Noname manuscript No.
(will be inserted by the editor)

1600 Faults in 100 Projects: Automatically Finding Faults While Achieving High Coverage with EvoSuite

Gordon Fraser and Andrea Arcuri

Received: date / Accepted: date

Abstract Automated unit test generation techniques traditionally follow one of two goals: Either they try to find violations of automated oracles (e.g., assertions, contracts, undeclared exceptions), or they aim to produce representative test suites (e.g., satisfying branch coverage) such that a developer can manually add test oracles. Search-based testing (SBST) has delivered promising results when it comes to achieving coverage, yet the use in conjunction with automated oracles has hardly been explored, and is generally hampered as SBST does not scale well when there are too many testing targets. In this paper we present a search-based approach to handle both objectives at the same time, implemented in the EvoSuite tool. An empirical study applying EvoSuite on 100 randomly selected open source software projects (the SF100 corpus) reveals that SBST has the unique advantage of being well suited to perform *both* traditional goals at the same time – efficiently triggering faults, while producing representative test sets for any chosen coverage criterion. In our study, EvoSuite detected twice as many failures in terms of undeclared exceptions as a traditional random testing approach, witnessing thousands of real faults in the 100 open source projects. Two out of every five classes with undeclared exceptions have actual faults, but these are buried within many failures that are caused by implicit preconditions. This "noise" can be interpreted as either a call for further research in improving automated oracles — or to make tools like EvoSuite an integral part of software development to enforce clean program interfaces.

Keyword: Search-based testing; automated test generation; test oracles

G. Fraser
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
E-mail: Gordon.Fraser@sheffield.ac.uk

A. Arcuri
Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
E-mail: arcuri@simula.no

1 Introduction

Software testing is an essential but complex task in software development. To support the developers, automated techniques have been devised that take a program as input and generate sets of test cases. There are two main usage scenarios of such *white-box* test generation techniques: If the developer has provided a partial specification – for example in terms of assertions in the code, generic descriptions of misbehavior such as program crashes, or a previous program version – then the aim of test generation is to find violations of these specifications (e.g., [11, 32, 35]). In this case the specifications serve as *automated oracles* during testing. The alternative approach, which is useful when specifications are not sufficiently available, is to produce a representative set of test cases that captures the essential behavior of the program. The developer is then expected to manually add test oracles (e.g., [20, 43]).

Search-based software testing (SBST) has resulted in mature and efficient test generation tools (e.g., [15, 26, 30]). Most SBST tools take a code coverage criterion as objective function, and then produce small test sets satisfying the coverage criterion, with the intention that the developer adds test oracles. SBST is often compared to alternative test generation approaches such as random testing (e.g., [11, 35]) or dynamic symbolic execution (DSE, e.g., [21, 43]), and the comparison is commonly based on the achieved code coverage (e.g., [27, 28]). Even though these other techniques can also achieve high coverage, they were designed with the intent to exercise automated oracles. Yet, the question of how suitable SBST is to exercise automated oracles remains unanswered.

On one hand SBST executes large numbers of tests and has guidance towards difficult to reach program states, so one might expect it to be well suited to exercise automated oracles. On the other hand, traditional SBST simply does not scale up to the task of optimizing an individual test input for each possible violation of an automated oracle. However, recent advances in SBST [20] have led to the insight that optimizing entire test suites towards satisfying a coverage criterion can be advantageous compared to optimizing individual tests. In this paper, we make use of this insight and optimize test suites towards *both* goals, i.e., satisfying a coverage criterion and exercising automated oracles. Experiments with our EvoSuite prototype on a representative sample of 100 open source software projects reveal that this approach is very effective at achieving both goals. Our study shows that two out of every five classes with failures determined by the most generic automated oracle — program crashes, or undeclared exceptions in the case of unit testing — actually have real faults. This implies that there is also a significant amount of false warnings, which would be very easily avoided if tools like EvoSuite would be used during software development from the beginning, forcing developers to properly declare method signatures.

In detail, the contributions of this paper are as follows:

SBST for Automated Oracles: We present an extended fitness function for a Genetic Algorithm in a whole test suite generation setting that optimizes towards both, code coverage and finding violations of automated oracles, in absence of specifications.

```
class Test {
  public int add(ComplexObject c) {
    if(c.getValue() == 1024) {
      throw new Exception("Bug 1");
    }

    return c.wrongFunctionCall();
  }
}
```

Fig. 1 Example code: Random testing is unlikely to cover the branch; DSE might struggle to generate a *ComplexObject* instance, and when these techniques do not find violations of automated oracles, what does it mean with respect to functional correctness? SBST, on the other hand, traditionally ignores automated oracles and *only* aims to produce representative test sets to which the user can manually add oracles.

Testability Transformation: We present a code transformation that adds program branches for error conditions, thus providing additional guidance for the test generation towards violations of automated oracles.

Empirical Study: We present a large empirical study on a representative sample of open source software, demonstrating the usefulness of SBST in revealing violations of automated oracles while optimizing towards code coverage.

Failure Analysis: We sample and manually analyze the detected violations, which allows us to quantify in a statistically representative way how many of the classes with failures in the case study have real faults, showing that our experiments have hit thousands of real faults.

This paper is organized as follows: Section 2 gives an overview of the state of the art in structural unit testing and oracles. Section 3 describes how SBST can be adapted to address both problems, generating representative test sets for manual oracles, and exercising automated oracles, at the same time. Finally, Section 4 evaluates the described technique on a large set of classes that together are a representative sample of open source software.

2 Background

2.1 Unit Test Generation

At a high level, the state of the art in automated structural test generation can be divided into three main groups: variants of *random testing* (e.g., Randoop [35]), approaches based, on *constraint solving* (e.g., DART [21] and CUTE [39]) and *search-based software testing* (e.g., [29]).

2.1.1 Random Testing

Random testing [4, 13] is perhaps the simplest form of test generation; it simply consists of randomly calling functions with random inputs. The power of this approach lies in its simplicity: Because there is virtually no computation effort in choosing suitable test inputs, large numbers of test cases can be produced in a short time. A large

number of tests means that the usage scenario assumes the availability of automated oracles; random testing has been shown to find violations of such automated oracles (e.g. [32, 35]). However, random testing may struggle to cover parts of programs that are difficult to reach, e.g., because they require specific values.

Consider the example code snippet in Figure 1: Random testing would execute the code with random input values; if a random test finds a violation of an automated oracle, then the test case causing this violation can be reported to the user. For example, random testing would easily find a *NullPointerException* by passing *null* as parameter to the method *add*. However, whether the *add* function is functionally correct cannot be determined with the automated oracles in the example; asking a user to determine correctness of hundreds and thousands of random tests is also infeasible. Furthermore, even the simple branch *if(c.getValue() == 1024)* would only be covered with a very low probability during simple random testing.

2.1.2 Dynamic Symbolic Execution

Approaches based on constraint solving use symbolic execution to assign path conditions to program paths, such that solving a path condition results in a test case that executes the corresponding path. Dynamic symbolic execution (DSE [21]) executes a program initially with a random value, and the path condition of the path taken by this input is derived during concrete execution. By negating an individual condition one can derive a new path condition, such that a solution to this will follow a different program path than the original path condition. DSE systematically explores all program paths by negating individual conditions. Thus, its main objective is to explore all program paths in order to find violations of automated oracles.

DSE would have no problems in generating integer inputs that satisfy the conditions in the example in Figure 1. Modern DSE tools would also explicitly consider different paths where line 3 results in a *NullPointerException* and where it does not. However, DSE might struggle to produce a *ComplexObject* object in the first place [44], thus not reaching the *throw* statement. Furthermore, the number of paths can quickly become very large, and so the problem of which tests to show the user to determine functional correctness remains.

DSE tools such as Pex [40] can produce branch coverage test sets as a byproduct of the exploration, yet other coverage criteria are rarely seen in the literature (e.g., modified condition/decision coverage [36]). Consequently, most DSE tools focus on the task of finding violations of automated oracles, rather than producing efficient test sets.

2.1.3 Search-based Testing

Search-based approaches [29] require that the objective of the test generation is encoded in a fitness function that guides the search. For example, a popular type of search algorithm is a Genetic Algorithm (GA), where a population of candidate solutions is evolved using operators inspired by natural evolution: Individuals are selected for reproduction based on their fitness (better individuals have higher probability for

reproduction), and with a certain probability operators such as crossover and mutation are applied, resulting in new offspring individuals. The algorithm breeds new generations until a solution is found or a stopping condition (e.g., timeout) is met.

Traditionally, SBST is applied to generate test sets satisfying coverage criteria, and each coverage goal is represented as a distinct fitness function; test generation is attempted for one such goal at a time. Yet, SBST traditionally does not make any use of automated oracles. Indeed, traditional SBST is unlikely to scale up to the task of deriving a test input for each possible violation of an automated oracle *and* for each coverage goal. Thus, in the example in Figure 1, a branch coverage test set would contain a test case that triggers the exception in line 4 – yet whether a *NullPointerException* or other exceptions are included is not prescribed by traditional coverage criteria like statement or branch coverage.

Consequently, even though the different approaches of random testing, DSE, and SBST are often compared with each other, they do not target the same usage scenarios of automated vs. manual test oracles.

