SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. (2016)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/stvr.1601

Seeding strategies in search-based unit test generation

José Miguel Rojas[1,*,†], Gordon Fraser[1] and Andrea Arcuri[2]

[1]*Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello, Sheffield, S1 4DP, UK*
[2]*Scienta, Norway, and SnT Centre, University of Luxembourg, Luxembourg City, Luxembourg*

SUMMARY

Search-based techniques have been applied successfully to the task of generating unit tests for object-oriented software. However, as for any meta-heuristic search, the efficiency heavily depends on many factors; *seeding*, which refers to the use of previous related knowledge to help solve the testing problem at hand, is one such factor that may strongly influence this efficiency. This paper investigates different seeding strategies for unit test generation, in particular seeding of numerical and string constants derived statically and dynamically, seeding of type information and seeding of previously generated tests. To understand the effects of these seeding strategies, the results of a large empirical analysis carried out on a large collection of open-source projects from the SF110 corpus and the Apache Commons repository are reported. These experiments show with strong statistical confidence that, even for a testing tool already able to achieve high coverage, the use of appropriate seeding strategies can further improve performance. © 2016 The Authors. *Software Testing, Verification and Reliability* Published by John Wiley & Sons Ltd.

Received 29 April 2015; Revised 16 December 2015; Accepted 10 January 2016

KEY WORDS: test case generation; search-based testing; testing classes; search-based software engineering; JUnit; Java

1. INTRODUCTION

Search-based techniques have been shown to be a promising approach to tackle many kinds of software engineering tasks [1], particularly software testing [2]. Although automated generation of test cases for structural coverage has received particular attention, for example, in the case of object-oriented software (e.g. [3]), such testing techniques are still not widely adopted by practitioners. This is partially due to current limitations in these techniques (e.g. in terms of efficiency and applicability) and because many of the different parameters that influence search-based software testing (SBST) are not well understood. Investigating these techniques is therefore of practical value.

One specific aspect requiring further investigation is *seeding*, which loosely refers to any technique that exploits previous related knowledge to help solve a search problem. For example, when generating unit tests for object-oriented software, there often arises the need to create specific string or numeric values to pass in as parameters. If there is existing knowledge about the class under test in terms of sample values, then these can be used instead of randomly generated values. This may lead to an improvement of the overall performance of the test generation, which is typically measured in terms of the achieved code coverage. However, it is not clear what influence seeding has on the achievable results and what are the best seeding strategies.

To study the influence of seeding on search-based generation of unit tests in detail, this paper considers different strategies targeting the Java language and using branch coverage as effectiveness measure; however, the presented techniques can be extended to other programming languages

*Correspondence to: Jose Miguel Rojas, Department of Computer Science, University of Sheffield Regent Court, 211 Portobello S1 4DP Sheffield, UK.

†E-mail: j.rojas@sheffield.ac.uk

as well and are not specific to any particular coverage criterion. In particular, this paper studies *seeding* strategies that are applied when generating new or modifying existing test cases throughout the search:

- Seeding of constants extracted from source code or bytecode (e.g. numbers and strings).
- Seeding of values observed at runtime during test executions required for fitness evaluation.
- Seeding of type information for dynamic or generic type instantiation.
- Reuse of previous solutions (e.g. previously generated or hand crafted test cases).

To study these seeding techniques, they have been integrated in the automated testing tool EvoSuite [4]. EvoSuite is an advanced tool based on a genetic algorithm (GA), featuring for example the *whole test suite optimization* approach to test data generation [5]. Evaluation of these seeding techniques is performed with a large case study, including the SF110 corpus of open-source Java projects [6] and a subset of projects from the Apache Commons repository [7]. The results show, with high statistical confidence, that seeding strategies improve the performance of the employed SBST technique. However, different strategies provide different ranges of improvement, and in some cases, this effect can be correlated with the type of the tested software (e.g. when the class under test makes strong use of string objects).

This paper extends previous work on seeding strategies in search-based unit test generation [8] in several ways. Three additional seeding strategies have been implemented and evaluated, that is, dynamic seeding of constants encountered at runtime, seeding of statically collected type information and seeding of instantiated objects from existing manually written test suites. Moreover, a more ambitious experimental evaluation is carried out on a much larger set of real-world Java projects.

The organization of the paper is as follows. Section 2 sets up the context of seeding strategies in SBST, and Section 3 discusses different seeding strategies when testing object-oriented classes. Section 4 describes the experiments, presents and interprets the results and provides detailed examples. Threats to validity are discussed in Section 5. Finally, the paper is concluded in Section 6.

2. BACKGROUND

2.1. *Evolutionary testing of classes*

When generating tests for object-oriented code, the aim is to produce small sets of tests that maximize the coverage of the underlying code in the classes under test. A test suite for a class is a set of test cases, where each test case in turn is a sequence of statements (e.g. a simple JUnit test case). Each statement in a test case can generate objects through constructors and can access fields and methods. The length of test cases is typically variable, as is the number of test cases in a test suite, as it is highly dependent on the class being tested.

The experiments described in this paper use the EvoSuite test generation tool [4], which implements a GA to derive test suites for classes. Individuals of the population of the GA are test suites as described earlier. The GA works by iteratively selecting individuals from the population based on their fitness with respect to the search objective and then applying crossover and mutation operators to the selected individuals. From generation to generation, the fitness of the individuals gradually improves, until either a solution has been found or the search is terminated another way (e.g. when it hits a fixed bound on the number of generations or fitness evaluations). Crossover and mutation operators have to be defined specifically for each type of chromosome; in the case of test suite chromosomes, crossover amounts to exchange of test cases between two parent test suites, while mutation can arbitrarily update the set of test cases by adding new ones or discarding or modifying existing ones. In turn, modifying an existing test case involves deletion, change and insertion of statements (e.g. method calls).

The search is guided by a fitness function that aims to maximize coverage [5]. For example, to measure the fitness of a test suite with respect to the well-known branch coverage criterion, the minimum branch distance [2] (estimate how close the branch was to being executed based on the guarding predicate) is calculated for each of the branches in the class under test, and then the normalized branch distance values are essentially summed up. An optimal solution thus has fitness

0.0, meaning all branches have been covered. In practice, classes often have difficult branches that require the generation of complex sequences of method calls as well as specific constant values (e.g. numbers or strings). In the standard case, the initial population is generated randomly, and any constants generated during this initialization step or during the search are chosen randomly out of their respective value domains.

2.2. *Seeding in evolutionary search*

In this paper, the term *seeding* is used to loosely refer to any technique that exploits previous related knowledge to help solve the testing problem at hand. The presence of this previous knowledge should not be a requirement to address the problem at hand (i.e. in theory, the problem should be solvable even without using such knowledge).

The literature on evolutionary computation contains several papers on seeding strategies to improve the search. For example, in genetic programming, seeding strategies have been used in the context of improving different aspects of programs, which the search should optimize: in the context of machine learning, Langdon and Nordin [9], for example, studied a seeding strategy in order to improve the ability of a classifier/regressor to generalize. Westerberg and Levine [10] studied different strategies to seed the initial population for a genetic planner, an AI planning system based on genetic programming. Similarly, White *et al.* [11] studied several different seeding strategies to initialize a genetic programming population for optimizing execution time of a given input program. If every individual in the initial population is an exact copy of the input program, then the search could get stuck in a sub-optimal area of the search landscape. Therefore, there is the need to use smart seeding strategies to reuse good ‘building blocks’ from the original input program without hampering the search progress.

2.3. *Seeding in search-based software testing*

In the context of SBST, the most common case of seeding regards the case when testing targets (e.g. branches to cover) are sought one at a time, as for example in the work of Wegener *et al.* [12]. The control dependence graph can be used to choose an order in which the targets are sought, and so reuse input data from previous runs when seeking to cover a dependent target. For example, if several targets are nested inside the same difficult branch, when generating test data for one of them, the result can later be reused as starting solution when seeking to cover other targets, instead of re-starting the search from scratch (which can be expensive considering the difficulty of the parent branch in which they are nested).

In the context of testing real-time systems to find worst case execution times, Tlili *et al.* [13] applied seeding strategies as well. Given the execution time of the system under test as the fitness function to optimize, instead of starting from scratch, they used a test case with high coverage as seed to start the search from.

In some cases, it can be useful to generate more test data, even if coverage for the chosen testing criterion is already maximized. This can be the case, for example, when automated oracles are available or when software testers can afford to manually evaluate more test cases. Starting from a test case, Yoo and Harman [14] investigated a seeding strategy in which local search is applied on a solution test case, such that the diversity of the test data is maximized while the achieved coverage is preserved. The rationale is that more different test cases would have more chances to find faults.

McMinn *et al.* [15] proposed seeding values taken from source code and documentation with the objective to reduce the human oracle costs. Similarly, Fraser and Zeller [16] used common object usage for seeding in the search to reduce the human oracle costs and to improve readability of the generated test cases.

A GA usually starts from an initial population that is randomly generated. However, domain knowledge can be used to choose this initial population. For example, in test data generation, there may be several targets that are not so difficult to cover. So it makes sense to do a first phase of random testing, before using a complex testing technique. For example, Miraz *et al.* [17] create the initial population by selecting the best individuals out of a larger pool of randomly generated individuals.

When testing software with predicates involving strings, generating the right strings for the input data can be very challenging, as the space of possible strings is much larger than, for example, the one of integers. Alshraideh and Bottaci [18] proposed and investigated a seeding technique in which the code of the system under test is analysed and then string constants are extracted and used as starting point for generation of string inputs. For example, consider the snippet `if(input.equals("complexAndLongString"))`: covering this branch becomes trivial, as the right input data would be present already in the first generation of the GA. As the seeded strings can be modified during the GA evolution (e.g. through the mutation operator), such seeding technique can be helpful even in more complex cases [18]. Besides extracting string constants from the system under test, a recent seeding approach has been investigated by McMinn *et al.* [19], in which candidate input strings are extracted from web queries on search engines based on the system under test information.

Another recent seeding strategy in SBST has been proposed by Alshahwan and Harman [20], named 'Dynamically Mined Value' seeding. In testing web applications, the resulting HTML pages generated as output of the test cases are then used as source of string inputs for the new test cases in the search.

3. SEEDING STRATEGIES FOR TESTING OBJECT-ORIENTED CLASSES

This section discusses different seeding strategies that can be applied during test generation for object-oriented classes. These are considered in the context of the Java language and its bytecode representation, but the general seeding strategies apply to any language.

3.1. Seeding constants from source code or bytecode

As previous work has shown, reusing constant values appearing in the source code of the system under test can have positive effects in test data comprehension [15] and code coverage [18]. When branches are dependent on particular values, the program code often contains values that are similar to the sought values. For example, branch conditions often contain the boundary values as constants:

```
if(x == 27 && y > 250) {
  // ...
}
```

Both values, 27 and 250, are candidates for seeding. This information can be easily collected and integrated into the search. Namely, a constant pool is populated with concrete values during the loading of the system under test in a so-called instrumentation phase. Then, during the search, whenever attempting to generate a new constant value (e.g. to satisfy a parameter necessary for a method call), with a certain probability P_{Constant}, a value from the constant pool is randomly chosen rather than a random new value. If by chance 27 is assigned to `x` directly, this would be lucky and make the condition `x == 27` immediately true. If by chance 250 is assigned to `y`, this would not immediately make the condition true, but the search would be very close to achieving this. Typically, there are different numerical types (e.g. 6, 16, 32 and 64 bit integer numbers, and 32 and 64 bit floating point numbers). A sensible approach is to keep distinct constant pools for each type.

This type of seeding is not only viable for numerical constants, but also for textual constants, that is, strings:

```
if(str1.equals("Example") && str2.startsWith("foo")) {
  // ...
}
```

Again, using the strings `"Example"` and `"foo"` as constant seeds when generating string inputs may help the search to satisfy the branching condition earlier. In general, evolving a randomly generated string to any concrete value can take a very long time. Therefore, seeding string constants have the potential to significantly reduce the time needed for the search, even if the seeded strings are just in the proximity of the necessary values.

The EvoSuite tool operates on bytecode, which is an intermediate representation used by many modern object-oriented languages: compilation produces a binary representation of the classes that is close to machine code yet retains parts of the original structure (e.g. classes and methods). Typically, bytecode also includes constant numerical values and strings. For example, in Java, every class has a dedicated constant pool as part of its bytecode representation, and so, it is easy to collect these constants from bytecode. In principle, constants can be extracted from source code just as well, if available.

3.2. Dynamic seeding

Whereas static seeding uses values found through static analysis of the source code or bytecode, *dynamic* seeding uses any values observed during execution for seeding. This idea was first proposed by Alshahwan and Harman [20] in the context of testing web applications, and in this section, we introduce new techniques to apply it in the context of unit test generation.

3.2.1. Seeding numerical values. Although comparisons with constants are common, very often comparisons are also made between two variables with values that are not known statically, for example:

```
if(x == y) {
 // ...
}
```

Assume that `x` contains an input value that can be set but `y` is a local variable containing the result of a complex calculation. Here, information cannot easily be extracted statically. However, at *runtime*, the exact values of `x` and `y` can be observed when this comparison is performed; in fact, during search-based testing these values are already compared in the context of calculating the branch distance metric [21]. The idea of dynamic seeding, first proposed by Alshahwan and Harman [20] in the context of web testing, is to collect these values at runtime and add them to the constant pool used for seeding. To distinguish between constant values extracted statically and dynamically, they are kept in two different pools, one for each type of constant. Thus, once the condition is reached and executed, the values for `x` and `y` are added to the dynamic constant pool, and the next time a variable is initialized with a seeded constant, with a probability P_{Dynamic}, these values in the dynamic pool are candidates.

To enable dynamic seeding, the class under test must be instrumented with instructions to collect values observed at runtime. In Java bytecode, one has to distinguish between integer datatypes (`int`, `short`, `byte`), which all use the same bytecode instructions for comparisons, 64-bit `long` variables with their own operations, and 32/64-bit floating point numbers `float` and `double`, again with their own comparison instructions. Each numerical type requires specific instrumentation for their corresponding comparison instructions. The instrumentation typically consists of inserting a call before the comparison that copies the values to the pool. All dynamic pools are emptied at the end of each run of the GA.

