Automated Unit Test Generation for Classes with Environment Dependencies

Andrea Arcuri
Certus Software V&V Center
Simula Research Laboratory
Lysaker, Norway

Gordon Fraser
University of Sheffield
Dep. of Computer Science
Sheffield, UK

Juan Pablo Galeotti
Saarland University –
Computer Science
Saarbrücken, Germany

ABSTRACT

Automated test generation for object-oriented software typically consists of producing sequences of calls aiming at high code coverage. In practice, the success of this process may be inhibited when classes interact with their *environment*, such as the file system, network, user-interactions, etc. This leads to two major problems: First, code that depends on the environment can sometimes not be fully covered simply by generating sequences of calls to a class under test, for example when execution of a branch depends on the contents of a file. Second, even if code that is environment-dependent can be covered, the resulting tests may be *unstable*, i.e., they would pass when first generated, but then may fail when executed in a different environment. For example, tests on classes that make use of the system time may have failing assertions if the tests are executed at a different time than when they were generated.

In this paper, we apply bytecode instrumentation to automatically separate code from its environmental dependencies, and extend the EvoSuite Java test generation tool such that it can explicitly set the state of the environment as part of the sequences of calls it generates. Using a prototype implementation, which handles a wide range of environmental interactions such as the file system, console inputs and many non-deterministic functions of the Java virtual machine (JVM), we performed experiments on 100 Java projects randomly selected from SourceForge (the SF100 corpus). The results show significantly improved code coverage — in some cases even in the order of +80%/+90%. Furthermore, our techniques reduce the number of unstable tests by more than 50%.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

Keywords. Unit testing; automated test generation; environment

1. INTRODUCTION

Automated test generation techniques are usually applied to produce test suites with high code coverage (e.g., statement or branch coverage), or to satisfy related criteria such as mutation testing [11] or failure exceptions [9]. The simplest technique is perhaps random testing [3], but more sophisticated techniques such as Dynamic Symbolic Execution (DSE) [15] and Search-Based Software Testing (SBST) [18] have been developed and combined [13]. In the case of procedural code (e.g., programs written in the C language), this task usually amounts to finding input data for the functions under test. For object-oriented software, there is the extra challenge of first putting the classes under test (CUT) and method parameters in the right internal state, which is usually achieved with sequences of function calls.

```
 1 public class EnvExample {
 2
 3  public boolean checkContent() throws Exception{
 4
 5   Scanner console = new Scanner(System.in);
 6   String fileName = console.nextLine();
 7   console.close();
 8
 9   File file = new File(fileName);
10   if(!file.exists())
11    return false;
12
13   Scanner fromFile = new Scanner(new FileInputStream(
          file));
14   String fileContent = fromFile.nextLine();
15   fromFile.close();
16   String date = DateFormat.getDateInstance(DateFormat.
          SHORT).format(new Date());
17   if(fileContent.equals(date))
18    return true;
19
20   return false;
21  }
22 }
```

Figure 1: Example class challenging automated test generation: Method `checkContent` first reads a filename from the console, then reads the first line of that file, and finally compares it to the current date.

However, sequences of function calls on a CUT and its parameter objects may not be sufficient in practice to achieve high code coverage. For example, consider the Java class `EnvExample` in Figure 1: The method `checkContent` has no input parameters in its signature. To achieve full coverage, a tester would first need to create a file that contains the current date on the first line, and then input the name of that file on the console. To the best of our knowledge, there is no research tool in the literature that would do this automatically. There is a further problem in that class: To cover the branch in Line 17 the file would need to contain a string representation of the current date. If the test is later executed on a different day, the result of the string comparison would change, and assertions that use the return value will lead to *unstable* tests that fail. The reason for this is that the success of a test generation technique and the reproducibility of the results for a given CUT may strongly depend on the class's *environment*.

ASE'14, September 15-19, 2014, Vasteras, Sweden.
Copyright 2014 ACM 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642986 .

```
1 public class EnvExampleEvoSuiteTest {
2   @BeforeClass
3   public static void initEvoSuiteFramework() {
4     org.evosuite.Properties.REPLACE_CALLS = true;
5     org.evosuite.Properties.VIRTUAL_FS = true;
6     org.evosuite.agent.InstrumentingAgent.initialize();
7     org.evosuite.runtime.Runtime.getInstance().
          resetRuntime();
8   }
9
10  @Before
11  public void initTestCase(){
12    org.evosuite.runtime.Runtime.getInstance().
          resetRuntime();
13    org.evosuite.agent.InstrumentingAgent.activate();
14    org.evosuite.utils.SystemInUtil.getInstance().
          initForTestCase();
15  }
16
17  @After
18  public void doneWithTestCase(){
19    org.evosuite.agent.InstrumentingAgent.deactivate();
20  }
21
22  @Test
23  public void test0() throws Throwable {
24    SystemInUtil.addInputLine("zF");
25    EvoSuiteFile evoSuiteFile0 = new EvoSuiteFile("/
          private/tmp/zF");
26    FileSystemHandling.appendLineToFile(evoSuiteFile0, "
          14/02/14");
27    EnvExample envExample0 = new EnvExample();
28    boolean boolean1 = envExample0.checkContent();
29    assertEquals(true, boolean1);
30  }
31
32  @Test
33  public void test1() throws Throwable {
34    SystemInUtil.addInputLine("z");
35    EnvExample envExample0 = new EnvExample();
36    boolean boolean0 = envExample0.checkContent();
37    assertEquals(false, boolean0);
38  }
39
40  @Test
41  public void test2() throws Throwable {
42    SystemInUtil.addInputLine("q");
43    EvoSuiteFile evoSuiteFile0 = new EvoSuiteFile("/
          private/tmp/q");
44    FileSystemHandling.appendStringToFile(evoSuiteFile0, "
          M");
45    EnvExample envExample0 = new EnvExample();
46    boolean boolean1 = envExample0.checkContent();
47    assertEquals(false, boolean1);
48  }
49 }
```