Recently, whole test suite generation [20] was introduced as an alternative approach in SBST, where the optimization objective is to produce a test suite that covers *all* coverage goals of a given criterion; individuals of the search are test suites. The advantage of this is that the feasibility or difficulty of individual coverage goals does not affect the overall performance. In both cases of SBST, the objective is usually to produce small sets of tests such that a developer can manually add test oracles.

2.2 Automated Oracles without Specifications

For a given test input, a *test oracle* describes the correct behaviour as well as the procedure to compare it to the implemented behaviour. A common solution proposed by researchers is to use various types of specifications [5] as automated oracles. Yet, when these are not available, one would face the so called *oracle problem* in which the oracles need to be manually added. However, several techniques have been proposed to find faults in programs even without explicit user input in terms of specifications or oracles. In the following, we describe these techniques in the context of the Java programming language. Our discussions will be based on the technical details of this language, although many concepts apply to other object-oriented languages as well (e.g., C#).

Some basic assumptions on programs always need to hold, and can always be used as automated oracles. For example, in general programs should not crash. In the context of unit testing of classes this means that a class should not throw any undeclared exceptions. Random testing has been used to exercise this type of automated oracle, for example by the JCrasher [11] and Randoop [35] tools.

Like many object-oriented languages, Java uses exceptions to handle errors and other exceptional conditions. In Java, checked exceptions represent invalid conditions in areas outside the immediate control of the program (e.g., operating system errors when reading/writing files, network problems), and a method either needs to handle such exceptions or explicitly declare that such exceptions are passed on (rethrown). Unchecked exceptions, on the other hand, are meant to represent defects

in the program, and a method is not expected to handle them; thus, unchecked exceptions are not declared in the API, unless throwing such exceptions is part of the expected normal behavior (e.g., checking preconditions). There is dispute on whether the distinction between checked and unchecked exceptions is useful, but in principle any observed exception that is not declared represents a failure.

A drawback of using unchecked exceptions as oracles is that this assumes that the developer explicitly declares all exceptions that a class can throw in normal usage (e.g., by using the keyword *throws* in the method signatures). However, as we will show in Section 4, developers of open source software fail to provide accurate method signatures. For example, even though a programmer might not defensively add null checks on all input parameters, he or she might accept the fact that wrong usage can lead to *NullPointerException*s. Without declaration, it is difficult to judge automatically whether a concrete *NullPointerException* represents an error that is of interest, or just a violation of an implicit precondition. This is a well known problem in software testing, and a common solution is to use heuristics to identify "unexpected" undeclared exceptions (e.g. [11]). Alternatively, we are exploring the possibility to drive testing through user interfaces to filter out exceptions that violate the implicit assumptions of the developer [24].

In addition to undeclared exceptions, Randoop [35] can check *contracts* on the code, which may be supplied by the user. By default, Randoop implements several default contracts that are present in the Java language, such as reflexivity of *Object.equals*. All objects in Java extends the type *Object*, which contains a predefined set of methods (e.g., *equals*) with contracts. In the Java language specification, these contracts are expressed in natural language (i.e., as JavaDoc comments), and classes that override these methods still need to satisfy their contracts. However, these methods would represent only a small fraction of the code of the system under test (SUT), and recent IDEs like Eclipse can generate some of these methods automatically (e.g., *equals* and *hashCode*).

Some approaches try to infer models of normal behaviour from known executions, and then assume that deviations from this behaviour are more likely to be faults [34]. Previous program versions can also serve as automated oracles in regression testing (e.g., [33]): A fault is found if the behavior of a previous version of the program differs compared to a new version under the same inputs, assuming that no intentional behavioral change was applied (e.g., a refactoring).

2.3 SBST and Automated Oracles

There has been previous work on applying SBST to a scenario of automated oracles: By statically selecting possible paths that can lead to memory access violations (*NullPointerException*) [38] it is possible to search for inputs that follow these paths. It is also possible to represent arithmetic errors such as division by zero faults as branches in the source code, such that traditional SBST metrics such as the branch distance can optimize towards these errors [8]. In this paper, we also add new branches to the code that represent error conditions; however, in real-world software the number of such

conditions may be so large that it becomes an issue of scalability for the traditional approach of targeting one branch at a time.

Korel and Al-Yami [25] described an approach that uses search techniques to try to find violations to assertions in code. Each assertion represents a branch in the bytecode, and so our approach will automatically try to cover these branches (i.e., try to make the assertions fail) as well. Tracey et al. [41] presented a search-based approach to optimize test cases towards raising exceptions in order to exercise exception handling code. The guidance is given through guarding conditions of the statements raising exceptions. By applying whole test suite optimization, our approach will include such branches in principle. Our implementation currently only considers branches directly in the SUT, which means that guarding conditions of exceptions outside the SUT do not contribute to the search. Including this guidance can be expected to increase the number of exceptions raised in the resulting test suites.

Del Grosso et al. [12] used SBST to exercise statements statically detected as vulnerable. By rewarding multiple execution of such vulnerable statements, the search is led towards filling buffers, likely finding buffer overflows. The testability transformation for buffer overflows presented in this paper has a similar intention, yet it requires no static analysis to identify vulnerable statements.

McMinn [31] presented two testability transformations aimed at revealing faults through automated oracles. Numerical imprecision and roundoff errors are detected by comparing floating point numbers with higher precision objects that mirror the same operations. The second transformation consists of adding and removing synchronization statements in order to find race conditions. The fitness function guiding the search is based on maximising the differences in the output between transformed and un-transformed version. In contrast, the search used in this paper is only applied to the SUT, and the search is guided by the program structure (e.g., by using branch coverage) towards reaching all parts of the program. However, in principle the oracles provided by McMinn's transformations could also be used in conjunction with our approach.

Transformations have not only been applied in the context of search: The idea to check error conditions was first mentioned in 1975 by Lori Clarke [9] in the context of symbolic execution. Active Property Checking [22] describes the use of explicit error branches in the path constraints during DSE. By having explicit constraints on the error conditions, the DSE exploration will try to negate also the error conditions, such that if there exists an input that leads to the error it will be found. This is also implemented in other state-of-the-art DSE tools; for example, Pex [40] automatically adds constraints that check references against null, divisions against zero, etc. Barr et al. [6] instrument programs with additional branches to find floating point exceptions with symbolic execution. These additional constraints are similar to the error conditions we introduce in the bytecode (see Section 3.2).

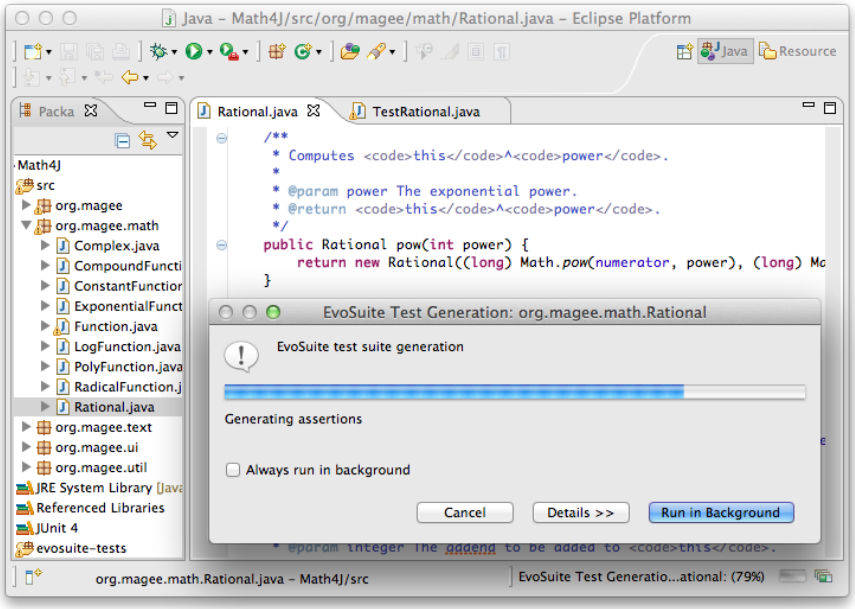

Fig. 2 The EVOSUITE Eclipse plugin, where test cases are generated with a click of the mouse.

2.4 The EVOSUITE Tool

As context of our experiment we chose the EVOSUITE [15] tool, which automatically generates test suites for Java classes, targeting branch coverage and several other coverage criteria (e.g., weak/strong mutation testing).

The only input EVOSUITE requires is the bytecode of the SUT and its dependencies. EVOSUITE automatically determines class dependencies, and generates a JUnit test suite for the SUT. For practitioners, a typical usage of EVOSUITE is through its Eclipse plugin, where they can simply right-click on a class to automatically generate unit tests (an example can be seen in Figure 2). For experiments, EVOSUITE can be used through a simple command line interface that just requires a correct classpath and the name of the target class (SUT). This level of automation was essential for the large experiment described in this paper, as the effort to manually adapt tools to case study subjects is often prohibitive. For example, popular tools like Java PathFinder [42] require manually written drivers for the SUT when used to generate test cases. Manually writing drivers would not be a viable option for empirical studies involving thousands of classes (as it is done in this paper).

