Instance Generator and Problem Representation to Improve Object Oriented Code Coverage

Abdelilah Sakti, Gilles Pesant, and Yann-Gaël Guéhéneuc

Abstract—Search-based approaches have been extensively applied to solve the problem of software test-data generation. Yet, test-data generation for object-oriented programming (OOP) is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code. To address this problem we present a new automated search-based software test-data generation approach that achieves high code coverage for unit-class testing. We first describe how we structure the test-data generation problem for unit-class testing to generate relevant sequences of method calls. Through a static analysis, we consider only methods or constructors changing the state of the class-under-test or that may reach a test target. Then we introduce a generator of instances of classes that is based on a family of means-of-instantiation including subclasses and external factory methods. It also uses a seeding strategy and a diversification strategy to increase the likelihood to reach a test target. Using a search heuristic to reach all test targets at the same time, we implement our approach in a tool, JTExpert, that we evaluate on more than a hundred Java classes from different open-source libraries. JTExpert gives better results in terms of search time and code coverage than the state of the art, EvoSuite, which uses traditional techniques.

Index Terms—Automatic Test Data Generation, Search Based Software Testing, Unit Class Testing, Seeding Strategy, Diversification Strategy, Java Testing.

1 Introduction

Software testing is a time consuming and tedious process, accounting for more than 50% of the total software cost [32]. The most expensive part of testing is test-data generation. Automating this generation is particularly challenging for unit-class testing due to the dynamic nature and the complex features of object-oriented programming languages (OOP), e.g., abstraction, encapsulation, and visibility. These features prevent the direct access to some parts of the source code and impose indirect accesses to such parts through accessible methods, e.g., public methods [8]. This problem of source-code accessibility leads to a decrease in code coverage.

To address the accessibility problem, a test-data generator must performs three actions: (1) instantiate the class-under-test (CUT) and all other required classes; (2) perform a sequence of method calls to put the instance of the CUT in a desired state (i.e., a state that may help to reach the test target); and, (3) call a method that may reach the test target. The problem is in finding an adequate combination of these three actions to represent a test data to reach a given test target. Solving this problem faces three difficulties: ($D1$) finding an adequate instance of the CUT and of each required object; ($D2$) finding an adequate sequence of method calls to put the instance of the CUT in a desired state; and, ($D3$) finding an adequate method to reach the test target.

Many approaches have been proposed to address the problem of automating test-data generation and that fall in Search Based Software Testing (SBST). SBST has been successfully applied to solve the problem of test-data generation [13], [28], [46]. SBST translates the test-data generation problem into a search problem by providing an abstract representation of a feasible solution to the original problem and by searching an actual solution using a search heuristic guided by a fitness function. SBST represents the problem of test-data generation for procedural programming by a vector of input data. When considering the problem of test-data generation for OOP, SBST must take into account the state of the objects. Before invoking a method that may reach a test target, an instance of the CUT and a sequence of method calls are required to put the CUT in a desirable state to reach a test target. Therefore, the search problem consists of finding a means to instantiate the CUT (D1), a sequence of method calls (D2+D3), and an instantiation for each required argument (D1). To address the difficulties of test-data generation for unit-class testing, the most widely used approach is exploring the whole search space of D1 and a reduced search space of D2+D3 that bounds the length of sequences of method calls [7], [13], [37], [46], [50], [51]. Initially, such an approach randomly generates instances of classes and sequences of method calls. To generate a sequence of method calls, a length ℓ_r bounded by a constant L is randomly generated, where L is specified by the user or chosen automatically by the approach, then an algorithm iteratively and randomly selects an accessible method until the sequence length reaches ℓ_r. Instances of classes and sequences of method calls are evolved or regenerated until a stopping condition is reached. With such an approach, the search space is large because of four reasons: (1) there is no restriction on methods to call; (2) there is no restriction on the length of sequences of method calls; (3) the order of method calls is undefined; (4) the possible instances of a CUT or an argument may be "unlimited".

Because any class may have an "unlimited" number of possible instances, the search space of D1 is very large. Splitting such search space into some subspaces and generating required instances from different subspaces may increase the likelihood to reach a test target. Using different means-of-instantiation (e.g., constructors, factory methods, subclasses) when generating instances of classes can split the search space, generate diversified instances, and speed up the automated test-data generation process. Further, as shown in previous work [1], [2], [14], [30], a seeding strategy may help in solving the test-data generation problem efficiently.

The search space of D2 and D3 may be reduced by selecting sequences from a subset of methods that contains only methods relevant to the test target. A method is relevant to D2 iff it may change the state of an instance of the CUT and it is relevant for D3 iff it may reach the test target. A static analysis is required to identify the set of relevant methods for each test target. The representation of the problem of test-data generation for OOP must be restructured to take into account the results of the static analysis.

We propose a search-based approach for object-oriented test-data generation that addresses D1, D2, and D3. We focus on the branch coverage criterion and Java language, although the approach can be extended to other coverage criteria and OOP languages.

To solve D1, we hypothesize that diversifying the needed instances of classes and seeding constants extracted from the source code when instantiating classes may significantly ease the search. Thus, our approach uses a customized instance generator that is based on three main components:

1) Means-of-instantiation generator: It finds and prepares the different means-of-instantiation existing in the classpath. It is responsible for splitting the search space of D1 into subspaces (i.e., a set of possible instances that can be generated using a same means-of-instantiation) where each subspace is represented by a different means-of-instantiation.
2) Diversification strategy: It generates the needed instances of a given class by using different means-of-instantiations. The number of reuses of a same means-of-instantiation depends on its complexity. The proposed diversification strategy computes a representative complexity measure for each means-of-instantiation. Initially, it supposes that any class requires a constant complexity to be instantiated and a means-of-instantiation requires the total complexity of its arguments, then it dynamically adjusts this measure at each attempt of instantiation. Thus, our approach can use a probabilistic algorithm to select a means-of-instantiation with low complexity without compromising the diversity of generated instances of a class.
3) Seeding strategy: For each primitive data type or string, it collects constants from the source code, generates new values, then seeds them while generating data. It defines a seeding probability for each data type and each constant according to the number of collected occurrences of the constants. Also, it seeds the `null` constant with a constant probability while generating instances of classes.

The proposed instance generator allows our approach to better explore the search space, reaching more test targets; thus increasing code coverage in less time.

To solve D2 and D3, we hypothesize that selecting methods from the set of methods that may change the state of an instance of the CUT and the set of methods that may reach the test target is likely to generate relevant sequences of method calls. The approach uses a novel representation of the test-data generation problem for unit class testing of OOP to explore only these relevant sequences of method calls. It uses a static analysis to determine the set of methods that are likely to change the state of a given data member and the set of accessible methods that contain a path to reach a test target. The proposed representation allows our approach to avoid the exploration of useless sequences and thus generate tests faster with more coverage.

Using the collected information, the test-data generation problem is represented by a vector composed of relevant means-of-instantiations of the CUT, methods that are likely to change the object state by changing a data member, and methods that may reach the test target. Thus, the approach represents a test-data by: (1) a means-of-instantiation of the CUT (i.e., it can be a constructor, a method factory, a data field, or an external method from the CUT); (2) a sequence of method calls whose length is bounded by the number of declared data members in the CUT, each method in a sequence being called to put a given data member in a relevant state; (3) a method call that is likely to reach the test target; (4) a means-of-instantiation for each needed argument.

The approach was implemented in a tool, JTExpert, that takes as inputs a classpath and a Java source file or a directory that contains Java source files to be tested. It automatically produces a set of JUnit [24], [44] test cases (i.e, a Java source file that defines a method for each test case) for every Java file under test. It is completely automated, works on ordinary Java classes, and does not need additional information.

JTExpert was evaluated on more than 100 classes from different open-source libraries (Joda-Time, Barbecue, Commons-lang, and Lucene), most of which present a challenge for test-data generation approaches. Using JaCoCo tool [22], the test-data set to cover all-branches generated by JTExpert is compared to the one generated by an existing test generation approach, EvoSuite [13]. The results of this comparison shows that JTExpert is more effective than EvoSuite because it reaches a higher code coverage while it requires less time.

The contributions of this paper are then as follows:

I) We propose a customized **instance generator** of classes that is based on a means-of-instantiation generator, a seeding strategy, and a dynamic strategy to diversify generated instances.

II) We propose a **novel representation** of the test-data generation problem for unit class testing of OOP.

III) We describe an **implementation** of our test-data generation approach, JTExpert.

IV) We report an **empirical evaluation** of JTExpert on more than 100 classes from different open-source libraries and a comparison to EvoSuite.

The remainder of the paper is organized as follows: Section 2 summarizes related work. Section 3 describes our approach. Section 4 describes the implementation of our approach, JTExpert. Section 5 presents an experimental study comparing Evosuite with JTExpert. Section 6 concludes with future work.

2 Related Work

Our approach is related to work on automated test-data generation for unit class testing. Automatic test-data generation is an active research area. The last decade has witnessed an increased interest concerning object-oriented test-data generation.

2.1 Static Analysis

Korel proposed a dynamic data-flow analysis approach [26] for path coverage. His approach consists of analyzing the influence of each input variable on the successful sub-path traversed during program execution. During a genetic algorithm evolution, only input variables that influence the successful sub-path can have their values changed. Also, Harman et al. [21], [29] studied the impact of search-space reduction on search-based test-data generation. In their study, for a given test-data generation problem (e.g., a branch), they use a static-analysis approach described in [9], [20] to remove irrelevant input variables from the search-space. The analyses proposed in these approaches target procedural programming and focus on the arguments of a function under test, whereas our static analysis targets OOP and focuses on sequences of method calls.

To improve over SBST, Ribeiro et al. [42] use a purity analysis to reduce the size of the search space for SBST in OOP: all pure methods are discarded while generating the sequences of method calls, i.e., pure methods are considered irrelevant to the test case generation process. However, many impure methods may also be irrelevant; for example, a class containing a hundred public methods that are impure just because they invoke another impure public method m: the method m is the most relevant method and it may be the only relevant method if the hundred methods call m through a difficult branch to reach. Therefore, the purity analysis may reduce the search space but still needs additional analysis to refine the set of relevant methods. In contrast to purity analysis, the analysis proposed in this work considers only accessible methods that are in the roots of the tree of relevance. Further, the analysis proposed by Ribeiro et al. [42] differs from our analysis because it only generates a set of relevant methods for the CUT, whereas our analysis generates a set of relevant methods for each data member. Because a test target does not necessary depend on all data members, the relevant set of methods may be reduced according to a given test target.

2.2 Seeding Strategy

Many work proposes different seeding strategies [1], [2], [14], [17], [30], [31]: when a branch condition involves constants, covering such a branch may be challenging but it may become easier to reach if the set of constants exist in the source code used when generating instances of classes. Alshraideh and Bottaci [2] propose a seeding strategy and show that the seeding of string literals extracted from the source code during the generation of instances is better than a good fitness function. Alshahwan and Harman [1] propose a dynamic seeding strategy for Web applications by extracting constants constructed by the application as it executes. Fraser and Arcuri [14] study different seeding strategies and show that the use of an adequate strategy can significantly improve the SBST. To cover branches involving strings, McMinn and al. [30] propose a seeding strategy that extracts string literals from the results of Web queries.

The main difference among previous works is in using different sources of constants. All previous approaches use a constant probability of seeding and seed either primitive types or strings. Fraser and Arcuri [14] show that a constant probability equal to 0.2 gave best results compared to other values but it may be harmful in some cases. Indeed, if the extracted set of constants contains only one value, it is undesirable to have 20% of a population formed of the same value because it substantially reduces diversity. Therefore, it is necessary to use a seeding strategy with a variable probability.

Although the `null` constant is often present in the source code, previous work did not discuss the seeding of this constant, i.e., seeding the `null` constant with a certain probability during instance generation. It considers that constant as an ordinary instance that may have been unintentionally discarded from some types of classes, e.g., strings, or arrays. Hence, It may not use this constant enough to cover branches involving the `null` constant. Therefore, a systematic seeding strategy of the `null` constant must to be defined.

2.3 Test Data Generation Approach

Because random testing scales to large systems, random test-data generation is a widely used approach. It was explored in several works to generate test data that meets different test objectives [3], [12], [33], [34], [35], [36]. JTest [36] is a commercial tool that uses random testing to generate test data that meets structural coverage. Jartege [33] randomly generates test driver programs to cover Java classes specified in JML. RUET-J [4] is a tool that is as an enhanced version of Jartege which adds some features, such as the minimization of a test case suite. JCrasher [12] generates test data that is susceptible to detect program crashes by performing a random search. Eclat and RANDOOP [34], [35] use random search to create tests that are likely to expose fault. To boost the random search, these two tools use a dynamic pruning approach: they prune sequences of methods that violate some predefined contracts. All these works use random search without enough guidance, therefore they achieve low code coverage. In contrast to such tools, our approach does not use random search blindly but guides the generation of sequences of method calls and instances of arguments.

