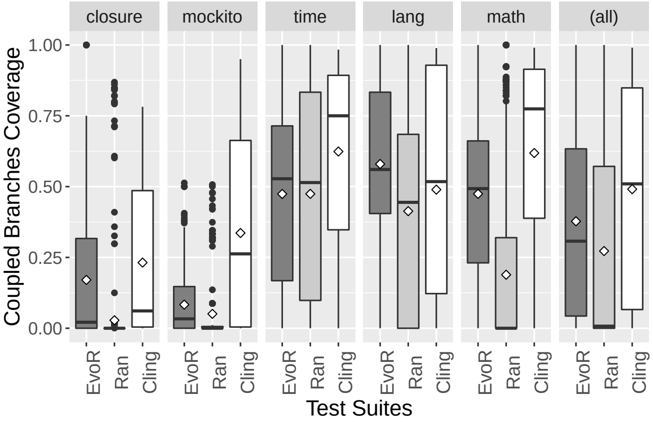


Fig. 6. Total coupled-branches coverage achieved by T_{CLING} (Cling), T_{Ran} (Ran) and T_{EvoR} (EvoR). (◇) denotes the arithmetic mean and (—) is the median.

5.1 CBC achieved by CLING (RQ1.1)

As explained in Section 4.2, CLING did not identify any coupled-branches for three pairs of classes. Figure 5 gives the distribution of the CBC coverage achieved by CLING for 137 pairs of classes. In total, CLING could generate at least one test suite achieving a coupled-branches coverage of at least 50% for 87 out of 137 class pairs. Figure 6 presents the coupled-branches coverage of T_{CLING} in all projects. On average (the diamonds in Figure 6) the test suites generated by CLING cover 49.1% of the coupled-branches.

The most covered couples are in the `time` project (62.4% on average), followed by `math` (61.9% on average) and `lang` (48.9% on average). The least covered couples are in the `closure` (23.2% on average) and `mockito` projects (33.6% on average), which are also the projects with the highest number of coupled-branches in Table 1 (10,542 coupled-branches on average for all the class pairs in `closure` and 1,185 coupled-branches on average in `mockito`).

For 9 caller-callee pairs, CLING could not generate a test suite able to cover at least one coupled branch during 20 executions: 3 pairs from `math`, 3 pairs from `mockito`, 2 pairs from `closure`, and 1 from `lang`. In the class pair from `lang`, CLING could not cover any coupled branch because the callee class (`StringUtils`) misleads the search process (we detail the explanation in Section 5.2). The remaining 8 pairs cannot be explained solely by the complexities of the caller (with a cyclomatic complexity ranging from 8 to 5,034 for those classes) and the callee (with a cyclomatic complexity ranging from 1 to 2,186) or the number of call sites (ranging from 1 to 177). This calls for a deeper understanding of the interactions between caller and callee around the call sites. In our future work, we plan to refine the caller-callee pair selection (for which we currently looked at the global complexity of the classes) to investigate the local complexity of the classes around the call sites.

Summary (RQ1.1). On average, the generated tests by CLING cover 49.1% of coupled-branches. In 87 out of 137 (59.2%) of the pairs, these test suites achieve a CBC higher than 50%.

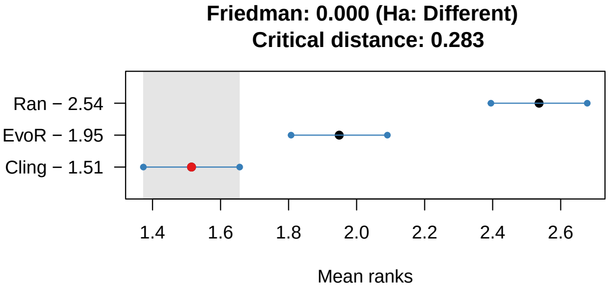


Fig. 7. Non-parametric multiple comparisons (*i.e.*, mean ranks with confidence interval) in terms of CBC score for T_{CLING} (Cling), T_{EvoR} (EvoR), and T_{Ran} (Ran) using Friedman's test with Nemenyi's post-hoc procedure.

5.2 CBC achieved by CLING vs. unit tests (RQ1.2)

Since T_{EvoE} test suites cover only branches in the callee class (*i.e.*, it does not call any methods in the caller class), the coupled-branches coverage achieved by these tests is always zero. Hence, for this research question, we compare the tests generated by CLING (T_{CLING}) against the tests generated by RANDOOP (T_{Ran}) and EVOSUITE applied to the caller class (T_{EvoR}) *w.r.t.* coupled-branches coverage.

Figure 6 presents the coupled-branches coverage of T_{CLING}, T_{Ran}, and T_{EvoR} for all projects. The number of covered coupled-branches by T_{CLING} is higher in total (*all* in Figure 6). On average (the diamonds in Figure 6), the test suites generated by CLING (49.1%) cover more coupled-branches compared to 37.8% for T_{EvoR}, and 27.2% for T_{Ran}. On average, the coupled-branches coverage achieved by unit tests is lower than the one achieved by CLING in all of the projects except lang. The average coupled-branches coverage of EVOSUITE in this project is 58%, compared to 48.9% for CLING. We also observe a wider distribution of the CBC coverage for T_{CLING} (with a median of 51.0% and an IQR of 78.2%) compared to T_{EvoR} (with a median of 30.7% and an IQR of 59.0%) and T_{Ran} (with a median $< 1.0\%$ and an IQR of 57.1%).

We further compare the different test suites using Friedman's non-parametric test for repeated measurements with a significance level $\alpha = 0.05$ [72] . This test is used to test the significance of the differences between groups (treatments) over the dependent variable (CBC coverage in our case). We complement the test for significance with Nemenyi's post-hoc procedure [73], [74]. Figure 7 provides a graphical representation of the ranking (*i.e.*, mean ranks with confidence interval) of the different test suites. According to the Friedman test, the different treatments (i.e., CLING, EVOSUITE, and RANDOOP) achieve significantly different CBC coverage (p-values $<$ 0.001). According to Figure 7, the average rank of CLING is much smaller than the average ranks of the two baselines. Furthermore, the differences between the average rank of T_{CLING} and the average rank of the two baselines are larger than the critical distance $CD = 0.283$ determined by Nemenyi's post-hoc procedure. This indicates that T_{CLING} achieves a significantly higher CBC coverage than T_{EvoR} and T_{Ran}.

Finally, we have manually analyzed the search progress of CLING for pairs of classes where the number of covered coupled-branches is low (*i.e.*, lower than 10). We noticed that CLING is counter-productive for specific class pairs where the callee class is `StringUtils`. In those cases, the test cases generated during the search initialization throw a `NoSuchFieldError` in the callee class (`StringUtils` here). Since these test cases achieve small approach levels and branch distances from the callee branches, they are fitter (*i.e.*, their fitness value is lower) than other test cases. Therefore, these test cases are selected for the next generation and drive the search process in local optima.