EVOSUITE is a mature research prototype and has been successfully applied to a range of different systems [18]. It is well suited for the experiments in this paper as it has been studied in detail with regard to its parameters (e.g., [3, 16, 17, 20, 23]). For more details on the tool and its abilities we refer to [15], and for more implementation details we refer to [19]. The results of a recent competition on unit test generation

tools [7] won by EvoSuite[1] also demonstrates that it is indeed one of the most advanced unit test generation tools and is thus well suited for our experiments.

3 Search-based Testing for Coverage and Automated Oracles

Traditionally, SBST derives coverage-based test sets. There is an overlooked opportunity here: During the exploration of the search-space, the search can come across violations of automated oracles. In this section we consider how to apply SBST to exercise automated oracles while generating test suites for coverage.

3.1 Fitness Function for Automated Oracles

The fitness function is at the core of any search-based technique. In SBST, the fitness function usually encodes an individual coverage goal, a coverage criterion, or an individual error condition. An alternative if offered by whole test suite generation [20], where test suites are optimized with respect to entire coverage criteria. Here, we present an extended fitness function for whole test suite generation that can be used in conjunction with a regular fitness function that targets code coverage.

As a baseline fitness function to drive exploration, we will use branch coverage in this paper, although any other coverage criterion could be used instead. The branch coverage fitness function is based on the branch distance, which estimates how close a branch was to evaluating to true or to false. For example, if we have the branch $x == 17$, and a concrete test case where x has the value 10, then the branch distance to make this branch true would be $17 - 10 = 7$, while the branch distance to making this branch false is 0 (i.e., it already is false).

Let $d_{min}(b,T)$ be the minimal branch distance of branch b for all executions of b on test suite T, then we define the distance $d(b,T)$ as follows:

$$d(b,T) = \begin{cases} 0 & \text{if the branch was covered,} \\ \nu(d_{min}(b,T)) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases}$$

Each branching statement needs to be executed twice to avoid the search oscillating between the two values of the branch [20] (i.e., otherwise optimizing a predicate towards evaluating to true might remove the case where the false branch is covered) . ν is a normalization function [1] in the range $[0,1]$. This leads to the following (minimizing) fitness function for branch coverage:

$$\text{covf}(T) = |M| - |M_T| + \sum_{b \in B} d(b,T) \; .$$

[1] The other participating tools were *t2* and *DSC*, as well as *Randoop* as a baseline. Tools were evaluated based on achieved code coverage, mutation score and execution time (linearly combined in a single score).

Here, B is the set of all branches in the class under test, M is the set of all methods in such class, while M_T is the subset of methods executed by T. The difference $|M| - |M_T|$ is used to reward the execution of methods in the SUT that have no conditional expressions.

In addition to this baseline fitness function, we also keep track of the unique number of exceptions in the SUT that are not caught in the public methods called within a test suite T. We consider as "unique" the pair of exception type (i.e., its class, as for example *java.lang.NullPointerException*) and the SUT public method in which it was observed (note that the exception could have been thrown in internal private methods but propagate up to public methods). We only keep track of exceptions that are not declared in the signature of the SUT methods. For example, if a method's signature allows throwing of null pointer exceptions, then such exceptions will *not* be counted in the fitness function.

We also make the distinction between *implicit* and *explicit* exceptions. We define an explicit exception any exception that is directly thrown in the code with the keyword *throw*; all others are implicit exceptions. For example, consider the following snippet of code:

```
public int foo(int x) {
  if(x > 100)
    throw new InvalidArgumentException();

  return 1/x;
}
```

Here, the *InvalidArgumentException* is thrown explicitly if the input *x* is greater than 100, but if *x* equals 0 then the division *1/x* will lead to an implicit *DivisionByZeroException*. Making such a distinction is important for filtering heuristics, as uncaught explicit exceptions are likely just violated undefined preconditions. A typical example is the unchecked exception *IllegalArgumentException*: A software developer could directly "throw" it in a method if the input is not valid (i.e., violated precondition), but as long as it is not included in the signature of the method it would not be considered normal behavior. By making the distinction between implicit and explicit exceptions, if we find a test case that throws a (possibly uninteresting) explicit exception of type A, then successive test cases that find implicit exceptions of same type A in that method (possibly related to real faults) would be kept in the test suite and not discarded.

Given n_e the number of unique thrown explicit exceptions, and n_i the number of implicit ones, the fitness function to minimize is:

$$\text{fitness}(T) = \frac{1}{1 + n_e + n_i} + \text{covf}(T) \ .$$

Note that $n_e + n_i$ does not necessarily represent the number of *faults* in the SUT. As long as the execution of the test cases in T share even a single line of SUT code (e.g., calling the same constructor), it could be that all the $n_e + n_i$ failures are the manifestation of a single fault in the SUT. This can only be determined through manual verification. Thus, although the final goal for a software developer is to fix faults in the SUT, it is still important to have a fitness function that rewards the number of

failures in the SUT, as long as the failures are different (e.g., different types and/or thrown in different methods). The reason is that they can provide useful extra information for debugging. For example, it is not uncommon that different types of exception thrown in the same SUT might actually be due to different faults (e.g., a null pointer in a method and an array out of bound in another method).

During evolutionary selection, if two individuals (i.e., test suites) have the same fitness, then the smaller one gets higher chances to reproduce. This is done to avoid too large test suites and bloat problems [16].

3.2 Testability Transformation

The fitness function presented in the previous section rewards the number of violations of automated oracles that are found. However, it does not provide any *guidance* towards maximizing them, except by achieving that all branches in the program are executed (if the underlying coverage criterion is branch coverage). As seen above, a common type of guidance offered in SBST is through branch distance estimation.

To apply the existing techniques to different target criteria, a common approach is to transform these other criteria to branch coverage problems. This has for example been done for division by zero errors [8] or null pointer exceptions [38]. In the following, we describe several transformations implemented in the EVOSUITE tool; some transformations are similar to the additional constraints that are added to DSE when applying Active Property Checking [22] (division by zero, array bounds, null pointer dereference). The transformations are implemented at the bytecode level, but for illustration purposes we show the semantically equivalent source code versions. Besides those transformation already present in the literature, several novel, additional test objectives are explicitly included in the program code in terms of new branch instructions. Note that these transformations can be therefore used by any testing tool, not just EVOSUITE.

3.2.1 Array access transformation

Managed languages such as Java and C# do not suffer from buffer overruns, but accessing an array outside of its dimensions will lead to an exception. These exceptions can be explicitly formulated as follows:

```
void test(int x) {
  if(x < 0)
  throw new NegativeArraySizeException();
  if(x >= foo.length)
  throw new ArrayIndexOutOfBoundsException();
  foo[x] = 0;
}
```

These additional branches explicitly guide the search towards accessing the lower and upper boundary of an array, ultimately rewarding a violation. Note that the same transformation on an unmanaged language (e.g., C) would be suitable to detect buffer overruns, e.g., in strings.

3.2.2 Division by zero transformation

Numerical divisions are not defined for 0-divisors and lead to exceptions or undefined behavior. Checking the divisor against zero offers appropriate guidance for the search, and can be represented as an explicit branch as follows:

```
void test(int x) {
  if(x == 0)
  throw new ArithmeticException();
  int y = z / x;
}
```

The divisor can of course be an arbitrarily complex expression. However, at the bytecode level at the point of the division instruction it is resolved to a number, and the instrumentation simply checks this concrete value. In other words, using testability transformations at bytecode level relieves us from handling possible side-effects in the evaluation of the divisor.

3.2.3 Numerical overflow transformation

Numerical overflows and underflows are easy to miss and can cause unexpected behavior. The details of how an over/underflow is reached depends on the arithmetic operator in use. For simplicity, our instrumentation therefore calls an external helper function that determines whether there is an over/underflow:

```
void test(int x) {
  if(checkOverflow(z, x, ADD) < 0)
  // report overflow
  if(checkUnderflow(z, x, ADD) < 0)
  // report underflow
  int y = z + x;
}
```

The distance calculation rules for overflows are given in Table 1; the rules for underflows are defined analogously but omitted here for space reasons. This set of rules is necessary to avoid overflow errors during the calculation of the overflow distances[2]. The *checkOverflow* function added in the instrumented branches first determines if there was an overflow or an underflow (e.g., an overflow can only be the case under the conditions in Table 1 that have an entry for distance to non-overflow, and in that case are determined by checking if the result of the operation is negative). If there is an over/underflow the value returned by *checkOverflow* equals the negated absolute value of the amount of the overrun, such that the search has guidance towards finding a case where there is no overflow. If there is no overflow, then there are different levels of guidance. The function s scales values in the range $[0,P]$ as follows: $s(x) = P \times \frac{x}{x+1}$. The value P is a constant, for example *Integer.MAX_VALUE*

[2] An alternative would be to resort to data structures that can cope with larger number ranges (e.g., *BigDecimal* in Java), but this would lead to a significant performance drop.