3.2.2. Seeding strings. Dynamic seeding is also valuable for collecting string values. However, here one has to distinguish between the different methods of the `String` class in Java. For example, consider the following expression:

```
if(str1.equals(str2 + "bar")) {
  // ...
}
```

Static seeding will collect the value "`bar`", but misses the value of the string variable `str2`. However, at runtime when `equals` is called, it will receive as argument the concatenated version of

Table I. Dynamic seeding of strings.

Class	Instrumented statement	Values added to dynamic pool
`String`	`s.equals(obj)`	`s`, `obj.toString()`
	`s.equalsIgnoreCase(s2)`	`s`, `s2`
	`s.startsWith(prefix[,start])`	`prefix + s`
	`s.endsWith(suffix)`	`s + suffix`
	`s.matches(regex)`	*if* `matches(s,regex)` *then* `nonMatchingRegexInstance(regex)` *else* `regexInstance(regex)`
	`s.regionMatches([ignoreCase,] start, s2, start2,len)`	`s[0,start] + s2[start2,len+start2]+ s[start+len,-]`, `s2[0,start2] + s[start,len+start] + s2[start2+len,-]`, `s[start, len+start]`, `s2[start2, len+start2]`*,†
`Pattern`	`p.matches(regex,charSeq)`	*if* `matches(charSeq.toString(),regex)` *then* `nonMatchingRegexInstance(regex)` *else* `regexInstance(regex)`
`Matcher`	`m.matches()`	*if* `matches(charSeq(m).toString(), m.pattern().pattern())` *then* `nonMatchingRegexInstance (m.pattern().pattern())` *else* `regexInstance(m.pattern(). pattern())`‡

*If `ignoreCase` equals `true`, then both `s` and `s2` are lowercased before adding anything to the pool.
†`s[start,end]` is short for the Java statement `s.subString(start,end)` (where '`-`' means end of the string).
‡`charSeq(m)` is a function that returns the char sequence stored in `m` at runtime.

`str2` and "`bar`". Again, this value is valuable for seeding. In order to retrieve this value for seeding, the bytecode is instrumented such that, rather than `String.equals`, a helper method is called that collects the argument string for the constant pool and then evaluates the string equality. Again, in the case of test generation, this replacement of the `equals` method may already happen as part of testability transformation [22], where rather than a boolean value a value providing more fine grained guidance is returned.

As another example, consider the following expression:

```
if(str1.startsWith(str2) && str1.endsWith(str3)) {
  // ...
}
```

Here, the values of `str1`, `str2` and `str3` can be collected at runtime, but unless all three values are equal, seeding the exact values will not satisfy the branching condition. Thus, rather than seeding the exact values of these three variables, instead, the concatenated string `str2 + str1` is added for the `startsWith` condition, and the concatenation of `str1` followed by `str3`, that is, `str1 + str3`, for the `endsWith` condition. Doing this leads the search to choose the right values to make both conditions evaluate to true. Again this is achieved by replacing the `String` methods with custom helper functions that collect the string values for the constant pool before evaluating the actual method. Table I shows the complete relation of string operations considered and the concrete values used for dynamic seeding.

Finally, to illustrate dynamic seeding of strings from regular expressions, consider the following conditional statement:

```
if (str.matches("^[_A-Za-z0-9-]+(\\.[_A-Za-z0-9-]+)*@[A-Za-z0-9]" +
             "+(\\.[A-Za-z0-9]+)*(\\.[A-Za-z]{2,})")) {
  // ...
}
```

For this condition to evaluate to true, the variable `str` needs to be assigned a valid email address, which is what this regular expression checks for. Here, seeding the static value of the regular expression will be of little help. However, a string variable that matches the regular expression can be generated.

The algorithm to generate strings that satisfy a regular expression (regex for short) encountered in the code starts with building an automaton that represents the regex. Then, the automaton is traversed whenever a matching or non-matching instance of a given regular expression is required, and such an instance is added to the dynamic pool. As can be seen in Table I, adding a matching or non-matching instance of a regular expression to the dynamic pool depends on whether the string (or char sequence) matches the regular expression or not. If the string (or char sequence) matches the regular expression, a non-matching string value that can potentially be used to cover the false branch of the conditional statement is added to the pool, and vice versa. In EvoSuite, regular expression operations are provided by the `dk.brics.automaton` Java library [23].

3.3. Type seeding

Consider the following snippet of code:

```
public boolean testMe(Object o) {
  String s = (String)o;
  if(s.equals("foobar")) {
    // ...
  }
}
```

During test generation, a choice has to be made as to what object to pass in as a parameter to the `testMe` method. Given the information provided in the signature, all that is known is that this method expects an instance of `java.lang.Object`. However, in Java, *every* class is a subclass of `java.lang.Object`. Thus, how should the test generator be able to select the correct class, which in this case would be `String`? A conservative approach would be to perform static type analysis or to only allow a small, fixed set of classes (e.g. `Object`, `Integer`). However, in order to cover branches like in this example, potentially *all* known classes would need to be included. Two main challenges are identified here. First, this requires knowledge about which classes are known at all, which in Java is not possible via standard API calls but requires inspection of the directories on the classpath. Second, the number of classes that are available on the classpath of a non-trivial project can be very large, often in the thousands.

This problem is amplified with Java generics [24]: in Java, a generic class has type parameters and can be instantiated for different types; for example, a collection can be parameterized with the type of values it contains. However, the Java compiler removes generic type information (‘type erasure’), and when accessing type information via Java reflection, generic parameters will be declared of type `Object`.

However, even if parameters are declared to be of type `Object`, if a specific class is expected, for example, in order to call a method of that class, then there will be a *cast* to that type. Hence, the types of interest can be determined by statically looking at all cast operations in the Java bytecode of the code under test and creating a pool of relevant types. Furthermore, looking at `instanceof` operations also provides types. When an instance of class `Object` is required by the signature of a method, then an instance of any of the types stored in the type pool can be used as parameter of an invocation of the method.

In contrast to the two previous seeding techniques, which build on existing research, the idea of seeding type information during search-based unit test generation is, to the best of our knowledge, novel in this paper.

3.4. Incorporating previous solutions

When testing classes, often one does not start from scratch but already has a certain set of test cases ready. These might for example originate from previous runs of the test generation tool, or they might be test cases written by hand by the developer of the class. Pre-existing test cases have been used in previous work as a starting point for test data generation [13, 14]. Similarly, in the context of this paper, such information can be useful to seed the initial population and mutation operators. However, in order to use this information during search, the existing test cases need to be converted to the representation used in the search algorithm. In the context of testing classes, previous test cases are also sequences of method calls, which need to be parsed and interpreted. This might be a non-trivial task, if a developer bases his tests on class hierarchies and inter-procedural test calls. Previous work [8] used a parser to read test cases without inter-procedural calls and with simple linear control flow. In this paper, unit tests are reconstructed from execution traces, which makes it easier to reconstruct inter-procedural sequences of calls: For each class in the project under test, each method entry is instrumented to add a trace entry consisting of the receiver object, called method, and parameters. A unit test can then be generated by reproducing the top-level method calls in the trace, while adhering to the parameter relations. In addition, it is possible to use the execution traces to generate sequences of calls to generate and exercise parameter objects, or objects created within other method calls.

3.4.1. Using existing unit tests. Given a set of parsed test cases T_P, the question is how to best use this information in the initial population. If T_P is too small, then using this information too much might lead to the search getting stuck in a sub-optimal area of the search landscape. On the other hand, if T_P is too large, then it may be difficult to use for producing small individual test suites, as desired in evolutionary testing of classes.

When generating the initial population for the GA in EvoSuite, each initial test suite is created with n random tests, where n is selected within predefined bounds. If such a set of parsed test cases T_P is available, then it can be used for seeding the initial population: Each time a new test case is produced, with probability P_{Clone} it is not produced randomly, but by cloning an existing test case randomly chosen from T_P. Then, to promote diversity, a number of mutations are applied to this clone, chosen randomly in the range of $[0, N_{\text{Mutation}}]$.

3.4.2. Carving objects from unit tests. Under the assumption that a set of parsed test cases T_p exists, a potential improvement consists in *carving* T_p. Carving a test suite consists in executing all its test cases in order to collect all potentially reusable objects defined therein. The resulting collection is referred to as the *object pool*. The following example illustrates how the object pool works.

The following class contains one method `foo`. In principle, and assuming that constant seeding is disabled, it is hard for a search-based unit test generation tool to achieve full branch coverage on this method. The difficulty lies in the need of having one test where the `isFive` condition is `true`, and one where it evaluates to `false`, which depends on setting the right specific value (`5`) in the argument object `dep`.

```java
public class DifficultWithoutCarving {
  public boolean foo(DifficultDependency dep) {
    if(dep.isFive())
      return true;
    else
      return false;
  }
}
```

Now, let us assume that the following test suite exists, which contains a test case that exercises the `else` branch in `foo`.

```
public class DifficultWithoutCarvingTest {
  @Test
  public void testFoo() {
    DifficultDependency dep = new DifficultDependency();
    dep.inc();
    dep.inc();
    dep.inc();
    DifficultWithoutCarving bar = new DifficultWithoutCarving();
    boolean result = bar.foo(dep);
    assertFalse(result);
  }
}
```

Executing this test suite allows to populate the object pool with two objects: one instance of class `DifficultDependency` with three invocations to method `inc` and one instance of class `DifficultWithoutCarving` with a call to method `foo` passing `dep` as input argument. Then, at test generation time, with a probability $P_{\text{ObjectPool}}$ the `dep` object will be chosen as input argument for a call to `foo`. Notice that `dep` does not exactly make the conditional evaluate to `true`: because method `inc` is only invoked three times, the call to `isFive()` in `foo` will still return false. However, seeding object `dep` does help in guiding the search towards generating a test case that covers that branch and hence achieving full branch coverage.

4. EVALUATION

Having defined the different seeding strategies, this section now addresses the following five research questions:

RQ1 : What is the impact on branch coverage of using constants from the bytecode for seeding?
RQ2 : What is the impact on branch coverage of using values observed at runtime for seeding?
RQ3 : What is the impact on branch coverage of using static type information for seeding?
RQ4 : What is the impact on branch coverage of using previous solutions for seeding?
RQ5 : What is the impact of seeding on fault finding effectiveness?

4.1. Experimental setup

4.1.1. Test generation tool. Five different sets of experiments were conducted, each one addressing one of the five research questions. In all the experiments, the default settings of EvoSuite [25] were used, and individual seeding strategies were enabled based on the research question addressed. The search is stopped after a 2-min timeout, which is a value that is suitable to achieve high coverage with EvoSuite based on past experience, but still permits thorough experiments.

In this paper, the effectiveness of each seeding strategy is measured by the branch coverage achieved when using it. In general, however, none of the strategies is specific to a particular coverage criterion. The rest of this paper thus uses the terms coverage and branch coverage interchangeably.

4.1.2. Case study objects. The choice of a case study is of paramount importance for any empirical analysis in software engineering. This paper uses two sets of benchmarks to evaluate the different seeding strategies. The SF110 corpus of open-source Java projects [6] is used to address **RQ1** to **RQ3** and **RQ5**. The SF110 corpus consists of 110 open-source Java projects from the SourceForge database[‡], totalling 23,886 Java classes. The corpus includes a statistically representative sample of one-hundred projects from SourceForge plus the ten most popular projects residing in SourceForge at the time of the corpus collection (June 2014).

Because of the large amount of classes in the corpus, a systematic approach to study the different seeding strategies was used, consisting of the following phases. First, a random stratified sample of 100 classes from SF110 was used to perform experiments to find out the best seeding configurations.

[‡]http://sourceforge.net/

To construct such a stratified sample of subject classes, iteratively, a random project in SF110 was selected, and then a random class from that project. This method provides fairness among projects with different sizes. Table II presents size metrics of the classes in the stratified sample grouped by project. Second, ten repetitions of the observed best seeding configurations were run on all classes in SF110, which lead to determine those classes on which seeding has an impact, that is, resulted in a different coverage value. Third, up to 30 repetitions with the best configuration were run again on this latter subset of classes to discern statistically significant benefits.

To address **RQ4**, Java classes with existing and available test suites were needed. The requirement of having existing test suites available rendered the SF110 corpus unsuitable for **RQ4**: only 88 classes (0.37%) therein have accompanying test suites. To satisfy this requirement, the Apache Commons repository of reusable Java components [7] is used, which has a reputation of being well tested. A sample of 28 Java projects with sizes ranging from 14 to 731 classes was collected, with 127 classes per project on average and 3558 classes altogether. A total of 1212 classes in the selected projects are accompanied by manually written test suites. Because there is no explicit mapping between classes and test suites, the following simple algorithm to match test suites and classes under test automatically was implemented: execute each test suite measuring its coverage on any class in the classpath; then, select the class on which the test suite achieves the highest coverage and assume that is the suite's target. Although not infallible in general, this method suffices for the purposes of this work.

The same systematic method described before for **RQ1-3** was followed. From the set of 1212 classes with test suites available, a first experiment was run on a randomly selected sample of 100 classes. Table III summarizes the resulting list of classes. Running all configurations on this sample lead us to determine the optimal seeding configurations, which were then applied on the initial set of 1212 classes. In order to evaluate statistical significance, 30 repetitions with the optimal configuration were run on the list of classes where seeding made a difference in coverage.

Finally, **RQ5** was addressed by means of mutation analysis, comparing the average mutation scores of test suites generated without seeding and test suites generated using optimal configurations for the three seeding strategies evaluated in **RQ1-3**. The analysis was performed on the randomly selected set of 100 classes from SF110 and refined for the subset of classes for which either of the three seeding strategies had an impact on branch coverage.