Summary (RQ1.2). On average, the generated test suites by CLING cover 11.3% more coupled-branches compared to EVOSUITE and 21.9% more coupled-branches compared to RANDOOP.

5.3 Line Coverage and Mutation Scores (RQ2.1)

Figure 8a shows line coverage of the callee classes (E) for the test suites generated by the different approaches. On average, CLING covers 39.5% of the lines of the callee classes. This is lower compared to unit-level tests generated using EVOSUITE (58.2% for T_{EvoE} and 59.4% for T_{EvoR}), and RANDOOP (47.4% for T_{Ran}).

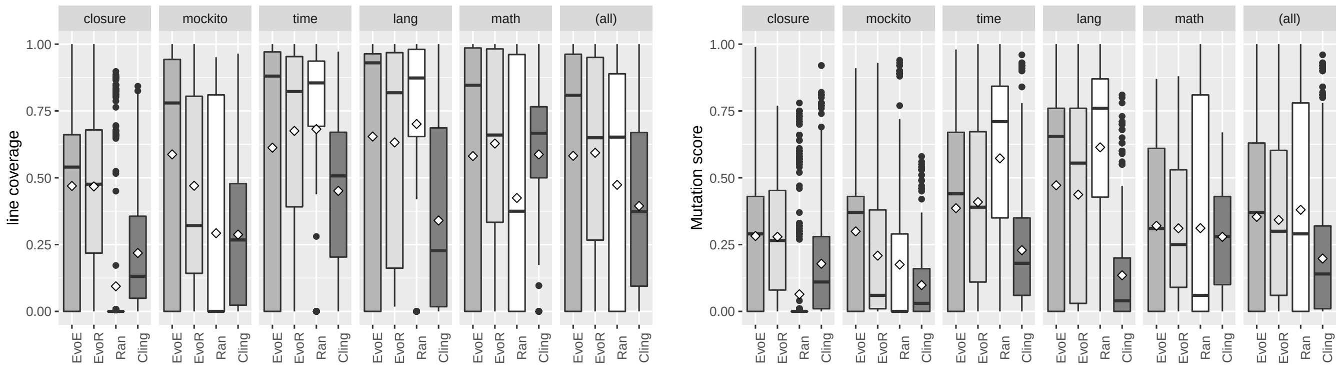
To understand the fault revealing capabilities of CLING compared to unit-level and random test suites, we first show in Figure 8b the overall mutation scores when mutating class E, and apply the test suite T_{EvoE}, T_{EvoR}, T_{Ran}, and T_{CLING}. Similar to line coverage, test suites optimized for overall branch coverage achieve a total higher mutation score (35.4% for T_{EvoE} and 34.2% for T_{EvoR} on average), simply because a mutant that is on a line that is never executed cannot be killed. RANDOOP achieves on average the best mutation score (38% for T_{Ran}), which would tend to indicate that despite a lower line coverage, indirect testing of the callee class through its dependencies enables discovering more faults. T_{CLING} scores lower (20.0% on average), since CLING searches for dedicated interaction pairs, but does not try to optimize overall line coverage. Note that T_{CLING} achieves the highest average mutation score for classes in `math`, while it achieves the lowest mutation score for classes in the `mockito` project.

Our results are consistent with the design and objectives of the tools: EVOSUITE seeks to cover all the branches of the class under test; CLING targets specific pairs of branches between the caller and callee; and RANDOOP performs (feedback-directed) random testing.

Summary (RQ2.1). The results in terms of line coverage are as expected, namely that EVOSUITE has the highest average line coverage (58.2% for T_{EvoE} and 59.4% for T_{EvoR}), followed by RANDOOP (47.4%) and CLING (39.5%). Regarding mutation score, RANDOOP achieved the highest mutation score on average (38%), followed by EVOSUITE (35.4% for T_{EvoE} and 34.2% for T_{EvoR} on average) and CLING (20.0%). This tends to indicate that despite a lower line coverage, indirect testing of the callee class through its dependencies in RANDOOP enables discovering more faults.

5.4 Combined Mutation Analysis (RQ2.2)

Figure 8b shows that unit test suites do not kill almost half of the mutants. CLING targets more mutants, including those



(a) Line coverage of the callee (E)

(b) Mutation score of the callee (E)

Fig. 8. Effectiveness of T_{CLING} (Cling), T_{EvoE} (EvoE), T_{EvoR} (EvoR), and T_{Ran} (Ran). (◇) denotes the arithmetic mean and (—) indicates the median.

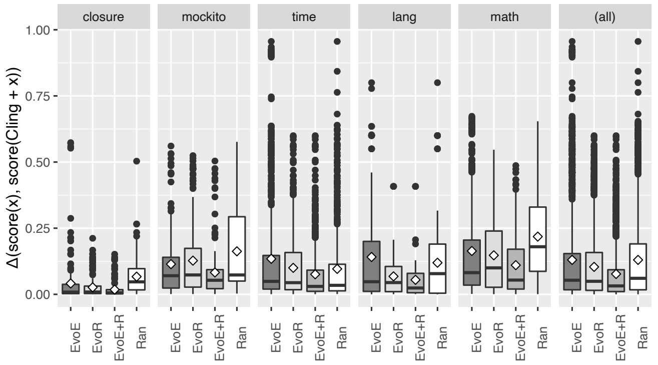


Fig. 9. Increases (Δ) of the mutation score when combining T_{CLING} with unit test suites T_{EvoE} (EvoE), T_{EvoR} (EvoR), and their unions $T_{EvoE+EvoR}$ (EvoE+R), and T_{Ran} (Ran). (◇) denotes the arithmetic mean and (—) is the median.

that remain alive with unit tests. In Figure 9, we report the *improvement* (Δ) in the mutation score when executing T_{CLING} in addition to different test suites (T_{EvoE}, T_{EvoR}, and T_{Ran}), and their unions ($T_{EvoE+EvoR}$).

On average, 13% of the mutants are killed only by T_{CLING} compared to both T_{EvoE}, the unit test suites optimized for the class under test (E), and T_{Ran}, randomly generated tests. This difference decreases to 10.4% if we use T_{EvoR} the unit test suites exercising E via the caller class R (as more class interactions are executed). The difference with traditional unit testing is still 7.7%, when comparing CLING with the combined unit test suites $T_{EvoE+EvoR}$, exercising E directly as much as possible as well as indirectly via call sites in R.