To improve over random testing, global and local search algorithms have been implemented in several ways. eToc [46] is a pioneering tool that uses genetic algorithms to generate test data that meet some structural criteria. It only deals with primitive types and strings and since its creation in 2004 it has not maintained to better exploit the strengths of recent testing approaches. EvoSuite [13], [16] is also a tool that automates test case generation by using genetic algorithms. Its objective is achieving a high code coverage with a small test suite. To achieve this objective, EvoSuite integrates recent state-of-the-art approaches.

As an alternative approach to SBST several approaches implement symbolic execution techniques [11], [25], such as JPF-symbc [38]. Such approaches are not scalable because they cannot deal with complex statements, native function call, or external libraries [39]. To overcome this limitation, dynamic symbolic execution [19], [43] uses concrete values that are extracted from an actual execution to simplify any complex constraint but may fail to generate test data for some test targets [39]. In addition both approaches have in common the exploration of all possible sequences of method calls (all paths), which faces the problem of combinatorial explosion because of the exponential growth of the search space [52].

Despite the large body of work on unit testing, previous work shares the use of a same problem representation that consists of generating sequences of method calls from the whole set of accessible methods. Also, the generation of instances of classes is always underestimated and random generation is usually used. As we will discuss in Section 5.3, there are some types of classes, such as sqlsheet[1]' classes, that require sophisticated algorithms to be instantiated.

Our approach differs from previous work in that it provides a formal expressive representation of the test-data generation problem that implicitly reduces the possible number of sequences of method calls. This is also the first approach to provide a probabilistic algorithm to generate a diversified set of needed instances of a class.

1. sqlsheet: An open source library that provides a JDBC driver for MS Excel. Available at https://code.google.com/p/sqlsheet/

```
public Iterator<K> keyIterator() {
  reap();
  final Iterator<IdentityWeakReference> iterator =
        backingStore.keySet().iterator();
  return new Iterator<K>() {
    private Object next = null;
    private boolean nextIsSet = false;
    public boolean hasNext() {
      ...
    }
    public K next() {
      ...
    }
    public void remove() {
      ...
    }
    private boolean setNext() {
      ...
    }
  };
}
```

Fig. 1: Skeleton of an anonymous class that can be instantiated, from class `org.apache.lucene.util.WeakIdentityMap`

3 Approach for Unit Class Testing

A test-data generation problem is an instantiation of the CUT and a sequence of method calls on that instance. The sequence of method calls can be split into two subsequences: (1) putting the CUT instance in an adequate state; (2) targeting the test target. Because the first subsequence aims to change the object state, we call it *state-modifier* methods and we call the second subsequence *target-viewfinder* methods because it aims to reach a test target. Thus, a representation of the test-data generation problem can be split into three main components: a *CUT-Instantiator*, a sequence of *state-modifier* methods, and a *target-viewfinder* method.

3.1 Instance Generator

In the object-oriented paradigm, generally, calling a constructor or a method requires some instances of classes. Given the large number of constructors and method calls needed for solving a testing problem, the generation of instances of classes requires a particular attention. It is key to successful test-data generation because without an adequate instance of a CUT or a needed objects the solving process may fail before it starts.

3.1.1 Means-of-instantiation

In our approach, we use the term *means-of-instantiation* to represent any means that allows generating an instance of the CUT or more generally to instantiate a class. A means-of-instantiation can be a constructor, a factory method, an accessible data member that is an instance of the required class, or a method defined in another class that returns an instance of the required class.

Means-of-instantiations can be categorized in two groups: internal and external. For a given class c, a means-of-instantiation is internal if it is offered by c itself, i.e., defined in c. A means-of-instantiation is external if it is defined in a class different from c.

To generate an instance of a given class our Instance Generator takes into account all accessible means-of-instantiation offered in a program and according to our diversification strategy one of them is selected. Thus, to instantiate a given class, our instance generator considers five different families of means-of-instantiation: (1) All accessible constructors (if there is no constructor defined the default one is considered); (2) all factory methods, i.e., all statics methods member returning an instance of that class; (3) all statics fields that are instances of the class; (4) all external methods that return an instance of that class, i.e., methods that return an instance of a needed class and are defined outside of that class; (5) recursively all means-of-instantiations of subclasses.

External Factory Methods and Anonymous Class Instantiation: In general, to instantiate a given class, only the internal means-of-instantiations are considered (i.e., constructors, factory methods, and fields). However, external factory methods, i.e., a method that returns an instance of a class and is defined outside of that class, may be a potential means not only for generating instances but also for diversifying the generated instances. Also, in some cases it may be the only solution to instantiate a class. For example, in Figure 1, an anonymous class defined at Line 4 is nested in the method `keyIterator`: there is a very weak likelihood to cover branches in that anonymous class without instantiating it as a CUT because all of its methods require an explicit call. One possible mean-of-instantiation of that anonymous class is the method `keyIterator`, which returns an instance of that class. Once that anonymous class is instantiated reaching its branches becomes a simple search problem.

An anonymous class is instantiable if and only if the method wherein it is declared returns its instance. Using such mean-of-instantiation, we can test an anonymous class separately and directly call all of its public methods.

Subclasses: Subclasses (stubs) are always used to instantiate abstract classes or interfaces but rarely to instantiate a class that offers some other means-of-instantiation (e.g., constructors). However, in some cases, using an instance of a subclass may be the only means to cover protected methods or methods implemented in abstract classes. Also, using means-of-instantiations offered by subclasses may significantly diversify generated instances, especially for arguments.

3.1.2 Diversification Strategy

To diversify the generated instances of a given class, we assume that its means-of-instantiations can split the search space into some subspaces where each means-of-instantiation is represented by a subspace. Generating required instances from different subspaces may increase the diversity and the likelihood to reach a test target. To enlarge the number of subspaces and have more diversity our Instance Generator takes into account all accessible means-of-instantiations offered in a program.

In the presence of different means-of-instantiation, choosing between them to generate a certain number of instances is problematic. For example, in the Joda-Time[2] library, most classes have more than a hundred means-of-instantiations, e.g., the class `org.joda.time.base.AbstractPartial` and the interface `org.joda.time.ReadablePartial` each have 225 different means-of-instantiations through their subclasses. The complexity of these means-of-instantiations vary greatly. For example, a mean-of-instantiation that does not need parameters is less complex than another that needs some instances of other classes, that needs an instance of a class that is difficult to instantiate, or, for accessibility reason, that is "not instantiable". In some cases, instantiating a class using one of its complex means-of-instantiations may be harmful, i.e., may negatively influence performance of a test-data generation search (e.g., a means-of-instantiation involves complex computation operations in its implementation and requires significant time to be executed).

To balance diversity and complexity our selection strategy favors less complex means-of-instantiations while diversifying generated instances. A probabilistic selection strategy is implemented for this propose that diversifies the generated instances without compromising performances. Such a selection strategy needs an evaluation of the complexity of a means-of-instantiation.

A given means-of-instantiation mi can be considered complex if it fails to generate an instance of a class because of one of the following reasons:

- mi involves complex computation;
- a precondition on its arguments is hard to satisfy;
- recursively one of its needed arguments is complex to instantiate.

To simplify the measurement of the complexity of a means-of-instantiation, we divide it into two complexities: the complexity to execute a means-of-instantiation and the complexity of instantiating its needed arguments. Initially, we suppose that the complexity to execute a means-of-instantiation or instantiating a class is constant and equal to a constant IC (Initial Complexity, equal to 20 in our approach). Formally an initial complexity of a means-of-instantiation is measured according to the following formula:

$$C^0_{mi} = (Nbr\_Arg + 1) \times IC$$

This expression uses the number of arguments as a measure to evaluate the complexity of a means-of-instantiation. Sometimes, preconditions or the complexity of instantiating an argument may make a mean-of-instantiation that needs only one argument more complex than another that requires many arguments.

2. Joda-Time: An open source library that provides a quality replacement for the Java date and time classes. Available at http://www.joda.org/joda-time/

To take this observation into consideration, we use the percentage of failure of generating instances to measure the complexity of a means-of-instantiation, i.e., we attempt to generate a number of instances using the same means-of-instantiation while observing the number of failures. Using such a computation, we obtain an accurate measure. However, evaluating the complexity of all means-of-instantiations before the search may be expensive. To simplify the computation, we measure the complexity of a means-of-instantiation on the fly while searching test data: initially, a complexity of a given means-of-instantiation mi is evaluated to C^0_{mi}, each time mi is called to generate an instance of its returned class, its complexity measure is updated based on failures.

Our updating of the complexity measures is based on a penalty system. We use two types of penalties:

- NON INSTANTIABLE PENALTY (NIP): this penalty is assigned to a class c if our instance generator could not generate an instance of c because c does not have any mean-of-instantiation. This penalty should be big enough to reduce the likelihood (to zero if possible) of the selection of a mean-of-instantiation that needs a non instantiable class. In this work, we use $NIP = 10^6$.
- FAILURE PENALTY (FP): this penalty describes the difficulty of executing a mean-of-instantiation. Every time a mean-of-instantiation could not generate an instance of its returned class, a failure penalty is added to its complexity measure. This may happen if at least one parameter does not satisfy the means-of-instantiation preconditions or time out of instantiation is reached. To allow other means to be selected, this penalty should be bigger than the most complex mean-of-instantiation. In this work, $FP = 10 \times IC$.

Finally, the complexity of a means-of-instantiation is measured, at a time t, as follows:

$$C^t_{mi} = C^0_{mi} + x^t_{mi} \times FP + y^t_{mi} \times NIP$$

where x^t_{mi} represents the number of failures of mi until t; y^t_{mi} represents the number of failures of mi caused by a non-instantiable argument until t.

Then it is possible to define a selection strategy based on the complexity measure to select means-of-instantiations with low complexity. Such a strategy always favors means-of-instantiations with low complexity, which may reduce the diversity of the generated instances. To balance complexity and diversity, we use a cost of diversity (DC_{mi}) that is determined by a penalty system. Each time a means-of-instantiation succeeds to generate an instance of a class, its diversity cost is increased by a constant DIVERSITY PENALTY (DP_{mi}). This constant is defined in term of complexity to keep an advantage for means-of-instantiations with low complexity. Therefore, each means-of-instantiation has its own DP_{mi}, which depends on its complexity. In our approach, DP_{mi} and DC_{mi} are computed as follows:

$$DP^t_{mi} = \begin{cases} 0 & mi \text{ fails} \\ C^t_{mi} - C^0_{mi} + IC & mi \text{ succeeds} \end{cases}$$

and:

$$DC^t_{mi} = \sum_{t' \leq t} DP^{t'}_{mi}$$

To instantiate a given class c, each means-of-instantiation in its set of possible means-of-instantiations Set_MI_c receives an instantiation probability proportionate to its own global cost value $GC^t_{mi} = C^t_{mi} + DC^t_{mi}$ and the total cost value of all other means-of-instantiations $TOTAL\_COST^t = \sum_{mi \in Set\_MI_c} GC^t_{mi}$. The instantiation probability to use a means-of-instantiation mi for instantiating c is determined according to the following formula:

$$p^t_{mi} = \frac{TOTAL\_COST^t - GC^t_{mi}}{(|Set\_MI_c| - 1) \times TOTAL\_COST^t}$$

This probability is used for the selection step through a roulette-wheel selection strategy.

3.1.3 Seeding Strategy of Constants

When a branch condition involves constants, covering such a branch may be challenging. To deal with such a problem, many works [1], [2], [14], [30] propose different seeding strategies, especially for strings and primitive types. We adopt a new seeding strategy, inspired by the works [1], [2], [14], [30] but with some differences: in addition to primitive and string, we seed also object constants, and the seeding probability is defined for each data type and each value in terms of the number of occurrences collected.

Seeding with a Variable Probability: In general, an instance generator seeds extracted constants with a fixed probability, i.e., for a primitive type or a string, a value is randomly selected from the set of extracted constants with a certain fixed probabilities. The study conducted by Fraser and Arcuri [14] shows that a probability equal to 0.2 gave best results compared to other values. It also shows that in some classes seeding can be harmful. Indeed, if the extracted set of integer constants contains only one value, it is undesirable to see 20% of a population formed of the same value because it substantially reduces diversity. We experimentally observed that using a seeding strategy with a fixed probability equal to 0.2 indeed affects the coverage negatively in some classes, especially if the number of extracted constants is small. Also, it is unbalanced to seed two constants with the same probability, when one is used a hundred times, whereas the other is used only once in the source code. Thus, each constant must have its own seeding probability according to the number of its occurrences in the source code.