Figure 2: A JUnit test suite automatically generated by EvoSuite that achieves full coverage on the class `EnvExample` defined in Figure 1.

In this paper, we extend the EvoSuite unit test generation tool to handle environmental dependencies by a) using bytecode instrumentation to control the environmental state and inputs during test generation, b) including the environmental state and inputs as part of the search space of EvoSuite, and c) observing environmental interactions during the tests and restoring the environmental state after test execution. This allows EvoSuite to control not only function inputs and call sequences, but also the state of the environment, resulting in stable tests with higher code coverage. Figure 2 shows a test suite for the `EnvExample` example class that achieves 100% branch coverage: `test0` (Line 23) creates a mocked file and sets its content to “14/02/14”, which is the default mocked date set by EvoSuite. When `checkContent` is invoked, the branch in Line 17 is thus true, and the method returns `true`. Test `test1` inputs the name of a file that does not exist, thus covering the branch in Line 10. Finally, test `test2` creates a file, but does not set the required contents, thus the false-branch in Line 17 is covered. The code between Line 2 and 20 in the test suite serves to control the environment during subsequent test executions outside of EvoSuite.

But how often do environment interactions happen in practice? In previous work [8], we ran experiments with the EvoSuite unit test generation tool on 100 Java projects randomly selected from SourceForge (i.e., the SF100 corpus). We observed an unexpectedly high number of environmental interactions and generally low code coverage, leading to the conjecture that these two observations are linked. We thus studied the effects on running test generation on the SF100 corpus. In detail, in this paper we provide the following contributions:

- An instrumentation technique to isolate Java code from the filesystem.
- An instrumentation technique to isolate Java code from the console input.
- An instrumentation technique to isolate Java code from the system state (time, properties, etc.)
- An extension of the EvoSuite test generation tool that includes the state of the environment (e.g., file contents, console input, etc.) as part of the input space.
- An extension of the EvoSuite test generation tool that efficiently handles changes to the environment performed by the tests (e.g., changes of the static state of classes).

Experiments on selected classes demonstrate the power of this approach in terms of large coverage increases and large reductions of the number of unstable tests. Experiments on the full SF100 corpus of classes show improvement in both coverage and unstable tests, but the overall improvements in coverage indicates that there are further obstacles to achieving high coverage.

Although our main motivation and evaluation is on coverage and unstable tests, the techniques introduced in this paper provide more general benefits: First, the use of a virtual file system is important for automated test generation as it prevents the real file system from being corrupted by. Second, isolating test cases from their environment makes tests execute faster than when they rely on an actual file system [4], and also makes them independent such that parallelisation becomes easy.

2. BACKGROUND

2.1 The Environment of a Class

In object-oriented software development, unit testing typically focuses on individual classes, or the methods of a class. Such a class is embedded in a complex environment with a range of dependencies. Consider Figure 3: The class under test (CUT) and its dependency classes have a state that depends on their class loader. That is, independently of the state of any instances of the CUT, the class itself can have a state in the context of its class loader; this state is typically defined by static variables. Class loaders, in turn, are embedded in a virtual machine (VM) environment, which itself has a complex state. For example, the state of a VM may be defined by system properties. The VM, in turn, interacts with the operating system environment, where the state depends on the system time, memory state, filesystems, hardware configuration, and many other factors. The operating system is part of a machine, which can interact with the user or other machines and services over a network. Interactions with the environment at the level of the JVM and beyond are performed via standard library calls provided with the host language (e.g., Java). Interacting with the class loader specific environment happens without any specific calls, it is simply reflected by the states of static variables.

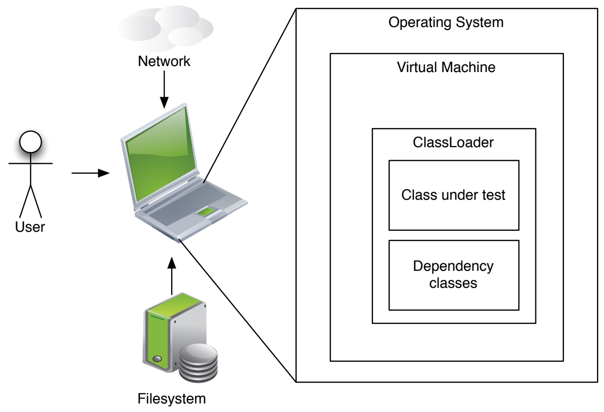

Figure 3: The environment of a class under test.