4.1.3. RQ1: Constant seeding. In the first set of experiments, 10 different values for the constant seeding probability were selected, that is, $P_{\text{Constant}} \in \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9\}$ where, intuitively, $P_{\text{Constant}} = 0$ means no seeding at all (i.e. only random values) and larger values mean increasing the probability of seeded values being used rather than random values during test generation. The value $P_{\text{Constant}} = 1$ is omitted because it means no random choice at all, which would limit the search to *exclusively* use seeded constants. First, EvoSuite was run with each possible P_{Constant} value on the stratified sample of classes from the SF110 corpus described in Table II. From this experiment, the optimal value for P_{Constant} was chosen, that is, the value with which the highest overall coverage was achieve on the sample. Then, on a second round of experiments, the optimal value for P_{Constant} was used to run 10 repetitions on all classes in SF110, with the aim of identifying classes for which static seeding has an impact on coverage. That is, only classes for which the average coverage achieved with and without constant seeding varied were included in the selection. Finally, 30 repetitions of the same configuration, that is, using the optimal value for P_{Constant}, were run on that selection of classes in order to statistically validate the benefits observed.

4.1.4. RQ2: Dynamic seeding. In the second set of experiments, to answer **RQ2**, EvoSuite was run with a range of probability values for dynamic seeding, namely, $P_{\text{Dynamic}} = \{0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$. The use of dynamically seeded values during test generation depends on the probability of using seeded primitive values (i.e. P_{Dynamic} depends on P_{Constant}). For example, using the configuration $P_{\text{Constant}} = 0.4$ and $P_{\text{Dynamic}} = 0.8$, with probability 0.4, a constant value will be used instead of a randomly generated one. In that case, with probability 0.8, the constant will be a dynamically seeded value and with probability 0.2 ($1 - P_{\text{Dynamic}}$); it will be a statically seeded value. Although the optimal value for P_{Constant} identified in isolation earlier in

Table II. Number of classes, non-commenting lines of code (calculated with JavaNCSS [26]), branches and methods for the stratified sample of 100 classes from the SF110 corpus, grouped by project.

Project	Classes	Lines of Code	Branches	Methods
a4j	2	44	28	12
apbsmem	1	78	30	18
beanbin	2	16	10	4
biblestudy	1	108	34	33
bpmail	1	15	0	10
byuic	2	19	0	11
celwars2009	2	60	18	7
corina	1	5	2	2
dash-framework	3	13	4	7
db-everywhere	2	14	0	9
dsachat	2	67	12	8
dvd-homevideo	1	1061	160	14
falselight	1	6	0	1
fixsuite	1	21	4	2
fps370	1	16	8	3
freemind	2	32	6	6
gae-app-manager	1	15	4	7
gaj	1	22	2	13
geo-google	3	64	10	39
gfarcegestionfa	3	82	28	13
greencow	1	2	0	1
heal	1	17	8	3
ifx-framework	1	2	0	2
imsmart	1	4	0	4
inspirento	1	3	0	2
io-project	2	84	66	17
ipcalculator	3	318	38	6
javabullboard	1	69	58	22
javathena	1	8	0	3
javaviewcontrol	3	1585	2444	52
jaw-br	1	41	16	2
jclo	1	8	0	1
jdbacl	3	91	34	34
jhandballmoves	2	108	94	27
jiggler	1	261	122	57
jmca	1	37	22	3
jni-inchi	1	65	8	3
jsecurity	1	46	10	7
jwbf	3	106	48	17
lavalamp	1	2	0	2
lilith	3	49	20	10
mygrid	2	91	60	18
nekomud	2	10	0	7
netweaver	3	261	108	53
newzgrabber	1	129	34	17
noen	2	19	6	10
nutzenportfolio	2	161	22	26
objectexplorer	1	9	0	6
omjstate	1	10	0	4
openhre	1	15	4	6
petsoar	1	11	4	5
quickserver	1	234	90	41
resources4j	1	60	14	14
rif	1	18	0	2
sfmis	1	78	26	5
shop	2	86	28	18
sugar	1	72	32	7

Table II. Continued.

Project	Classes	LOC	Branches	Methods
summa	1	2	0	2
sweethome3d	1	64	22	21
templatedetails	1	41	12	9
tullibee	1	20	20	3
twfbplayer	1	4	0	2
vuze	1	1	0	2
water-simulator	2	35	10	13
weka	1	145	76	22
xbus	2	117	48	13
Grand total	100	6357	3964	820

Table III. Number of classes, non-commenting lines of code (calculated with JavaNCSS [26]), branches, methods and test suites available for the random sample of 100 classes from the Apache Commons repository, grouped by project.

Project	Classes	LOC	Branches	Methods	Test suites
beanutils-1.9.2	4	73	32	24	4
chain-1.2	2	182	104	28	3
cli-1.2	1	30	10	9	1
codec-1.7	4	184	108	47	5
collections-3.2.1	19	709	274	245	20
compress-1.4.1	4	427	226	82	7
configuration-1.10	2	104	32	14	2
dbcp-1.4	1	123	34	52	2
dbutils-1.5	2	170	24	64	2
digester3-3.2	1	34	14	11	1
email-1.3.1	2	341	180	75	2
exec-1.1	2	65	70	22	2
imaging-1.0	1	3	0	2	1
io-2.4	6	888	466	156	11
jxpath-1.3	3	181	104	54	14
lang3-3.3.2	10	1417	1056	275	11
logging-1.1.1	1	81	28	15	1
math3-3.3	24	1858	798	206	26
net-3.3	1	548	64	118	2
pool-1.6	1	54	0	26	1
scxml-0.9	3	105	36	33	7
validator-1.4.0	3	319	152	69	3
vfs-2.0	3	67	10	42	8
Total	100	7963	4022	1669	136

RQ1 could have been used, a more interesting scenario is to evaluate several combinations of both probability variables. In total, $1 + 9 \times 11 = 100$ combinations were tested (the first one corresponds to $P_{\text{Constant}} = 0$, whose combination with any value for P_{Dynamic} would be redundant). The results of these experiments are used to find the combination of P_{Constant} and P_{Dynamic} that achieves the highest average coverage. Having identified this optimal combination of constant and dynamic seeding, two more experiments were conducted. In the first one, 10 repetitions were run to filter out SF110 classes for which dynamic seeding has no influence on the achieved coverage (i.e. the average coverage equals the average coverage achieved using the baseline configuration with no seeding). In the second experiment, 30 repetitions were completed for this selection of classes in order to evaluate the extent of the benefits observed.

Table IV. Ranking of P_{Constant} values according to the average coverage they achieve.

P_{Constant}	Avg. Branch Coverage
0.5	0.7273
0.7	0.7263
0.6	0.7261
0.8	0.7261
1.0	0.7260
0.9	0.7260
0.3	0.7258
0.1	0.7252
0.2	0.7249
0.4	0.7245
0.0	0.7155

4.1.5. RQ3: Type seeding. The third set of experiments looks at the impact of type seeding on coverage, in order to answer **RQ3**. The same set of combinations of values for P_{Constant} and P_{Dynamic} described for **RQ2**, that is, $1 + 9 \times 11 = 100$, was used to run EvoSuite on the stratified sample from SF110, but this time with type seeding enabled. Again, the optimal values were identified on the stratified sample, and then 10 repetitions were applied on all classes to gather the list of classes where type seeding matters (there is some difference in the average coverage with and without type seeding) and, finally, 30 repetitions completed on that list of classes to assess statistical significance.

4.1.6. RQ4: Reusing previous solutions. To gather empirical evidence for answering **RQ4**, a similar methodology as for **RQ1** to **RQ3** was followed. This time, however, classes from the Apache Commons collection, which contains considerably more test suites than the SF110 corpus, were used as subjects.

As an experimental design decision, and to prevent the combinatorial explosion of possible configurations, for these experiments, the optimal constant and dynamic seeding values with type seeding enabled were used. The combinations of possible values for P_{Clone} and N_{Mutation} were considered, in particular $P_{\text{Clone}} \in \{0.3,\ 0.5,\ 0.7,\ 0.9\}$ and $N_{\text{Mutation}} \in \{0,\ 4,\ 8\}$. This resulted in 12 configurations for EvoSuite. Running this experiment resulted in the selection of the optimal combination of P_{Clone} and N_{Mutation}, which was then used to run EvoSuite on the complete set of classes in the Apache Commons collection for which hand-written test cases were available. As discussed in Section 3.4, existing test suites can be reused not only to initialize test suites during test generation (using P_{Clone} and N_{Mutation}) but also to seed objects carved from the execution of the existing tests. Therefore, the experiments for **RQ4** were replicated to find out the optimal probability of reusing carved objects $P_{\text{ObjectPool}}$ and its impact on the search. In this case, both P_{Clone} and N_{Mutation} were reset to 0.

4.1.7. RQ5: Fault finding effectiveness. **RQ1–4** considered the effects of seeding in terms of code coverage, which often serves as the primary objective of test generation. However, ultimately, the purpose of these generated tests is to detect faults. Intuitively, higher branch coverage means higher fault detection potential: if erroneous code is not covered, then the fault cannot be detected. However, the relation between coverage and fault detection effectiveness is far from trivial [27] as fault detection might require the faulty code to be covered in a specific way. To investigate whether seeding has any effects on fault finding effectiveness, the major mutation framework [28] was used to perform mutation analysis. As customary, the mutation score is calculated as the ratio of the number of mutants killed by the generated test suite to the total number of mutants for the class under test, that is, $\text{MS} = |\text{Mutants Killed}| \,/\, |\text{Mutants Generated}|$. The values for P_{Constant} and P_{Dynamic} were set to those determined optimal in combination with type seeding in **RQ3** and used as subjects the randomly sampled set of 100 classes from SF110.

4.1.8. Experiment analysis. All experiments were repeated 30 times to take the randomness of the employed algorithms into account, and the results were analysed following the guidelines proposed by Arcuri and Briand [29]. When the performance of two different algorithms/configurations is compared (e.g. which seeding strategy leads to higher branch coverage?), the effect sizes of the comparisons are quantified with the Vargha–Delaney $\hat{A}_{12}$ statistics. In the context of this paper, the $\hat{A}_{xy}$ is an estimation of the probability that, if EvoSuite is run with seeding configuration x, it will result in better coverage than running it with configuration y. When two randomized algorithms are equivalent, then $\hat{A}_{12} = 0.5$. A high value $\hat{A}_{xy} = 1$ means that, in *all* of the 30 runs of EvoSuite with configuration x, higher coverage values were obtained than those in *all* of the 30 runs with configuration y.

Statistical difference between two algorithms/configurations on the same class has been measured with a two-tailed Mann–Whitney U-test. When analysing a whole project, we collected all the A_{12} values (one per class) and still used a U-test on the symmetry of the A_{12} around 0.5 (e.g. to check if it is the case if there are as many values above 0.5 in the succeeding text).

Table V. For each project with statistically significant results, average coverage on all of its classes when no bytecode/source code constant is seeded ($P_{\text{Constant}} = 0$) and when they are used with probability $P_{\text{Constant}} = 0.5$.

Project	# of classes	Base	Constant seeding	$\hat{A}_{12}$	% > 0.5	% < 0.5
apbsmem	16	0.62	0.76	0.70	0.50	0.00
caloriecount	202	0.72	0.74	0.59	0.18	0.01
corina	106	0.53	0.55	0.58	0.19	0.03
db-everywhere	13	0.31	0.36	0.75	0.54	0.00
dsachat	9	0.39	0.42	0.73	0.56	0.00
firebird	62	0.51	0.55	0.64	0.29	0.02
fixsuite	10	0.45	0.47	0.70	0.50	0.00
follow	13	0.70	0.71	0.63	0.15	0.00
freemind	141	0.50	0.50	0.55	0.11	0.04
heal	43	0.62	0.66	0.66	0.33	0.02
hft-bomberman	51	0.67	0.71	0.59	0.22	0.02
javaviewcontrol	7	0.62	0.68	0.72	0.29	0.00
jcvi-javacommon	116	0.63	0.65	0.58	0.20	0.00
jiggler	97	0.69	0.77	0.71	0.43	0.00
jiprof	37	0.61	0.66	0.68	0.35	0.05
jmca	27	0.56	0.81	0.85	0.67	0.00
jwbf	12	0.75	0.77	0.66	0.25	0.00
lagoon	22	0.56	0.62	0.76	0.50	0.00
liferay	1380	0.67	0.69	0.58	0.19	0.02
lilith	114	0.64	0.65	0.58	0.18	0.04
noen	58	0.69	0.77	0.68	0.40	0.02
openhre	27	0.58	0.62	0.69	0.41	0.00
openjms	194	0.56	0.60	0.62	0.27	0.02
pdfsam	89	0.60	0.67	0.70	0.42	0.00
quickserver	43	0.53	0.56	0.59	0.21	0.07
schemaspy	27	0.38	0.45	0.72	0.56	0.00
squirrel-sql	205	0.63	0.67	0.63	0.29	0.01
summa	234	0.51	0.54	0.59	0.22	0.01
twfbplayer	30	0.57	0.64	0.65	0.37	0.00
vuze	1060	0.41	0.42	0.54	0.10	0.02
weka	587	0.54	0.56	0.59	0.19	0.01
wheelwebtool	55	0.62	0.65	0.61	0.25	0.00
xbus	35	0.43	0.50	0.67	0.34	0.00
Average	155	0.57	0.62	0.59	0.32	0.01

The $\hat{A}_{12}$ of these comparisons are calculated by averaging all runs of all classes per project. On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

Because of the large number of experiments, we could not report the p-values of all the comparisons. Therefore, at times, we rather report only how often they are significant at $\alpha = 0.05$ level. Note that a *statistically significant* result (e.g. p-value very close to 0) might not have much 'practical relevance'(e.g. effect size A_{12} close to 0.5). In the extreme case, with very large sample sizes, it would be possible to obtain statistical difference on any compared distribution, as it is unlikely that they would be exactly the same.

4.2. *RQ1: Seeding constants from bytecode*

Table IV ranks the values for P_{Constant} according to the average coverage achieved by EvoSuite using each of them on the stratified sample described in Table II. Consequently, $P_{\text{Constant}} = 0.5$ is chosen as the optimal probability.