We propose a variable seeding probability that is based on the number of occurrences of constants extracted from the source code. Empirically, we found that a probability equal to 0.05 is better than 0.2 if the number of extracted occurrences of constants is less than 10; otherwise, as in previous works, we use a probability equal to 0.2. Thus, a constant with a large number of occurrences in the source code has a higher likelihood of being selected. For example, if the vector of integer constants exacted from the source code is $\{\langle 5, 2, 4, 5, 5, 1, 2, 3, 4\rangle\}$ then the seeding probability to generate an integer is equal to 0.05 because the number of occurrences is less than ten and the probability to seed the value 5 is equal to $\frac{3}{9} \cdot 0.05$.

Seeding the `null` constant: In general, when constants are discussed, only strings and primitive types are considered, although any object may be a constant and this constant may be extracted from the source code (e.g., array constant is often present in the source code). We consider also the `null` constant. The `null` constant is often involved in a branch's condition that checks for a null object, i.e., it requires an equality between a variable and the `null` constant. This type of condition is difficult to satisfy, but it may become easier with a seeding strategy. In a OOP source code, there is often some branch involving the `null` constant (e.g., $object == null$). If such branches are forgotten, there is a high likelihood to generate null pointer exceptions. For example, in the library Apache Commons-Lang[3], in class `org.apache.commons.lang3.ArrayUtils`, among 1,096 branches 20% (170) have a predicate involving the `null` constant. When we tested this class using EvoSuite [13], [16], only 24 branches out of 170 were covered, i.e., 14% coverage of branches involving the `null` constant. This weak coverage does not mean that EvoSuite does not use the `null` constant at all, but it does not use this constant enough to cover branches involving the `null` constant. We think that EvoSuite does not use a systematic seeding strategy with the `null` constant: perhaps it uses the `null` constant sometimes with some classes or when it meets a difficult class to instantiate. However, to satisfy branches' conditions that involve the `null` constant, it is necessary to seed every class with this constant using an adequate seeding probability. In this work, our instance generator systematically seeds the `null` constant while generating instances of classes with a seeding probability equal to 0.05, i.e., for every one hundred of instances, five null instances are used. We chose a probability equal to 0.05 because we have only one value to seed.

3.2 A Representation of the Test-data Problem

To generate unit-test data using SBST techniques, the main component is the test-data problem representation. The key idea behind our representation of the test-data generation problem is in using a static analysis to determine relevant means-of-instantiations (CUT-Instantiator), state-modifier methods, and target-viewfinder methods, and then use them to generate test-data candidates.

3. Apache Commons-Lang provides extra methods for classes in the standard Java libraries. Available at http://commons.apache.org/proper/commons-lang/

```
public class A{
    private Map<String,Integer>Dm1;
    private double Dm2;
    public A(B b, C c, int i ) {···}
    public void setDm1(String s, int i ) {Dm1.put(s,i);}
    public void remDm1(int i) {Dm1.put(s,i);}
    public void setDm2(double d) {Dm2=d;}
    public mTV(C c, String s, int i) {
        ...
        mUT(c, "mTV");
        ...
    }
    private mUT(C c, String s) {
        ...
        //test target
        ...
    }
    ...
}
```

Fig. 2: Example of CUT

3.2.1 CUT-Instantiator

To reach a test target in a non-static method, an instance of the CUT is required. Different means can be used to generate that instance. If the test target is in a constructor or accessible only via a constructor, then we call this type of constructor *CUT-Instantiator*. For a given test target in a CUT, two reasons may make a constructor a CUT-Instantiator: (1) through it the test target is reachable or (2) through it a data member can be modified.

A means-of-instantiation is considered a CUT-Instantiator if and only if it contains the test target, contains a statement that modifies a data member, or calls an inaccessible method directly or transitively via inaccessible methods and the latter contains the test target or contains a statement that modifies a data member.

We denote the set of all CUT-Instantiator of a test target t in a CUT c by $MI_{c,t}$. If $MI_{c,t}$ is not empty then to generate potential instances of the CUT to reach t, only CUT-Instantiators in $MI_{c,t}$ are considered, otherwise all means-of-instantiations of the CUT are considered.

3.2.2 State-modifier Methods

Because of encapsulation, in general, the state of an instance is not directly accessible. To address this accessibility problem, in a test datum, a sequence of method calls is used to put an instance of a CUT in an adequate state by modifying some data members. Because the aim of ths sequence of method calls is changing the state of the CUT, instead of exploring random sequences of method calls as previous works, we focus on methods that may modify a data member.

To change the state of an instance of a CUT, we define *state-modifier* methods as all accessible methods that may directly or indirectly assign to some data members a new value, instantiate them or change their states by invoking one of their methods.

An accessible method is a state-modifier method if and only if it contains a statement that modifies a data member or it calls an inaccessible method directly or transitively via inaccessible methods and the latter contains a statement that modifies a data member.

For example, in Fig. 2, the methods $setDm1$ and $remDm1$ are state-modifier methods because they change the state of data member $Dm1$.

Generally, in a given class, not all methods are state-modifier. Thus, using the subset of state-modifier methods instead of all methods to generate the sequence of method calls may significantly reduce the number of possible sequences, i.e., the search space. We denote the set of all state-modifiers for the i^{th} declared data member in a class c by $SM_{c,i}$.

3.2.3 Target-viewfinder Methods

A test target is either in an accessible or an inaccessible method of a CUT. Thus, it may not be directly accessible. In general, in a test datum, the last method in the sequence of method calls is called in the hope to reach the test target. Because a test target is already known, instead of calling a method randomly, we focus only on methods that may reach the test target. A target-viewfinder method aims to reach the test target.

An accessible method is considered a target-viewfinder method if and only if it contains the test target or it calls an inaccessible method directly or transitively via inaccessible methods and the latter contains the test target.

For example, in Fig. 2, if we consider that the test target is reaching Line 15, then the accessible method mTV is a target-viewfinder because it is accessible and calls the inaccessible method mUT that contains the test target. We denote the set of all target-viewfinder methods for a given test target t in a CUT c by $TV_{c,t}$.

3.2.4 Static Analysis

For a CUT we use static analysis on the source code to extract its set of CUT-Instantiators, state-modifier methods for each data member, and target-viewfinder methods. To determine these three sets, static analysis identifies branches that modify the data members and execution paths ascending from a branch to an accessible method. A branch can modify a given data member dm if it contains a constructor call that assigns a new instance to dm; an assignment statement wherein dm is on the left side; or, a method call on dm. Thus, static analysis generates three types of information: (I1) information about all branch-modifiers for each data-member; (I2) information about the parent of each branch; (I3) information about all branch-callers for each inaccessible method. Using I1, I2, and I3, static analysis can generate the set of state-modifier methods and a subset of CUT-Instantiators. When a test target is defined, using I2 and I3, the set of CUT-Instantiators is completed and the set of target-viewfinder methods is generated.

3.2.5 Domain-vector

A domain-vector represents the general skeleton of potential test-data candidates. For a given test target, it is a vector composed of the set of CUT-Instantiators, followed by the set of state-modifier methods for each data member and ending with the set of target-viewfinder methods. Correspondingly, a solution is a vector that assigns to each component one instance from its domain with a fixed list of needed instances of arguments. For a nested class (i.e, local, member or anonymous class), its vector is extended with the parent class's vector. We fix an argument or input data by recursively defining its means-of-instantiations. Except for the target-viewfinder method, any component in a solution vector can take an $empty$ assignment. Also, the means-of-instantiation of CUT in a solution vector can be empty if and only if all methods in the CUT are static.

Thus, a presentation of a problem of test-data generation that aims to cover a target t in a class c is as follow:

$$\begin{aligned} DV_{c,t} \quad &\longmapsto \langle \{MI_{c,t} \cup \{empty\}\}, \{SM_{c,1} \cup \{empty\}\}, \\ &..., \{SM_{c,n} \cup \{empty\}\}, \{TV(c,t)\}\rangle \end{aligned}$$

For example, if the class A in Fig. 2 is under test and the test target is Line 15 and the constructor defined at Line 4 initializes a data member, then the domain-vector is defined as: $DV_{A,15} = \langle MI_{A,15}, SM_{A,1}, SM_{A,2}, TV_{A,15}\rangle$

with:

- $MI_{A,15} = \{A(B, C, int)\}$
- $SM_{A,1} = \{setDm1(String, int), remDm1(int), empty\}$
- $SM_{A,2} = \{setDim2(decimal), empty\}$
- $TV_{A,15} = \{mTV(C, String, int)\}$

In this example, a test data necessary uses the constructor $A(B, C, int)$ to instantiate the CUT and calls $mTV(C, String, int)$ to reach the test target. To change the state of the generated instance, a test datum calls the methods $setDm1(String, int)$ or $remDm1(int)$, then it calls the method $setDim2(decimal)$.

This example shows that the general form of a test datum is almost fixed by the domain vector; it remain only a search algorithm to find adequate arguments to reach the test target.

4 Implementation

We have implemented our approach in a tool, JTExpert, that takes as inputs a classpath and a Java file to be tested. JTExpert automatically produces a set of JUnit test cases [24], [44] that aims to cover all branches. JTExpert is completely automated, works on ordinary Java source-code (`.java`), and does not need additional information. An executable of JTExpert with all required libraries is available for download at https://sites.google.com/site/saktiabdel/JTExpert.

JTExpert performs two phases: a preprocessing phase and a test-data generation phase.

4.1 Preprocessing

The preprocessing phase consists of instrumenting and collecting required information for the second phase. To extract relevant information from the source code, we use static analysis as a first phase before starting test-data generation. The Java file under test is parsed and its Abstract Syntax Tree (AST) is generated. Before extracting data, each node in the AST representing a branch or a method is encoded with its parent branch or method into a unique code, i.e., an integer value.

We implement this phase in two main components: Instrumentor and Analyzers.

4.1.1 Instrumentor

To instrument the Java file under test, its AST is modified to call a specific method on entering each branch. This method call takes as inputs the branch code and notifies the testing process when the branch is executed. After the instrumentation, based on the AST, a new version of the Java file under test is saved and compiled to generate its new Java bytecode file (`.class`).

4.1.2 Analyzers

This component generate the problem representation described in the previous section. To extract the information needed for the problem representation and the instance generator, several explorations of the AST are performed: (1) for each method, we identify the set of all branches callers, i.e., branches wherein the method is called; (2) for each data member, we identify the set of all branches modifiers, i.e., branches wherein the data member is modified; (3) for strings and each primitive type, the set of constant values are saved.

To simplify the implementation of parsing the Java file under test and exploring the AST, we used the parser provided with Eclipse JDT [23]. JDT makes our static analysis easy because it allows creating an AST visitor for each required information.

4.2 Test Data Generation

The test-data generation phase is the core of JTExpert to find the set of test data that satisfies all-branch coverage criterion. We implement this phase into three main components: Instance Generator, Search Heuristic, and Test Data generator.