The outliers in Figure 9 are also of interest: for 20 classes (out of 137), CLING was able to generate a test suite where more than half of the mutants were killed only by T_{CLING}, compared to T_{EvoE} (i.e., +50% of mutation score). When compared to $T_{EvoE+EvoR}$ there are 4 classes for which T_{CLING} kills more than half of the mutants that are killed by neither T_{EvoE} nor T_{EvoR}. This further emphasizes the complementarity between the unit and integration testing. Comparing to randomly generated tests T_{Ran}, T_{CLING} kills more than half of the mutants for 13 classes, demonstrating the need for guidance when generating class integration tests.

Table 2 presents the status of the mutants that are killed by T_{CLING} but not by unit-level or random test cases. What stands out is that many mutants are in fact covered, but not killed by unit-level or random test suites. Here CLING leverages the context of the caller, not only to reach a mutant, but also to propagate the (modified) values inside the caller's context, so that the mutants can be eventually killed.

5.4.1 Mutation Operators

We analyzed the mutation operators that generate mutants that are exclusively killed by T_{CLING}. We categorize the mutation operators implemented in PIT into *integration-level* and *non-integration-level*. For this categorization, we rely on the definition of mutation operators for integration testing provided by Delamaro *et al.* [75]. We observed that ten of the mutation operators implemented in PIT inject integration-level faults. These operators can be mapped to two integration-level operators defined by Delamaro *et al.* [75]: *RetStaRep*, which replaces the return value of the called method, and *FunCalDel*, which removes the calls to void method calls and replaces the non-void method calls by a proper value.

Table 3 lists the number of mutants killed exclusively by T_{CLING} and grouped by mutation operators. Integration-level operators are indicated in bold with the mapping to either *RetStaRep* or *FunCalDel* between parenthesis. As we can see in this table, the most frequently killed mutants are produced by an integration-level operator, and other integration-level operators also produce frequently killed mutants. We can see that all of the ten integration-level mutation operators generate mutants that can be killed using CLING.

Furthermore, some of the most frequently killed mutants are not produced by integration-level operators. For instance, operator *NegateConditionalsMutator*, which mutates the conditions in the target class, produces the second most frequently killed mutants. These mutants are not killed but also not covered by tests generated by EVOSUITE.

TABLE 2
Status (for T_{EvoR}, T_{EvoE}, and T_{Ran}) of the mutants killed solely by T_{CLING}. `Not-covered` denotes the number of mutants killed by T_{CLING}, which are not covered by EvoSuite (or Randoop) test suites, and `survived` denotes the number of mutants killed by T_{CLING}, which are covered by EvoSuite (or Randoop) tests but not killed. The numbers between parentheses denote the percentage of mutants.

Test Suite	**closure**		**lang**		**math**		**mockito**		**time**	
	not-covered	survived	not-covered	survived	not-covered	survived	not-covered	survived	not-covered	survived
T_{EvoE}	1,988 (<1%)	881 (<1%)	3,247 (1%)	403 (<1%)	6,178 (5%)	1,747 (1%)	5,604 (4%)	2,414 (2%)	10,905 (3%)	5,920 (1%)
T_{EvoR}	2,480 (<1%)	780 (<1%)	2,797 (<1%)	851 (<1%)	5,310 (4%)	2,558 (2%)	4,867 (4%)	3,144 (2%)	7,431 (2%)	9,150 (2%)
T_{Ran}	18,935 (2%)	246 (<1%)	1,834 (1%)	343 (<1%)	13,316 (1%)	1,381 (1%)	18,419 (1%)	949 (1%)	18,073 (4%)	5,942 (1%)

```
1   boolean evaluateStepC(StepInterpolator interpolator
       ){
2     if (functions.isEmpty()){[...]}
3     if (! initialized) {[...]}
4     for ([...]) {
5         [...];
6         // calling the callee class in the next line.
7         if (state.evaluateStep(interpolator)){
8             // Changing variable first
9             [...]
10        }
11    }
12    return first != null;
13  }
```

(a) Method `evaluateStepC` declared in the caller class `SwitchingFunctionsHandler`.

```
1   boolean evaluateStep(final StepInterpolator
           interpolator){
2      [...]
3      for([...]){
4          if([...]){
5              [...];
6          }
7          if([...]){
8              [...];
9          }
10     }
11     [...];
12     ~~return false;~~ return true; //mutant
13  }
```

(b) Mutant `evaluateStep` declared in the callee class `switchsState`.

Fig. 10. Example of a integration-level mutant killed only by Cling From `Apache commons-math`.

TABLE 3
Number of mutants killed solely by T_{CLING} and grouped by mutation operators. Integration-level operators are highlighted in **bold** face and the corresponding integration-level mutation operator defined by Delamaro *et al.* [75] is indicated between parenthesis.

Against	**EvoSuite**		**Randoop**	
Mutation operator	**Rank**	**#kills**	**Rank**	**#kills**
NonVoidMethodCallMutator (*RetStaRep*)	1	1,983	1	2,340
NegateConditionalsMutator	2	1,638	2	2,020
InlineConstantMutator	3	1,201	5	1,183
ReturnValsMutator (*RetStaRep*)	4	1,195	4	1,414
RemoveConditionalMutator_EQUAL_IF	5	1,110	3	1,424
RemoveConditionalMutator_EQUAL_ELSE	6	1,015	6	1,138
NullReturnValsMutator (*RetStaRep*)	7	578	7	695
ArgumentPropagationMutator (*FunCalDel*)	8	518	9	539
MathMutator	9	513	11	398
MemberVariableMutator	10	458	8	576
ConstructorCallMutator (*FunCalDel*)	11	379	10	481
RemoveConditionalMutator_ORDER_IF	12	375	13	348
VoidMethodCallMutator (*FunCalDel*)	13	374	12	394
RemoveConditionalMutator_ORDER_ELSE	14	348	16	270
ConditionalsBoundaryMutator	15	322	15	272
PrimitiveReturnsMutator (*RetStaRep*)	16	309	14	295
NakedReceiverMutator	17	264	17	235
IncrementsMutator	18	143	19	154
BooleanTrueReturnValsMutator (*RetStaRep*)	19	142	18	162
RemoveIncrementsMutator	20	106	22	89
RemoveSwitchMutator	21	89	20	134
EmptyObjectReturnValsMutator (*RetStaRep*)	22	71	21	105
BooleanFalseReturnValsMutator (*RetStaRep*)	23	63	23	83
InvertNegsMutator	24	38	24	44
SwitchMutator	25	16	25	36

Listing 3
Cling test case killing mutant in Figure 10.

```
1      public void test07()  throws Throwable  {
2          [...]
3          boolean boolean1 = switchingFunctionsHandler0.
               evaluateStepC(stepInterpolator0);
4          assertTrue(boolean1 == boolean0);
5          assertFalse(boolean1);
6      }
```

As an example of a mutant killed only by T_{CLING}, Figure 10b illustrates one of the mutants in method `evaluateStep` in class `SwitchState` (callee class) from the *Apache commons-math* project. This mutant is produced by an integration-level mutation operator (*RetStaRep*) that replaces a boolean return value by `true`. Method *evaluateStep* is called from the method `evaluateStepC` (Figure 10a) declared in `SwitchingFunctionsHandler` (caller class). Method `evaluateStepC` must return false if it calls the callee class in a certain situation: (i) the variable `first` in the caller class is null, and (ii) the callee method returns false because of the execution of line 12 in Figure 10b.