2.2 System Test Environment Simulators

When developing systems that interact with hardware components (e.g., sensors and actuators), it is common practice in industry to build environment simulators. These simulators can be used for system level testing, and allow the tester to run the developed system without the need for real hardware (e.g., Software-in-the-loop testing). This is particular useful for when the hardware is not available yet during the software development process, and also for large scale testing when hardware is expensive.

There is a lot of research on how to best develop those environment simulators, and how to use them for testing purposes. For example, Arcuri et al. [2] developed a technique to define the environment as UML state machines, and automatically generate environment code. However, even though the testing was automated, there was still the initial *manual* effort of defining the UML models.

While such an approach is reasonable to simulate hardware, in this paper we focus on unit level testing, and not system level testing. Furthermore, here we desire full automation for test generation: There should be no need of manual intervention to write any models or test drivers. Ideally, once EVOSUITE is installed, the user should only need to specify which class/project she wants to generate tests for (e.g., with a mouse-click in IDEs like Eclipse).

2.3 Mocking at the Unit Level

A common approach to isolate a class from its dependencies is by *mocking*. In general, mocking refers to the use of replacement classes during testing rather than real classes. There are different related terms, e.g., a *stub* usually has a fixed default behavior, whereas a *mock* typically has to be set up as part of a test, and then can verify expected interactions. Other common terms are *fakes*, *dummies*, or *test doubles*. In this paper, we will use the term "mocks" to denote replacement classes where the behavior can be set up as part of the test.

The creation of mocks is often supported by mocking frameworks, which are are widely used in industry when unit tests are manually written. For Java, there are many tools like Mockito[1], EasyMock[2] and JMock[3]. Isolating a class using mocks and stubs is good from a testing point of view, but may create a maintenance problem, as the behavior of the mock classes needs to be kept in sync with the real behavior. In this paper, we do not mock user classes, which in principle are under control of the programmer. Rather, we aim to bring the environment under control; but to achieve this, we do use an approach based on mocking.

The use of mocking in automated test case generation is less common, albeit some promising techniques have been developed. For example, when there are CUTs that take references to some specific interface as input, but no concrete class exists that implements these interfaces, concrete classes can be created on the fly [16]. If there are formal specifications (i.e., formal pre and post conditions), such information can be used to create more "realistic" mock classes [14]. Automated mocking can also be used to replace input instances that have operations that take long time to run [19]. Furthermore, it has also been used to partially handle interactions with databases [20] and file systems [17].

This latter work is perhaps the one that shares most similarities with what presented in this paper. In [17], mocks were created to handle interactions with the file system, and an initial case study was carried out on eight methods. However, those mocks were not sharing their behavior (i.e., the writing of a text in a mocked file would not have effect to the reading of the same file from another mock), and could only be used when given as input to the CUT. Therefore, such an approach would not work on a class like the one depicted in Figure 1, as the object interacting with the file system is not given as input to the CUT.

The Pex [21] test generation tool is integrated with the Moles framework [6], which supports integration of user-supplied mocks (*moles*) during testing. Once a mole has been manually created, it permits Pex to derive path conditions during symbolic execution in cases where calls to the environment would otherwise prohibit that. Promising results were obtained on a 15 line of code method [6].

The approach presented in this paper builds on this general line of research and provides a full working solution, addressing different technical challenges of environment mocking. In contrast to previous work, our approach is fully automated, e.g., it does not require the users to manually provide mocked classes or do any manual processing on the CUTs. Furthermore, we provide an evaluation using more than two million lines of code (the SF100 corpus), suggesting practical usefulness beside initial promising results.

2.4 Automated Unit Test Generation

In this paper, we take the EVOSUITE [7] unit test generation tool as a starting point for our investigations. EVOSUITE is an advanced research prototype that automatically generates unit test suites for Java programs. It works at Java bytecode level (so it can also be used on third-party systems with no available source code), and it is fully automated: it does not require any manually written test drivers or parameterized unit tests. For example, when EVOSUITE is used from its Eclipse plugin, a user just needs to select a class, and tests are generated with a mouse-click.

EVOSUITE uses a genetic algorithm in which it evolves whole test suites, an approach that has been shown to be more efficient at achieving code coverage than generating tests individually [10, 11]. Once unit tests with high code coverage are generated, EVOSUITE applies various post-processing steps to improve readability (e.g., minimizing) and adds test assertions that capture the current behavior of the tested classes. To select the most effective assertions, EVOSUITE uses mutation analysis [12]. EVOSUITE can generate test suites covering different kinds of coverage criteria, like for example weak and strong mutation testing [11], and it can also aim at triggering undeclared exceptions; for example, this has been demonstrated to automatically find real faults in several open source projects [9]. EVOSUITE can be integrated into a programmer's development environment with its Eclipse plugin, or it can be used on the command line.