Table 1 Overflow checking rules

Operation	Condition	Distance to Non-Overflow	Distance to Overflow
$a + b$	$a \geq 0 \wedge b \geq 0$	$a + b$	$P - s(a + b)$
$a + b$	$a < 0 \wedge b < 0$	-	$P + s(\|a\| + \|b\|)$
$a + b$	$a \geq 0 \wedge b < 0$	-	$P + s(\|b\|)$
$a + b$	$a < 0 \wedge b \geq 0$	-	$P + s(\|a\|)$
$a - b$	$a \geq 0 \wedge b \leq 0$	$a - b$	$P - s(a - b)$
$a - b$	$a < 0 \wedge b > 0$	-	$P + s(\|a\| + \|b\|)$
$a - b$	$a \geq 0 \wedge b > 0$	-	$P + s(b)$
$a - b$	$a < 0 \wedge b \leq 0$	-	$P + s(\|a\|)$
$a \times b$	$a > 0 \wedge b > 0$	$a \times b$	$P - s(a \times b)$
$a \times b$	$a < 0 \wedge b < 0$	$a \times b$	$P - s(a \times b)$
$a \times b$	$a > 0 \wedge b < 0$	-	$P + s(\|b\|)$
$a \times b$	$a < 0 \wedge b > 0$	-	$P + s(\|a\|)$
$a \times b$	$a = 0 \vee b = 0$	-	P
$a \ / \ b$	$a = \text{MIN} \wedge b = -1$	-1	$s(\|-\text{MIN} - a\|)$ $+s(\|-1 - b\|)$

$/2$, such that the maximum value returned by $s(x)$ is P, and thus the maximum value returned by the overflow checking rules in Table 1 is $P + P =$ *Integer.MAX_VALUE*.

For example, if the operation is an addition, then both operands need to be larger than 0 for an overflow to happen. Thus, if one of the operands is smaller than 0, then the distance guides towards increasing this operand until it is greater than 0. If both operands are negative, the search guides both operands towards becoming greater than 0. Finally, if both operands are greater than zero but there is no overflow, the distance returned is smaller than in the previous cases (achieved through the constant value P), and the value is smaller the closer the sum is to an overflow. In Java, an over- or underflow is silently ignored, so one can choose a custom way to report them (e.g., a custom exception).

3.2.4 Reference access transformation

A *NullPointerException* happens whenever a method invocation or a field access on a null reference is attempted. The transformation therefore inserts null-checks before every single method invocation or field access as follows:

```
void test(Foo x) {
  if(x == null)
  throw new NullPointerException();
  Foo.bar();
}
```

This transformation is important to distinguish between different ways in which a null pointer exception is thrown in the SUT, as every single method invocation becomes a target that EvoSuite will try to cover.

Table 2 Contracts used in EVOSUITE

AS	Assertion [35]	No *AssertionError* is raised
Es	Equals self [35]	*a.equals(a)* = true
EH	Equals hash code [35]	*a.equals(b)* $\rightarrow$ *a.hashCode()* = *b.hashCode()*
EN	Equals null [35]	*a.equals(null)* = false
ES	Equals symmetric [35]	*a.equals(b)* = *b.equals(a)*
HN	HashCode returns normally [35]	*a.hashCode()* does not throw an exception
JCE	Unexpected exception [11]	Method throws no undeclared exception that is unexpected based on JCrasher's heuristic
NPE	Null pointer [35]	No *NullPointerException* is raised on a method that received only non-null arguments
TSN	ToString returns normally [35]	*a.toString()* does not throw an exception
UE	Undeclared exception	Method throws no undeclared exception

3.2.5 Class cast transformation

A class cast leads to an exception at runtime if the concrete type of the object to be casted does not match the expected cast type. Consequently, the transformation consists of an explicit branch before each cast as follows:

```
void test(Foo x) {
  if(!(x instanceOf Bar))
  throw new ClassCastException();
  Bar y = (Bar)x;
}
```

3.2.6 Overhead

All the transformations describe in this section add an overhead in the execution of the test cases. Given the same amount of testing budget, then such transformations could lead to fewer fitness evaluations, and so maybe even decrease the efficacy of the testing tool. In such a context, when different algorithms and variants are compared, is hence important to use as stopping condition the same amount of time (e.g., two minutes per search), and not a fixed number of fitness evaluations.

3.3 Generic API Contracts

The fitness function and testability transformation discussed so far address automated oracles in terms of undeclared exceptions. However, even in absence of a specification it is sometimes possible to have programming language specific properties that have to hold on all programs. For example, the Java language specification describes some general contracts that have to hold on all Java classes. These API contracts were used, for example, in the Randoop [35] test generator. In the original Randoop experiments several failures were identified using these contracts.

Table 2 lists the standard API contracts that can be checked on Java classes, and that area implemented in EVOSUITE. In a random testing tool like Randoop [35], these contracts are checked after each statement for every possible object. In contrast, EVOSUITE is a unit testing tool, and so the contracts only need to be checked on instances of the SUT and not all objects in a test. Contracts are checked after every executed statement.

4 Evaluation

To determine how well SBST performs with respect to automated oracles, we conducted a set of experiments on a representative sample of open source software. In these experiments we used two types of generic, program independent automated oracles: Undeclared exceptions, and generic object contracts. We aim to answer the following research questions:

RQ1: What type and how many undeclared exceptions can be found with SBST in open source software?

RQ2: Does search-based testing detect more exceptions than random search?

RQ3: How does the testability transformation affect the search in terms of coverage and exceptions found?

RQ4: What type and how many contract violations can be found with SBST in open source software?

RQ5: What is the overhead of checking contracts during the search?

RQ6: How many real faults does EVOSUITE find in SF100 through violations of object contracts and assertions?

RQ7: How many real faults does EVOSUITE find in SF100 through undeclared exceptions?

4.1 Experimental Setup

To answer our research questions, we carried out an extensive empirical analysis using the EVOSUITE tool on the SF100 corpus as case study [18]. SF100 is a collection of 100 Java projects *randomly* selected from SourceForge, which is one of the largest repositories of open source projects on the web. In total, SF100 consists of 8,844 classes[3] with more than 290 thousand bytecode level branches. The use of a *large* case study that was selected in an *unbiased* manner is a pre-requisite for an empirical study aiming to achieve sound results of practical importance; in contrast, a small hand-picked case study would only show feasibility of the proposed techniques.

The testability transformation described in Section 3.2 adds an overhead in the execution of test cases. On average, it adds 141 new branches per class in SF100. Given a fixed testing budget, this could lead to fewer fitness evaluations, and so maybe even decrease the overall performance. We therefore used a fixed time as stopping condition rather than a fixed number of fitness evaluations.

[3] Note that we used the 1.01 version of SF100. The original version in [18] had 8,784 classes, but more classes became available once we fixed some classpath issues (e.g., missing jars) in some of the projects.

EVOSUITE was configured to run for two minutes on each class, using a population size of 50, maximum test length of 20 statements, and the default parameters determined during a study on parameter tuning [3]. Furthermore, EVOSUITE was run using a custom security manager which was configured to deny most permission requests. This is necessary to avoid programs to interact with their environment in undesired ways (e.g., programs might create or delete files randomly).

EVOSUITE was run with six different configurations: We compared the basic version of EVOSUITE with and without the testability transformation, and we used random search as sanity check. We further considered these configurations with contract checking activated (more details will be provided in the following sections). Each configuration was run on each of the 8,844 classes in SF100. Each run was repeated 13 times with different seeds to take the randomness of the algorithm into account. In total, EVOSUITE was run $6 \times 13 \times 8{,}844 = 689{,}832$ times. Considering a two minute timeout, the study took $(689{,}832 \times 2)/(60 \times 24) = 958$ days of computational time.

All data resulting from this empirical study were analyzed using statistical methods following the guidelines in [2]. In particular, we used the Vargha-Delaney $\hat{A}_{12}$ effect size and Wilcoxon-Mann-Whitney U-test. This test is used when algorithms (e.g., result data sets X and Y) are compared on single classes (in R this is done with $wilcox.test(X,Y)$). We also used this test to check on the entire case study if effect sizes are symmetric around 0.5. On some classes, an algorithm can be better than another one (i.e., $\hat{A}_{12} > 0.5$), but on other classes it can be worse (i.e., $\hat{A}_{12} < 0.5$). A test for symmetry (in R this is done with $wilcox.test(Z,mu = 0.5)$, where for example Z contains 8844 effect sizes, one per class in SF100) determines if there are as many classes in which we get better results as there are classes in which we get worse results. Note that this test makes sense if and only if the case study is a valid statistical sample (as it is the case for the SF100 corpus). Otherwise, on hand-picked case studies, the bias in their selection (e.g., proportion of different application types) would make this type of analysis hard to interpret.

All the experiments were run on a cluster which has 80 nodes, each with eight computing cores and eight gigabytes of memory running a Linux operating system. The use of a cluster was necessary due to the large number of experiments. For example, repeating each experiment several times is necessary to take into account the randomness of the algorithm. However, this is different from a normal usage scenario of EVOSUITE, in which for example a software engineer using EVOSUITE, on the software he is developing, would just need to run it once per class.

4.2 Undeclared Exceptions

To answer **RQ1**, we checked what kind of failures were found in all $689{,}832$ runs of EVOSUITE. In particular, we checked all exceptions thrown in the SUT that propagated to the test case (i.e., they were not caught in the SUT). We found 187 different types of exceptions in 6,376 different classes out of the 8,844 in SF100. In total, $32{,}594$ distinct exceptions were found in different methods. For reasons of space, Table 3 only shows the 10 most frequent ones.