The unit test suites generated by EvoSuite targeting `SwitchState` (T_{EvoE}) or class `SwitchingFunctionsHandler` (T_{EvoR}) both cover the mutant but do not kill it. T_{EvoE} easily cover the mutant statement, but it does not have any assertion to check the return value. T_{EvoR} also covers this statement by calling the right method in `SwitchingFunctionsHandler`. However, as is depicted by Figure 10, both methods in caller and callee class have multiple branches. So, T_{EvoR} covers the mutant from another path, which does not reveal the change in the boolean return value.

In contrast, this mutant is killed by T_{CLING}, targeting `SwitchingFunctionsHandler` and `SwitchState` as the caller and callee classes, respectively (Listing 3). According to the assertion in line 5 of this test case, `switchingFunctionsHandler0.evaluateStep` must return false. However, the mutant changes the returned value in line 7 of the caller class (Figure 10a), and thereby the true branch of the condition in line 7 is executed. This true branch changes the value of variable `first` from null to a non-

TABLE 4
Categorization and number (**#**) of the fault revealing test cases.

Category	#	Description
Confirmed	7	The test case exposes a fault that has been fixed (*e.g.,* by updating the code or the documentation), or has been marked as such in the source code (*e.g.,* using a comment).
Pending	4	The test case potentially exposes a fault that has not been fixed.
Deprecated	14	The test case is not relevant anymore as the source code it executes has been deleted from the project (*e.g.,* in the case of a deprecated method).

null value. Hence, the `evaluateStep` method in the caller class returns true in line 12. So, the assertion in the last line of the method in Listing 3 kills this mutant.

Summary (RQ2.2). The test suite generated by CLING for a caller R and callee E, can kill *different* mutants than unit and random test suites for E, R or their union, increasing the mutation score on average by 13.0%, 10.4%, and 7.7%, respectively, for EVOSUITE, and 13% for RANDOOP, with outliers well above 50%. Our analysis indicates that many of the most frequently killed mutants are produced by integration-level mutation operators.

5.5 Integration Faults Exposed by CLING (RQ3)

In our experiments, CLING generates 50 test cases that triggered unexpected exceptions in the subject systems. None of those exceptions were observed during the execution of the test cases generated by EVOSUITE and RANDOOP.

The first and second author independently performed a manual root cause analysis for all 50 unexpected exceptions to check if they actually stemmed from an integration-level fault. For this analysis, we check the API documentation to see if the generated test cases break any precondition. We indicated a test case as a fault revealing test if it does not violate any precondition according to the documentation, and it truly exposes an issue about the interaction between the caller and callee class. We found that out of the 50 test cases generated by CLING, 25 are fault revealing. The remaining 25 test cases trigger exceptions expected according to the documentation (5 tests), violate preconditions specified in the documentation (14 tests) or in the existing tests (1 test), return a wrong value for a method call on a mocked object (3 tests), or do not actually expose an issue between the caller and the callee class (2 tests).

To analyze if developers have already identified the faults in the following commits, the first three authors analyzed the code history of the classes involved in the detected faults. In this analysis, we manually checked all of the modifications made to the involved classes to see if the faults are fixed. Based on this analysis, we classify the 25 fault revealing test in one of the categories reported in Table 4. According to this Table, seven faults (found only by tests generated via CLING) were detected, confirmed, and fixed by developers in the next commits. We describe hereafter a representative example of these faults. The detailed descriptions of the analysis for all 25 fault revealing test cases are available in our replication package [67].[4]

Example. To illustrate the type of problem detected by CLING, consider the generated test case in Figure 11a and the induced stack trace (for a `NullPointerException`) in Figure 11b.[5] This test is produced by CLING for classes `UnionType` (caller class) and `JSType` (callee class). In this scenario, the `UnionType` is a sub-class of `JSType`. The test (Figure 11a Line 3) instantiates a `UnionType` object and passes a `null` value for the first parameter of its constructor. This constructor sets the value of a local variable (`registry`) to the value passed as the first parameter of the constructor (here, `null`). After instantiating `UnionType`, the generated test calls `getTypesUnderInequality` (Figure 11a Line 6), which in turns indirectly calls `isEmptyType` in the superclass. The `isEmptyType` method tries to use the attribute `registry`. Since this attribute is `null`, calling `getTypesUnderInequality` leads to a `NullPointerException`. No indication in the documentation specifies that the `registry` parameter should not be `null`, and no checks are done on the value of the input parameters.

By reviewing the code history of `UnionType` class, we observed that this fault has been fixed.[6] A `UnionType` should be instantiated only by a `UnionTypeBuilder` to ensure that it is instantiated properly, but this was not enforced in the source code nor documented in the class. The fixing commit message indicates that it refactors the "`UnionTypeBuilder` into `UnionType.Builder`, a nested class of `UnionType`" to "better reflect the entangled nature of the two classes." Concretely, the commit (i) refactors the `UnionTypeBuilder` class into `UnionType.Builder`, a nested class of `UnionType`; (ii) makes `UnionType`'s constructor private; and (iii) updates the `UnionType` constructor's documentation to indicate that this class has to be instantiated using its builder.

We also analyzed the tests generated by baseline tools for this case to understand why this specific fault is substantially less likely to be captured by EVOSUITE and RANDOOP. For EVOSUITE, since the tool concentrates on coverage of a single class, the tests generated by EVOSUITE only concentrate on covering the branches in the given class under test. So, in the EVOSUITE test generation process, tests cases that achieve higher branch coverage have higher priority than this failure capturing test case. This prioritization leads EVOSUITE to exclude this test case from the most optimized solutions during the search process. In contrast with EVOSUITE, CLING's search objectives (*i.e.,* CBC coverage) are designed to exercise the interactions between given class pairs and thereby give a higher priority to failures that can be captured in this interaction. Moreover, since RANDOOP gets a set of classes under test (*i.e.,* classes that

[4] Also available online at https://github.com/STAMP-project/Cling-application/blob/master/data_analysis/manual-analysis/failure-explanation.md.