[1]http://code.google.com/p/mockito/, accessed March 2014
[2]http://www.easymock.org, accessed March 2014
[3]http://jmock.org, accessed March 2014

3. HANDLING THE ENVIRONMENT DURING TEST GENERATION

The environment of a class under test needs to be controlled during test generation in order to make code dependent on the environmental state or inputs reachable, and to make test cases stable and deterministic in subsequent executions. In this section, we describe our approach to control the environment during test generation in EVOSUITE and in the resulting test cases. When designing the techniques presented in this paper, we tried to satisfy the following important constraints:

- The mocked environment has to be semantically equivalent to the real one. For example, if a running thread modifies a file (e.g., by writing text to it), such modifications should be reflected in all other threads and successive interactions on such a file (e.g., it should be possible to read what was written in the file).
- The user should not need to do any manual installation or configuration of any third-party tool (e.g., installing a virtual operating system, or set up a customised JVM). Everything should be fully automated and running within a standard JVM, independently of the operating system.
- The developed solutions for environment mocking should not break any current practice in the development process of the final users. For example, if the testers use tools like JaCoCo/EclEmma[4] to measure achieved coverage, then those should still work with a mocked environment.

3.1 Instrumenting Bytecode in Unit Tests

EVOSUITE, like many other test generation tools, relies heavily on bytecode instrumentation. In Java, it is easy to attach instrumenting code to a class loader, such that all classes will receive the same instrumentation. Thus, during test generation all test executions use a customized class loader that ensures the instrumentation. However, we need to ensure that the generated JUnit test cases faithfully represent the exact same behavior as observed during test generation. Using a custom class loader is not a viable option in the generated JUnit tests: Once a JUnit file is generated, we cannot really make assumptions on how it will be used. For example, class loaders used in Eclipse can be different from the ones used in Maven when test cases are run. Furthermore, the user might employ special plugins which use their own special class loaders that do their own set of bytecode changes. This is the typical case of plugins used to measure code coverage (e.g., EclEmma), which add their probes to the analyzed classes when loaded in memory.

Our solution uses JavaAgent technology: When a JUnit test suite is run, it automatically starts a JavaAgent that *automatically* binds to the JVM in which the tests are executed. This agent is responsible for instrumenting and modifying the bytecode of classes loaded, regardless of the used class loader. Figure 2 shows how this is done by calling methods on the class `InstrumentingAgent`. For example, `InstrumentingAgent.initialize()` (Line 6) will set up the agent, and `activate` (Line 13) and `deactivate` (Line 19) will instruct the agent to perform or stop doing bytecode instrumentation.

3.2 Controlling Static State

During the execution of a test case, the CUT can not only access object and primitive values stored in static fields (i.e., the static state of the program), it might also *modify* them, leading to different behaviour if the test case is executed again or if the execution order of the tests in a test suite is altered. Let us illustrate this problem with a simple example: Figure 4 shows a class that uses a static integer field to provide unique identifiers to each object that is created.

```
public class Counter {
 private static int count = 0;

 public int id;
 public Counter() {
  id = count++;
 }
}
```

Figure 4: A simple class that modifies static state.

```
@Test
public void test() {
 Counter counter = new Counter();
 assertEquals(0, counter.id);
}
```

Figure 5: A generated test case for `Counter`.

Whenever a new instance of `Counter` is created, the field `count` is increased. EVOSUITE generates the test case shown in Figure 5, but this test will only pass if no other `Counter` instance was ever created during the execution of any other test case in the same test suite. If another test case is executed before `test` and creates a `Counter` instance, the value of `count` will be wrong, and the test will fail. Even worse, during test generation, *many* tests will be executed, and thus the value of `count` is quite unpredictable.

A possible workaround to this problem is to discard all classes that were loaded by the JVM's class loader after each test case execution is concluded, for example by using a different class loader for every test execution (e.g., [5]). The advantage of such a solution is that, whenever a class is used again, the JVM will need to reload it, and the static state will be the initial one. In the context of a test generation tool, the obvious disadvantage of this solution is a significant decrease of performance, since class files have to be re-read from the file system (or the network, if a `URLClassLoader` is being used), instrumented, and the static initialisation blocks all have to be re-executed, even if no static state was changed at all. Furthermore, other tools relying on their own class loading schemas (e.g., JaCoCo or EclEmma), may not be compatible to this solution (e.g., as they require their own class loaders and instrumentation), breaking current practices of the final user.

3.2.1 *Handling updates in static fields*

An alternative to reloading all classes lies in resetting their state. In Java, the static state of a class is initialized in a special method (`<clinit>`), and an idea introduced with JCrasher [5] is to create a copy of that method that can later be invoked to reset the static state of a class. This copy needs to be modified to avoid attempts to re-initialize fields with the `final` modifier, as the JVM would not permit this. This approach has some limitations; in particular, although it saves the effort of loading classes repeatedly it still requires execution of the reset methods for every class. This is not very efficient if the number of classes loaded is large — and in our experience, projects often have thousands of dependency classes.