Table 3 Top 10 exceptions out of 187, ordered by how many classes they appeared as failures in. We also counted the number of distinct SUT public methods in which those exceptions were not caught.

Name	Classes	Methods
java.lang.NullPointerException	4,250	14,891
java.lang.IllegalArgumentException	2,241	4,468
java.lang.NoClassDefFoundError	1,047	2,015
java.lang.ClassCastException	800	1,749
java.lang.ArrayIndexOutOfBoundsException	574	1,794
java.lang.ArithmeticException	567	1,175
java.lang.ExceptionInInitializerError	480	562
java.awt.HeadlessException	312	500
java.lang.StackOverflowError	230	491
java.lang.NegativeArraySizeException	220	558

RQ1*: In our experiments,* EVOSUITE *found 187 types of exceptions in 6,376 different classes of* SF100*, for a total of* 32,594 *distinct pairs of exception and method.*

There are several interesting things to point out here: During our experiments, EVOSUITE was configured to run tests using a custom security manager to ensure that tests do not interact with their environment in undesired ways. Without this, tests created by EVOSUITE might for example create random files, or change and delete existing files. However, often file access is already attempted in the static constructor of a class (e.g., to set up a logger, to load font files, etc.) When using the custom security manager, this class initialization will thus fail, leading to *NoClassDefFoundError*s or *ExceptionInInitializerError*s. We mainly observed this behavior for GUI components, which additionally also often failed because EVOSUITE runs test using Java's "Headless Mode", which can potentially lead to *HeadlessException*s. Tuning the security manager and simulating the environment are planned for future work and will remove these types of exceptions. For the time being, however, it is necessary to run tests with the security manager as is, because running code randomly downloaded from SourceForge could otherwise have unforeseen consequences.

The most common exceptions are *NullPointerException* and *IllegalArgumentException*. As discussed previously, these exception types likely include failures that are of less interest to the developer as they are based on implicit preconditions. *ClassCastException*s may be caused directly by inputs or by faults: Java type erasure removes type information when using Java Generics, such that to EVOSUITE the signature looks like it takes objects of type *Object*; improved type support is planned as future work for EVOSUITE. If these parameters are cast to concrete classes, this may be a source of *ClassCastException*s. Instances of *ArrayIndexOutOfBoundsException* and of *NegativeArraySizeException* may represent actual faults, or they may again be violations of implicit preconditions, if the array index is directly determined by the test input. Finally, instances of *ArithmeticException* and *StackOverflowError* usually represent faults (e.g., division by zero or infinite loops).

Table 4 Branch coverage and number of distinct implicit exception types/classes that resulted in distinct failures (e.g., in different methods). For example, if there are two null pointer exceptions for the same method and two array out of bound exceptions in two distinct methods, then it would result in two types/-classes and and three exceptions (the two null pointers would count just as a single exception, as they are thrown in the same method), i.e. "# Types" equal to two, and "# Exc." equal to three. The number of times in which the same exception type was thrown by different methods in the SUT (i.e., the "# Exc." column) is used to calculate $\hat{A}_{12}$ effect sizes compared to the base version GA. All values are averaged per class. We counted how often the other two configurations led to worse ($\hat{A}_{12} < 0.5$), equivalent ($\hat{A}_{12} = 0.5$) and better ($\hat{A}_{12} > 0.5$) results compared to GA. Values in brackets are for the comparisons that are statistically significant at $\alpha = 0.05$ level; p-values are of the test for $\hat{A}_{12}$ symmetry around 0.5.

Name	Cov.	# Types	# Exc.	Worse	Eq.	Better	$\hat{A}_{12}$	p-value
GA	0.59	0.72	1.60	-	-	-	-	-
Random	0.53	0.56	0.82	3,013 (2,144)	5,435	396 (50)	0.40	<0.001
TT	0.60	0.77	1.64	1,341 (180)	5,986	1,517 (399)	0.51	<0.001

4.3 SBST vs. Random Testing

As sanity check to see what the effect of the guided search is, we compared the results produced by the GA with random search. The employed random search is closely related to state-of-the-art tools (e.g., [11,35]), but also aims to optimize a test suite with respect to the fitness function as follows: Randomly generate one test case at a time. If the test case improves fitness (i.e., code coverage or thrown exceptions) of the current test suite (which is initially empty), then store it in the test suite. Keep generating test cases while there is still enough testing budget (i.e., until timeout).

Table 4 shows the results of the analyses. This data includes all undeclared exceptions, but only the implicit ones (recall the definition and motivation given in Section 3.1). Comparing GA to Random, the guidance increases not only coverage but also the number of found exceptions with strong statistical significance, from which we can conclude that the use of SBST to exercise automated oracles makes sense.

RQ2*:* EVOSUITE *outperforms random search significantly in terms of coverage and at finding failures.*

4.4 Testability Transformation

To evaluate the effects of the testability transformation, we compare the performance of the default version of EVOSUITE (which we call GA in this paper) with a variant using testability transformation (TT). Comparisons are based on the ability to trigger failures in the SUT.

Table 4 shows that on average TT finds $1.64/0.82 = 2$ times more failures than Random search. On the other hand, the testability transformation does not adversely affect the achieved branch coverage; in fact it is even slightly higher than that produced by GA. This demonstrates that whole test suite optimization is not affected by the number of coverage goals or whether they are infeasible [20] (the transformation may add new infeasible branches, for example by adding a division by zero check for

a value that cannot become 0). We can offer several conjectures regarding the slight increase: EVOSUITE by default uses null with only a low probability, while explicit error branches will reward the use of null, and the increased use of null might in turn lead to higher coverage. The additional error branches may also create gradients in plateaus such that random walks are avoided, and the transformation might also lead to larger test suites.

Considering the detected failures, Table 4 shows that TT achieves, on average, a very small increase, albeit statistically significant. Although there are 180 cases in which it provides statistically worse results, there are more than twice (i.e., 399) the number of cases in which it provides better results. Consequently, we can safely conclude that the testability transformation can be beneficial and should be used by default.

There are several reasons why the effect size is not larger: First of all, the ability to reveal additional failures depends on the existence of faults in the SUT in the first place. Second, the search budget of two minutes we used in our experiments may in many cases not be enough to guide the search towards exceptions related to arithmetic operations or array accesses. Finally, the effect size is shadowed by the sheer number and typology of classes (e.g., GUI components, TCP/UDP connectors) contained in the SF100 corpus that current automated unit testing technologies do not efficiently handle yet.

RQ3*: The testability transformation reveals additional failures but does not negatively affect coverage.*

4.5 Checking API Contracts

So far we have evaluated the performance in terms of all undeclared exceptions. Previous work on testing automated oracles further attempted to filter these exceptions using heuristics [11] or contracts [35]. As the Java programs in SF100 do not include well defined contracts (e.g., like the Eiffel programs used for the AutoTest experiments [32]) we were not able to evaluate how SBST performs with such contracts. However, there are default-API contracts as discussed in Section 3.3.

To reduce the number of contract violations that need to be inspected we report the numbers of *unique* contract violations, based on the heuristic described in the Randoop paper [35]: Two violations of the same contract are considered to be in the same equivalence partition if they followed after the same method call (or call to a constructor or assignment to a field).

We ran all three configurations (Random, GA, TT) on all classes again 13 times with contract checking enabled. Table 5 lists statistics on the violated contracts, averaged per class. Table 6 lists the numbers of unique violations observed. The largest share of contract violations is related to undeclared exceptions. Compared to the total number of undeclared exceptions (UE) the JCrasher heuristic (JCE) reduces the number of reported violations significantly, but the number is still very high. Almost half of the undeclared exceptions are *NullPointerExceptions*, even after filtering using Randoop's heuristic (NPE).

Table 5 Average contract violations per class. Effect size $\hat{A}_{12}$ is calculated on number of violations compared to GA; p-values are of the test for $\hat{A}_{12}$ symmetry around 0.5.

Name	Types	Violations	$\hat{A}_{12}$	p-value
GA	7.20	15,656	-	-
Random	6.69	8,974	0.35	0.000
TT	7.28	17,698	0.54	0.000

Table 6 Number of classes that exhibited a contract violation.

Name	AS	Es	EH	EN	ES	HN	JCE	NPE	TSN	UE
GA	69	14	156	0	8	0	5,068	2,662	236	6,352
Random	64	16	159	0	9	0	4,938	2,575	214	6,336
TT	71	15	149	0	9	0	5,192	2,654	233	6,440

There were no violations of the equals-null (EN) and hashcode-returns-normally (HN) contracts, but all other contracts found violations. ToString lead to exceptions in a number of cases (TSN), and in most cases these are *NullPointerException*s. The violations of Es, EH, ES all point to actual faults.

Comparing the results between the different techniques, we see that the GA leads to more contract violations than random testing with a high statistical significance. The testability transformation does increase the number of detected violations with statistical significance, although interpreted over the large number of classes in SF100 the effect size is of course only small.