[5] The details are available at https://github.com/STAMP-project/Cling-application/blob/master/data_analysis/manual-analysis/failure-explanation.md#st28

[6] The fixing commit is https://github.com/google/closure-compiler/commit/cfc0fab3dc2be49692a4fe9162b8095c934f6c41.

```
public void testFraction() {
    [...]
    UnionType unionType0 = new UnionType((
         JSTypeRegistry) null, immutableList0);

    // Undeclared exception!
    unionType0.getTypesUnderInequality(unionType0);
}
```

(a) Cling test case triggering the crash in Figure 11b.

```
java.lang.NullPointerException:
 at [..].JSType.isEmptyType([..]:159)
 at [..].JSType.testForEqualityHelper([..]:666)
 at [..].JSType.testForEquality([..]:655)
 at [..].NumberType.testForEquality([..]:63)
 at [..].JSType.getTypesUnderInequality([..]:962)
 at [..].UnionType.getTypesUnderInequality([..]:486)
 at [..].JSType.getTypesUnderInequality([..]:957)
 at [..].UnionType.getTypesUnderInequality([..]:486)
```

(b) Exception captured only by Cling.

Fig. 11. Example of test case generated by Cling and exposing a fault in the *Closure* project.

are direct or indirect dependencies of the given caller and callee classes), it has a higher search space to explore. In this case, Randoop generates tests using 30 classes indicated by jdeps (25 classes from the project under test and five from Java). In total, these classes contain 867 visible (non-private) methods. Also, the tests generated by Randoop initialize and use many objects, and hence the length of test cases are relatively higher than test cases generated by EvoSuite and Cling. Consequently, by looking at tests generated by Randoop, we can see that this tool generates many test cases that lead to higher coverage in the given set of classes but could not explore the particular part of the search space to capture this fault in the given time budget. However, theoretically, by giving enough time budget to Randoop, this tool should be able to cover this failure. In contrast, since Cling focuses on the interactions between two given classes, thereby having a smaller search space, it manages to capture this failure in a shorter time (*i.e.,* 5 minutes).

Summary (RQ3). Our manual analysis indicates that Cling-based automated testing of ⟨caller, callee⟩ class pairs can expose actual problems that are not found by unit testing either the caller or callee class individually. These problems relate to conflicting assumptions on the safe use of methods across classes (*e.g.,* due to undocumented exception throws, implicit assumptions on parameter values, *etc.*). Several of these faults are identified, confirmed, and fixed later by developers in subsequent commits.

6 Discussion

6.1 Applicability

The CBC criterion and its implementation in Cling consider pairs of classes and targets the integration between them. We did not propose any procedure for selecting pairs of classes to give in input to Cling. Since the technique requires pairs of classes to test, it would be time-consuming and tedious for developers to manually collect and provide the class pairs. Hence, we suggest using an automated process for class pair selection, as well. In this study, we implemented a tool that automatically analyzes each class pair to find the ones with high cyclomatic complexity and coupled branches (according to the CBC criterion defined in this article). This procedure is explained in Section 4.2.

Besides, our approach can be further extended by incorporating automated integration test prioritization approaches and selecting classes to integrate according to a predefined ordering [30], [32], [33], [34], [35], [37], [38], [39], [40], [41]. So, the end-to-end process of generating test for class integrations can be automated to require a minimal manual effort from the developer.

Moreover, since it is easier for developers to handle and integrate generated test cases in continuous integration, the number of tests generated by test generation approaches is also playing a crucial role in the effectiveness and applicability of techniques. Although Cling generates test cases that kill mutants and capture integration-level failures that cannot be covered by unit and random testing, this approach generates less test cases compared to EvoSuite and Randoop. In total, Cling generated 64,537 test cases in this experiment. This number is lower than EvoSuite (183,795 test cases) and Randoop (29,536,928 test cases).

6.2 Test generation cost

One of the challenges in automated class integration testing is detecting the integration points between classes in a SUT. The number of code elements (*e.g.,* branches) that are related to the integration points increases with the complexity of the involved classes. Finding and testing a high number of integration code targets increases the time budget that we need for generating integration-level tests.

With CBC, the number of coupled branches to exercise is upper bounded to the cartesian product between the branches in the caller R and the callee E. Let B_R be the set of branches in R and B_E the set of branches in E, the maximum number of coupled branches $CB_{R,E}$ is $B_R \times B_E$. In practice, the size of $CB_{R,E}$ is much smaller than the upper bound as the target branches in the caller and callee are subsets of R and E, respectively. Besides, CBC is defined for pairs of classes and not for multiple classes together. This substantially reduces the number of targets we would incur when considering more than two classes at the same time.

While a fair amount of the test generation process can be automated, multiple instances of this approach can be executed simultaneously, and thereby, this approach can be used to generate test suites for a complete project at once in a reasonable amount of time. For instance, in this study, we managed to test each of the 140 class pairs with Cling for 20 times in less than a day thanks to a parallelization of the executions.

Finally, we have used a five minutes time budget to test each class pair's interactions. Since Cling considers each coupled branch as an objective for the search process, we could have defined a different search budget per pair, depending on the number of objectives. Similarly to EVOSUITE and RANDOOP, the outcome of CLING may differ depending on the given time budget. Defining the best trade-off between the search-budget and effectiveness of the tests generated using CLING is part of our future work.

6.3 Effectiveness

To answer **RQ2**, we analyzed the set of mutants that are killed by CLING (integration tests), but not by the unit and random test suites for the caller and callee separately (boxes labeled with T_{EvoE+R} and T_{Ran} in Figure 9). The test suite T_{CLING} was generated using a search budget of five minutes. Similarly, the unit-level suites were generated with a search budget of five minutes for each caller and callee class separately. Therefore, the total search budget for unit test generation (T_{EvoE+R}) is twice as large: 10 minutes, which corresponds to the time budget allocated to RANDOOP (T_{Ran}) as it generates random tests for both the caller and the callee. Despite the larger search budget spent on unit and random testing, there are still mutants and faults detected only by CLING and in less time. It is worth mentioning that, theoretically, all of these approaches might capture these failures with an infinite time budget. The point is that Cling can capture these failures faster, thanks to the CBC criterion.