In order to handle the changes in the static state during test case execution more efficiently, EVOSUITE monitors any change to the static fields of the CUT; i.e., every writing access to a static field (`PUTSTATIC` in Java bytecode) is instrumented with a method call that records the occurrence. If along the test execution an update of a static field occurs (e.g. `count`), we re-execute the static initialisation method (i.e., the copy of `<clinit>`) of the declaring class of the static field (in this case, class `Counter`) when the test has finished. This solution avoids re-loading all classes, and also limits the re-execution of static initialiser methods only to those classes where an actual change was detected during test case execution.

[4]http://www.eclemma.org/, accessed March 2014

```
public class Counter {
 private static AtomicInteger atomicInteger = new
      AtomicInteger();

 public int id;
 public Counter() {
  id = atomicInteger.getAndIncrement();
 }
}
```

Figure 6: A simple class that modifies static state using the singleton pattern.

3.2.2 *Reverting changes in the state of objects*

However, this *optimistic* approach has serious limitations. To understand these limitations, let us introduce the modified version of the previous example in Figure 6. In this modified version, the integer primitive value is replaced by the complex type `AtomicInteger` from the JDK library. Singletons like this one are very common in practice, and they can be found extensively in open source projects such as those included in SF100. Observe that the value of the field `atomicInteger` will not change, but the state of the object referenced by that field will change. This change will escape our monitoring of writing accesses to static fields during test execution, as there is no explicit writing to a static field[5].

Since extending our monitoring approach to cover such cases would require a substantial amount of overhead (i.e. monitoring each change to any field of any object reachable from all the static fields), we complement the optimistic approach with a *pessimistic* extension. Given all classes that were loaded during the execution of the whole test suite, we re-execute the class initialisers for all of them after each test case is finished. This solution is by far less performant that the optimistic approach presented earlier. Nevertheless, to control the impact in performance we combine both approaches. During the regular genetic algorithm execution of EVOSUITE, we rely on the *optimistic* approach to handle changes to the static state of the CUT. Also, along the entire evolution of the chromosomes we track all the classes that are initialised by the CUT. When EVOSUITE finishes the genetic algorithm, a more focused phase starts to generate the assertions that will act as oracles. During this assertion generation phase, test cases are re-executed repeatedly, but less often than during the search algorithm. In this stage of EVOSUITE test generation, we apply the *pessimistic* approach, and we reset all classes that were loaded during the search of the test suite.

This schema is completed by two additions. First, since the order in which classes are loaded might have an impact on the static state, we also try to reproduce the exact sequence of class initialisation that occurred during the genetic algorithm. Secondly, `final` fields are instrumented as `non-final` by the instrumentation framework. The latter change has no effect from the final user perspective (since the original bytecode does not modify the final field more than once) and allows the re-execution of the class initialiser to appropriately restore the value of the static fields to their original state. Take for instance the `atomicInteger` field. Even if the class `<clinit>` method is executed again, it will not update the value of the field. In contrast, if the field is non-final, then the execution of the class initialiser will replace the *used* instance with a fresh one.

In the generated test cases, the `initEvoSuiteFramework` method is invoked once before any test of the test suite is executed (`@BeforeClass`) and ensures that EVOSUITE's instrumentation is used, and that all classes are loaded in the correct order (see Figure 2, Line 2). The instrumentation is activated before, and deactivated after every test execution, such that any further loaded classes are correctly instrumented (e.g., to receive the additional static reset methods based on `<clinit>`). Finally, after every test execution `doneWithTestCase` invokes `resetClasses` (not shown in Figure 2 as there is no static state in this example) which calls the static state resetting method on all classes for which the static state was changed.

[5]The static variable is loaded with GETSTATIC, and then a method is invoked using INVOKEVIRTUAL.

3.3 Mocking Environment Interactions

Handling static state only covers the environment in the scope of a class loader (see Figure 3). To be able to control the environment inputs of a class beyond that, we need to *mock* them. In other words, we create a simulated environment, which is controlled by the generated test cases. The type of simulator/mock will depend on the type of the environment input we want to simulate. For example, mocking the file system will be different from mocking the CPU clock-time. Mocking the environment has several advantages:

- Test cases will be faster to run, as they do not have to deal with expensive I/O operations on the environment (e.g., reading and writing files from a physical hardisk).
- Test cases become independent from each other. E.g., two test cases manipulating the same file can be run in parallel without the worry of one influencing the other, as each of them will have its own instance of the simulated file system.
- No negative side-effects. A tester will not need to worry to apply a tool like EVOSUITE on a class that deletes files, as all those operations will be automatically made on a virtual file system. Even code that generates new temporary files can be problematic for search-based test generation, because during the search hundreds of thousands of CUT statements could be executed, and even code like `File.deleteOnExit()` would have effect only after the search.
- Higher coverage due to exception handling. When using a simulated environment, we can easily simulate error conditions. For example, if there are problems in the hardisk, Java classes manipulating files might end up throwing an `IOException`. In the search, as part of the test data, we can mark virtual files to simulate that they should throw an `IOException` if any operation is called on them.
- The test cases become more *stable*. For example, consider a CUT accessing to the CPU time-clock. The accessed time value could affect not only the achieved coverage, but also the JUnit assertions that capture the executed behavior. If run again, the JUnit test case will likely fail, as a new clock value is used. By mocking the CPU clock-time (and other non-deterministic events), we can prevent this kind of unstable tests. This is particular important when test suites generated by tools like EVOSUITE are used for regression testing.