RQ4*: In our experiments, the majority of contract violations are related to exceptions. We found violations of all contracts but EN and HN.*

Checking oracles incurs a computational overhead, which may reduce the number of test cases that can be evaluated given a fixed amount of time. To study whether such an overhead is negligible or not, we compared the GA configuration with and without contract checking. We calculate how many statements the two configurations could execute during each run (recall, each run was stopped after two minutes). On average on the entire case study, GA executed $34{,}489$ statements per class when not checking contracts, whereas checking contracts decreased this to $30{,}518$, i.e., 13% difference. To take into account the randomness of EvoSuite (different runs will lead to different number of executed statements), for each class we also calculated the $\hat{A}_{12}$ effect sizes on the executed statements. On average, we obtained a "small" effect size $\hat{A}_{12} = 0.47$, where a test on symmetry around 0.5 gave a p-value close to zero.

RQ5*: On average, checking contracts results in a* 13% *execution overhead.*

4.6 Real Faults due to Assertions and Object Contracts

The majority of failures found by EVOSUITE are due to undeclared exceptions. As described in Section 2.2, an undeclared exception is not necessarily an indication of an important bug — it may simply be a violation of a precondition that was not specified. Note that the problem of implicit preconditions is not specific to SBST, as it is independent of the test generation technique. For example, in a recent study we observed the same problem in Randoop's output [24]: Not a single of the 112 failures on a simple address book application was due to a real fault, but all were caused because Randoop violated the implicit precondition that there can only be one instance of an address book in that application.

There are, however, some cases in which it is possible to be completely sure if a failing test case is due to a real fault: Those are when contracts on the *Object* methods are violated and when *assert* statements in the code are evaluated as false (and therefore throw an error that propagates to the test case).

The use of assertions in the code does not seem to be very common in Java, at least as far as open source software is concerned. For example, in the 8,844 classes of SF100, only **38** have at least one *assert* statement, for a total of **89** assertions. However, EVOSUITE was able to generate test cases for which **71** of these assertions were violated, thus pointing to real faults in those SUTs.

It could be argued that, in some cases, a violated assertion is not necessarily representing a real fault. This could happen if the are implicit pre-conditions and those are violated. Although it is a common practice to do not specify method pre-conditions, it is also true that, once you spend effort to define either a post-condition or an invariant with an *assert* statement, it could sound strange to leave the pre-conditions undefined/implicit. In such cases, the fact that the pre-conditions are left implicit could be considered as real faults that need to be fixed.

Besides assertion violations, the generic object contracts that are guaranteed to represent faults are equals self, equals hash code, equals null, and equals symmetric. Furthermore, as neither the method *Object.hashCode* nor *Object.toString* declares to throw any exceptions, we can also consider violations of the hashCode returns normally, and toString returns normally contracts as real faults. Considering Table 6, EVOSUITE thus found a total of **477** real faults in SF100.

RQ6*:* EVOSUITE *found at least 477 real faults in* SF100*, not considering undeclared exceptions.*

4.7 Qualitative Analysis of Undeclared Exceptions

We observed that the majority of violations found are related to undeclared exceptions, even if considering heuristics to reduce the number of undeclared exceptions counted (see Table 6, columns JCE and NPE). To see how many of the undeclared exceptions found by EVOSUITE are due to actual bugs, we randomly selected 20 classes that resulted in uncaught exceptions. We manually analyzed these classes to see if the failures were indeed symptoms of real faults. As expected considering Table 3, the

majority of exceptions were instances of *NullPointerException*s and of *IllegalArgumentException*s.

All cases of *IllegalArgumentException* were due to a missing *throws* declaration, i.e., they were violations of implicit preconditions, not actual faults in the SUT. However, one cannot simply discard these exceptions, as they might be symptoms of actual faults. For example, if method A in the SUT calls another method B with wrong input due to a fault in A, then that second method B might correctly throw an exception (e.g., in the signature of B we have *throws IllegalArgumentException*), which might propagate to the caller of A. Among the 20 analyzed classes, there was also a case of a user defined exception that was used exactly like a customized *IllegalArgumentException*.

The *NullPointerException*s were raised due to several different conditions. The simplest case is when, given an object obj as input parameter, the SUT calls a method on it resulting in an exception if the input is *null*. This is why Randoop, for example, ignores *NullPointerException*s if a method parameter was null. A more complex example is when a method is called on obj that returns another object, which might be *null* and so the SUT throws a *NullPointerException* if calling any method on it. Another case we found in our manual evaluation is when a SUT constructor is called that leaves some of the internal fields equal to *null*. Successive calls on the instantiated SUT could so result in exceptions even if they do not take anything as input. The last case we encountered is when the SUT directly accesses a static object in another class which is not initialized yet (i.e., default value *null*). In general, all these *NullPointerException*s seem violations of implicit preconditions, and not really critical faults.

Among the manually evaluated test cases, there was an interesting case of a *StackOverflowError*. A tree-like data structure had a method traversing all of its children. However, it was possible to insert in that tree a reference of itself, leading to an infinite recursion when that method was called (and so a *StackOverflowError* after a while).

In another case, a method required an *Object* as input, which was cast to a particular class. Calling such method with a different type of class instance as input led to a *ClassCastException*. Another case of *ClassCastException* is due to a current limitation of EvoSuite related to type erasure: On a method taking as input *ArrayList<String>*, EvoSuite can give as input an *ArrayList* with undefined type, containing instances that are not of type *String*. Note, the test case still compiles.

The last type of exception we encountered were two cases of *ArrayIndexOutOfBoundsException*. In one, an internal array was accessed directly with *set/get* methods that had no checks on input indexes. The other case was more complex: A method took two different objects (A, B) as input. Inside the method, a function on A was called that returned an index that was used as input to a method called on B, which accessed an internal array in B.

4.8 Statistical Analysis of Undeclared Exceptions

Out of the 20 classes with failures manually analyzed in the previous section, eight have actual faults (at least one), i.e., $8/20 = 40\%$ of them. Because those 20 classes were randomly selected among the $6{,}376$ with failures, we can estimate how many of these $6{,}376$ classes have at least one actual fault.

It would be incorrect to state that for sure 40% of those classes (i.e., $2{,}550$) have real faults, as that 40% value is only an estimate based on 20 observations. There could be two extreme cases: there are only eight classes with faults out of the $6{,}376$, or all but $20 - 8 = 12$ classes have faults. Those cases, although extremely unlikely, are still a possibility (the reader interested in the exact probability of these events is referred to [14], as this problem is an instance of the classical urn problem). Therefore, when estimating how many faults there are in total based on a smaller sample, it is important to quantify the *reliability* of the estimate.

4.8.1 Determining confidence intervals

If we call ρ the probability that a class chosen uniformly at random (with replacement) from these $6{,}376$ has at least one real fault, then ρ could be anything between $8/6{,}376 = 0.00125$ and $(6{,}376 - 12)/6{,}376 = 0.998$ (considering that the eight classes with faults were all different, e.g., none counted twice), where $\rho = 0.4$ is the most likely estimate based on the 20 observations. Because whether a class has at least one fault or not is a binary decision, such process follows a binomial distribution [14], where ρ is the probability parameter.

For the binomial distribution, given an observed number x of "successes" (e.g., sampling a class with faults) out of n samples (e.g., $n = 20$), the probability of success is estimated with $\rho = x/n$. For this parameter ρ, it is possible to provide *confidence intervals* at any level $1 - \alpha$. A confidence interval $CI = [a,b]$ for ρ means that there is a $1 - \alpha$ probability that the real ρ^* is within that interval. Many statistical toolkits provide functions for calculating confidence intervals for the parameters of the binomial distribution. For example, in R [37], one can use the function *binom.confint(x,n,*$1 - \alpha$*,methods='exact')*.

If we want to create a 95% confidence interval (i.e., $\alpha = 0.05$) for the estimate $\rho = 0.4$, we obtain $CI = [0.191, 0.639]$. In other words, there is a 95% chance that the actual ρ value is between 19% and 64%, where 40% is the most likely estimate. In this scenario, it is important to stress out that the total number of elements $N = 6{,}376$ from which we sample the $n = 20$ classes to manually evaluate is simply irrelevant to the accuracy of the estimate ρ, only of course as long as $n < N$ (if $n \geq N$ then there would be no point to sample at random).

How many n classes should be sampled and manually evaluated? It all depends on which research questions we are trying to answer. The higher the value n, the better the estimate ρ will be, i.e., the interval $[a,b]$ will be smaller. Conversely, assuming the same sample size as discussed above (i.e., keep n as a constant), if one wants higher confidence (e.g., 99%), then the confidence interval will be larger (i.e., $[0.145, 0.700]$). On the other hand, if one does not need such a high confidence, and for example is already satisfied with a 50% level, then the confidence interval would

be $[0.307, 0.500]$. What is the confidence level $1 - \alpha$ one should aim at? The higher the better, but that goes in contrast to the fact that such analysis of the n samples is manual, and so there are practical constraints.