Table V compares the coverage obtained with the optimal value $P_{\text{Constant}} = 0.5$ versus the configuration with no seeding at all, that is, $P_{\text{Constant}} = 0$, on the selection of 5270 classes where constant seeding has *some* effect. These results show that seeding primitive values from the bytecode or source code is beneficial for the search, with an average $\hat{A}_{12} = 0.59$, where the coverage is increased from 57% to 62%. The magnitude of improvement is in line with expectations: non-trivial software often has *infeasible* branches, in which case 100% coverage is impossible. Without a manual verification (which would not be possible because of the large number of classes involved), it may very well be the case that 62% might already be the maximum (or close to) achievable average bytecode coverage in the analysed set of classes. Some branches, even if feasible, could not be coverable (yet) by a tool such as EvoSuite, because they rely on environment interactions like GUI and network connections (EvoSuite currently has only limited support for environment interactions, like for example using files as test data [30]). Furthermore, not all software relies on primitive constants, and even when they do, those might affect the control flow of only few branches.

Depending on how the case study is chosen, there can be a lot of variation in the results. This is why this paper uses a large and variegated case study and why the results are also grouped by project. For example, the details in Table V reveal that the improvements for a project such as `jmca` are huge. Average coverage increases from 56% to 81%, with a statistically significant $\hat{A}_{12} = 0.85$. Improvements are statistically significant for 67% of the classes in `jmca`, and there is no case for which the results are statistically worse. Overall, the results in Table V show that, even on a large and variegated case study, seeding is beneficial, and there are cases in which it can be extremely beneficial. This can, for example, be seen for projects that heavily rely on string manipulations; for example, the Java Method Cohesion Analyzer project (`jmca`) performs many different manipulations on parse tree nodes by checking the node type through explicit string comparisons.

Note also that seeding, as any heuristics, can be less beneficial or even harmful in some cases. Table V reports only classes where statistical significance was observed, but there are 74 projects, totalling 870 classes, for which no statistically significant difference was observed and where using constant seeding was better on average in 24% of the classes and worse in 1% of them. A reduction in coverage can result if the constants used for seeding are irrelevant for the search and worse than randomly generated values (for example, if a branch requires very large numerical values, but the constant pool only contains small values).

RQ1: *Seeding constants from the bytecode or source code significantly improves the performance of* EvoSuite *on 30% of the projects in the SF110 corpus.*

4.2.1. Example of seeding numeric constants. The following is an excerpt from class `com.lts.pest.gatherer.TimeConstants` from the `caloriecount` SF110 project. The method `toDurationString` takes as input argument a long integer value and returns its representation as a duration string:

```
public class TimeConstants
{
   ...
   public static final int MSEC_PER_MINUTE = 60 * 1000;
   public static final int MSEC_PER_HOUR = 60 * MSEC_PER_MINUTE;
   public static final long MSEC_PER_DAY = 24 * MSEC_PER_HOUR;
   public static final long MSEC_PER_YEAR = 365 * MSEC_PER_DAY;
   ...
   public static String toDurationString(long duration)
   {
      StringBuffer sb = new StringBuffer();

      if (duration < 1000)
      {
         sb.append(duration);
         sb.append(" msec");
      }
      else if (duration < MSEC_PER_MINUTE)
      {
         long chronons = duration/1000;
         sb.append(chronons);
         sb.append(" sec");
      }
      else ...
      else if (duration < MSEC_PER_YEAR)
      {
         long ticks = duration/MSEC_PER_DAY;
         sb.append(ticks);

         if (ticks > 1)
            sb.append(" days");
         else
            sb.append(" day");
      }
      else
      {
         sb.append(duration);
         sb.append("msec");
      }

      return sb.toString();
   }
}
```

The four numeric constants defined in this class can be statically seeded as potentially useful values when generating tests for this class. In concrete and as a result of the constant folding compile-time optimization, the values `60000` (`MSEC_PER_MINUTE`), `3600000` (`MSEC_PER_HOUR`), `86400000` (`MSEC_PER_DAY`) and `31536000000` (`MSEC_PER_YEAR`) are available in the class's bytecode constant pool. By seeding these constants, the probabilities of reaching each of the conditional branches in the method improve notably, enabling EvoSuite to increase the coverage achieved on this class from 36% to 88% on average. The following are examples of test cases generated thanks to the seeding of constant values:

```
 @Test
 public void test04() throws Throwable {
    String string0 = TimeConstants.toDurationString(31535999985L);
    assertEquals("364 days", string0);
 }
 @Test
 public void test08() throws Throwable {
    String string0 = TimeConstants.toDurationString(3600046L);
    assertEquals("1 hour", string0);
 }
```

```
@Test
public void test09() throws Throwable {
   String string0 = TimeConstants.toDurationString(31536000089L);
   assertEquals("31536000089msec", string0);
}
```

Interestingly, observe that the third test case reveals what seems to be a fault in the method `toDurationString`: there is a missing space between the numeric representation and the `"msec"` string.

4.2.2. Example of seeding string constants. Class `org.exolab.jms.selector.Identifier` from the SF110 project `openjms` provides an example of when seeding string constants is beneficial. Covering the `then` branch of the `if`-statement in the constructor of the class requires evaluating method `Identifiers.isJMSIdentifier` to true, which can only happen if the string input argument starts with a specific prefix.

```
class Identifier implements Expression {
   ...
   public Identifier(final String name) throws SelectorException {
      _name = name;
      if (Identifiers.isJMSIdentifier(_name)) {
         if (!Identifiers.isQueryableJMSIdentifier(_name)) {
            throw new SelectorException("Invalid header field: " + _name);
         }
         _headerField = true;
      } else {
         _headerField = false;
      }
   }
   ...
}
```

Through seeding of string constants, EvoSuite is able to generate test cases that exercise the desired branch, for example,

```
@Test
public void test3() throws Throwable {
   Identifier identifier0 = new Identifier("JMSTimestamp");
   MapMessageImpl mapMessageImpl0 = new MapMessageImpl();
   SObject sObject0 = identifier0.evaluate((Message) mapMessageImpl0);
   assertNull(sObject0);
}
```

4.3. *RQ2: Seeding values at runtime*

Figure 1 presents a heatmap that shows the coverage achieved for each combination of static and dynamic seeding on the SF110 stratified sample (Table II). Plotted coverage values are calculated as means of all classes and all runs for each class. Although other combinations, for instance $P_{\text{Constant}} = 0.8$ and $P_{\text{Constant}} = 0.6$, also achieve similar coverage values, the optimal values are $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$.

The average coverage values obtained on the 5303 classes on which dynamic seeding makes a difference using the best combination, $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$, are compared with those obtained with no seeding at all. Table VI presents the results of this experiment. Moreover, Table VII reports on the comparison between the best combination of P_{Constant} and P_{Dynamic} and the best configuration for constant seeding in isolation (i.e. $P_{\text{Constant}} = 0.5$).

When compared with the default configuration of EvoSuite with no seeding, constant and dynamic seeding together achieve an overall average coverage increase of 5% on 41 projects, with 32% of classes in each project showing statistically significant positive difference on average. The

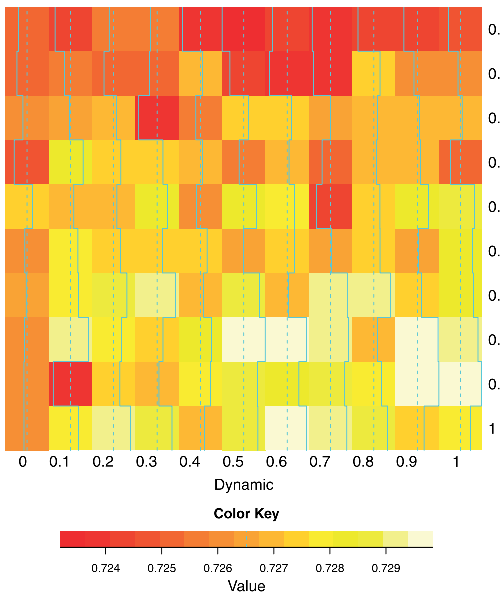


Figure 1. Heatmap showing the average coverage achieved by each combination of P_{Constant} and P_{Dynamic} with type seeding *disabled*. The lighter the colour, the higher the coverage achieved.

Table VI. For each project with statistically significant results, comparison of average coverage between the base configuration with no seeding and the best configuration with both constant and dynamic seeding enabled ($P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$).

Project	# of classes	Base	Constant+dynamic	$\hat{A}_{12}$	% > 0.5	% < 0.5
caloriecount	202	0.72	0.74	0.60	0.22	0.03
corina	106	0.53	0.56	0.62	0.25	0.00
db-everywhere	13	0.31	0.34	0.64	0.31	0.00
diebierse	8	0.64	0.77	0.73	0.38	0.00
echodep	24	0.40	0.48	0.63	0.29	0.00
fim1	18	0.52	0.54	0.60	0.22	0.00
firebird	62	0.51	0.55	0.64	0.31	0.00
freemind	141	0.50	0.51	0.56	0.15	0.04
heal	43	0.62	0.67	0.68	0.37	0.02
hft-bomberman	51	0.67	0.71	0.58	0.22	0.02
inspirento	9	0.72	0.74	0.60	0.11	0.00
javabullboard	24	0.59	0.61	0.63	0.25	0.00
javathena	16	0.63	0.69	0.71	0.44	0.00
javaviewcontrol	7	0.62	0.70	0.78	0.43	0.00
jcvi-javacommon	116	0.63	0.67	0.63	0.28	0.03
jdbacl	52	0.69	0.74	0.65	0.35	0.00
jhandballmoves	27	0.66	0.67	0.59	0.22	0.04
jiggler	97	0.69	0.80	0.75	0.55	0.02
jiprof	37	0.61	0.66	0.68	0.41	0.08
jmca	27	0.56	0.81	0.86	0.78	0.00
jsecurity	53	0.59	0.61	0.57	0.15	0.02
jwbf	12	0.75	0.78	0.67	0.33	0.00
lagoon	22	0.56	0.63	0.76	0.50	0.00
liferay	1380	0.67	0.69	0.59	0.20	0.01
lilith	114	0.64	0.65	0.58	0.19	0.00
newzgrabber	8	0.41	0.46	0.68	0.50	0.00
noen	58	0.69	0.73	0.63	0.38	0.16
openhre	27	0.58	0.63	0.68	0.44	0.04
openjms	194	0.56	0.60	0.63	0.28	0.01
pdfsam	89	0.60	0.67	0.69	0.44	0.02

Table VI. Continued.

Project	# of classes	Base	Constant+dynamic	$\hat{A}_{12}$	% > 0.5	% < 0.5
schemaspy	27	0.38	0.43	0.64	0.30	0.00
shop	19	0.47	0.49	0.58	0.11	0.00
squirrel-sql	205	0.63	0.67	0.65	0.33	0.02
sugar	11	0.49	0.57	0.75	0.45	0.00
summa	234	0.51	0.54	0.62	0.29	0.03
sweethome3d	86	0.47	0.48	0.55	0.14	0.03
twfbplayer	30	0.57	0.67	0.73	0.50	0.00
vuze	1060	0.41	0.42	0.53	0.11	0.05
weka	587	0.54	0.56	0.61	0.21	0.02
wheelwebtool	55	0.62	0.66	0.61	0.27	0.04
xbus	35	0.43	0.49	0.64	0.31	0.06
Average	132	0.57	0.62	0.60	0.32	0.02

The $\hat{A}_{12}$ are in respect to the latter configuration and are calculated by averaging all runs of all classes per project. On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

Table VII. For each project with statistically significant results, comparison of average coverage between the best configuration for constant seeding (PConstant = 0.5) and the best configuration with both constant and dynamic seeding enabled ($P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$).

Project	# of classes	Constant	Constant+dynamic	$\hat{A}_{12}$	% > 0.5	% < 0.5
apbsmem	16	0.76	0.63	0.39	0.00	0.25
corina	106	0.55	0.56	0.54	0.12	0.04
javathena	16	0.65	0.69	0.60	0.31	0.06
jcvi-javacommon	116	0.65	0.67	0.55	0.14	0.05
jdbacl	52	0.70	0.74	0.62	0.31	0.04
jhandballmoves	27	0.68	0.67	0.55	0.15	0.04
jiggler	97	0.77	0.80	0.55	0.11	0.03
lagoon	22	0.62	0.63	0.59	0.23	0.05
schemaspy	27	0.45	0.43	0.41	0.04	0.19
squirrel-sql	205	0.67	0.67	0.55	0.16	0.03
sugar	11	0.54	0.57	0.67	0.45	0.00
sweethome3d	86	0.47	0.48	0.55	0.08	0.02
twfbplayer	30	0.64	0.67	0.62	0.23	0.00
vuze	1060	0.42	0.42	0.49	0.05	0.07
weka	587	0.56	0.56	0.52	0.09	0.04
Average	164	0.61	0.61	0.52	0.16	0.06

The $\hat{A}_{12}$ are in respect to the latter configuration and are calculated by averaging all runs of all classes per project. On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

coverage increase using this configuration seems to be more moderate in general, but, at the same time, more consistent across projects. There are more projects for which combining constant and dynamic seeding helps in achieving higher coverage. On the downside, there are some projects, for example, `dsachat` and `fixsuite`, for which a significant benefit was observed when using only constant seeding (Table V) but that increase is not observed when combining constant and dynamic seeding, which suggests that the globally 'optimal' combination of probabilities for P_{Constant} and P_{Dynamic} is sometimes detrimental.

The results observed in Table VII are in line with the observation that there is not one specific combination of probabilities that provides optimal values on all projects. There are only 12 projects for which the combination of both techniques with $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$ achieves significantly better results than constant seeding only ($P_{\text{Constant}} = 0.5$ and $P_{\text{Dynamic}} = 0$), which also suggests that constant seeding often contributes the most of the two techniques to coverage increase. In fact, for three projects, namely, `apbsmem`, `schemaspy` and `vuze`, constant seeding is significantly better than the combination of both techniques. For example, this may be a result of too many irrelevant values being added to the constant pool; in our implementation, all string values are added to the pool, although values observed in the class under test may be more relevant than values observed in other classes. Nevertheless, in the presence of regular expressions and pattern matching, as in project `sugar` for instance, only dynamic seeding can help enhance coverage.