The CBC criterion and its implementation in CLING are not an alternative to unit or random testing. In fact, integration test suites do not subsume unit-level and random suites as the different types of suites focus on different aspects of the system under test. Our results (**RQ2**) confirm that integration and unit and random testing are complementary. Indeed, some mutants can be killed exclusively by unit or random test suites: *e.g.*, the overall mutation scores for the unit tests T_{EvoE} and T_{EvoR}, and random tests T_{Ran} are larger than the overall mutation scores of CLING. This higher mutation score is expected due to the larger branch coverage achieved by the unit and random tests (*i.e.*, coverage is a necessity but not a sufficient condition to kill mutant).

Instead, the CBC criterion and its implementation in CLING focuses on a subset of the branches in the units (caller and callee), but target the integration between them more extensively. In other words, the search is less broad (fewer branches), but more in-depth (the same branches are covered multiple times within different pairs of coupled branches). This more in-depth search allows killing mutants that could not be detected by satisfying unit-level criteria.

Furthermore, our results in **RQ3** indicate that CBC and its implementation in CLING steer us toward finding bugs that are not detectable by other tests. In Section 5.5, we have shown that the tests generated by CLING capture exceptions, which are not detectable by unit or random tests. We have carefully performed an extensive manual analysis on these stack traces to identify whether they expose software faults. According to this manual analysis, we have detected 25 failures. Finally, for external confirmation, we have investigated if these 25 faults are identified and fixed by developers in the subsequent commits. The results of our investigation have confirmed that developers have actually fixed some of the faults in the following commits. To demonstrate the impact of the CLING in finding bugs, we have presented an example in Section 5.5. Moreover, the other faults, which were confirmed by our investigation, are available in our replication package [67]. While our evaluation pointed to 25 real faults, we have not yet applied CLING in a live setting in a currently active project. Doing so requires a project that does intensive (unit) testing already, and whose developers are interested in exploring issues raised by tests dedicated to exercising various inter-class interactions. As part of our future work, we intend to set up and conduct such a (longitudinal) study.

7 THREATS TO VALIDITY

Internal validity. Our implementation may contain bugs. We mitigated this threat by reusing standard algorithms implemented in EVOSUITE, a widely used state-of-the-art unit test generation tool. And by unit testing the different extensions (described in Section 3.3) we have developed.

To take the randomness of the search process into account, we followed the guidelines of the related literature [76] and executed CLING, EVOSUITE, and RANDOOP 20 times to generate the different test suites (T_{CLING}, T_{EvoE}, T_{EvoR}, T_{RanE}, and T_{RanR}) for the 140 caller-callee classes pairs. We have described how we parametrize CLING, EVOSUITE, and RANDOOP in Sections 3.2 and 4. We left all other parameters to their default value, as suggested by related literature [52], [77], [78].

External validity. We acknowledge that we report our results for only five open-source projects. However, we recall here their diversity and broad adoption by the software engineering research community. We also did not use the latest version of those five projects. On the one hand, it prevented us from reporting potential faults to the developers, which could have provided anecdotal evidence of the capability of the approach to find faults, but would have not provided any information in case of a rejection of a pull request by the developers. On the other hand, it allowed us to investigate the history of the code base and identify whether developers fix these faults, which were identified in our study, in the further commits. Additionally, considering the broad adoption of the projects by the software engineering community, it enables comparisons with the state-of-the-art and future approaches.

Construct validity. The identification and analysis of the integration faults done in **RQ3** have been performed by the first and second authors independently. The subsequent code history analysis and categorization have been performed by the three first authors independently. Each documented analysis was reviewed by one of the other authors involved.

Reproducibility. We provide CLING as an open-source publicly available tool as the data and the processing scrips used to present the results of this paper.[7] Including the subjects of our evaluation (inputs) and the produced test cases (outputs). The full replication package has been uploaded on Zenodo for long-term storage [67].

[7] https://github.com/STAMP-project/Cling-application

8 Conclusion and future work

In this paper we have introduced a testing criterion for integration testing, called the *Coupled Branches Coverage* (CBC) criterion. Unlike previous work on class integration testing focusing on (costly) data-flow analysis, CBC relies on a (lighter) control flow analysis to identify couples of branches between a caller and a callee class that are not trivially executed together, resulting in a lower number of test objectives.

Previous studies have introduced many automated unit and system-level testing approaches for helping developers to test their software projects. However, there is no approach to automate the process of testing the integration between classes, even though this type of testing is one of the fundamental and labor-intensive tasks in testing. To automate the generation of test cases satisfying the CBC criterion, we defined an evolutionary-based class integration testing approach called CLING.

In our investigation of 140 branch pairs, collected from 5 open source Java projects, we found that CLING has reached an average CBC score of 49.1% across all classes, while for some classes we have reached 90% coverage. More tangibly, if we consider mutation coverage and compare automatically generated random and unit tests with automatically generated integration tests using the CLING approach, we find that our approach allows to kill 7.7% (resp. 13%) of mutants per class that cannot be killed by tests generated with EVOSUITE (resp. RANDOOP). Finally, we identified 25 faults causing system crashes that could be evidenced only by the generated class-integration tests.

The results indicate a clear potential application perspective, more so because our approach can be incorporated into any integration testing practice. Additionally, CLING can be applied in conjunction with other automated unit and system-level test generation approaches in a complementary way.

From a research perspective, our study shows that CLING is not an alternative for unit or random testing. However, it can be used for complementing unit testing for reaching higher mutation coverage and capturing additional crashes which materialize during the integration of classes. These improvements of CLING are achieved by the key idea of using existing usages of classes in calling classes in the test generation process.

For now, CLING only tests the call-coupling between classes. In our future work, we will extend our approach to explore how other types of coupling between classes (*e.g.*, parameter coupling, shared data coupling, and external device coupling) can be used to refine the couples of branches to target. Indeed, our study indicates that despite the effectiveness of CLING in complementing unit tests, lots of objectives (coupled branches) remain uncovered during our search process. Hence, in future studies, we will enhance the detection of infeasible branches to remove them from the search objectives and perform a fitness landscape analysis of the search process to identify potential bottlenecks.

Finally, this study mostly focuses on examining the results of this approach on coupled branches coverage, mutation coverage, and detected faults. In our future work, we will explore how CLING can be effectively integrated with a development lifecycle (for instance, in a continuous integration process) and how automatically generated class integration tests can help developers to detect potential faults and debug their software.

In this study, we have evaluated CLING against state-of-the-art test generation tools (*i.e.*, EVOSUITE and RANDOOP). In our future work, we would like to compare the tests generated by CLING with the manually written tests. Also, since CBC is a new criterion, we aim to perform another study to investigate how well the class integration tests written by developers cover CBC targets.

Finally, since this paper is the first step toward generating class integration tests, we only collected the call-sites from the static analysis. However, a dynamic analyzer is able to detect more call-sites, and thereby CLING can generate more tests that cover class interactions that can only be identified dynamically.