To make the discussion regarding n and α accessible also for readers less familiar with statistics, let us make a (simplified) example in which binomial distributions are widely used: electoral polls. Assume a small village in which $N = 6{,}376$ citizens have the right to vote, and we are interested if candidate A is going to win the race to mayor of the village against a second candidate B. Out of N citizens, it is possible to select $n = 20$ at random from them, and ask them if they are going to vote for A (recall, this is a simplified example, as we do not want to deal with details such as people lying, not wanting to answer, etc.). Out of these $n = 20$ citizens, eight are going to support candidate A, i.e., the 40% of them. Based on this poll, can we state that candidate A is going to lose or win the election with "high confidence"? If we translate "high confidence" into $\alpha = 0.05$ (which is a somehow arbitrary decision widely used in the literature based on properties of the normal distribution [10]), then there would be no conclusive answer considering the confidence interval $[0.191, 0.639]$. This is because, on one hand, the lower bound 0.19 is lower than 50% (and so cannot be sure of his victory) and, on the other hand, the upper bound is above 50% (and so cannot be sure of his defeat). If one wants to answer those questions without lowering the confidence level $1 - \alpha$, there is no other choice than increasing n till either $a > 0.5$ or $b < 0.5$. However, if rather than a candidate we consider a political party, and we want to know if it will receive at least 10% of the votes (e.g., if 10% is the minimum to obtain a seat in the council), then $n = 20$ is already enough to answer "yes" even at higher confidence level 99%, as $0.145 > 0.1$.

4.8.2 Results

In our context, we are mainly interested to find out if the techniques described in this paper are of practical value for practitioners releasing open source software. Considering a 95% confidence interval, it means that $\rho^* \geq 0.19$, and so there are *at least* 1,217 *real* faults automatically discovered through undeclared exceptions by EvoSuite in SF100 by a simple click of the mouse (EvoSuite only needs the bytecode of the SUT as input, and nothing more than that). The value 1,217 is a lower bound, as ρ^* could be as high as 0.63 (and so 4,074 faults) and each class may contain more than one fault (note, we do not consider this latter case, as it is not necessary to calculate lower bounds and would increase the complexity of the math involved).

The gap between 0.19 and 0.63 is quite large (i.e., a 0.44 difference). If one wanted to have a more precise estimate for ρ^*, and not just a good enough lower bound, how many more n classes would we have to manually evaluate? If, for the sake of discussion, ρ^* was indeed 0.4, then for example a $n = 30$ sample size would lead to a confidence interval $[0.226, 0.593]$ (so a 0.37 difference). Using a sample size as big as $n = 100$ would get a closer confidence interval $[0.303, 0.502]$, but still not particularly tight (i.e., a 0.2 difference). One would need a sample size as large as $n = 400$ to get a confidence interval where the difference between upper and lower bound is lower than 10%, i.e., $[0.351, 0.449]$. Considering the large sample size needed to get tight bounds, and considering the high cost of manually evaluating each

single failure, to answer the research questions in this paper there was no compelling reason to consider more than a $n = 20$ sample size.

RQ7*: In our experiments, considering a 95% confidence level,* EVOSUITE *found between 1,217 and 4,074 classes with real faults leading to undeclared exceptions.*

4.9 Libraries vs. Applications

In our experiments, EVOSUITE found at least $477 + 1{,}217 = 1{,}694$ faults in SF100 using simple automated oracles. This is a surprisingly high number, considering that SF100 includes publicly available and released software. One important observation is that in the literature unit testing is often evaluated on open source *libraries* (e.g., [20]), but the projects sampled for SF100 are mainly *applications*.

In an application, there is usually a single entry point, and all classes are somehow directly or indirectly called from this entry point. For example, if entry class A calls public methods in class B, then only a subset of possible valid inputs will be ever called on B. For example, it could be possible that A will never call B with a *null* value. In contrast, in unit testing it is possible to call all public methods in class B, which could lead to finding unit level failures that would be impossible to have at system level (cf. [24]).

For a library, these failures are critical faults, but for an application they would be less important. They would still be important for maintainability reasons — even if an application works fine in the current version, if it has failures at unit level, it will be more difficult for developers to modify those classes in the future (e.g., to fix bugs or add new features), especially when there is no documentation but the code itself.

4.10 Threats to Validity

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 (e.g., more than 2,000 unit and system tests). 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, we ran each experiment 13 times, and we followed rigorous statistical procedures to evaluate their results.

To cope with possible threats to *external validity*, the SF100 corpus was employed as case study, which is a collection of 100 Java projects randomly selected from SourceForge [18]. In contrast to hand-picked case studies, the use of the SF100 corpus provides high confidence in the possibility to generalize our results to other open source software as well.

At this point, we need to point out that the fault triggering results ρ on SF100 (recall Section 4.8.2) do not directly extend to all the (roughly) 50 thousand Java projects hosted on SourceForge, even though SF100 is a statically valid sample of 100 projects sampled at random. The reason is that, although these 100 projects are a valid sample of projects on SourceForge, their classes are NOT a valid sample of all

the classes contained in all the projects hosted on SourceForge. On one hand, when sampling projects at random, a project containing only one class would have the same chances of being selected as a project composed of thousands of classes. On the other hand, if one does sample $8{,}844$ classes at random from all the available ones, it would be very unlikely that any of those $8{,}844$ classes belong to a small project. As the size of a project might (or might not) be strongly correlated to the probability of its classes having faults that EVOSUITE can find, such a generalization of the results is currently not justified.

Why is SF100 composed of the classes of 100 projects selected at random instead of for example sampling $10{,}000$ classes directly from all the projects? The reason is of practical nature. Downloading and compiling projects from SourceForge requires a significant amount of manual labor, as different projects use different technologies (if any) for building the jar files (e.g., Ant and Maven) and set up scripts with the correct classpath of the required third-party libraries. And, even if it was possible to collect and prepare all the (possibly millions of) classes in SourceForge, it likely will be several gigabytes (if not terabytes) of data, which would be difficult (if possible at all) to handle for research purposes in academic contexts (i.e., limited computational resources). Note that, even if one samples a small subset of classes as SUTs for experimentation, still all the dependent classes would need to be available on the classpath.

In this paper, we evaluated testing algorithms based on their ability of triggering failures. This, however, all depends on whether there are faults in the case study in the first place. On one hand, on poorly written software full of faults even random testing can be very effective, and in those cases more sophisticated tools would not achieve better results. On the other hand, if a project follows an appropriate verification and validation process (e.g., if the developers use test driven development and commit their changes only when all tests pass), then we would not expect many faults in any particular revision version, especially the trivial faults. The faults in the SF100 corpus represent faults that slipped in real-world open source projects, after the developers made a commit (all projects were using a software versioning and revision control system, and for each project we only used a single revision version). However, a tool like EVOSUITE can also be used while engineers develop software, i.e., between code commits. But the type of faults that are introduced and fixed in those cases will not end up in any code repository. To study this important kind of faults, controlled empirical studies in industry would be necessary.

The faults found by EVOSUITE are related to triggered exceptions and violations of assertions/oracles. This does not include all possible kinds of faults, like for example resource leaks and deadlocks. However, during software development, automated unit testing can be used together with other techniques, like for example static analysis (e.g., a popular static analysis tool for Java is FindBugs[4]).

In this paper, we only used EVOSUITE and did not compare to other tools. This has several reasons: First, fair tool comparisons always pose challenges. EVOSUITE aims at being of practical use for software engineers, which means it has to be fully automated and applicable on real-world software, like for example the SF100 cor-

[4] http://findbugs.sourceforge.net, accessed July 2013.

pus (including GUI elements, multi-threading, read/delete of files, opening of TCP sockets, etc). Comparing with tools that require the user to write test drivers manually would not only be difficult, but would be comparing tools with different usage scenarios. Second, we are aware of no other Java tool that can be *automatically* and *safely* applied to SF100. Extending existing tools to apply to SF100 is not just a matter of adding a security manager, as there are many details to consider (see [19] for more discussions on some of the technical challenges involved). Finally, comparing to tools for other programming languages is not possible for the chosen case study. However, the SF100 corpus is freely available, and it will allow tool comparisons in the future when other Java tools are mature enough to handle SF100. However, as discussed in Section 2.4 and demonstrated by past comparisons (e.g., the SBST 2013 tool competition), EvoSuite is representative of the state of the art.

5 Conclusions

Search-based Software Testing (SBST) is a test generation approach that boasts many advantages — it supports many different coverage criteria, can optimize tests towards non-functional criteria (e.g., execution time), and it can be applied to many different test representations (test data, sequences of method calls, GUI event sequences, etc.). However, to date it was not clear how well SBST would perform at the task of exercising automated oracles, and how best to apply it in this context. In this paper, we have presented an extension of SBST and a large empirical study that demonstrates that SBST is able to exercise automated oracles *and* to produce high coverage test suites at the same time. In our experiments, EvoSuite found 32,594 distinct failures in 8,844 classes, which can be attributed to at least **1,694** real faults. At the same time, EvoSuite produced minimized test suites achieving an average bytecode level branch coverage of 60% (which is a good number considering the difficulties such as environmental dependencies contained in SF100 [18]).

To improve the approach further, we have described a testability transformation. This transformation can be used by any SBST tool, not just EvoSuite, and in many cases the testability transformation helped to trigger more failures. However, several of the branches introduced in this transformation offer only coarse guidance, e.g., a reference either is *null* or it is not *null*, but our fitness function does not yet offer guidance towards making the reference *null*. Future work will consider turning this into more fine grained guidance, which will help detecting more faults.