RQ2: *Seeding values encountered at execution time significantly improves the average coverage obtained for 37% of the SF110 projects. However, the choice of dynamic seeding probability can also affect the performance of constant seeding negatively.*

4.3.1. Example of dynamically seeded strings. The utility class `org.jcvi.jillion.trace.fastq.IlluminaUtil` from the JCVI Java Common project (`jcvi-javacommon`) exemplifies the potential of using dynamic seeding of strings to increase code coverage. Method `isIlluminaRead` receives a string as input argument, and its output depends on whether the input string matches the pattern `NAME_PATTERN` or the pattern `CASAVA_1_8_PATTERN`.

```java
public final class IlluminaUtil {

   private static final Pattern NAME_PATTERN = Pattern.compile(
     "^(SOLEXA\\d+).*:(\\d+):(\\d+):(\\d+):(\\d+)#(\\D+)?(\\d+)?\\/(\\d+)$");
   private static final Pattern CASAVA_1_8_PATTERN = Pattern.compile(
     "^(\\S+):(\\d+):(\\S+):(\\d+):(\\d+)#(\\D+)?(\\d+)?\\/(\\d+)$");
   ...
   public static boolean isIlluminaRead(String readId){
      if(readId == null){
         throw new NullPointerException();
      }
      Matcher matcher = NAME_PATTERN.matcher(readId);
      if( matcher.matches()){
         return true;
      }

      return CASAVA_1_8_PATTERN.matcher(readId).matches();
   }
   ...
}
```

Seeding matching strings for both patterns, as described in Table I, allows EvoSuite to achieve full coverage of method `isIlluminaRead` by generating, for example, the following test case:

```java
 @Test
 public void test02() throws Throwable {
    boolean boolean0 = IlluminaUtil.isIlluminaRead("SOLEXA0:0:0:0:0#/0");
    assertTrue(boolean0);
 }
```

For this particular class, the average coverage obtained in the experiments without dynamic seeding is 69%. However, dynamically seeding strings matching the patterns in the class leads EvoSuite to achieve a substantial coverage increase, reaching 100% on average.

4.4. RQ3: Seeding types

Figure 2 shows the average coverage achieved for each combination of P_{Constant} and P_{Dynamic} when type seeding is enabled. The heatmap shows that with type seeding enabled, $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$ obtained the highest average coverage. Using these values, an experiment is run on the selection of 5491 classes where type seeding had some impact. Tables VIII and IX compare the results obtained using this optimal configuration and those obtained using,

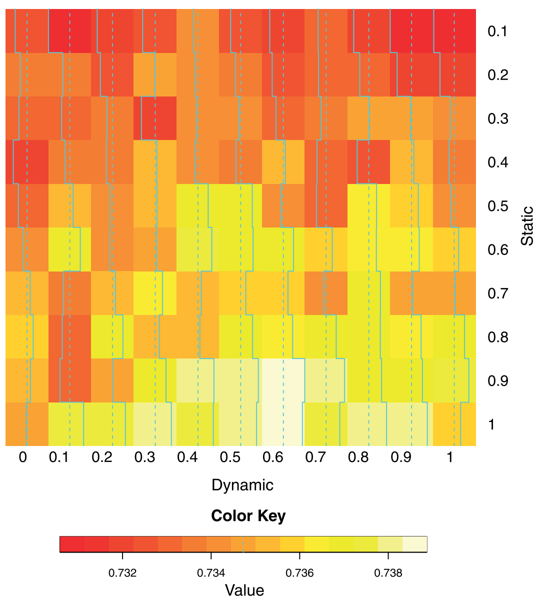


Figure 2. Heatmap shows the average coverage achieved by each combination of P_{Constant} and P_{Dynamic} with type seeding *enabled*. The lighter the colour, the higher the coverage achieved.

Table VIII. For each project with statistically significant results, comparison of average coverage between the best configuration with constant, dynamic and type seeding enabled ($P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$) and the base configuration with no seeding.

Project	# of classes	Base	Types	$\hat{A}_{12}$	% > 0.5	% < 0.5
apbsmem	16	0.62	0.73	0.66	0.38	0.00
beanbin	13	0.59	0.67	0.72	0.54	0.00
caloriecount	202	0.72	0.77	0.65	0.31	0.03
checkstyle	32	0.44	0.53	0.60	0.34	0.03
corina	106	0.53	0.57	0.62	0.25	0.02
db-everywhere	13	0.31	0.36	0.78	0.54	0.00
diebierse	8	0.64	0.77	0.77	0.50	0.00
dsachat	9	0.39	0.45	0.77	0.67	0.11
echodep	24	0.40	0.52	0.67	0.42	0.00
ext4j	9	0.63	0.86	0.86	0.67	0.00
fim1	18	0.52	0.57	0.65	0.39	0.06
firebird	62	0.51	0.55	0.66	0.35	0.02
fixsuite	10	0.45	0.48	0.68	0.40	0.00
freemind	141	0.50	0.51	0.55	0.14	0.05
heal	43	0.62	0.68	0.71	0.47	0.02
hft-bomberman	51	0.67	0.73	0.61	0.22	0.02
javabullboard	24	0.59	0.64	0.67	0.38	0.04
javathena	16	0.63	0.68	0.68	0.44	0.06
jcvi-javacommon	116	0.63	0.69	0.69	0.49	0.03
jdbacl	52	0.69	0.74	0.66	0.38	0.02
jhandballmoves	27	0.66	0.71	0.64	0.37	0.00
jiggler	97	0.69	0.80	0.75	0.56	0.00
jiprof	37	0.61	0.67	0.71	0.46	0.05
jmca	27	0.56	0.82	0.83	0.74	0.00
jsecurity	53	0.59	0.63	0.61	0.26	0.02
jwbf	12	0.75	0.79	0.75	0.42	0.00
lagoon	22	0.56	0.64	0.76	0.55	0.00
lhamacaw	35	0.58	0.68	0.65	0.34	0.06
liferay	1380	0.67	0.70	0.59	0.24	0.05
lilith	114	0.64	0.70	0.68	0.40	0.03

Table VIII. Continued.

Project	# of classes	Base	Types	$\hat{A}_{12}$	% > 0.5	% < 0.5
netweaver	52	0.80	0.83	0.58	0.21	0.02
newzgrabber	18	0.41	0.46	0.64	0.33	0.06
noen	58	0.69	0.76	0.72	0.53	0.14
objectexplorer	21	0.64	0.66	0.61	0.24	0.05
openhre	27	0.58	0.63	0.70	0.48	0.00
openjms	194	0.56	0.62	0.67	0.36	0.02
pdfsam	89	0.60	0.69	0.71	0.47	0.01
quickserver	43	0.53	0.58	0.63	0.30	0.02
schemaspy	27	0.38	0.46	0.76	0.56	0.00
squirrel-sql	205	0.63	0.68	0.64	0.38	0.04
sugar	11	0.49	0.67	0.85	0.64	0.00
summa	234	0.51	0.56	0.64	0.32	0.03
sweethome3d	86	0.47	0.49	0.55	0.15	0.02
tullibee	8	0.72	0.77	0.77	0.50	0.00
twfbplayer	30	0.57	0.68	0.75	0.47	0.03
vuze	1060	0.41	0.43	0.55	0.16	0.04
weka	587	0.54	0.59	0.67	0.35	0.01
wheelwebtool	55	0.62	0.67	0.65	0.33	0.02
xbus	35	0.43	0.53	0.72	0.49	0.03
xisemele	6	0.82	0.87	0.78	0.67	0.00
Average	112	0.58	0.65	0.62	0.41	0.03

The $\hat{A}_{12}$ are in respect to the latter configuration and are calculated by averaging all runs of all classes per project. On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

Table IX. For each project with statistically significant results, comparison of average coverage between the best configuration with constant, dynamic and type seeding enabled ($P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$) and the best configuration for constant and dynamic seeding with type seeding disabled ($P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.9$).

Project	# of classes	Constant+dynamic	Types	$\hat{A}_{12}$	% > 0.5	% < 0.5
caloriecount	202	0.74	0.77	0.54	0.10	0.02
checkstyle	32	0.46	0.53	0.57	0.16	0.03
db-everywhere	13	0.34	0.36	0.64	0.38	0.00
ext4j	9	0.64	0.86	0.80	0.56	0.00
hft-bomberman	51	0.71	0.73	0.54	0.06	0.02
jcvi-javacommon	116	0.67	0.69	0.56	0.17	0.02
jtailgui	11	0.56	0.66	0.65	0.27	0.00
lilith	114	0.65	0.70	0.62	0.28	0.04
noen	58	0.73	0.76	0.59	0.14	0.00
openjms	194	0.60	0.62	0.56	0.15	0.02
schemaspy	27	0.43	0.46	0.62	0.30	0.00
squirrel-sql	205	0.67	0.68	0.48	0.11	0.14
summa	234	0.54	0.56	0.52	0.07	0.05
vuze	1060	0.42	0.43	0.53	0.08	0.02
weka	587	0.56	0.59	0.57	0.16	0.02
wheelwebtool	55	0.66	0.67	0.54	0.07	0.00
Average	186	0.59	0.63	0.55	0.19	0.02

The $\hat{A}_{12}$ are in respect to the latter configuration and are calculated by averaging all runs of all classes per project. On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

respectively, no seeding at all and the best combination for constant and dynamic seeding with type seeding off.

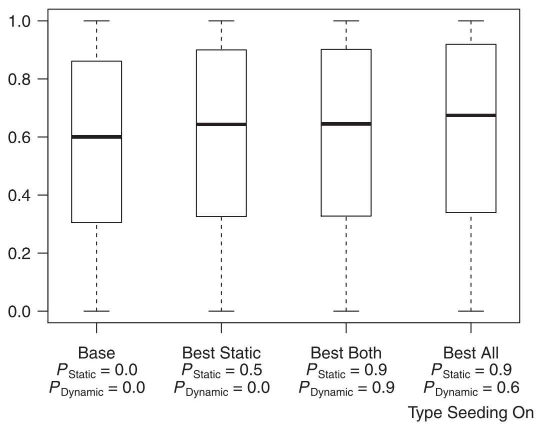


Figure 3. Boxplots comparing best seeding configurations versus base configuration with no seeding.

On average, significantly higher coverage is obtained for 41% of the classes in these 50 projects, with an overall coverage increase of 7% (from 58% to 65%) and an average 0.62 effect size. As per Table IX, when type seeding is compared with the best combination of constant and dynamic seeding without type seeding, significantly better results are observed for 15 projects, and significantly worse results are observed for one project (`squirrel-sql`). This indicates that there are cases where static and dynamic seeding together are more beneficial than type seeding alone. On average, a 5% increase is observed in these projects with 19% of classes reaching significantly better results. An interesting case is project `ext4j`, for which the average coverage increases by 22% when using type seeding. Overall, these results show that seeding type information also contributes towardsreaching higher coverage. As the next example will help demonstrate, type seeding can be useful, if not essential, in the presence of object-oriented features such as polymorphism, method overloading or object typecasting.

RQ3: *Seeding type information leads to significantly better coverage on 50 of the SF110 projects, and improves over dynamic seeding on 15 projects.*

To conclude the analysis of the results obtained for **RQ1**, **RQ2** and **RQ3**, Figure 3 presents boxplots that summarize the comparison of the average coverage obtained for the four seeding configurations evaluated on *all* 5992 classes where some difference was observed for any of the seeding techniques. The plots show that all the seeding techniques discussed improve the coverage achieved by the baseline configuration without seeding. Type seeding in combination with dynamic seeding seems to offer the most noticeable improvements, and dynamic seeding results in only moderate enhancement over constant seeding.

4.4.1. Example of type seeding. Consider the following class `ListChannelHelper` from the SF110 project `caloriecount`:

```java
public class ListChannelHelper extends ListenerHelper
{
   @Override
   public void notifyListener(Object client, int type, Object data)
   {
      ListChannelListener listener = (ListChannelListener) client;
      ListChannelEvent event = (ListChannelEvent) data;
      ListChannel list = event.getList();
      int oldIndex = event.getOldIndex();
      int newIndex = event.getNewIndex();
      switch (event.getEventType())
      {
          case ListChannelEvent.EVENT_ADD :
             listener.addElement(event, list, oldIndex);
             break;
          case ListChannelEvent.EVENT_ALL_CHANGED :
             listener.allChanged(event, list);
             break;
          case ListChannelEvent.EVENT_MOVE :
             listener.moveElement(event, list, oldIndex, newIndex);
             break;
          case ListChannelEvent.EVENT_REMOVE :
             listener.removeElement(event, list, oldIndex);
             break;
          default :
             throw new IllegalArgumentException();
       }
    }
}
```

Without type seeding, EvoSuite is unable to cover any of the cases in the `switch`-statement in method `notifyListener`. On the contrary, using type seeding information, EvoSuite realises that the input arguments `client` and `data` must be of type `ListChannelListener` and `ListChannelEvent`, respectively, which is enough to generate tests covering all of the cases in the `switch`. The following is thus one of the test cases EvoSuite generates for class `ListChannelHelper`. Notice that although there is no direct call to `ListChannelHelper`, an instance of the class is created in the constructor of class `ListChannel`, and a call to its `notifyListener` method is triggered by the `ListChannel.add` method.

```
@Test
public void test0() throws Throwable {
   ListChannel listChannel0 = new ListChannel();
   ListChannelEvent listChannelEvent0 = new ListChannelEvent((-2939),
       listChannel0, (-2939));
   ListChannelListenerAdaptor listChannelListenerAdaptor0 = new
       ListChannelListenerAdaptor();
   listChannel0.addListener((ListChannelListener)
       listChannelListenerAdaptor0);
   listChannel0.add((Object) listChannelEvent0);
   listChannel0.remove((Object) listChannelEvent0);
   List list0 = listChannel0.myList;
   listChannel0.setList(list0);
   assertEquals(0, list0.size());
}
```

4.5. RQ4: Incorporating previous solutions

Figure 4 plots the results obtained for each combination of values for P_{Clone} and N_{Mutation} when running EvoSuite on the random sample from Apache Commons described in Section 4.1.2 (Table III). The plot suggests that higher values for N_{Mutation} are preferable, whereas P_{Clone} does not show a clear trend. Based on these results, $P_{\text{Clone}} = 0.9$ and $N_{\text{Mutation}} = 8$ were chosen as the combination of values that achieves the highest coverage benefits. Furthermore, Table X provides the optimal value for the probability of reusing objects carved from existing tests $P_{\text{ObjectPool}} = 0.9$.