4.2.1 Instance Generator

This component implements the generation of means-of-instantiations, the seeding strategy, and the diversity strategy described in the previous section. Algorithm 1 presents the different steps of instance generation and complexity measurement. The algorithm distinguishes between three types of classes: Atomic, Container, and any other type of classes (simple classes). All primitive types, classes that encapsulate only primitive types, and the string class are considered Atomic. Each Atomic class has its own random generator that implements the seeding strategy described in the previous section, which uses the whole domain corresponding to the Atomic class (e.g., a short takes its value from the domain $[-2^{15}, 2^{15}-1]$). A Container is a standard way for grouping objects (e.g., List and Array), i.e., it is an object that can contain other objects, often referred to as its elements. All containers in Java are either simple arrays, classes derived from the `java.util.Collection` interface, or classes derived from the `java.util.Map` interface. The documentation of the Java Collections Framework[4] gives an exhaustive description of all containers that can be used in a Java program. The current version of JTExpert treats these three types of collections as containers, whereas any other collection that does not derive either from `java.util.Collection` or from `java.util.Map` is considered as a simple class. Containers have many particular features [7], [48]. Test data that involves a container should be generated in a different way than other objects. Instead of using means-of-instantiaitons to generate a container, we implement a random instance generator to generate some types of container: it randomly selects a bounded length then

Algorithm 1 Instance Generator

```
Input: $c$ is the class to be instantiated
Output: $i_c$ is an instance of $c$
1: if ($c$ is $Atomic$) then
2:     $i_c \leftarrow Atomic(c).getInstance()$
3: else
4:     if ($c$ is $Container$) then
5:         $i_c \leftarrow Container(c).getInstance()$
6:     else
7:         if ( $Set\_MI_c == \emptyset$ ) then
8:             $C_c \leftarrow NIP$
9:             return $ClassNotInstantiable$
10:        end if
11:        $im \leftarrow selectMeans()$
12:        for (each $p$ in $im.getParameters()$) do
13:            $i_p \leftarrow InstanceGenerator(p)$
14:            $params.add(i_p)$
15:        end for
16:        $i_c \leftarrow im.getInstance(params)$
17:        if ( $im.succeeds()$ ) then
18:            $DC^t_{mi} \leftarrow DC^{t-1}_{mi} + DP^t_{mi}$
19:        else
20:            $C^t_{mi} \leftarrow C^{t-1}_{mi} + FP$
21:            if ( $attempt < MAX\_ATTEMPT$) then
22:                $i_c \leftarrow InstanceGenerator(c)$
23:            end if
24:        end if
25:    end if
26: end if
27: return $i_c$
```

4. Java Collections Framework Overview: http://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html

Algorithm 2 Generator of Sequences of Method Calls

Input: DV domain vector; B branch to reach
Output: TDC a test data candidate

```
1: $TDC.object \leftarrow InstanceGenerator(CUT)$
2: $ntv \leftarrow DV.length - 2$
3: for ( $int\ i = 0; i < ntv; i++$ ) do
4:     $m \leftarrow$ randomly select a method from $SM_{CUT,i}$
5:     for (each $p$ in $m.getParameters()$) do
6:         $i_p \leftarrow$ InstanceGenerator(p)
7:         $params.add(i_p)$
8:     end for
9:     $TDC.methods.add(m, params)$
10:    new params
11: end for
12: $m \leftarrow$ randomly select a method from $TV_{CUT,B}$
13: for (each $p$ in $m.getParameters()$) do
14:     $i_p \leftarrow$ InstanceGenerator(p)
15:     $params.add(i_p)$
16: end for
17: $TDC.methods.add(m, params)$
```

it recursively calls Algorithm 1 to generate all needed elements. For any other class, at Line 11, the set of all its possible means-of-instantiations is generated and a mean-of-instantiation is selected. To generate this set, for a given class, we use the Java Reflection API [40] to get the means-of-instantiations offered by that class and the open-source library Reflections [41] to get subclasses and external factory methods. After getting the set of all possible means-of-instantiations, an instantiation probability is assigned to each means-of-instantiation according to its complexity and a roulette-wheel selection strategy is used to select a means-of-instantiation. Then, all needed arguments are recursively instantiated at Line 13. Finally, at Line 16, an attempt to generate an instance of the class is made, after which the complexity of the selected mean-of-instantiation is updated.

4.2.2 Search Heuristic

We combine the proposed approach with a random search. Although random search is the simplest among the search heuristics, it is largely used in software testing because it may reach a high level of coverage [3], [7], [12], [33], [34], [35], [36]. It also makes it easier to evaluate our proposal without having to isolate its effect from that of a more sophisticated search.

The common random search relies on generating a candidate solution vector randomly to execute the instrumented class and to reach the targeted branch. It stops either if the generated candidate solution vector executes the targeted branch or a stop condition is reached. Such an implementation may hang the search in difficult or unreachable branches [16].

In contrast, in JTExpert, a random search is implemented to target all uncovered branches at the same time: it does not focus only on one branch, instead it

Algorithm 3 Test Data Generation.

Input: U is the unit under test
Output: TD a set of test data (in JUnit Format)

```
1: $U' \leftarrow instrument(U)$
2: $A \leftarrow analyse(U')$
3: $T \leftarrow A.getBranches2Cover()$
4: while ( $T! = \emptyset$ && $!maxTime$ ) do
5:     $b \leftarrow$ randomly select a branch from T
6:     $dv \leftarrow A.getDV(b)$
7:     $tdc \leftarrow generateMethodsSequence(dv)$
8:     $execute(tdc)$
9:     if ($tdc.isTestData()$) then
10:        $TD \leftarrow TD \cup tdc$
11:        $T.remove(tdc.getCoveredBranches())$
12:    end if
13:    $T.remove(b)$
14:    if (b $\notin$ tdc.getCoveredBranches()) then
15:        $WT.add(b)$
16:    end if
17:    if ( $T == \emptyset$ ) then
18:        $T \leftarrow WT$
19:        $WT.clear()$
20:    end if
21: end while
22: $writeJUnitTestCases(TD)$
```

generates a candidate solution uniformly at random for every uncovered branch. This implementation is likely to reach a good branch coverage quickly because it does not waste efforts on unreachable branches and it benefits from the significant number of branches that may be covered fortuitously. Lines 4 to 21 in Algorithm 3 are a pseudo-code of the search algorithm implemented in JTExpert. Line 7 calls the generator of sequences of method calls that is presented in Algorithm 2. The later uses the domain-vector and the instance generator to guide the random search. In Algorithm 2, at Line 1, an instance of the CUT is generated using the instance generator and the set of CUT-Generators. Hence the implemented random heuristic benefits from all the features in our instance generator, e.g., the large number of means-of-instantiations. Lines 2 to 10 generate a sequence of state-modifier methods: at Line 4 a method is randomly selected from the current set of state-modifier methods, then all required instances of classes are generated in the loop for at Line 5. Finally, at Lines 12 to 16 a target-viewfinder method is generated.

The instance generator and the domain-vector are at the core of our search heuristic. The domain-vector guides the search by restricting the possible sequences of method calls. The instance generator guides the search by diversifying instances of classes.

4.2.3 Test Data generator

This component operates and coordinates other components to generate test data. It implements the skeleton of the whole process of test data generation. Algorithm 3 presents the different steps of this component to satisfy the all branch coverage criterion for a file under test. First, at Line 1, a file under test is instrumented and a new instrumented version of the file is generated. The file is analyzed at Line 2 to get all relevant information (e.g., constants) needed for the next steps. For each branch to be covered in the file, selected at Line 5, a domain vector dv that represents the problem is generated at Lines 6 by an analyzer A. Lines 7 and 8 represent a guided random generation of a test-datum candidate to reach the branch b. A test-datum candidate tdc is generated at Line 7 and executed at Line 8 using the Java Reflection API [40]. If this tdc covers either b or some uncovered branches, then it is inserted in the set of test data TD at Line 10 and all its covered branches are removed from the set of test targets T. At Line 13, b is removed from T. If b is not covered yet, then it is inserted in a waiting set of test targets WT at Line 15. When the search has run through all branches (i.e., T becomes empty), then a new round starts by initializing T with WT at Line 18. Finally, at Line 22, JDT is used to translate the set of test-data TD into a Java file that contains test cases in JUnit format.

5 Empirical Study

This section presents our evaluation of JTExpert for exploring restricted bounded method sequences, instantiating classes by using the proposed diversification and seeding strategies, and generating test data. We investigate the advantages and limitations of JTExpert by comparing it with EvoSuite [13], [16] that uses a genetic algorithm to generate test data, which starts with an initial population of chromosomes (i.e., a set of sequences of method calls) randomly generated from the set of all accessible methods. When an argument is needed, EvoSuite randomly selects a means-of-instantiation to generate an instance. In this study, we use version 20130905 of EvoSuite. We have performed the experiments on a Oracle Grid Engine comprising 42 similar nodes, each of them equipped with 2-dual core CPU 2.1 GHZ, 5GB of memory, a Fedora core 13 x86_64 as OS, and Java Development Kit 7.

5.1 Experimental Setup

5.1.1 Empirical Study Subjects

The study was performed on 115 of classes from four open-source libraries. All the selected CUTs have been previously used as benchmark to evaluate competing tools that participated to the SBST contest of the Java unit-testing tool version 2013 [49] which EvoSuite won [15]. Some of these classes were also used in evaluating μTest [18], which has become a part of EvoSuite [16]. The set of classes in this benchmark are carefully selected by the SBST contest committee of the Java unit-testing tool to represent different challenges of unit-testing. Most of those classes are complex for test generation [18]. For example, in the library Joda-Time, accessibility is a problem for several classes: only default access is possible, i.e., classes are visible only in their packages. Also, many classes are difficult to access because they are private and embedded as class members in other classes. Others are difficult to instantiate because they do not offer any accessible constructor and can only be instantiated through factory methods or accessible data members [18]. Such accessibility problems create the difficulties D1, D2, and D3 described in Section 1.

Table 1 lists the Java libraries that we used in our study. Each line presents one of the libraries while columns show the library names, numbers of Java files under test, numbers of classes, numbers of methods, numbers of branches, numbers of lines of code, and numbers of instructions. These metrics are computed at the byte-code level using the JaCoCo tool [22], so that no empty or comment line is included.

5.1.2 Procedure

In the SBST contest 2013 [49], tools were evaluated using two metrics: code coverage and mutation score. In this study, we evaluate the two tools, JTExpert and EvoSuite, only with code coverage because mutation score does not apply to our approach and the current version of JTExpert does not generate assertions to kill mutants.

For every class in the benchmark, we generate complete test suites to cover all branches with JTExpert and compare them with those generated by EvoSuite. Each tool applies on a different level of source code: EvoSuite applies on Java bytecode (`.class`), whereas JTExpert applies on Java source code (`.java`), which may generate differences in the observed coverage of each tool because (1) the Java compiler translates some instructions over boolean into conditional statements and (2) one conditional statement in the source code may be translated into many conditional statements if it contains many clauses. To compare the approaches needed, instead of the coverage measured by each tool, we use JaCoCo [22] to measure the coverage: JTExpert and EvoSuite generate test suites and JaCoCo takes them and measures their coverage, at the bytecode level, in terms of four metrics: method coverage, branch coverage, line coverage, and instruction coverage. Each search for test data that meets branch coverage for every CUT is performed 20 times. This repetition allows reducing any random aspect in the observed values. The 20 executions are performed using an identical set of seeds for random number generation. To make the experimentation scalable, for each execution, a maximum of 200 seconds is allowed per search including the instrumentation and preliminary analysis stages. If this time is spent by JTExpert or EvoSuite, the tool is asked to stop and, after a maximum of 5 minutes, its process is forced to stop (killed). We stop at 200 seconds because we observe empirically that, after this duration, the coverage progress of each tool becomes very slow. In total, this

Libraries	#Java Files	#Classes	#Methodes	#Branches	#Lines	#Instructions
Joda-Time	50	87	1,411	3,052	5,876	25,738
Barbecue	18	18	161	323	1,041	14,558
Commons-lang	5	6	366	1,942	2,134	9,139
Lucene	2	4	58	202	262	1,364
All	**75**	**115**	**1,998**	**5,519**	**9,313**	**50,799**

TABLE 1: Experimental subjects.

experiment took $(77 \times 20 \times 200 \times 2) = 616 \times 10^3$ seconds, i.e., more than seven days of computational time.

During all experiments, except the time-out parameter, EvoSuite is configured using its default parameter settings, which according to Arcuri and Fraser [6] work well. Also, we configure EvoSuite to skip its test-case optimization phase that consists of optimizing the number of generated test cases because this optimization slows down EvoSuite and does not impact coverage.

5.1.3 Comparing JTExpert to EvoSuite

We compare JTExpert and EvoSuite using box-plots that represent the actual obtained coverage values and an array showing the average coverage values. To compute the average branch (respectively, method, line, or instruction) coverage achieved for a given library, the number of all covered branches (respectively, methods, lines, or instructions) in all executions is summed and divided by the total number of branches in the library multiplied by the number of executions (20).

To identify the origin of any observed differences between JTExpert and EvoSuite, we minutely analyze the classes wherein a significant difference of coverage is observed: for each library, we create an Eclipse[5] project containing the classes under test and the generated test suites. Then we observe the branches covered by one tool and missed by the other using the plugin EclEmma[6]. We make our interpretations according to the type of branches at the root of the differences.

We also perform statistical tests on the coverage results. We choose Mann-Whitney U-test [27] and Vargha-Delaney's $\widehat{A}_{12}$ effect size measure [47]. Both measures are recommended as statistical tests to assess whether a novel random testing technique is indeed useful [5].

- To assess the statistical significance of the average difference between JTExpert's results and EvoSuite's, Mann-Whitney U-test [27] is used and the p-values are computed. Generally, the U-test is applied to compare whether the average difference between two groups is really significant or if it is due to random chance. The average difference between two groups is considered statistically significant if its p-value is less than 0.05;
- The U-test may be statistically significant but the probability for JTExpert to outperform EvoSuite may be small. To assess this probability, we also compute Vargha-Delaney's $\widehat{A}_{12}$ effect size measure [47]. The $\widehat{A}_{12}$ measures the probability that JTExpert yields higher code coverage than EvoSuite, e.g., $\widehat{A}_{12} = 0.9$ means JTExpert would obtain better results than EvoSuite 90% of the time.