Acknowledgments

This research was partially funded by the EU Project STAMP ICT-16-10 No.731529, the EU Horizon 2020 H2020-ICT-2020-1-RIA "COSMOS" project (No.957254), and the NWO Vici project "TestShift" (No. VI.C.182.032).

References

[1] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering," ACM Computing Surveys, vol. 45, no. 1, pp. 1–61, nov 2012.

[2] P. McMinn, "Search-based software test data generation: A survey," Software Testing Verification and Reliability, vol. 14, no. 2, pp. 105–156, 2004.

[3] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 2017, pp. 263–272.

[4] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for test suite generation," in Symposium on Search Based Software Engineering (SSBSE '17), ser. LNCS, T. Menzies and J. Petke, Eds., vol. 10452. Cham: Springer International Publishing, 2017, pp. 33–48.

[5] A. Panichella, F. M. Kifetew, and P. Tonella, "A large scale empirical comparison of state-of-the-art search-based test case generators," Information and Software Technology, vol. 104, pp. 236–256, 2018.

[6] G. Fraser and A. Arcuri, "1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite," Empirical Software Engineering, vol. 20, no. 3, pp. 611–639, 2015.

[7] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges," Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, pp. 201–211, 2016.

[8] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella, "Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency," ACM Trans. Softw. Eng. Methodol., vol. 25, no. 1, pp. 5:1–5:38, 2015.

[9] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall, "The impact of test case summaries on bug fixing performance: An empirical investigation," Proceedings - International Conference on Software Engineering, vol. 14-22-May-2016, pp. 547–558, 2016.

[10] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl, "The risks of coverage-directed test case generation," IEEE Transactions on Software Engineering, vol. 41, no. 8, pp. 803–819, 2015.

[11] A. Schwartz, D. Puckett, Y. Meng, and G. Gay, "Investigating faults missed by test suites achieving high code coverage," Journal of Systems and Software, vol. 144, pp. 106–120, 2018.

[12] Y. Wei, B. Meyer, and M. Oriol, "Is branch coverage a good measure of testing effectiveness?" in Empirical Software Engineering and Verification. Springer, 2012, pp. 194–212.

[13] Z. Jin and A. J. Offutt, "Coupling-based criteria for integration testing," Software Testing, Verification and Reliability, vol. 8, no. 3, pp. 133–154, 1998.

[14] A. Offutt, A. Abdurazik, and R. Alexander, "An analysis tool for coupling-based integration testing," in Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000. IEEE Comput. Soc, 2000, pp. 172–178.

[15] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su, "A Survey on Data-Flow Testing," ACM Computing Surveys, vol. 50, no. 1, pp. 1–35, 2017.

[16] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," SIGSOFT Softw. Eng. Notes, vol. 19, no. 5, pp. 154–163, Dec. 1994.

[17] A. Souter and L. Pollock, "The construction of contextual def-use associations for object-oriented systems," IEEE Transactions on Software Engineering, vol. 29, no. 11, pp. 1005–1018, 2003.

[18] R. Alexander and A. Offutt, "Criteria for testing polymorphic relationships," in Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000. IEEE Comput. Soc, 2000, pp. 15–23.

[19] R. T. Alexander, J. Offutt, and A. Stefik, "Testing coupling relationships in object-oriented programs," Software Testing, Verification and Reliability, vol. 20, no. 4, pp. 291–327, 2010.

[20] M. Vivanti, A. Mis, A. Gorla, and G. Fraser, "Search-based data-flow test generation," in 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2013, pp. 370–379.

[21] G. Denaro, A. Margara, M. Pezze, and M. Vivanti, "Dynamic Data Flow Testing of Object Oriented Systems," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1. IEEE, may 2015, pp. 947–958.

[22] G. Denaro, A. Gorla, and M. Pezzè, "Contextual Integration Testing of Classes," in Fundamental Approaches to Software Engineering (FASE '08), ser. LNCS, vol. 4961. Springer, 2008, pp. 246–260.

[23] A. Scott, "Building object applications that work, your step-by-step handbook for developing robust systems using object technology," 1997.

[24] G. Fraser and A. Arcuri, "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, ser. ESEC/FSE '11. ACM, 2011, pp. 416–419.

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-Directed Random Test Generation," in 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007, pp. 75–84.

[26] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri, "An empirical evaluation of evolutionary algorithms for unit test suite generation," Information and Software Technology, vol. 104, pp. 207–235, 2018.

[27] M. Soltani, A. Panichella, and A. van Deursen, "A guided genetic algorithm for automated crash reproduction," in 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). IEEE, 2017, pp. 209–220.

[28] F. E. Allen, "Control flow analysis," ACM SIGPLAN Notices, vol. 5, no. 7, pp. 1–19, jul 1970.

[29] Y. Wang, M. Wen, R. Wu, Z. Liu, S. H. Tan, Z. Zhu, H. Yu, and S.-C. Cheung, "Could i have a stack trace to examine the dependency conflict issue?" in 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). IEEE, 2019, pp. 572–583.

[30] Z. Wang, B. Li, L. Wang, M. Wang, and X. Gong, "Using Coupling Measure Technique and Random Iterative Algorithm for Inter-Class Integration Test Order Problem," in 2010 IEEE 34th Annual Computer Software and Applications Conference Workshops, vol. 1. IEEE, 2010, pp. 329–334.

[31] M. Steindl and J. Mottok, "Optimizing Software Integration by Considering Integration Test Complexity and Test Effort," in Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems. IEEE, 2012, pp. 63–68.

[32] N. Hashim, H. Schmidt, and S. Ramakrishnan, "Test Order for Class-based Integration Testing of Java Applications," in Fifth International Conference on Quality Software (QSIC'05), vol. 2005. IEEE, 2005, pp. 11–18.

[33] S. R. Vergilio, A. Pozo, J. C. G. Árias, R. da Veiga Cabral, and T. Nobre, "Multi-objective optimization algorithms applied to the class integration and test order problem," International Journal on Software Tools for Technology Transfer, vol. 14, pp. 461–475, 2012.

[34] P. Bansal, S. Sabharwal, and P. Sidhu, "An investigation of strategies for finding test order during Integration testing of object Oriented applications," in 2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS). IEEE, 2009, pp. 1–8.

[35] S. Jiang, M. Zhang, Y. Zhang, R. Wang, Q. Yu, and J. W. Keung, "An Integration Test Order Strategy to Consider Control Coupling," IEEE Transactions on Software Engineering, vol. 47, no. 7, pp. 1350–1367, 2021.

[36] L. Borner and B. Paech, "Integration Test Order Strategies to Consider Test Focus and Simulation Effort," in 2009 First International Conference on Advances in System Testing and Validation Lifecycle. IEEE, 2009, pp. 80–85.