Using these optimal configurations obtained on the list of classes from Table III, EvoSuite was run on the complete set of classes for which hand-written test cases were available (1212 classes). The performance of the optimal test initialization configuration was compared with the case of no seeding from hand-written test cases, that is, $P_{\text{Clone}} = 0$ and $N_{\text{Mutation}} = 0$, that is, the default “random” initialization strategy. Table XI presents the results of these experiments; Figure 5(a) summarizes these results in a boxplot. To study the effects of reusing objects carved from existing test suites, the performance of the optimal probability $P_{\text{ObjectPool}} = 0.9$ and the baseline configuration

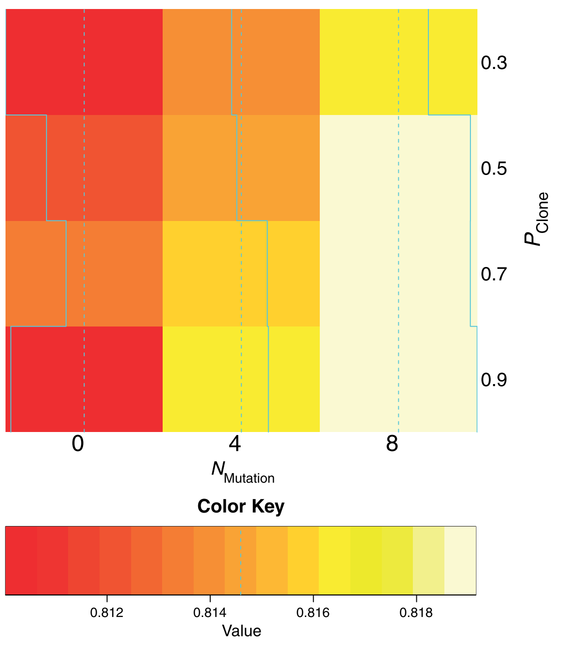


Figure 4. Heatmap that shows the average coverage achieved by each combination of values for P_{Clone} and N_{Mutation}. Best values are used for P_{Constant}, P_{Dynamic} and P_{Types}. The lighter the colour, the higher the coverage achieved.

Table X. Ranking of $P_{\text{ObjectPool}}$ values according to the average coverage they achieve.

$P_{\text{ObjectPool}}$	Avg. Coverage
0.9	0.8161
0.8	0.8154
0.6	0.8140
0.7	0.8127
0.4	0.8122
0.5	0.8106
0.3	0.8103
0.2	0.8099
0.1	0.8086
1.0	0.7887
0.0	0.7856

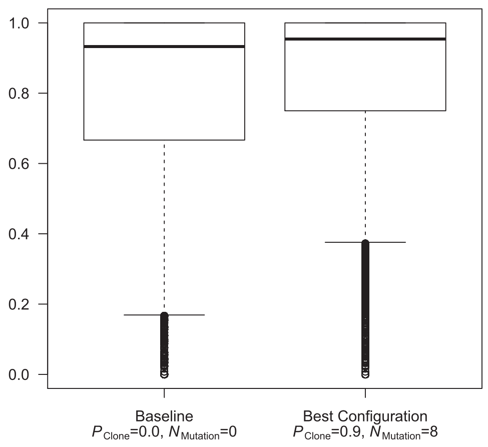
Table XI. For each project, comparison of coverage achieved using the baseline configuration with random test initialization ($P_{\text{Clone}} = 0.0$ and $N_{\text{Mutation}} = 0$) and the best configuration for test initialization from previous solutions ($P_{\text{Clone}} = 0.9$ and $N_{\text{Mutation}} = 8$).

	# of	Coverage					Seeded test suites	
Project	classes	Baseline	Seeding	$\hat{A}_{12}$	% > 0.5	% < 0.5	Average # of tests	Average coverage
beanutils-1.9.2	65	0.86	0.88	**0.57**	0.23	0.03	11	0.50
chain-1.2	12	0.75	0.75	0.48	0.08	0.25	9	0.33
cli-1.2	12	0.93	0.96	0.63	0.42	0.00	13	0.49
codec-1.7	37	0.98	0.98	0.49	0.05	0.11	6	0.60
collections-3.2.1	144	0.80	0.83	**0.57**	0.25	0.06	65	0.41
compress-1.4.1	40	0.71	0.72	0.54	0.10	0.03	3	0.28
configuration-1.10	61	0.78	0.83	**0.61**	0.31	0.07	16	0.32
dbcp-1.4	21	0.72	0.70	0.52	0.19	0.14	9	0.14

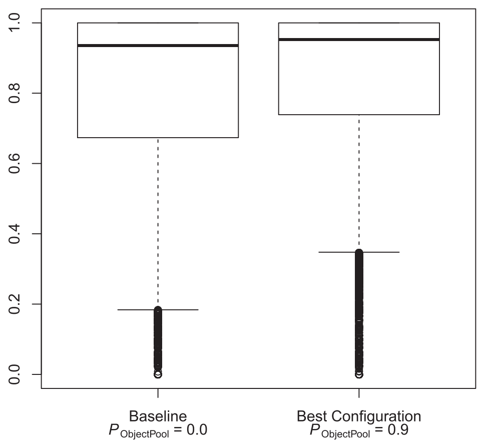
Table XI. Continued.

Project	# of classes	Coverage Baseline	Coverage Seeding	$\hat{A}_{12}$	% > 0.5	% < 0.5	Seeded test suites Average # of tests	Seeded test suites Average coverage
dbutils-1.5	20	0.50	0.51	0.51	0.05	0.05	5	0.24
digester3-3.2	15	0.83	0.85	0.54	0.13	0.00	4	0.10
discovery-0.5	1	0.93	0.93	0.47	0.00	0.00	12	0.50
email-1.3.1	10	0.72	0.71	0.46	0.10	0.20	4	0.32
exec-1.1	7	0.77	0.69	0.58	0.29	0.00	5	0.27
fileupload-1.2.2	8	0.59	0.59	0.51	0.12	0.00	1	0.36
imaging-1.0	22	0.64	0.62	0.45	0.00	0.23	2	0.12
io-2.4	65	0.88	0.90	**0.56**	0.23	0.05	5	0.36
jexl-2.1.1	9	0.49	0.50	0.50	0.00	0.11	8	0.18
jxpath-1.3	10	0.65	0.73	0.69	0.40	0.00	17	0.22
lang3-3.3.2	95	0.85	0.88	**0.61**	0.29	0.04	14	0.50
logging-1.1.1	5	0.87	0.87	0.49	0.00	0.00	3	0.37
math3-3.3	388	0.82	0.86	**0.57**	0.26	0.04	7	0.43
net-3.3	34	0.68	0.80	**0.66**	0.38	0.03	6	0.50
pool-1.6	13	0.52	0.51	0.49	0.00	0.15	1	0.02
proxy-1.0	22	0.54	0.65	0.61	0.23	0.00	3	0.50
scxml-0.9	26	0.73	0.74	0.51	0.08	0.12	4	0.27
validator-1.4.0	34	0.84	0.89	**0.64**	0.32	0.00	6	0.52
vfs-2.0	29	0.56	0.57	0.53	0.07	0.00	0.2	0.03
Average	45	0.74	0.76	0.57	0.17	0.06	9	0.33

In both configurations, $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$. Bold $\hat{A}_{12}$ indicate statistically significance (p-value lower than 0.05). On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0:05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.



(a) Boxplot comparing the best test initialization configuration ($P_{\text{Clone}} = 0.9$ and $N_{\text{Mutation}} = 8$) and the baseline configuration with random test initialization ($P_{\text{Clone}} = 0.0$ and $N_{\text{Mutation}} = 0$).



(b) Boxplot comparing the best object seeding configuration ($P_{\text{ObjectPool}} = 0.9$) and the baseline configuration without object seeding ($P_{\text{ObjectPool}} = 0$).

Figure 5. Boxplots comparing techniques for seeding from previous solutions.

with $P_{\text{ObjectPool}} = 0$ was compared. The results of this experiment are presented in Table XII; Figure 5(b) summarizes these results in a boxplot. Recall that in all these two sets of experiments for **RQ4**, the optimal constant and dynamic seeding values $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$ with type seeding enabled were used.

For 10 projects, there is statistically significant evidence that seeding from existing test suites provides better results either by reusing statement sequences or by reusing instantiated objects (projects with bold $\hat{A}_{12}$ in Tables XI and XII, respectively).

Table XII. For each project, comparison of coverage achieved using the baseline configuration with no seeding ($P_{\text{ObjectPool}} = 0.0$) and the best object seeding configuration ($P_{\text{ObjectPool}} = 0.9$).

	# of	Coverage					Seeded test suites	
Project	classes	Baseline	Seeding	$\hat{A}_{12}$	% > 0.5	% < 0.5	Average # of tests	Average coverage
beanutils-1.9.2	65	0.85	0.90	**0.56**	0.14	0.03	11	0.55
chain-1.2	12	0.74	0.76	0.51	0.08	0.08	8	0.33
cli-1.2	12	0.94	0.96	0.61	0.33	0.00	13	0.48
codec-1.7	37	0.98	0.98	0.50	0.03	0.03	6	0.59
collections-3.2.1	144	0.80	0.83	**0.54**	0.20	0.10	66	0.41
compress-1.4.1	40	0.72	0.74	0.55	0.17	0.10	3	0.28
configuration-1.10	61	0.78	0.82	**0.62**	0.31	0.07	10	0.30
dbcp-1.4	21	0.72	0.75	**0.65**	0.48	0.05	8	0.14
dbutils-1.5	20	0.49	0.50	0.50	0.05	0.00	5	0.24
digester3-3.2	15	0.84	0.86	**0.55**	0.07	0.00	5	0.10
discovery-0.5	1	0.94	0.92	0.39	0.00	1.00	12	0.50
email-1.3.1	10	0.72	0.75	0.54	0.10	0.00	3	0.35
exec-1.1	7	0.77	0.78	0.52	0.14	0.00	5	0.31
fileupload-1.2.2	8	0.60	0.62	0.55	0.12	0.00	1	0.36
imaging-1.0	22	0.61	0.60	0.49	0.09	0.18	2	0.13
io-2.4	65	0.88	0.90	**0.54**	0.17	0.05	5	0.36
jexl-2.1.1	9	0.51	0.51	0.56	0.22	0.11	2	0.17
jxpath-1.3	10	0.65	0.66	0.45	0.20	0.20	6	0.16
lang3-3.3.2	95	0.85	0.87	**0.56**	0.15	0.00	14	0.51
logging-1.1.1	5	0.86	0.86	0.57	0.20	0.00	3	0.37
math3-3.3	388	0.83	0.85	**0.54**	0.18	0.06	7	0.43
net-3.3	34	0.69	0.79	**0.66**	0.35	0.00	6	0.50
pool-1.6	13	0.51	0.51	0.49	0.00	0.00	1	0.02
proxy-1.0	22	0.53	0.53	0.50	0.00	0.00	3	0.49
scxml-0.9	26	0.72	0.73	0.52	0.12	0.08	4	0.26
validator-1.4.0	34	0.84	0.85	0.54	0.09	0.03	6	0.52
vfs-2.0	29	0.56	0.57	0.52	0.07	0.00	0.2	0.03
Average	45	0.74	0.76	0.55	0.15	0.08	8	0.33

In both configurations, $P_{\text{Constant}} = 0.9$ and $P_{\text{Dynamic}} = 0.6$. Bold $\hat{A}_{12}$ indicate statistically significance (p-value lower than 0.05). On higher granularity, it is reported the percentage % of classes for which there is a significant (p-value lower than 0.05) $\hat{A}_{12} > 0.5$ and $\hat{A}_{12} < 0.5$.

On average, both cloning and mutating existing test cases and seeding objects constructed in these test cases led to a 2% increase in coverage. However, it is important to notice that in this case, in contrast to the experiments that were run for the other research questions, there is a human factor involved, that is, the quality of the hand-written test cases. On one hand, if the test cases are poor, then even the best seeding strategy would likely have little impact on performance. On the other hand, if the hand-written test cases are optimal, then it would be pointless to try to improve upon them. An alternative conjecture is that manually written tests tend to cover the ‘easy’parts of the code under test, which state-of-the-art test generation techniques can routinely cover as well.

When run on the set of classes considered for this research question, EvoSuite can achieve 74% average coverage even without any seeding enabled. The existing hand-written test suites, on the other hand, cover much less of the code under test (average 33%). Because of the limitation of the current implementation of the carving approach, this coverage is not necessarily equivalent to the real coverage of the existing test suites. Nevertheless, reusing even these limited-coverage carved test cases can already help EvoSuite reach 76% coverage (a 2% increase). To gain more insight on the relation between the coverage achieved using seeding and the carved coverage of the existing test suites, Figures 6 and 7 present plots depicting how the performance of EvoSuite with seeding (labelled ‘Coverage’ on the y -axis) is affected by the coverage of the existing test suites (labelled ‘Carved Coverage’ on the x -axis). Overall, these results can be seen as a proof of concept that shows the ability of SBST techniques to successfully exploit existing solutions and improve upon them. The boxplots indicate that seeding existing test suites, both for test initialization and for reuse

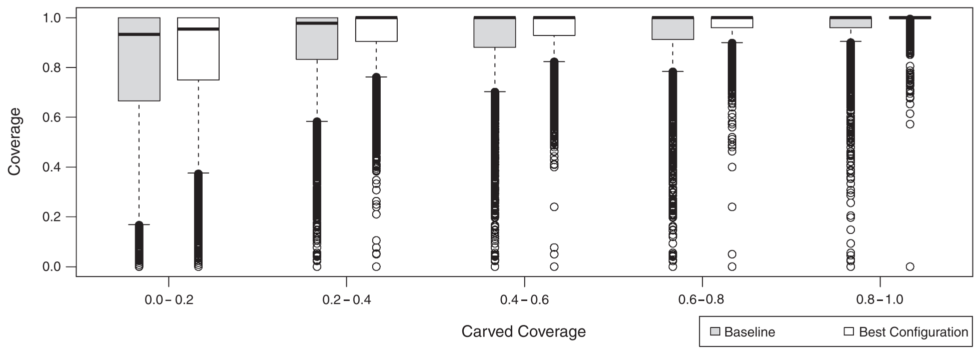


Figure 6. Boxplots comparing the best test initialization configuration ($P_{\text{Clone}} = 0.9$ and $N_{\text{Mutation}} = 8$) and the baseline configuration with random test initialization ($P_{\text{Clone}} = 0.0$ and $N_{\text{Mutation}} = 0$) for each interval of coverage achieved by the test suites available for carving.