5. https://www.eclipse.org

6. EclEmma is based on the JaCoCo code coverage library available at: https://www.eclipse.org

5.1.4 Understanding JTExpert Behavior

To gain a better understanding of the behavior of our approach and the contribution of each proposed component in its results, we carry out several different sets of experiments, each one focusing on one of the proposed components. First, we analyze JTExpert without any of its components (`JTE-All`) except a basic instance generator and the random search that targets all branches at the same time. Then, for a given component (e.g., problem representation, seeding strategy, diversification strategy) that is proposed to reduce one of the test-data generation difficulties D1, D2, or D3, we analyze JTExpert coverage over time in the absence of this component, its contribution being measured by the difference observed between JTExpert and the version of JTExpert without this component.

Because we use JaCoCo to measure coverage, we can not get the progress of coverage over time, so to get this information, we perform a set of experimentations: an experimentation is performed for each search time in the set {10s, 30s, 50s, 100s, 150s, 200s}. Each experimentation was performed with the same conditions as the first one (20 executions for each unit under test). The average branch coverage in terms of time is analyzed.

5.2 Results

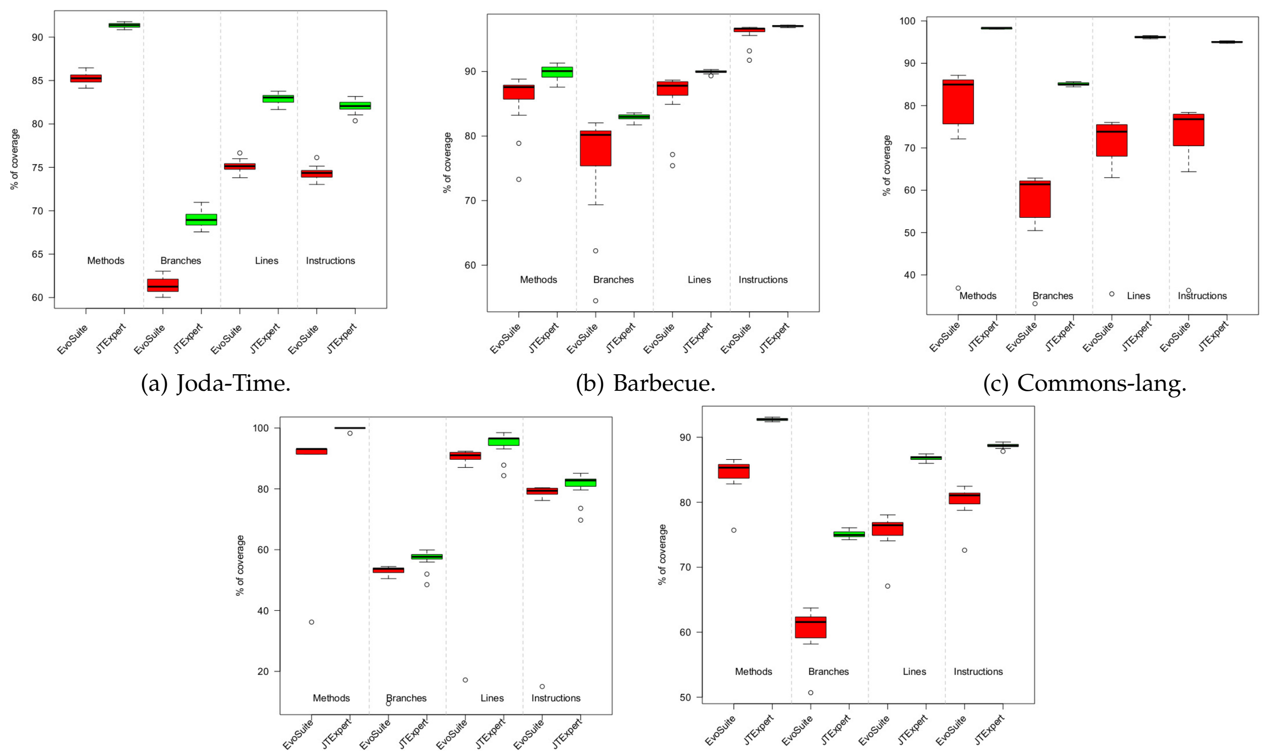
All graphics and statistical analyses presented in this study are performed with R[7] version 3.1.0 [45].

5.3 Comparing JTExpert to EvoSuite

Figures 3a, 3b, 3c, 3d, and 3e report the box-plots of the achieved coverage in 200 seconds per CUT by each tool on the Joda-Time, Barbecue, Commons-lang, and Lucene library, and all libraries together, respectively. Each box-plot compares JTExpert to EvoSuite using the four metrics of code coverage offered by JaCoCo: methods, branches, lines, and instructions.

Table 2 summarizes the results in terms of average coverage achieved by each tool at 200 s for each CUT. It shows the average of the four metrics. Triangles directed upward show the points of comparison wherein

7. Available at http://www.R-project.org



(a) Joda-Time. (b) Barbecue. (c) Commons-lang.

(d) Lucene. (e) All libraries.

Fig. 3: Comparison of JTExpert and EvoSuite on all libraries in terms of method coverage, branch coverage, line coverage, and instruction coverage.

Libraries	**Tools**	**% of average coverage in terms of**			
		Methods	**Branches**	**Lines**	**Instructions**
Joda-Time	EvoSuite	84.92	60.82	74.80	73.94
	JTExpert	▲ 91.33	▲ 69.01	▲ 82.90	▲ 82.09
Barbecue	EvoSuite	86.80	77.49	87.38	96.40
	JTExpert	▲ 89.75	▲ 82.89	▲ 89.94	■ 97.02
Commons-lang	EvoSuite	83.72	61.15	73.54	76.12
	JTExpert	▲ 98.27	▲ 85.06	▲ 96.17	▲ 95.00
Lucene	EvoSuite	92.58	52.40	90.26	78.59
	JTExpert	▲ 99.91	▲ 57.15	▲ 95.17	▲ 81.53
All	EvoSuite	**85.08**	**61.61**	**76.26**	**80.92**
	JTExpert	▲ **92.73**	▲ **75.04**	▲ **86.82**	▲ **88.70**
Diff.	%	**+7.65**	**+13.43**	**+10.56**	**+7.78**
	#	**+152.85**	**+741.20**	**+983.45**	**+3,952.16**

TABLE 2: Summary of the experimental results. Comparison with EvoSuite in terms of average coverage.

JTExpert outperforms EvoSuite by a percentage ranging from 2.5 % to 24 % while a square represents a difference of less than 2.5 %. Table 3 reports the $\widehat{A}_{12}$ effect size measure as well as the p-values of the statistical U-test.

The benchmark [49] contains two other classes from the library Sqlsheet but we do not report the results of these two classes because neither JTExpert nor Evosuite could generate any test data. The challenge in these classes is generating an instance of the CUT. These classes offer a constructor that needs two parameters: one is a URL and the other is a string. The first parameter must be a URL referencing an existing Microsoft Excel file and the second parameter must hold the name of an existing sheet in the file. The likelihood of randomly instantiating this type of class is almost null. Hence, to automatically generate an instance for such a class, the instance generator must understand the context of use of the class, which we will tackle in our future work.

A glance at Figures 3a, 3b, 3c, 3d, 3e and Table 2 shows that JTExpert outperforms EvoSuite by covering test targets that EvoSuite failed to cover. Among 20 comparisons, EvoSuite achieved almost the same performance as JTExpert for instruction coverage on Barbecue whereas JTExpert is dominant for all other metrics and libraries. The difference over EvoSuite reached 23.91 % using the metric branch coverage on the library Commons-lang. In total, JTExpert covers on average 3,952 instructions more than EvoSuite.

Libraries	Comparing JTExpert to EvoSuite in terms of average coverage							
	Methods		Branches		Lines		Instructions	
	U-test (p)	$\widehat{A}_{12}$	U-test (p)	$\widehat{A}_{12}$	U-test (p)	$\widehat{A}_{12}$	U-test (p)	$\widehat{A}_{12}$
Joda-Time	6.71e-08	1	6.76e-08	1	6.78e-08	1	1.45e-11	1
Barbecue	3.54e-07	0.96	6.95e-08	0.99	6.32e-08	1	1.05e-07	0.99
Commons-lang	3.27e-08	1	6.77e-08	1	6.67e-08	1	6.74e-08	1
Lucene	5.41e-09	1	8.98e-06	0.91	9.48e-06	0.90	3.88e-05	0.88
All	6.69e-08	1	1.45e-11	1	6.78e-08	1	6.79e-08	1

TABLE 3: Results of computing U-test and the $\widehat{A}_{12}$ effect size measure on JTExpert's results compared to EvoSuite's.

In Table 3, all U-test's p-values are less than 10^{-4}. To say that there is a difference, in all cases, is taking less than 10^{-2} percent risk of being wrong. Thus, we conclude that there is a statistically significant difference between JTExpert's results and EvoSuite's. Also, almost all $\widehat{A}_{12}$ effect size measure values are equal to 1. Therefore, JTExpert is practically certain to achieve a higher code coverage than EvoSuite. Even when $\widehat{A}_{12}$ is less than 1, there is a high probability (at least equal to 0.88) for JTExpert to achieve better coverage than EvoSuite.

To summarize, box-plots, average code coverage, Mann-Whitney U-test, and Vargha-Delaney?s $\widehat{A}_{12}$ effect size measure results support the superiority of our approach over EvoSuite in terms of code coverage.

5.4 Comparing JTExpert and EvoSuite in Details

To better understand the origin of the observed differences in terms of code coverage between JTExpert and EvoSuite, we analyze the code coverage at the class level. We would have liked to automatically analyze the lower levels (e.g., methods, branches, statements) but the JaCoCo reports do not offer us this information. Thus, we manually analyze and interpret the code coverage on lower levels.

Figure 4 presents the box-plots of the branch coverage achieved by each tool on the classes where a significant difference is observed. The coverage achieved on a given class is compared between two vertical lines. Each comparison contains a box-plot for EvoSuite's coverage, a box-plot for JTEpert's coverage, the name of the class, and the total number of branches in that class written in brackets. In Figure 4, JTExpert has higher coverage than EvoSuite's on the first fourteen classes, from the left, that represent 60% of the total number of branches under test. EvoSuite has higher coverage than JTExpert on the last seven classes that represent 5.6% of the total number of branches under test. Evosuite achieved better coverage in some small or medium size classes whereas JTExpert has higher coverage in some other small or medium size classes and is dominant on large classes. Overall, Figure 4, supports the observation that JTExpert is more effective than EvoSuite on large classes. This observation can be explained by the complexity of identifying a required sequence of methods to reach a test target in a large class, i.e., a significant number of sub-classes and methods substantially decreases the likelihood to get a relevant sequence without a static analysis. The proposed problem presentation helps JTExpert to reach more test targets efficiently by trying only relevant sequences of method calls, whereas EvoSuite may try a significant number of sequences of method calls without getting a good one.

```
public static PeriodFormatter alternateWithWeeks() {
    if (cAlternateWithWeeks == null) {
        cAlternateWithWeeks = new ↩
            PeriodFormatterBuilder()
            .appendLiteral("P")
            .printZeroAlways()
            .minimumPrintedDigits(4)
            .appendYears()
            .minimumPrintedDigits(2)
            .appendPrefix("W")
            .appendWeeks()
            .appendDays()
            .appendSeparatorIfFieldsAfter("T")
            .appendHours()
            .appendMinutes()
            .appendSecondsWithOptionalMillis()
            .toFormatter();
    }
    return cAlternateWithWeeks;
}
```

Fig. 5: Source code of method `org.joda.time.format.ISOPeriodFormat.alternateWithWeeks()`

Class `org.joda.time.format.ISOPeriodFormat` has five private data members and five public methods. Each method contains two branches and uses a different data member for its conditional statement. Figure 5 presents the source code of one of the methods. The data member `cAlternateWithWeeks` is used and modified only by this method. Hence, to reach both branches, the method must be called twice in a same sequence. All the other methods follow same pattern. The twenty test suites generated by EvoSuite cover only the five branches that require a null data member, whereas those generated by JTExpert cover all ten branches, thanks to its problem representation that allows JTExpert to understand that reaching those branches requires two calls of the methods.