The basic principle of this approach is to identify the standard library classes/methods that are responsible for environment interactions, and to replace them with mocked versions. The behavior of these mocked versions should be determined by the test generator in order to set up the environment in whatever way is necessary to achieve coverage. This is achieved by providing dedicated helper methods that set the environment state (e.g., to set the system time), and make them part of the methods that EVOSUITE considers during test generation. However, including such calls in the input space when there is no actual interaction with the environment may have a negative effect on the efficiency of the test generator. For example, if the CUT does not do any I/O on the file system, then there is no point in generating files as test data, as those will never

Table 1: Classes mocked in EvoSuite.

Filesystem	Non-determinism
`java.io.File`	`java.lang.Exception`
`java.io.FileInputStream`	`java.lang.Throwable`
`java.io.FileOutputStream`	`java.lang.Runtime`
`java.io.RandomAccessFile`	`java.lang.System`
`java.io.FileReader`	`java.lang.Class`
`java.io.FileWriter`	`java.lang.Thread`
`java.io.PrintStream`	`java.lang.Math`
`java.io.PrintWriter`	`java.util.Date`
`java.util.logging.FileHandler`	`java.util.Calendar`
`javax.swing.JFileChooser`	`java.util.GregorianCalendar`
`javax.swing.filechooser`	`java.util.Random`
`.FileSystemView`	`java.util.logging.LogRecord`

be used and would just hamper the search. Therefore, the environment mock classes dynamically detect if they are used, and only if a part of the environment has been accessed, the corresponding helper methods are made available to the test generator.

3.3.1 Console Inputs

When a Java program is run from a console, then the program can read user's inputs on the console from `System.in`, which is an instance of `java.io.InputStream` in Java. For example, the class `PasswordReader` from Sourceforge project Schemaspy reads passwords while masking the output with "*" characters. When EvoSuite is run on such a class, it replaces all occurrences of the static variable `System.in` in all instrumented class with a customized `InputStream`, which is called `SystemInUtil`. By using a customized stream, we can detect after a test case execution if the CUT has tried to read from `System.in`.

If the CUT has tried to read from `System.in`, in the next generation of the evolutionary algorithms we add the static method `addInputLine(String str)` of the class `SystemInUtil` to the set of methods EvoSuite can use in the generated JUnit test cases. This method, which will be now part of the search, adds its input string to the stream, and so it simulates a console input from the user. EvoSuite will now also try to find input values for this method that lead to higher coverage. Calls to this method will be included in the generated JUnit tests, as can be seen in Figure 2 (e.g., Lines 24, 34, and 42). Note that the mocked system input will be reset before every test execution using the call to `initForTestCase` in `SystemInUtil` (Line 14).

3.3.2 Virtual File System

In Java, there are different ways to access the filesystem. Handling the filesystem is important to allow a test generator to cover code dependent on files, but that is not the only reason: In our past experiments on SF100 [8], we had incidents where several parts of the hardrive were wiped out during test generation by unfortunate sequences of calls and inputs. As a consequence, we added a custom security manager that allows files to be *read*, but not to *write*, *delete* or *execute* any of them. However, as this security manager may adversely affect the code coverage, as part of this paper we have implemented a virtual file system for EvoSuite that fully runs inside the JVM. To force the CUTs to use this virtual file system, we have mocked the standard Java API classes for IO. This includes the classes in Table 1.

In this context, a mock is class that extends the class it is meant to mock, overriding *all* its methods. For example, `java.io.File` has 37 methods we needed to change to operate on the virtual file system, plus a further 15 methods that needed no changes. All methods of all classes operate on the same virtual file system, making the changes done by one mock class affecting the behavior of all the other mock classes. The instrumentation consists of replacing all invocations of methods and constructors of the original class (e.g., `java.io.File`) to invocations of the mocked class (e.g., `MockFile`). In addition, EvoSuite is instructed not to use any of the methods of the mocked classes, but the methods of the mock classes instead. Because each mock class is a subclass of the class it mocks it can be used like the original class, and all class signatures remain unchanged.

When a mocked class has methods that return an instance of another class dealing with the file system, we provide mocks for those returned instances. For example, the method `getChannel` in `FileInputStream` returns an instance of the abstract class `FileChannel` of the `java.nio.channels` package. In this case, our mocked `getChannel` returns a concrete instance for `FileChannel` that is consistent with the current state of the given file input stream. However, we still not fully support the `nio` package (e.g., calls to static methods like `FileChannel.open` are not handled yet).

The virtual file system is re-initialized after each test case execution. In addition, after a test case execution, we can check if the CUT has tried to read from any file. If so, then in the next generations of the evolutionary algorithm we add to search several different methods to create and manipulate such files, which will now be a new kind of test data. One such method is `appendLineToFile`, which can be seen for example in Figure 2 (Lines 26 and 44).