A large share of failures reported are due to undeclared exceptions, but many of these are simply violations of missing preconditions. For example, developers tend to ignore declaring exceptions such as *NullPointerException*s in the method signatures when method preconditions are violated. Technically speaking, these are still faults, but this kind of unit level fault may not directly manifest to the users of the developed application. As such, they might be of less interest for the software engineers (especially when software is developed in tight time/budget constraints).

There are two directions to address this issue in the future. First, there is a need for better techniques to filter out such "false warnings", e.g., using heuristics [11, 35]) or by driving test generation through user interfaces [24]. Once reliable false

warning detectors are available, prioritization techniques could be used to sort the generated test suites, such that test cases that are failing due to actual critical faults are shown first. Second, to improve usability, it is conceivable that the output of a tool like EVOSUITE should not be just a set of test cases, but also annotations in an editor (e.g., markers in the Eclipse editor as used for compiler warnings). For example, methods for which EVOSUITE finds test cases throwing exceptions could be highlighted, and refactoring tools (e.g., developed as Eclipse plug-ins) could give the option to automatically add the needed *throws* declaration if the failures were deemed to be just violations of implicit pre-conditions. The programs used in our experiments were developed without such tools. Having such tools at hand would lead to software where false warnings are never a problem.

For more information about EVOSUITE and the SF100 corpus of classes, please visit our website at:

`http://www.evosuite.org/`

Acknowledgements

This project has been funded by a Google Focused Research Award on "Test Amplification" and the Norwegian Research Council.

References

1. Arcuri, A.: It really does matter how you normalize the branch distance in search-based software testing. Software Testing, Verification and Reliability (STVR) **23**(2), 119–147 (2013)
2. Arcuri, A., Briand, L.: A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Software Testing, Verification and Reliability (STVR) (2012). DOI: 10.1002/stvr.1486
3. Arcuri, A., Fraser, G.: Parameter tuning or default values? An empirical investigation in search-based software engineering. Empirical Software Engineering (EMSE) pp. 1–30 (2013). DOI: 10.1007/s10664-013-9249-9
4. Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: Theoretical results and practical implications. IEEE Transactions on Software Engineering (TSE) **38**(2), 258–277 (2012)
5. Baresi, L., Young, M.: Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, USA (2001). `http://www.cs.uoregon.edu/~michal/pubs/oracles.html`
6. Barr, E., Vo, T., le, V., Su, Z.: Automatic detection of floating-point exceptions. In: Proceedings of the International Conference on Principles of Programming Languages (POPL'13). ACM (2013)
7. Bauersfeld, S., Vos, T., Lakhotiay, K., Poulding, S., Condori, N.: Unit testing tool competition. In: International Workshop on Search-Based Software Testing (SBST) (2013)
8. Bhattacharya, N., Sakti, A., Antoniol, G., Guéhéneuc, Y.G., Pesant, G.: Divide-by-zero exception raising via branch coverage. In: Proceedings of the Third international conference on Search based software engineering, SSBSE'11, pp. 204–218. Springer-Verlag, Berlin, Heidelberg (2011)
9. Clarke, L.A.: A system to generate test data and symbolically execute programs. IEEE Transactions on Software Engineering (TSE) **2**(3), 215–222 (1976)
10. Cowles, M., Davis, C.: On the origins of the .05 level of statistical significance. American Psychologist **37**(5), 553–558 (1982)
11. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw. Pract. Exper. **34**, 1025–1050 (2004). DOI 10.1002/spe.602
12. Del Grosso, C., Antoniol, G., Merlo, E., Galinier, P.: Detecting buffer overflow via automatic test input data generation. Comput. Oper. Res. **35**(10), 3125–3143 (2008)

13. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Transactions on Software Engineering (TSE) **10**(4), 438–444 (1984)
14. Feller, W.: An Introduction to Probability Theory and Its Applications, Vol. 1, 3 edn. Wiley (1968)
15. Fraser, G., Arcuri, A.: EvoSuite: Automatic test suite generation for object-oriented software. In: ACM Symposium on the Foundations of Software Engineering (FSE), pp. 416–419 (2011)
16. Fraser, G., Arcuri, A.: It is not the length that matters, it is how you control it. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 150 – 159 (2011)
17. Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 121–130 (2012)
18. Fraser, G., Arcuri, A.: Sound empirical evidence in software testing. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 178–188 (2012)
19. Fraser, G., Arcuri, A.: EvoSuite: On the challenges of test case generation in the real world (tool paper). In: IEEE International Conference on Software Testing, Verification and Validation (ICST) (2013)
20. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2013)
21. Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. In: ACM Conference on Programming language design and implementation (PLDI), pp. 213–223 (2005)
22. Godefroid, P., Levin, M.Y., Molnar, D.A.: Active property checking. In: Proceedings of the 8th ACM international conference on Embedded software, EMSOFT '08, pp. 207–216. ACM, New York, NY, USA (2008)
23. Gordon Fraser, A.A., McMinn., P.: Test suite generation with memetic algorithms. In: Genetic and Evolutionary Computation Conference (GECCO) (2013)
24. Gross, F., Fraser, G., Zeller, A.: Search-based system testing: High coverage, no false alarms. In: ACM Int. Symposium on Software Testing and Analysis (ISSTA) (2012)
25. Korel, B., Al-Yami, A.M.: Assertion-oriented automated test data generation. In: Proceedings of the 18th international conference on Software engineering, ICSE '96, pp. 71–80. IEEE Computer Society, Washington, DC, USA (1996)
26. Lakhotia, K., Harman, M., Gross, H.: AUSTIN: A tool for Search Based Software Testing for the C Language and its Evaluation on Deployed Automotive Systems. In: International Symposium on Search Based Software Engineering (SSBSE), pp. 101–110 (2010)
27. Lakhotia, K., McMinn, P., Harman, M.: An empirical investigation into branch coverage for c programs using cute and austin. J. Syst. Softw. **83**(12) (2010)
28. Malburg, J., Fraser., G.: Combining search-based and constraint-based testing. In: IEEE/ACM Int. Conference on Automated Software Engineering (ASE) (2011)
29. McMinn, P.: Search-based software test data generation: A survey. Software Testing, Verification and Reliability **14**(2), 105–156 (2004)
30. McMinn, P.: Iguana: Input generation using automated novel algorithms. a plug and play research tool. Tech. rep., The University of Sheffield (2007)
31. McMinn, P.: Search-based failure discovery using testability transformations to generate pseudo-oracles. In: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, Genetic and Evolutionary Computation Conference (GECCO), pp. 1689–1696. ACM, New York, NY, USA (2009)
32. Meyer, B., Ciupa, I., Leitner, A., Liu, L.L.: Automatic testing of object-oriented software. In: Proceedings of the 33rd conference on Current Trends in Theory and Practice of Computer Science, SOFSEM '07, pp. 114–129. Springer-Verlag, Berlin, Heidelberg (2007)
33. Orso, A., Xie, T.: Bert: Behavioral regression testing. In: Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008), WODA '08, pp. 36–42. ACM, New York, NY, USA (2008). DOI 10.1145/1401827.1401835
34. Pacheco, C., Ernst, M.D.: Eclat: Automatic generation and classification of test inputs. In: ECOOP 2005 — Object-Oriented Programming, 19th European Conference, pp. 504–527 (2005)
35. Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T.: Feedback-directed random test generation. In: ACM/IEEE International Conference on Software Engineering (ICSE), pp. 75–84 (2007)
36. Pandita, R., Xie, T., Tillmann, N., de Halleux, J.: Guided test generation for coverage criteria. In: IEEE International Conference on Software Maintenance (ICSM), pp. 1–10 (2010)

37. R Development Core Team: R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2008). URL http://www.R-project.org. ISBN 3-900051-07-0
38. Romano, D., Di Penta, M., Antoniol, G.: An approach for search based testing of null pointer exceptions. In: Proceedings of the 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST '11, pp. 160–169. IEEE Computer Society, Washington, DC, USA (2011)
39. Sen, K., Marinov, D., Agha, G.: CUTE: a concolic unit testing engine for C. In: ESEC/FSE-13: Proc. of the 10th European Software Engineering Conf. held jointly with 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering, pp. 263–272. ACM (2005)
40. Tillmann, N., de Halleux, N.J.: Pex — white box test generation for .NET. In: International Conference on Tests And Proofs (TAP), pp. 134–253 (2008)
41. Tracey, N., Clark, J., Mander, K., McDermid, J.: Automated test-data generation for exception conditions. Softw. Pract. Exper. **30**(1), 61–79 (2000)
42. Visser, W., Pasareanu, C.S., Khurshid, S.: Test input generation with Java PathFinder. ACM SIGSOFT Software Engineering Notes **29**(4), 97–107 (2004)
43. Williams, N., Marre, B., Mouy, P., Roger, M.: PathCrawler: automatic generation of path tests by combining static and dynamic analysis. In: EDCC'05: Proceedings ot the 5th European Dependable Computing Conference, *LNCS*, vol. 3463, pp. 281–292. Springer (2005)
44. Xiao, X., Xie, T., Tillmann, N., de Halleux, J.: Precise identification of problems for structural test generation. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE '11, pp. 611–620. ACM, New York, NY, USA (2011)