Class `net.sourceforge.barbecue.CompositeModule` has a private data member, `modules`, and contains three loops over `modules` in three different methods. Figure 6 presents a part of the source code of `CompositeModule`. To enter inside a loop, the list `modules` must contain at least one item. Hence, to reach a given loop, the

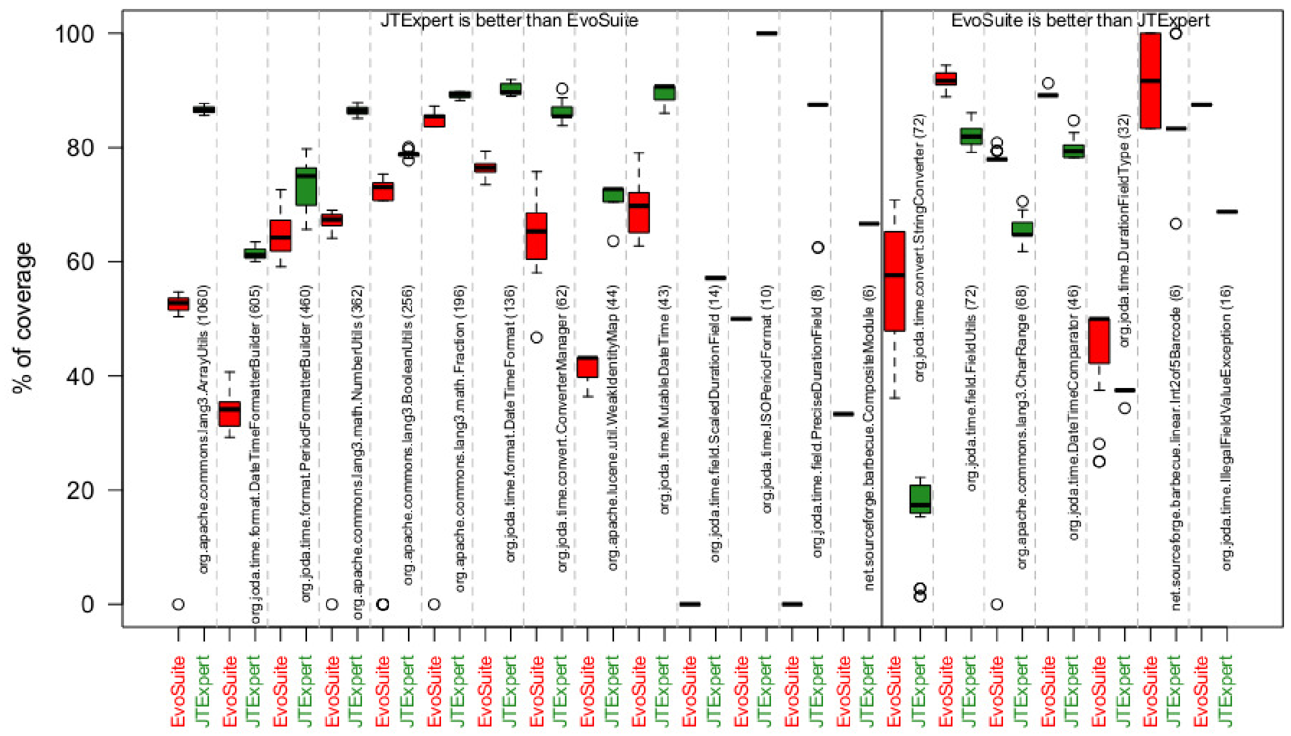


Fig. 4: Comparison of JTExpert and EvoSuite on classes in terms of branch coverage.

```
    public int widthInBars() {
        int width = 0;
        for (Iterator iterator = modules.iterator(); iterator.↩
            hasNext();) {
            Module module = (Module) iterator.next();
            width += module.widthInBars();
        }
        return width;
    }

    public void add(Module module) {
        modules.add(module);
    }
```

Fig. 6: Part of the source code of class `net.sourceforge.barbecue.CompositeModule`

method `add(Module module)` must be called before any method containing a loop. The twenty test suites generated with EvoSuite could not enter in any loop, whereas those generated by JTExpert cover two of the three loops. JTExpert missed covering the third loop because it is inside a protected method. JTExpert's problem representation makes the difference.

Classes `org.joda.time.format.DateTimeFormatterBuilder` and `PeriodFormatterBuilder` contain seventeen class members: ten in `DateTimeFormatterBuilder` and seven in `PeriodFormatterBuilder`. The classes contain 144 methods: 92 in `DateTimeFormatterBuilder` and 52 in `PeriodFormatterBuilder`. EvoSuite has difficulty in reaching methods declared in class members: the test suites generated with EvoSuite could not reach 44 methods whereas the test suites generated with JTExpert missed only 19 methods. The 19 methods missed by JTExpert are essentially in four class members: `DateTimeFormatterBuilder$Fraction`, `DateTimeFormatterBuilder$UnpaddedNumber`, `DateTimeFormatterBuilder$FixedNumber`, and `PeriodFormatterBuilder$Composite`. JTExpert was unable to instantiate the first three class members because they do not offer any accessible means-of-instantiation. EvoSuite could not or did not try to instantiate class members and reached a weak coverage compared to JTExpert. Its problem representation and means-of-instantiations make JTExpert more effective than EvoSuite by buildings more relevant sequences of methods and means-of-instantiations to reach methods in class members.

As shown in Figure 1, class `org.apache.lucene.util.WeakIdentityMap` contains an anonymous class with two private data members, a private method, and three public methods. Because there is no call to the three public methods, only a direct call can reach them. Also, branches in the anonymous class depend on the state of its enclosing class, `WeakIdentityMap`. Hence, to reach the methods in the anonymous class, an instance of this class is required in a relevant state. The twenty test suites generated by EvoSuite reached only two methods out of four and could not cover any branch, whereas those generated by JTExpert reached the four methods and covered 12 branches out of 16. Thanks to the means-of-instantiations that allow JTExpert to instantiate an anonymous class and with the help of the problem representation, JTExpert puts the instances of the class, `WeakIdentityMap` and the anonymous class in desired states to reach the methods and branches inside the anonymous class.

EvoSuite could not generate test data for two classes: `org.joda.time.field.ScaledDurationField` and `org.joda.time.field.PreciseDurationField`. The source code of EvoSuite is not available, so we cannot know the reason for this behavior but it may be due to its inability to generate an instance of the CUT. Consequently, JTExpert outperforms EvoSuite by covering eight out of 14 branches in `ScaledDurationField` and seven out of eight in `PreciseDurationField`.

In class `org.joda.time.convert.ConverterManager`, on average, each test suite generated by EvoSuite missed 13 branches compared to a test suite generated by JTExpert. The 20 test suites generated by EvoSuite missed four branches that require a `null` object (e.g., `if(object==null)`). The 20 test suites generated by EvoSuite, on average, covered only one of those four branches, whereas the test suites generated by JTExpert covered all four branches. The seeding of a `null` constant benefits JTExpert.

Three classes from the library Commons-lang, `ArrayUtils`, `BooleanUtils`, and `NumberUtils`, contain a significant number of conditional statements to check `null` objects and arrays of lengths equal to 0. Class `ArrayUtils` contains 170 branches requiring a `null` object and 60 branches requiring an array of length 0. Class `BooleanUtils` contains 26 branches requiring a `null` object and six branches requiring an array of length 0. Class `NumberUtils` contains 30 branches requiring a `null` object and 12 branches requiring an array of length 0. The 20 test suites generated by EvoSuite reached 31 branches requiring a `null` object: 10 branches in `ArrayUtils`, 14 branches in `BooleanUtils`, and seven branches in `NumberUtils`. These test suites could not reach the branches that required an array of length 0. In contrast, the test suites generated by JTExpert covered all branches requiring a `null` object or an array of length 0. The generator of containers and the seeding of `null` constants benefits JTExpert. The strategy used in EvoSuite to seed `null` constants is not enough and seeding the constant `null` with a constant probability equal to 5% is necessary to reach all branches involving that constant. Containers should be generated in a different way than other objects as proposed in Section 4.2.1.

Class `org.joda.time.MutableDateTime` extends class `BaseDateTime` and defines 103 methods that are mostly setters that take integers as parameters. In class `MutableDateTime`, there are no visible preconditions on the parameters but the parameters must satisfy the preconditions defined in the superclass, e.g., a value must be an hour of the day in the range [0,23] or a day of the week in the range [1,7]. These "hidden" preconditions make the task of covering methods in `MutableDateTime` harder. The test suites generated by EvoSuite could not cover 14 methods, whereas the test suites generated by JTExpert missed only four methods. This difference comes from the seeding strategy and the search heuristic. The seeding strategy offers adequate values that satisfy preconditions in the superclass but the way each tool uses them is different: EvoSuite relies on a genetic algorithm. If its initial population lacks values that satisfy the hidden preconditions, then EvoSuite may make a lot of attempts using the same population before introducing new values through its mutation operator, whereas the random search in JTExpert uses different values. Further, the absence of preconditions may make the GA blind and worst than a guided random search. Therefore, the seeding strategy combined with the random search is at the root of the observed difference between JTExpert and EvoSuite on class `MutableDateTime`.

Classes `org.joda.time.field.FieldUtils` and `org.joda.time.DateTimeComparator` contain branches and return statements that rely on disjunctive and conjunctive conditions. For such statements, the number of branches at the Java Bytecode level is different from at the Java source code level. For example, to cover all branches in `boolean foo(boolean a, boolean b){return a && b;}`, generates only one test data, whereas this function contains four branches at the bytecode level. Thus JaCoCo favors EvoSuite that works with bytecode and penalizes JTExpert. Consequently, EvoSuite reaches seven branches in class `FieldUtils` and four branches in class `DateTimeComparator` more than JTExpert.

Class `org.joda.time.convert.StringConverter` contains 72 branches, 42 of which are in method `getDurationMillis(Object object)` in which all the branches depend on the return value of `object.toString()`: To reach more than 28 branches, the method `object.toString()` must return a string that starts with the substring `PT` and finished with `S`, i.e., gets the value as a string in the ISO8601 duration format. For example, "PT6H3M7S" represents 6 hours, 3 minutes, 7 seconds. The method `object.toString()` must match the `toString()` method of the class `ReadableDuration`. In the 20 test suites generated, JTExpert failed to generate such an object, whereas EvoSuite generated the required objects. Consequently, EvoSuite outperforms JTExpert on class

`StringConverter` with 29 branches. After inspecting JTExpert source code, we found that we restricted the list of stubs to instantiate class `java.lang.Object` to some classes, e.g., Integer, String, and the CUT. Thus, JTExpert could not generate an instance of a class that implements the interface `ReadableDuration`. This limitation may be fixed by enlarging the list of stubs of the class `java.lang.Object` to contain any other class that was instantiated during the search.

To summarize, the sample of classes analyzed above is representative because it covers classes in which we observed differences between the two approaches. This analysis shows the actual effects of the proposed components and clarifies the advantages and weaknesses of the compared approaches. It also shows the types of test target for which JTExpert outperforms Evosuite: branches involving a data member, branches requiring the `null` constant, branches requiring a container or string with length equal to 0, methods that are declared inside classes members or anonymous classes, and methods that contains hidden preconditions. Also, it reveals the limitation in our generator for class `java.lang.Object`.

5.5 Understanding JTExpert behavior

We now quantify and measure the contribution of each component. Figure 7 shows the difference in terms of average branch coverage between JTExpert and six other versions wherein at least one proposed component is disabled: JTExpert without all components (JTE-All), JTExpert without the generator of sequences of method calls (JTE-Seq), JTExpert without the seeding strategy (JTE-Seed), JTExpert without the variable probability in the seeding strategy (JTE-Se%), JTExpert without the diversification strategy (JTE-Div), and JTExpert without the instantiation of anonymous classes (JTE-Ano).

Figure 7 reflects the achieved results in terms of average branch coverage for all classes. It shows that JTExpert performs better than Evosuite in terms of efficiency as well. JTExpert is more efficient because, at 10 seconds, it reaches a high branch coverage, almost 70 %, whereas Evosuite reaches 31 % coverage at 10 seconds and 61 % at 200 seconds. There is a large difference in terms of time required to reach the same level of coverage because Evosuite does not reduce the search domain of D2 and D3 and because it does not have a diversification strategy, its seeding strategy uses a fixed seeding probability, and it covers only primitive types and strings. Also, the random search implemented in JTExpert does not waste time with complex branches. Furthermore, the state-modifier methods, target-viewfinder methods, and the instance generator guide the random search to quickly reach less complex branches.