3.3.3 General Java Virtual Machine Calls

Besides filesystem and console input, there are many less obvious sources of environmental inputs to a class. For example, there are different ways in which a class can access the current system time (`System.currentTimeMillis`, `System.nanoTime`, `Date`, `Calendar`, `GregorianCalendar`). Other classes and methods, such as `Random`, `SecureRandom`, and `Math.random`, may indirectly depend on the system time when setting the initial seed. Some more subtle examples of environmental state are information about the operating system state; for example, the name and string representation of a `Thread` object depends on how many threads are currently running in the system, and this information is used in various places such as loggers (e.g., `LogRecord` or exception messages). Also the stack trace of a thread is used, e.g., in `Exception` and `Throwable`, and this is typically different depending on which JUnit runner is used. In all these cases, we use mock classes or methods with deterministic behaviour. In addition, if the current system time is accessed, then a special method `System.setCurrentTimeMillis` is added to the methods EvoSuite uses for test generation, which allows the current system time to be set explicitly. Furthermore, each call to a function that retrieves the time increases the mocked time by 1. Similarly to this, the next random number can be explicitly set by EvoSuite.

Figure 7 illustrates an example where the system time is involved: Covering the `MessageBean` class is quite trivial — the class has seven methods but not a single conditional statement or loop. However, the method `getCreation` will return a string that represents the exact time at which the `MessageBean` was instantiated during test generation. During later test execution, an assertion using this value would fail. By mocking the system time and dependent classes such as `GregorianCalendar`, the time is under control of the test. Although the time could be set by the test, in `MessageBean` this is not even necessary, and thus the default time used in EvoSuite is reported, and the test is now stable.

There are more sources of nondeterminism: For example, when using Java reflection, the order in which `Method` objects or other reflection objects are returned is nondeterministic. To avoid this

```
public class MessageBean implements Serializable{
 private static final SimpleDateFormat SDF = new
       SimpleDateFormat("dd/MM/yyyy␣HH:mm:ss␣SSS");
 private GregorianCalendar creation = new
       GregorianCalendar();

 public String getCreation() {
  return SDF.format(creation.getTime());
 }

 // ...
}
```

```
 @Before
 public void initTestCase(){
  org.evosuite.runtime.Runtime.getInstance().
        resetRuntime();
 }

 @Test
 public void test1() throws Throwable {
  MessageBean messageBean0 = new MessageBean("+CV_[B!oP␣
        CbfRC");
  String string0 = messageBean0.getCreation();
  assertEquals("14/02/2014␣20:21:21␣320", string0);
 }
```

Figure 7: Excerpt from the class `MessageBean` that depends on the system time (Sourceforge project DB-Everywhere). Achieving full code coverage on this class is easy, but `test1` would fail unless executed at exactly the right time. However, EvoSuite instrumentation controls the system time, and the time reported in the assertion is EvoSuite's default time.

problem, we mock all reflection calls with a wrapper class that sorts any returned list of reflection objects by name.

Hash codes are a particularly tricky source of failing tests. To produce deterministic hash codes, we provide a mocked version of `System.identityHashCode`, which looks up a sequential object ID based on the actual identity hash code of a given object. If a class does not implement its own `hashCode` method, then we add one using instrumentation which calls the mocked `identityHashCode` function.

Finally, `java.lang.Runtime` provides various system specific values such as `availableProcessors` or `freeMemory`, which would differ depending on the machine on which a test is executed. Again, we provide a mocked version of that class. Table 1 summarizes which classes are currently fully or partially mocked in EvoSuite.

4. EMPIRICAL STUDY

In this paper, we aim to address the following research questions:

RQ1: Does controlling the environment successfully increase coverage on known cases of environmental interactions?

RQ2: Does controlling the environment successfully resolve known issues of unstable tests?

RQ3: How do results generalize to the SF100 corpus?

RQ4: How many unsafe environmental interactions does EvoSuite now handle?

4.1 Experimental Setup

To answer the research questions posed in this paper, we used the SF100 corpus [8], which is a collection of 11,219 Java classes from 100 Java projects randomly selected from SourceForge. We carried out three different sets of experiments:

For the first set of experiments, we *manually* selected 30 Java classes from the SF100 corpus that interact with the environment. The selection was made by looking for usage of environment-related API in the source code, and by observing security exceptions triggered by EvoSuite's restrictive security manager (for example, the security manager will prohibit and report all attempted write accesses to files). Such a *biased* selection is important to study what kind of improvements could be obtained on classes that could most benefit from the proposed techniques. For example, not much improvement would be seen when using a mocked file system during test generation for CUTs that do not interact with any files. Furthermore, using such a small set of classes has the following benefits: we can run more experiments with different random seeds, we can manually study the classes in more detail, and it can also be easier for researchers to use such a set in the future for replicated experiments and comparisons.