[37] T. Mariani, G. Guizzo, S. R. Vergilio, and A. T. Pozo, "Grammatical Evolution for the Multi-Objective Integration and Test Order Problem," in Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16. ACM Press, 2016, pp. 1069–1076.

[38] G. Guizzo, G. M. Fritsche, S. R. Vergilio, and A. T. R. Pozo, "A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem," in Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15. ACM Press, 2015, pp. 1343–1350.

[39] A. Abdurazik and J. Offutt, "Using Coupling-Based Weights for the Class Integration and Test Order Problem," The Computer Journal, vol. 52, no. 5, pp. 557–570, 2009.

[40] R. da Veiga Cabral, A. Pozo, and S. R. Vergilio, "A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem," in IFIP International Conference on Testing Software and Systems, ser. ICTSS 2010, vol. 6435 LNCS. Springer, 2010, pp. 16–29.

[41] L. Briand, Y. Labiche, and Yihong Wang, "An investigation of graph-based class integration test order strategies," IEEE Transactions on Software Engineering, vol. 29, no. 7, pp. 594–607, 2003.

[42] S. Ali Khan and A. Nadeem, "Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Evolutionary Approaches," in 2013 10th International Conference on Information Technology: New Generations. IEEE, 2013, pp. 369–374.

[43] S. A. Khan and A. Nadeem, "Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Particle Swarm Optimization (PSO)," in Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, ICGEC 2013, ser. Advances in Intelligent Systems and Computing, J.-S. Pan, P. Krömer, and V. Snášel, Eds., vol. 238. Cham: Springer International Publishing, 2014, pp. 115–124.

[44] B. Danglot, O. Vera-Perez, Z. Yu, A. Zaidman, M. Monperrus, and B. Baudry, "A snowballing literature study on test amplification," Journal of Systems and Software, vol. 157, p. 110398, 2019.

[45] A. Arcuri, "RESTful API automated test case generation with Evomaster," ACM Transactions on Software Engineering and Methodology, vol. 28, no. 1, pp. 1–37, 2019.

[46] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in Presented as part of the 21st USENIX Security Symposium (USENIX Security 12). USENIX, 2012, pp. 445–458.

[47] M. Beyene and J. H. Andrews, "Generating string test data for code coverage," in 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, 2012, pp. 270–279.

[48] D. Coppit and J. Lian, "Yagg: an easy-to-use generator for structured test inputs," in Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. ACM, 2005, pp. 356–359.

[49] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in ACM Sigplan Notices, vol. 43, no. 6. ACM, 2008, pp. 206–215.

[50] R. Padhye, C. Lemieux, M. Papadakis, and Y. L. Traon, "Semantic Fuzzing with Zest," in Proceedings ofthe 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '19), Beijing, China, 2019, pp. 329–340.

[51] C. Baier and J.-P. Katoen, Principles of model checking. MIT Press, 2008.

[52] A. Panichella, F. M. Kifetew, and P. Tonella, "Reformulating Branch Coverage as a Many-Objective Optimization Problem," in Proceedings of the International Conference on Software Testing, Verification and Validation (ICST). IEEE, 2015, pp. 1–10.

[53] S. Jan, A. Panichella, A. Arcuri, and L. Briand, "Search-based multi-vulnerability testing of xml injections in web applications," Empirical Software Engineering, vol. 24, pp. 3696–3729, 2019.

[54] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," IEEE Transactions on Software Engineering, vol. 44, no. 2, pp. 122–158, 2018.

[55] U. Rueda, R. Just, J. P. Galeotti, and T. E. Vos, "Unit testing tool competition: round four," in 2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST). IEEE, 2016, pp. 19–28.

[56] A. Panichella and U. R. Molina, "Java unit testing tool competition-fifth round," in 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST). IEEE, 2017, pp. 32–38.

[57] U. R. Molina, F. Kifetew, and A. Panichella, "Java unit testing tool competition-sixth round," in 2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST). IEEE, 2018, pp. 22–29.

[58] F. Kifetew, X. Devroey, and U. Rueda, "Java unit testing tool competition: seventh round," in Proceedings of the 12th International Workshop on Search-Based Software Testing. IEEE Press, 2019, pp. 15–20.

[59] X. Devroey, S. Panichella, and A. Gambi, "Java Unit Testing Tool Competition - Eighth Round," in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops. ACM, 2020, pp. 545–548.

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

[61] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014. ACM Press, 2014, pp. 437–440.

[62] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset," Empirical Software Engineering, vol. 22, no. 4, pp. 1936–1964, Aug. 2017.

[63] M. Martinez and M. Monperrus, "Astor: A program repair library for java," in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016, pp. 441–444.

[64] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in Proceedings of the 39th International Conference on Software Engineering. IEEE Press, 2017, pp. 609–620.

[65] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016, pp. 177–188.

[66] T. B. Noor and H. Hemmati, "A similarity-based approach for test case prioritization using historical failure data," in 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE). IEEE, 2015, pp. 58–68.

[67] P. Derakhshanfar and X. Devroey, "Replication package of generating class-level integration tests using call site information," Dec. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.4300633

[68] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "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. ACM, 2014, pp. 654–665.

[69] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in Proceedings of the 27th international conference on Software engineering. ACM, 2005, pp. 402–411.

[70] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for Java (demo)," in Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016, pp. 449–452.

[71] Q. Zhu, A. Panichella, and A. Zaidman, "A systematic literature review of how mutation testing supports quality assurance processes," Softw. Test. Verification Reliab., vol. 28, no. 6, 2018. [Online]. Available: https://doi.org/10.1002/stvr.1675

[72] S. García, D. Molina, M. Lozano, and F. Herrera, "A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization," Journal of Heuristics, vol. 15, no. 6, pp. 617–644, Dec. 2009.

[73] N. Japkowicz and M. Shah, Evaluating Learning Algorithms: A Classification Perspective. Cambridge University Press, 2011. [Online]. Available: https://books.google.com/books?id=VoWIIOKVzR4C

[74] A. Panichella, "A systematic comparison of search-based approaches for lda hyperparameter tuning," Information and Software Technology, vol. 130, p. 106411, 2021.

[75] M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," IEEE Transactions on Software Engineering, vol. 27, no. 3, pp. 228–247, 2001.

[76] A. Arcuri and L. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," Software Testing, Verification and Reliability, vol. 24, no. 3, pp. 219–250, 2014.

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

[78] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri, "Random or evolutionary search for object-oriented test suite generation?" Software Testing, Verification and Reliability, vol. 28, no. 4, p. e1660, jun 2018.