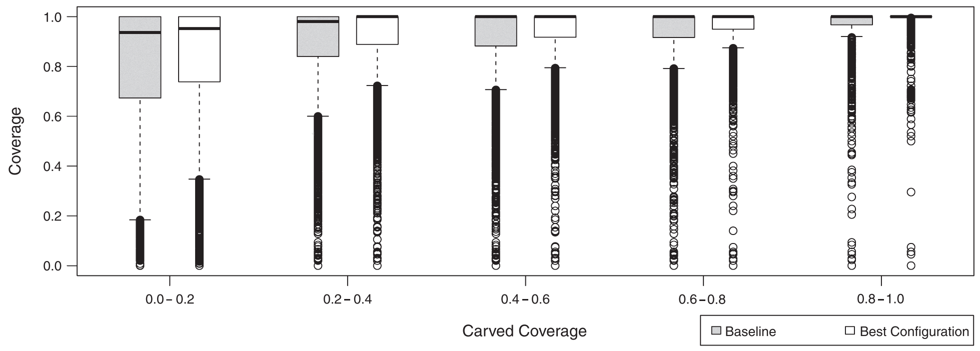


Figure 7. Boxplots comparing the best object seeding configuration ($P_{\text{ObjectPool}} = 0.9$) and the baseline configuration with no object seeding ($P_{\text{ObjectPool}} = 0.0$) for each interval of coverage achieved by the test suites available for carving.

of instantiated objects, helps in augmenting the coverage of the generated test suites, even when the carved coverage is low.

RQ4*: Seeding from existing hand-written test cases improves performance with high statistical confidence in 37% of the projects of our experiment.*

4.5.1. Example of reusing existing tests. Class `KohonenUpdateAction` from the Apache Commons Mathematics library (project `math3-3.3`) implements the method `update(Network net, double[] features);`, which updates the neural network `net` in response to the training sample `features`. Automatically generating test cases for this class is a far from trivial job. As can be observed in the following existing test case, setting up a testing scenario for this class requires instantiating a neural network, a distance measure, a learning factor function, a neighbourhood size function and an array of features.

```
@Test
public void testUpdate() {
   final FeatureInitializer init = new
       OffsetFeatureInitializer(FeatureInitializerFactory.uniform(0, 0.1));
   final FeatureInitializer[] initArray = { init };

   final int netSize = 3;
   final Network net = new NeuronString(netSize, false,
       initArray).getNetwork();
```

```
    final DistanceMeasure dist = new EuclideanDistance();
    final LearningFactorFunction learning =
        LearningFactorFunctionFactory.exponentialDecay(1, 0.1, 100);
    final NeighbourhoodSizeFunction neighbourhood =
        NeighbourhoodSizeFunctionFactory.exponentialDecay(3, 1, 100);
    final UpdateAction update = new KohonenUpdateAction(dist, learning,
        neighbourhood);

    final double[] features = new double[] { 0.3 };
    final double[] distancesBefore = new double[netSize];
    int count = 0;
    for (Neuron n : net) {
       distancesBefore[count++] = dist.compute(n.getFeatures(), features);
    }
    final Neuron bestBefore = MapUtils.findBest(features, net, dist);

    Assert.assertTrue(dist.compute(bestBefore.getFeatures(), features) >=
        0.2);

    update.update(net, features);
    ...
}
```

Using its default configuration without seeding, EvoSuite is incapable of generating any test for the `KohonenUpdateAction` class. On the contrary, when given the chance to clone and mutate existing test cases such as earlier, EvoSuite is able to achieve full coverage of the class, enhancing the coverage of the original test suite by 2.5%. The following is an example of a generated test, which is an evolved version of the previous existing one.

```
@Test
public void test1() throws Throwable {
    EuclideanDistance euclideanDistance0 = new EuclideanDistance();
    LearningFactorFunction learningFactorFunction0 =
        LearningFactorFunctionFactory.exponentialDecay(1.0, 0.1, 100L);
    NeighbourhoodSizeFunction neighbourhoodSizeFunction0 =
        NeighbourhoodSizeFunctionFactory.exponentialDecay((double) 100L, 0.1,
        100L);
    KohonenUpdateAction kohonenUpdateAction0 = new
        KohonenUpdateAction((DistanceMeasure) euclideanDistance0,
        learningFactorFunction0, neighbourhoodSizeFunction0);
    Network network0 = new Network(0L, 1);
    double[] doubleArray0 = new double[1];
    long long0 = network0.createNeuron(doubleArray0);
    long long1 = network0.createNeuron(doubleArray0);
    Neuron neuron0 = network0.getNeuron(0L);
    Neuron neuron1 = network0.getNeuron((long) 1);
    network0.addLink(neuron0, neuron1);
    kohonenUpdateAction0.update(network0, doubleArray0);
    assertEquals(0L, kohonenUpdateAction0.getNumberOfCalls());
}
```

Observe that this generated test case in essence corresponds to the first iteration of the loop in the existing test `testUpdate()`.

4.5.2. Example reusing objects carved from existing tests. Initializing objects with concrete states such that they can be used as useful parameters to trigger method calls, thus increasing coverage, is a complex task for EvoSuite. When previously created test suites are available, EvoSuite can execute them in a pre-processing step to collect for reuse all the class instances they create. A suitable example results from generating tests for class `org.apache.commons.beanutils.RowSetDynaClass` from the Apache project `beanutils-1.9.2`.

```
public class RowSetDynaClass extends JDBCDynaClass implements DynaClass,
    Serializable {
   ...
   public RowSetDynaClass(ResultSet resultSet, boolean lowerCase, int limit,
       boolean useColumnLabel) throws SQLException {

      if (resultSet == null) {
         throw new NullPointerException();
      }
      this.lowerCase = lowerCase;
      this.limit = limit;
      setUseColumnLabel(useColumnLabel);
      introspect(resultSet);
      copy(resultSet);

   }
}
```

The constructor of class `RowSetDynaClass` initializes the new object, creating a copy of the input argument `resultSet`, only if `resultSet` is not `null`; otherwise, it throws an exception. Therefore, to fully cover the constructor, at least two tests are needed: one in which `resultSet` equals `null` (trivial) and one in which is different from null. The latter is not trivial, because the type of the argument (`ResultSet`) is an interface, not an instantiable class. Fortunately, a manually written test suite for class `RowSetDynaClass` exists and can be *carved*:

```
public class DynaRowSetTestCase extends TestCase {
   ...
   public void testLimitedRows() throws Exception {
      // created one with low limit
      RowSetDynaClass limitedDynaClass = new
          RowSetDynaClass(TestResultSet.createProxy(), 3);
      List<DynaBean> rows = limitedDynaClass.getRows();
      assertNotNull("list exists", rows);
      assertEquals("limited row count", 3, rows.size());
   }
   ...
}
```

Observe that both in the test case, `testLimitedRows` contains a call to the constructor of class `RowSetDynaClass` in which a call to `TestResultSet.createProxy()` is passed as argument. Using this test case, EvoSuite is able to quickly generate the necessary test case to fully cover the constructor.

```
@Test
public void test0() throws Throwable {
   Proxy proxy0 = (Proxy)TestResultSet.createProxy();
   RowSetDynaClass rowSetDynaClass0 = new RowSetDynaClass((ResultSet)
       proxy0, true);
}
```

In fact, for this particular class, seeding existing tests allows EvoSuite to cover eight goals that otherwise are much harder to cover, that is, all goals in method `RowSetDynaClass.copy`.

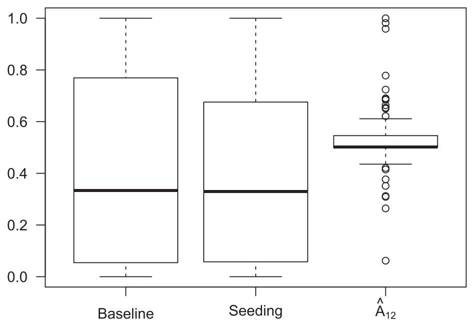
4.6. RQ5: Fault finding effectiveness

Table XIII presents the results of comparing the mutation scores achieved without seeding (baseline) and with the tuned seeding configuration (seeding) determined in the previous research questions. For brevity, only the 21 classes with statistical significance are listed. These results are also presented graphically as boxplots in Figure 8(a), which also shows the $\hat{A}_{12}$ effect size. There are only minor differences in the achieved mutation scores, with 39 classes having higher mutation scores with

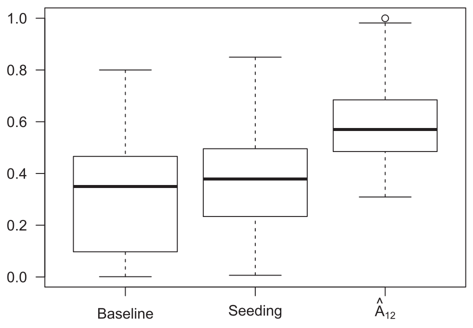
Table XIII. For each class with statistically significant results (p-value < 0.05), average mutation score when the baseline configuration, that is, no seeding, is used ($P_{\text{Constant}} = 0$, $P_{\text{Dynamic}} = 0$, type seeding off), and when the best seeding configuration is used ($P_{\text{Constant}} = 0.9$, $P_{\text{Dynamic}} = 0.6$, type seeding on).

Project	Class	Baseline	Seeding	$\hat{A}_{12}$	$\Delta_{Coverage}$
a4j	Accessories	0.49	0.55	0.66	no
a4j	Directors	0.51	0.55	0.62	no
celwars2009	CatmullRomSpline	0.33	0.37	0.65	no
celwars2009	CubicSpline	0.14	0.16	0.72	no
fps370	KeyUpBehavior	0.18	0.27	0.78	no
gfarcegestionfa	DaoFactoryException	0.97	0.29	0.06	no
io-project	ClientGroup	0.40	0.38	0.31	no
io-project	Server	0.12	0.11	0.41	no
javaviewcontrol	JVCParserTokenManager	0.05	0.07	0.69	yes
javaviewcontrol	SimpleCharStream	0.42	0.46	0.68	yes
jaw-br	Abrir	0.07	0.13	0.98	yes
jhandballmoves	MoveEvent	0.47	0.50	0.65	yes
jmca	Statement	0.10	0.27	1.00	yes
jwbf	Get	0.43	0.44	0.61	no
lilith	LoggingEventWrapperProtobufEncoder	0.74	0.85	0.69	yes
mygrid	ArrayOfString	0.36	0.38	0.60	no
quickserver	BlockingClient	0.28	0.29	0.61	yes
sfmis	Base64	0.26	0.24	0.26	no
shop	JSSubstitution	0.51	0.56	0.65	yes
sugar	FSPathResultListImpl	0.31	0.50	0.96	yes
xbus	XSLTTransformer	0.29	0.25	0.35	no

The $\hat{A}_{12}$ of these comparisons are calculated by averaging all runs on the class. $\Delta_{Coverage}$ indicates whether a significant increase in coverage was observed for a class in the previous research questions.



(a) Boxplots comparing the mutation score using the best test seeding configuration ($P_{\text{Constant}} = 0.9$, $P_{\text{Dynamic}} = 0.6$, Type Seeding on) and the baseline configuration with no seeding ($P_{\text{Constant}} = 0$, $P_{\text{Dynamic}} = 0$, Type Seeding off), considering all sampled classes.



(b) Boxplots comparing the mutation score using the best test seeding configuration ($P_{\text{Constant}} = 0.9$, $P_{\text{Dynamic}} = 0.6$, Type Seeding on) and the baseline configuration with no seeding ($P_{\text{Constant}} = 0$, $P_{\text{Dynamic}} = 0$, Type Seeding off), considering only classes with increased coverage.

Figure 8. Boxplots comparing mutation scores for seeding versus no seeding.

seeding (16 significant at $p < 0.05$), and 18 having lower mutation score with seeding (5 significant at $p < 0.05$). The average effect size is 0.54, which is similar to the effect size on coverage for this particular configuration (0.55). Thus, overall, the influence of seeding on the fault finding capability of test suites is similarly positive as the influence on coverage.

It is not surprising that an increase in coverage also leads to an increase in mutation score, as mutants that are not executed in the first place cannot be detected. Figure 8(b) shows a similar comparison, but by focusing only on those classes out of the sample on which coverage increases with seeding. Effect size grows to 0.60, and there are no classes where the mutation score significantly decreases (increase is significant for 9 out of 14 classes, and there is a non-significant decrease in six cases). On the other hand, there are 26 classes where the mutation score increases even though coverage is not increased by seeding (seven significant), while there are 12 classes (five significant) where coverage and mutation score are reduced.