In a first experimentation, `JTE-All`, each proposed component was disabled or replaced with a simple version as explained in the next paragraphs. `JTE-All` performed better than EvoSuite, especially on the Common-lang library and but is less effectiveness on Barbecue. For

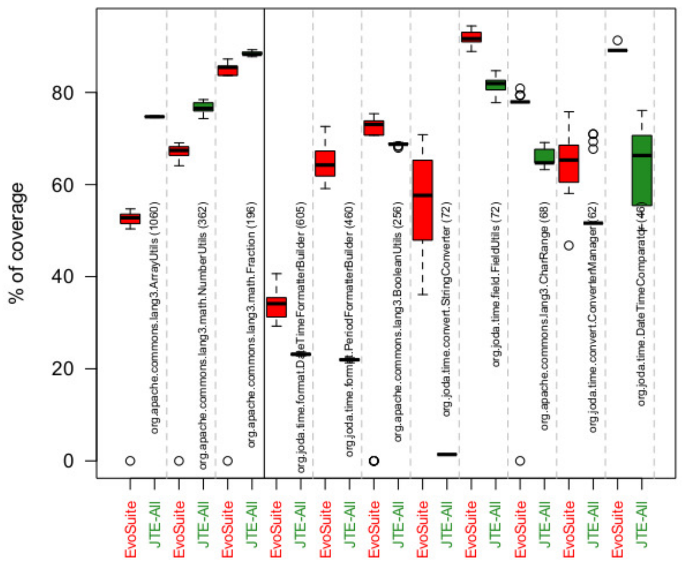


Fig. 8: Comparison of JTE-All and EvoSuite on classes in terms of branch coverage at 200 s.

the two other libraries, Joda-Time and Lucene, EvoSuite performed better than `JTE-All`.

On all libraries, `JTE-All` is more efficient than EvoSuite because at 10 s it reached 56% branch coverage, whereas EvoSuite reach only 31%. This difference can be explain by the different search heuristics implemented in `JTE-All` and EvoSuite: If EvoSuite generates an initial population that lacks some methods leading to some easy branches, then it must wait until the mutation operator injects them, whereas the random search used in `JTE-All` is more likely to quickly select these methods in a sequence, hence it reaches more branches early.

At 200 s, EvoSuite (genetic algorithm) is supposed to outperform `JTE-All` (random search) but we observed almost the same branch coverage. To understand this behavior, we analyzed the branches reached on each class. Figure 8 presents the box-plots of the achieved branch coverage with `JTE-All` and EvoSuite at 200 s on the classes where a significant difference was observed.

EvoSuite outperforms `JTE-All` on classes from different libraries, particularity Joda-time. In these classes, branches are complex to reach because they are embedded in sub-classes or private methods, e.g., `org.joda.time.format.DateTimeFormatterBuilder` contains 14 sub-classes. EvoSuite benefits from genetic algorithm to reach a complex test targets compared to a random search.

JTE-All outperforms EvoSuite on three large classes in Commons-lang. Most of the methods in these classes take containers as parameters: 208 methods in class `ArrayUtils` and 13 in class `NumberUtils` use arrays. Also, the difficulty to reach branches in these classes lies

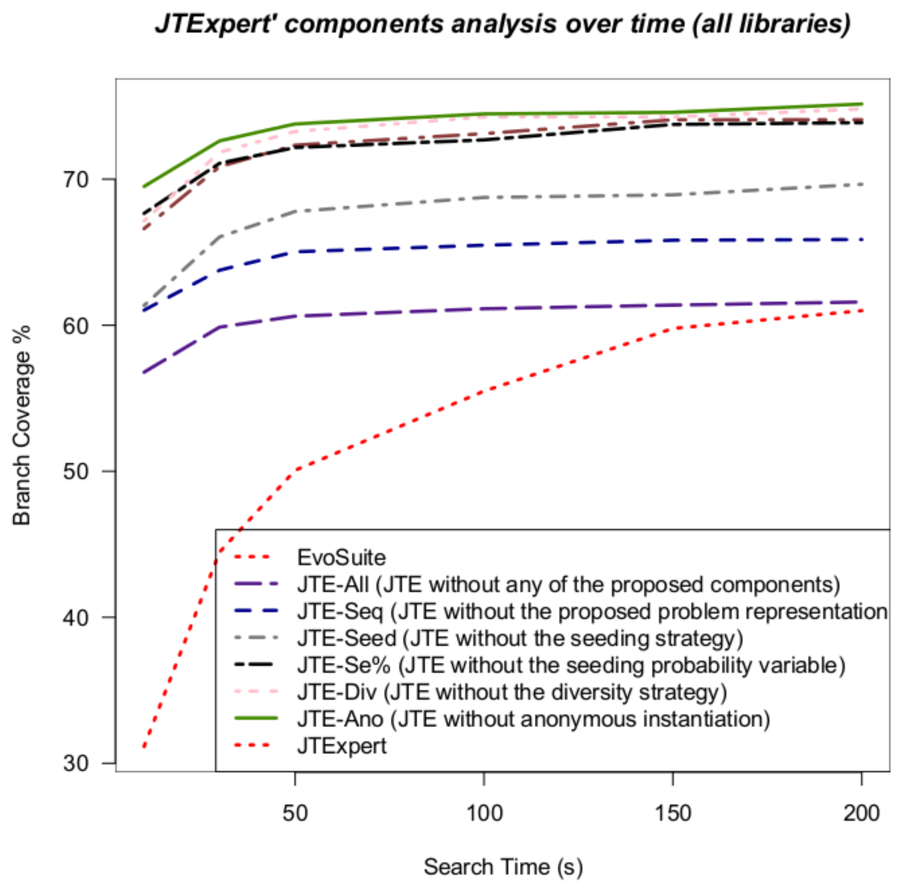


Fig. 7: Contribution of each proposed component in terms of average branch coverage over time.

in the generation of diversified instances of the methods' parameters. For example, a same method may contain a condition requiring a null array, a condition requiring an array with length equal to 0, and a condition requiring an array containing many elements. To cover branches in such a method, it is enough to call it with different types of array, i.e., to have an array generator that generates diversified types of array.

We think that we could not see the difference between `JTE-All` and EvoSuite at 200 s because the containers generator implemented in `JTE-All` hides the advantage of the genetic algorithm implemented in EvoSuite on complex branches.

In a second experimentation, `JTE-Seq`, the generator of sequences of method calls was replaced by a random generation of sequences of method calls. The comparison to JTExpert in Figure 7 shows that the use of the proposed representation is beneficial for the search, with an average branch coverage equal to 9.28 % at 200 s, where the average branch coverage increased from 65.87 % to 75.15 %. The improvement is significant for the classes in Joda-Time and Lucene: the average branch coverage for Joda-Time increases from 53.55 % to 69.01 % and for Lucene, from 47.20 % to 57.15 %. For the other libraries, the obtained results are almost the same, i.e., there are no improvements. The better performance obtained on Joda-Time and Lucene and not on Commons-lang and Barbecue can be explained by the complex class structures found in the first two: in Joda-Time and Lucene, one Java file or class defines more than one class (e.g., the Java file `DateTimeFormatterBuilder.java` defines 15 classes). A CUT that contains nested classes (e.g., member class, local class, or anonymous class) needs guidance during the test-data generation. We show that the proposed representation improves coverage, particularly for complex classes under test.

In a third experimentation, `JTE-Ano`, anonymous classes are tested through their enclosing class, i.e., the closest accessible class that contains the anonymous CUT. The comparison to JTExpert in Figure 7 shows that the proposed means-of-instantiations of anonymous classes slightly enhances the search process, with an average branch coverage equal to 0.34 % at 200 s. This improvement may seem insignificant but such magnitude on all classes was expected because such anonymous classes are rare in the chosen libraries. To estimate the actual improvement that the instantiation of anonymous classes may bring, we analyzed class `org.apache.lucene.util.WeakIdentityMap` separately. We found that the proposed means-of-instantiations of anonymous class enhance significantly the average branch coverage of this class, where the coverage increased from 39.77 % (`JTE-Ano`) to 71.47 % (JTExpert). This is a large enhancement in terms of branch coverage because the anonymous class defined in `org.apache.lucene.util.WeakIdentityMap` contains a significant number of branches.

In a fourth experimentation, `JTE-Div`, the diversification strategy was replaced by a random selection of means-of-instantiations. The comparison to JTExpert in Figure 7 shows that the proposed diversification strategy enhances the search process, with an increase of average branch coverage equal to 1.28 % at 200 s, where the average branch coverage increased from 73.87 % to 75.15 %. At first glance, this improvement seems insignificant. However, this magnitude is expected because of the small number of classes that are hard to instantiate. For example, class `org.apache.lucene.util.FixedBitSet` is hard to instantiate because calling some of its constructors with random arguments may hang the test data generation process. Thus, the diversification strategy is beneficial to the search, with an average branch coverage equal to 26.53 %, where the coverage on this particular class passed from 26.09 % to 52.62%. We show that the proposed diversification strategy of means-of-instantiations improves coverage, especially for classes that defines some means-of-instantiations that may be harmful with randomly generated arguments.

translated into many conditional statementsIn a fifth experimentation, `JTE-Seed`, the proposed seeding enhancements were simply disabled, i.e., without seeding `null` and with a constant seeding probability equals to 0.2. Thus, the seeding strategy used in this experimentation is equivalent to one studied in [14]. The comparison to JTExpert in Figure 7 shows that seeding `null` and using a variable seeding probability are beneficial for the search, with an increase of average branch coverage equal to 5.60 % at 200 s, where the average branch coverage increased from 69.65 % to 75.15 %. In this 5.60 % enhancement, the variable probability of seeding contributes by 1.08 % as shown in the graph `JTE-Se%` in Figure 7. The improvement is significant for the classes in Commons-lang and Lucene, with an enhancement almost equal to 10 %, where the average branch coverage increased from 75.63 % to 85.06 % on Commons-lang and from 44.80 % to 57.15 % on Lucene. On Joda-Time and Barbecue, the improvement is less significant, below 3 %. The significant enhancement on the first two libraries can be explained by the significant number of branches that rely on null pointer checks. The reason for the small improvement observed on Joda-Time and Barbecue is the small number of conditions that they use to prevent null pointer exception. During the execution of the test suites generated for these two libraries, we observed that a significant number of null pointer exceptions occurred. Seeding `null` may be also beneficial to raise null pointer exceptions. We conclude that seeding `null` improves the coverage, especially for classes that systematically check for null pointers before using the instance of a class.

To summarize, every proposed component improves the coverage for all or some cases of the CUT. The large difference in terms of code coverage achieved over EvoSuite comes from the accumulated contributions of all proposed components.

5.6 Threats to Validity

The results showed that using JTExpert to generate test data, improves SBST performance in terms of runtime and code coverage. Yet, several threats potentially impact the validity of the results of our empirical study. We discuss in details these threads on our results, following the guidelines provided in [53].

Construct validity threats concern the relationship between theory and observation. They are mainly due to errors introduced in measurements. In our study, they related to the measure of the performance of a testing technique. We compared JTExpert and EvoSuite in terms of coverage, which is widely used to measure the performance of a testing technique. As a second measure of performance, we chose time rather than the number of fitness evaluations because a fitness evaluation has a different meaning in the two tools: in EvoSuite, a chromosome is a set of test-data candidates and one evaluation may require the execution of many test-data candidates [16] whereas in JTExpert, we evaluate each test-data candidate separately. Moreover, time is the most important constraint in the testing phase of an industrial system [10].

Internal validity threats arise from the empirical study process methodology. A potential threat comes from the natural behavior of any search-based approach: the random aspect in the observed values, which may influence the internal validity of the experiments. In general, to overcome this problem, the approach should be applied multiple times on samples with a reasonable size an statistical tests should be used to estimate the probability of mistakenly drawing some conclusions. In our empirical study, each experiment took 200 s and was repeated 20 times. The libraries contain in total 50,799 instructions and 5,519 branches. Also, we computed Mann-Whitney U-tests and evaluated Vargha-Delaney's $\widehat{A}_{12}$ effect size measure. Therefore, experiments used a reasonable size of data from which we can draw some conclusions.

External validity threats concern the possibility to generalize our observations. A potential threat is the selection of the libraries and classes used in the empirical study. All these libraries and classes have been used to evaluate different structural testing approaches in the past [18], [49], thus they are a good benchmark for evaluating our approach.

Another potential threat comes from the EvoSuite parameters: we did not try different combinations of parameters values to show empirically that our approach is robust to EvoSuite parameters. However, according to Arcuri and Fraser [6], the default configuration of EvoSuite performs well, but it may not be as good as a specific configuration for each class. Because it is hard and impractical to find a best configuration for each class, the default parameter settings of EvoSuite can be considered as a good practical configuration.

Another thread is the choice of Java to implement our approach, which could potentially affect its external validity. We implemented our prototype, JTExpert, to generate test data for classes written in Java, although the presented components can be adapted to any OOP language, e.g., C++. The choice of Java is technical because: (1) we believe that the best programming language is the language that you master better, and in our laboratory Ptidej[8], we have an extensive experience in analyzing Java source code; (2) there are many available open-source tools (e.g., JDT, Reflexions) that made our development task easier; (3) it is much easier to debug the tool in Java; (4) it is easier to get a correspondence between a CUT, its instrumented class, and its test-data suite. In the described implementation, we referred to some APIs, such as the meta-data analyzer, Java Reflection API [40], and the AST generator, Eclipse JDT [23], to ease our development task. In general, for many OOP languages, there exist different meta-data analyzers and AST generators. In the worst case, if the approach must be implemented for an OOP language for which there is no meta-data analyzer or no AST generator, then four additional analyzers must be developed: (1) to get all means-of-instantitions of a given class, (2) to get all branches callers of a given method, (3) to get all branches modifiers of a given data member, and (4) to extract constants from the source code.