RQ5*: Seeding increases fault detection ability by increasing coverage, but may in some cases reduce fault detection ability.*

4.6.1. Examples of negative effects on fault finding effectiveness. In some cases, seeding may lead to lower branch coverage, which is usually accompanied by a reduction in mutation score. To see whether there is an influence of seeding on fault detection ability that is independent of coverage, a manual investigation of the classes where the mutation score decreased revealed different scenarios. An example for the first scenario is given by the `DaoFactoryException` class from the `gfarce` project:

```java
package fr.unice.gfarce.dao;

public class DaoFactoryException extends RuntimeException{

  private static final long serialVersionUID = 1L;
  private int code;
  public DaoFactoryException(int code) {
     super();
     this.setCode(code);
   }

  public DaoFactoryException(){
    super();
  }

   public DaoFactoryException(String message, int code) {
     super(message);
     this.setCode(code);
   }

  public void setCode(int code) {
    this.code = code;
  }

  public int getCode() {
    return code;
  }

}
```

Major creates only two mutants for this class; for each of the two calls to `this.setCode (code)`, there is a mutant that removes this call. As both calls are in constructors of the class and initialize `code`, this member variable remains uninitialized in the mutants, which in Java in practice means that `code` has the value 0. Covering the code is trivial, and when not using seeding, the tests use random integer values:

```java
@Test
public void test0() throws Throwable {
   DaoFactoryException daoFactoryException0 = new DaoFactoryException(205);
   int int0 = daoFactoryException0.getCode();
   assertEquals(205, int0);
}
```

Without seeding, it is thus trivial to kill the two mutants. However, when enabling seeding, there are few values that can be gathered through either static or dynamic seeding, and the pool remains initialized with default values of -1, 0, 1 (as implemented in EvoSuite). This often leads to tests like the following example:

```
@Test
public void test0() throws Throwable {
  DaoFactoryException daoFactoryException0 = new DaoFactoryException(0);
  daoFactoryException0.setCode(0);
  assertEquals(0, daoFactoryException0.getCode());
}
```

This test will pass even if the mutant in the constructor is activated, as `code` is initialized to the same value it also has when not initialized, such that the mutant survives.

In a second example scenario, the values of strings can also have a direct impact on the fault detection ability. For example, the class `com.hf.sfm.crypt.Base64` from the `sfmis` project contains complex code that iterates over string values, thereby performing various checks and calculations:

```
  private static byte[] _$23180(String s, boolean flag)
  {
     byte bb[] = flag ? _$23169 : _$23168;
     int i = s.length();
     int j = i / 4;
     if(4 * j != i)
        throw new IllegalArgumentException("String length must be a
            multiple of four.");
     int k = 0;
     int l = j;
     if(i != 0)
     {
        if(s.charAt(i - 1) == '=')
        {
           k++;
           l--;
        }
        if(s.charAt(i - 2) == '=')
           k++;
     }
     byte bc[] = new byte[3 * j - k];
     int i1 = 0;
     int j1 = 0;
     for(int k1 = 0; k1 < l; k1++)
     {
        int l1 = _$23183(s.charAt(i1++), bb);
        int j2 = _$23183(s.charAt(i1++), bb);
        int l2 = _$23183(s.charAt(i1++), bb);
        int j3 = _$23183(s.charAt(i1++), bb);
        bc[j1++] = (byte)(l1 << 2 | j2 >> 4);
        bc[j1++] = (byte)(j2 << 4 | l2 >> 2);
        bc[j1++] = (byte)(l2 << 6 | j3);
     }

     if(k != 0)
     {
        int i2 = _$23183(s.charAt(i1++), bb);
        int k2 = _$23183(s.charAt(i1++), bb);
        bc[j1++] = (byte)(i2 << 2 | k2 >> 4);
        if(k == 1)
        {
           int i3 = _$23183(s.charAt(i1++), bb);
           bc[j1++] = (byte)(k2 << 4 | i3 >> 2);
        }
     }
     return bc;
  }
```

Without seeding, random strings such as `"k&Afu8#W+ZUzL Rj"` are used in the test cases, whereas seeding includes shorter and simpler values such as `"g"` or `"AABe"`. Consequently, even when the tests generated with seeding reach the same target branches, possibly more iterations of the loops and more checks are performed, leading to more mutants detected overall.

5. THREATS TO VALIDITY

Threats to *construct validity* are on how the performance of a testing technique is defined. Priority was given to the achieved coverage, with the secondary goal of minimizing the length. This paper did not consider the potential cases where, even if a seeding strategy leads to higher coverage, it might also lead to larger test suites. This yields two problems: (1) in practical contexts, one might not want a much larger test suite if the achieved coverage is only slightly higher, and (2) this performance measure does not take into account how difficult it will be to manually evaluate the test cases for writing assert statements (i.e. checking the correctness of the outputs). However, it is conceivable that using seeded values leads to improved readability compared with purely generated values.

Threats to *internal validity* might come from how the empirical study was carried out. To reduce the probability of having faults in our testing framework, it has been carefully tested. But it is well known that testing alone cannot prove the absence of defects. Furthermore, randomized algorithms are affected by chance. To cope with this problem, each experiment was run 30 times, and rigorous statistical procedures were used to evaluate their results.

Another possible threat to internal validity is that the interactions/relations of the different parameter configurations of EvoSuite (e.g. population size and crossover probability) with the seeding strategies and the chosen search budget were not studied. This paper claims that seeding strategies help EvoSuite (and SBST in general) to achieve higher branch coverage. However, in theory, it might be possible that there exist parameter settings for which EvoSuite gives better results and seeding might not improve upon them. To shed light on this possible issue, one would need to carry out large tuning phases and studying the possible correlations among all the different parameters (i.e. seeding strategies could be seen as further parameters to tune), but this would be very time-consuming. However, 'default' parameter settings coming from the literature already tend to lead to reasonable performances [25].

Although a large case study consisting of a total of 25,098 classes from the SF110 corpus and the Apache Commons repository was used, there is the threat to *external validity* regarding the generalization to other types of software, which is common for any empirical analysis.

Furthermore, this paper evaluated the different seeding strategies only for EvoSuite. In principle, none of these seeding strategies depend on implementation aspect of the tool. Consider for instance the Randoop tool [31]. Randoop implements seeding of constant primitive numeric and string values observed in the Java bytecode. Running Randoop with seeding enabled on the example shown in Section 4.2.1, for instance, leads to a huge increase of 70% in coverage over the default configuration with no seeding (93% vs 23%, averaged over 30 repetitions). These results are comparable with the ones observed in our experiments with EvoSuite and support the claim that seeding techniques can be beneficial to other SBST techniques and tools. Nevertheless, effectiveness levels might vary across tools. For instance, whereas a random testing tool may benefit from contant seeding when specific values lead to coverage improvement, the advantage would not be as tangible as that observed for EvoSuite when the coverage improvement only results from the use of seeded values as guidance for the search algorithm (contrast variables `x` and `y` in the first example in Section 3.1).

6. CONCLUSIONS

Search-based testing techniques are dependent on a multitude of parameters and individual choices throughout the search. *Seeding* is one such technique that may strongly influence the result of an evolutionary search. In this paper, the effects of different seeding techniques were analysed in the context of search-based testing for object-oriented languages. The results provide evidence that a good choice of seeding techniques can lead to an overall improvement of the search results.

In general, the more domain-specific information can be included in the seeding strategies, the better the results will be. However, when applying seeding in a new context, there are several aspects that require consideration, and the experiments presented in this paper allow concluding some general guidelines on how to apply seeding:

- The effectiveness of static and dynamic constant seeding directly depends on the number of constants available in the context. If the project being tested contains many constant declarations, higher probability values for P_{Constant} and P_{Dynamic} are likely to be more effective, as they increase the chances of sampling the correct value from the pool. On the other hand, when the number of constants in the project is low, P_{Constant} and P_{Dynamic} should likely be low as well, to ensure diversity and to allow the search to progress using more random values. The experiments in this paper determined sensible default values that should in many cases lead to an improvement ($P_{\text{Constant}} = 0.9$, $P_{\text{Dynamic}} = 0.6$). These values suggest that dynamic constant seeding is slightly more effective, as the probability of using dynamically seeded values is higher than using statically derived constants.
- Type seeding is likely to be more effective in projects that make use of advanced object-oriented features (e.g. downcasting), in contrast to projects with more simple data structures (e.g. limited inheritance among classes). However, if a project does not depend on such features, then using type seeding usually also has no negative effects, so it is a safe optimization to enable.
- Previously existing unit tests can aid in the generation of more tests. Even more so when the pre-existing tests were manually written, because they often involve more complex testing scenarios and reflect common object usages and interactions. Again, the optimal probabilities depend on the pool of available individuals to sample from: the fewer individuals there are, the more diversification needs to be carried out, for example by increasing N_{Mutation} or by decreasing P_{Clone}. On the other hand, larger pools of tests to sample from would require lower probabilities. The experiments in this paper suggest that default values of $N_{\text{Mutation}} = 8$ and $P_{\text{Clone}} = 0.9$ will lead to improvement in many cases.

The specific values derived experimentally should serve as useful default values, but in general, these parameters could be specifically tuned to adapt to the specific traits of each project under test.

The experiments in this paper considered several seeding strategies and applied them to the context of testing object-oriented code in terms of the EvoSuite tool with the aim of maximizing branch coverage. The same seeding strategies can be applied also to other test generation approaches, such as random testing and to other testing domains than unit testing, such as for example GUI testing. Further seeding strategies are possible, and these as well as investigations of how individual seeding strategies interact with each other will be part of our future work.

To learn more about EvoSuite, visit the website:
`http://www.evosuite.org`

ACKNOWLEDGEMENTS

This work is supported by the EPSRC project 'EXOGEN' (EP/K030353/1), a Google Focused Research Award on 'Test Amplification' and by the National Research Fund, Luxembourg (FNR/P10/03).

REFERENCES

1. Harman M, Jones BF. Search-based software engineering. *Journal of Information & Software Technology* 2001; **43**(14):833–839.
2. McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156.
3. Tonella P. Evolutionary testing of classes. *ACM International Symposium on Software Testing and Analysis (ISSTA)*, Boston, Massachusetts, USA, 2004; 119–128.
4. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. *ACM Symposium on the Foundations of Software Engineering (FSE)*, Szeged, Hungary, 2011; 416–419.
5. Fraser G, Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering* 2013; **39**(2):276–291.
6. Fraser G, Arcuri A. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM TOSEM* 2014; **24**(2):8:1–8:42.

7. The Apache Commons, 2014. Available from: http://commons.apache.org/ [last accessed 16 December 2015].
8. Fraser G, Arcuri A. The seed is strong: seeding strategies in search-based software testing. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Montreal, Canada, 2012; 121–130.
9. Langdon WB, Nordin P. Seeding genetic programming populations. *Proceedings of the European Conference on Genetic Programming (EuroGP)*, Edinburgh, Scotland, 2000; 304–315.
10. Westerberg CH, Levine J. Investigation of different seeding strategies in a genetic planner. *Proceedings of EvoWorkshops*, Como, Italy, 2001; 505–514.
11. White D, Arcuri A, Clark J. Evolutionary improvement of programs. *IEEE Transactions on Evolutionary Computation (TEC)* 2011; **15**(4):515–538.
12. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 2001; **43**(14):841–854.
13. Tlili M, Wappler S, Sthamer H. Improving evolutionary real-time testing. *Genetic and Evolutionary Computation Conference (GECCO)*, Seattle, Washington, USA, 2006; 1917–1924.
14. Yoo S, Harman M. Test data regeneration: generating new test data from existing test data. *Software Testing, Verification and Reliability (STVR)* 2012; **22**(3):171–201.
15. McMinn P, Stevenson M, Harman M. Reducing qualitative human oracle costs associated with automatically generated test data. *Proceedings of the First International Workshop on Software Test Output Validation*, STOV '10, ACM: New York, NY, USA, 2010; 1–4.
16. Fraser G, Zeller A. Exploiting common object usage in test case generation. *ICST 2011: Proceedings of the International Conference on Software Testing, Verification, and Validation*, IEEE Computer Society: Los Alamitos, CA, USA, 2011; 80–89.
17. Miraz M, Lanzi PL, Baresi L. Improving evolutionary testing by means of efficiency enhancement techniques. *IEEE Congress on Evolutionary Computation (CEC)*, Barcelona, Spain, 2010; 1–8.
18. Alshraideh M, Bottaci L. Search-based software test data generation for string data using program-specific search operators. *Software Testing, Verification, and Reliability* 2006; **16**(3):175–203.
19. McMinn P, Shahbaz M, Stevenson M. Search-based test input generation for string data types using the results of web queries. *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Montreal, Canada, 2012; 141–150.
20. Alshahwan N, Harman M. Automated web application testing using search based software engineering. *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Lawrence, Kansas, USA, 2011; 3–12.
21. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* 1990; **16**(8):870–879.
22. Li Y, Fraser G. Bytecode testability transformation. In *Search Based Software Engineering*, vol. 6956, Lecture Notes in Computer Science. Springer, 2011; 237–251.
23. Møller A. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. Available from: http://www.brics.dk/automaton/ [last accessed 16 December 2015].
24. Fraser G, Arcuri A. Automated test generation for java generics. *Software Quality Days (SWQD)*, 2013.
25. Arcuri A, Fraser G. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)* 2013; **18**(3):594–623.
26. JavaNCSS - a source measurement suite for Java 2014. Available from: http://www.kclee.de/clemens/java/javancss, version 32.53 [last accessed 16 December 2015].
27. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. *ACM/IEEE International Conference on Software Engineering (ICSE),* ACM: Hyderabad, India, 2014; 435–445.
28. Just R. The Major mutation framework: efficient and scalable mutation analysis for java. *ACM International Symposium on Software Testing and Analysis (ISSTA),* ACM: San Jose, California, USA, 2014; 433–436.
29. Arcuri A, Briand L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability (STVR)* 2014; **24**(3):219–250.
30. Arcuri A, Fraser G, Galeotti JP. Automated unit test generation for classes with environment dependencies. *IEEE/ACM International Conference on Automated Software Engineering (ASE),* ACM: Vasteras, Sweden, 2014; 79–90.
31. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *ACM/IEEE International Conference on Software Engineering (ICSE)*, Minneapolis, Minnesota, USA, 2007; 75–84.