Reliability validity threats concern the possibility of replicating our study. We attempted to provide all the necessary details to replicate our study: analysis process is described in detail in Section 5 and all the classes tested in this study are publicly available [49]. Moreover, all tools, JTExpert, EvoSuite, JaCoCo, Eclipse, EclEmma, and R, used in this study are publicly available.

6 Conclusion

In the last decade, search-based software testing (SBST) has been extensively applied to solve the problem of automated test-data generation for procedural programming as well as for object-oriented programming (OOP). Yet, test-data generation for OOP is challenging due to the features of OOP, e.g., abstraction, encapsulation, and visibility that prevent direct access to some parts of the source code.

This paper introduced an approach for OOP software test-data generation for unit-class testing whose novelty relies on analyzing statically the internal structure of a class-under-test (CUT) to reduce the search space and on a diversification strategy and seeding strategy. The approach sees the test-data generation problem as facing three difficulties: ($D1$) obtaining an instance of a CUT and other required objects; ($D2$) finding an adequate sequence of method calls to put the instance of the class in a desired state; and, ($D3$) finding an adequate method to reach the test target through it.

To solve $D1$, an instance generator based on a diversification strategy, instantiation of anonymous classes, and a seeding strategy are proposed to boost the search. Instantiating a class using different means-of-instantiations according to their complexities is beneficial to the search with an increase of the average coverage equals to 1.28%, up to 26.53% in certain cases. The instantiation of anonymous classes brought an enhancement to the average coverage equal to 0.34% but up to 41% in the case of class `org.apache.lucene.util.WeakIdentityMap`. Using a variable seeding probability for primitive types and seeding the `null` value for classes while generating instances increase the average branch coverage by 5.60%.

To solve $D2$ and $D3$, only methods that may change an instance of the CUT state and methods that may reach the test target are explored. This leads to a restructuring of the test data generation problem that improved the branch coverage by 9.28%.

As a result, and contrary to earlier approaches that use a traditional representation of the problem and randomly generate instances of objects, our approach and its implementation JTExpert, find a test-data suite and achieve a high code coverage (70%) in less than 10 s. We showed on more than a hundred classes taken from different open-source Java libraries that JTExpert has a higher code coverage than EvoSuite and needs less time.

JTExpert currently relies on a guided random search and supports branch coverage. We are working on adding further search algorithms, such as genetic algorithms and hill climbing. Also we are focusing on three research directions: (1) enlarging the scope of JTExpert to support other structural testing criteria, such as data-flow coverage or mutation-coverage; (2) enhancing the instances generator to generate instances of classes that require understanding the context of their use; and, (3) studying the impact of seeding `null` with the instances of objects on the generation of null pointer exceptions.

8. Ptidej: Pattern Trace Identification, Detection, and Enhancement in Java. Website: http://www.ptidej.net

References

[1] Alshahwan, N., Harman, M.: Automated web application testing using search based software engineering. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering. pp. 3–12. IEEE Computer Society (2011)

[2] Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification and Reliability 16(3), 175–203 (2006)

[3] Andrews, J.H., Haldar, S., Lei, Y., Li, F.C.H.: Tool support for randomized unit testing. In: Proceedings of the 1st international workshop on Random testing. pp. 36–45. ACM (2006)

[4] Andrews, J.H., Menzies, T., Li, F.C.: Genetic algorithms for randomized unit testing. Software Engineering, IEEE Transactions on 37(1), 80–94 (2011)

[5] Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability 24(3), 219–250 (2014), http://dx.doi.org/10.1002/stvr.1486

[6] Arcuri, A., Fraser, G.: On parameter tuning in search based software engineering. In: Search Based Software Engineering, Lecture Notes in Computer Science, vol. 6956, pp. 33–47. Springer Berlin Heidelberg (2011)

[7] Arcuri, A., Yao, X.: Search based software testing of object-oriented containers. Information Sciences 178(15), 3075–3095 (2008)

[8] Barbey, S., Strohmeier, A.: The problematics of testing object-oriented software. In: SQM 94 Second Conference on Software Quality Management. vol. 2, pp. 411–426. Citeseer (1994)

[9] Binkley, D., Harman, M.: Analysis and visualization of predicate dependence on formal parameters and global variables. Software Engineering, IEEE Transactions on 30(11), 715–735 (2004)

[10] Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: Artoo. In: Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on. pp. 71–80. IEEE (2008)

[11] Clarke, L.: A system to generate test data and symbolically execute programs. Software Engineering, IEEE Transactions on SE-2(3), 215 – 222 (sept 1976)

[12] Csallner, C., Smaragdakis, Y.: Jcrasher: an automatic robustness tester for java. Software: Practice and Experience 34(11), 1025–1050 (2004)

[13] Fraser, G., Arcuri, A.: Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. pp. 416–419. ACM (2011)

[14] Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. pp. 121–130. IEEE (2012)

[15] Fraser, G., Arcuri, A.: Evosuite at the sbst 2013 tool competition. Software Testing Verification and Validation Workshop, IEEE International Conference on 0, 406–409 (2013)

[16] Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276 –291 (feb 2013)

[17] Fraser, G., Zeller, A.: Exploiting common object usage in test case generation. In: Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on. pp. 80–89. IEEE (2011)

[18] Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. Software Engineering, IEEE Transactions on 38(2), 278–292 (2012)

[19] Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. SIGPLAN Not. 40, 213–223 (June 2005)

[20] Harman, M., Fox, C., Hierons, R., Hu, L., Danicic, S., Wegener, J.: Vada: a transformation-based system for variable dependence analysis. In: Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on. pp. 55–64 (2002)

[21] Harman, M., Hassoun, Y., Lakhotia, K., McMinn, P., Wegener, J.: The impact of input domain reduction on search-based test data generation. In: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. pp. 155–164. ESEC-FSE '07, ACM, New York, NY, USA (2007)

[22] Hoffmann, M.R., Janiczak, B., Mandrikov, E.: Jacoco is a free code coverage library for java (2012), http://www.eclemma.org/contact.html, [Online; accessed 01-OCT-2013]

[23] development tools (JDT), E.J.: The jdt project provides the tool plug-ins that implement a java ide supporting the development of any java application, including eclipse plug-ins. (2013), http://www.eclipse.org/jdt/, [Online; accessed 02-FEB-2013]

[24] JUnit: Junit is a simple framework to write repeatable tests. http://www.junit.org (2013), [Online; accessed 19-JUN-2013]

[25] King, J.C.: Symbolic execution and program testing. Commun. ACM 19, 385–394 (July 1976)

[26] Korel, B.: Automated software test data generation. Software Engineering, IEEE Transactions on 16(8), 870–879 (1990)

[27] Mann, H.B., Whitney, D.R.: On a test of whether one of two random variables is stochastically larger than the other. The annals of mathematical statistics pp. 50–60 (1947)

[28] McMinn, P.: Search-based software test data generation: a survey. Software Testing Verification & Reliability 14, 105–156 (2004)

[29] McMinn, P., Harman, M., Lakhotia, K., Hassoun, Y., Wegener, J.: Input domain reduction through irrelevant variable removal and its effect on local, global, and hybrid search-based structural test data generation. Software Engineering, IEEE Transactions on 38(2), 453–477 (2012)

[30] McMinn, P., Shahbaz, M., Stevenson, M.: Search-based test input generation for string data types using the results of web queries. In: Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on. pp. 141–150. IEEE (2012)

[31] McMinn, P., Stevenson, M., Harman, M.: Reducing qualitative human oracle costs associated with automatically generated test data. In: Proceedings of the First International Workshop on Software Test Output Validation. pp. 1–4. ACM (2010)

[32] Myers, G.J.: The art of software testing. John Wiley and Sons (1979)

[33] Oriat, C.: Jartege: a tool for random generation of unit tests for java classes. In: Quality of Software Architectures and Software Quality, pp. 242–256. Springer (2005)

[34] Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. Springer (2005)

[35] Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: Software Engineering, 2007. ICSE 2007. 29th International Conference on. pp. 75–84. IEEE (2007)

[36] Parasoft: Jtest: Java static analysis, code review, unit testing, security (2013), http://www.parasoft.com, [Online; accessed 13-OCT-2013]

[37] Pargas, R.P., Harrold, M.J., Peck, R.R.: Test-data generation using genetic algorithms. Software Testing Verification and Reliability 9(4), 263–282 (1999)

[38] Păsăreanu, C.S., Rungta, N.: Symbolic pathfinder: symbolic execution of java bytecode. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) ASE. pp. 179–180. ACM (2010)

[39] Păsăreanu, C.S., Rungta, N., Visser, W.: Symbolic execution with mixed concrete-symbolic solving. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis. pp. 34–44. ISSTA '11, ACM, New York, NY, USA (2011)

[40] Reflection: The java reflection api. http://docs.oracle.com/javase/tutorial/reflect/ (2013), [Online; accessed 01-SEP-2013]

[41] Reflections: Java runtime metadata analysis. https://code.google.com/p/reflections/ (2013), [Online; accessed 01-SEP-2013]

[42] Ribeiro, J.C.B., Zenha-Rela, M.A., de Vega, F.F.: Strongly-typed genetic programming and purity analysis: input domain reduction for evolutionary testing problems. In: Proceedings of the 10th annual conference on Genetic and evolutionary computation. pp. 1783–1784. ACM (2008)

[43] Sen, K., Agha, G.: Cute and jcute: concolic unit testing and explicit path model-checking tools. In: Proceedings of the 18th international conference on CAV. pp. 419–423. CAV'06, Springer-Verlag, Berlin, Heidelberg (2006)

[44] Tahchiev, P., Leme, F., Massol, V., Gregory, G.: Junit in action (2011)

[45] Team, R.C., et al.: R: A language and environment for statistical computing (2012)

[46] Tonella, P.: Evolutionary testing of classes. SIGSOFT Softw. Eng. Notes 29(4), 119–128 (Jul 2004)

[47] Vargha, A., Delaney, H.D.: A critique and improvement of the cl common language effect size statistics of mcgraw and wong. Journal of Educational and Behavioral Statistics 25(2), 101–132 (2000)

[48] Visser, W., Păsăreanu, C.S., Pelánek, R.: Test input generation for java containers using state matching. In: Proceedings of the 2006 international symposium on Software testing and analysis. pp. 37–48. ACM (2006)

[49] Vos, T.: Sbst contest: Java unit testing at the class level (2013), http://sbstcontest.dsic.upv.es

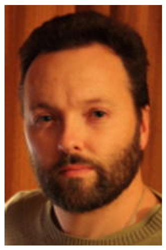
[50] Wappler, S., Wegener, J.: Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Proceedings of the 8th annual conference on Genetic and evolutionary computation. pp. 1925–1932. ACM (2006)

[51] Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 365–381. Springer (2005)

[52] Xie, T., Tillmann, N., de Halleux, J., Schulte, W.: Fitness-guided path exploration in dynamic symbolic execution. In: Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on. pp. 359 –368 (29 2009-july 2 2009)

[53] Yin, R.K.: Case study research: Design and methods. Sage publications (2014)

Abdelilah Sakti is a PhD candidate in the department of computer Science and software engineering, École Polytechnique de Montréal, Canada. He is proud being a member of Quosséca research laboratory, Ptidej team, and CiRRELT research center and supervised by the professors Gilles Pesant and JYann-Gaël Guéhéneuc. He holds a Master Sc.A. in software engineering from École Polytechnique de Montral, Canada since 2012. He also holds a Master Eng. in computer science from the Hassan II University, Casablanca, Morocco since 2009. His research interests are Software Testing, Constraint-based Software Testing, Search Based Software Testing (SBST), Constrained-search Based Software Testing, Automatic Test Data Generation (ATDG). He is currently working on combining Constraint Programing (CP) and Search Based Software Engineering techniques to improve the software-testing process, in particular through using CP to better guide SBST for ATDG.



Gilles Pesant ...

JYann-Gaël Guéhéneuc is full professor at the Department of computer and software engineering of École Polytechnique de Montréal where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns, at the language-, design-, or architectural-levels. He is IEEE Senior Member since 2010. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from cole des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of patterns. He is interested also in empirical software engineering; he uses eye-trackers to understand and to develop theories about program comprehension. He has published many papers in international conferences and journals, including IEEE TSE, Springer EMSE, ACM/IEEE ICSE, and IEEE ICSM.