On this set of 30 classes, we applied EvoSuite with six different configurations: the default one ("Base"), one configuration for each technique discussed in Section 3, i.e, "Console" (Section 3.3.1), "VFS" (Section 3.3.2), "JVM" (Section 3.3.3), "Static" (Section 3.2), and we also considered a configuration in which all four techniques are activated at the same time ("All"). For each configuration, we ran EvoSuite 100 times on each CUTs. Each run lasted up to three minutes. In total, this took $(20 \times 6 \times 100 \times 3)/(60 \times 24) = 38$ days of computational resources.

For the second set of experiments, we manually selected 30 Java classes from the SF100 corpus that lead to unstable tests. Note that not all classes interacting with the environment automatically lead to unstable tests; indeed many of the classes chosen for the first set of experiments simply lead to very low coverage without the additional instrumentation, but not unstable tests. After test generation, EvoSuite attempts to compile and run all generated tests and will thus detect and report unstable tests. We selected 30 classes from this information. Again, experiments were repeated 100 times with the same six configurations.

In the third set of experiments, we used the whole SF100 corpus, consisting of all the 11,219 classes it currently includes. Due to the large number of experiments, we ran only two configurations: "Base" and "All". Each experiment was repeated five times with different random seeds. This led to $(11{,}219 \times 2 \times 5 \times 3)/(60 \times 24) = 233$ days of computational resources.

Put together, these three sets of experiments took 308 days of computational resources. To run such large amount of experiments, we used a cluster of computers using six cores (12 considering hyper-threading) at 2.6 GHz., running a Linux distribution. To properly analyze the results of these large sets of experiments, we followed the guidelines in [1]. In particular, we used the Wilcoxon-Mann-Whitney U-test (at $\alpha = 0.05$ significance level) and the Vargha-Delaney $\hat{A}_{12}$ standarized effect size.

4.2 Results

4.2.1 *RQ1: Effects on code coverage*

Table 2 shows the results for the first set of experiments on 30 environment dependent Java classes. On this manually selected set of classes, performance improvements are very large, going from 29% average branch coverage of the "Base" configuration to the 82% achieved by the "All" configuration.

In some cases the improvement is due to a single technique. For instance, the classes `Log4jImportCallable` and `Renaming-FileStream` require an input file with content that has to satisfy certain constraints. A large coverage increase can already be achieved by simply creating these files, but with the three minutes of search time EvoSuite was generally not able to produce much meaningful file content. For example, `FileEditor` requires content in an ad-hoc format for storing application data, and `Log4jImportCallable` requires valid XML valid inputs.

Table 2: Results for RQ1: Average branch coverage achieved by each of the six analyzed configurations. Effect size $\hat{A}_{12}$ is calculated for the "All" configuration compared to the "Base" one. Statistically significant differences at 0.05 level are shown in bold.

Class	Base	Console	VFS	JVM	Static	All	$\hat{A}_{12}$
com.imsmart.parser.MCSVParser	0.10	0.10	**0.76**	0.10	0.10	**0.78**	**1.00**
net.sourceforge.beanbin.reflect.ReflectionShelf	0.75	0.75	0.75	0.75	**1.00**	**1.00**	**1.00**
com.allenstudio.ir.core.ConfigurationManager	0.09	0.08	**0.53**	0.08	**0.41**	**0.80**	**1.00**
com.soops.CEN4010.JMCA.JParser.SimpleNode	0.26	**0.74**	0.27	0.25	0.26	**0.73**	**0.99**
net.sourceforge.jwbf.core.bots.util.SimpleCache	0.40	0.40	**0.89**	0.40	0.40	**0.99**	**1.00**
jipa.Variable	0.60	0.60	0.60	0.61	**0.99**	**0.99**	**0.71**
apbs_mem_gui.FileEditor	0.20	0.20	**0.77**	0.20	0.20	**0.77**	**1.00**
net.kencochrane.a4j.file.FileUtil	0.29	0.29	**0.40**	**0.28**	**0.29**	**0.40**	**1.00**
httpanalyzer.HeaderSettings	0.94	0.94	0.94	0.95	**1.00**	**1.00**	**0.79**
corina.browser.RelativeDate	0.83	0.83	0.83	**0.95**	0.83	**0.94**	**1.00**
corina.util.GZIP	0.12	0.12	**0.71**	0.12	0.12	**0.70**	**1.00**
net.sourceforge.schemaspy.util.PasswordReader	0.08	**0.68**	0.09	0.09	**0.11**	**0.64**	**1.00**
net.sourceforge.schemaspy.view.StyleSheet	0.24	0.24	**0.24**	0.24	**0.79**	**0.86**	**1.00**
framework.base.ValueListHandler	0.12	0.12	0.12	0.12	0.12	**0.93**	**1.00**
de.beiri22.stringincrementor.helper.StringFromFile	0.33	0.33	**0.80**	0.33	0.33	**0.80**	**0.97**
de.huxhorn.lilith.log4j.xml.Log4jImportCallable	0.22	0.22	**0.74**	0.22	0.22	**0.73**	**1.00**
dk.statsbiblioteket.summa.ingest.stream.RenamingFileStream	0.00	0.00	**0.87**	0.00	0.00	**0.87**	**1.00**
ch.bfh.egov.nutzenportfolio.persistence.customizing.CustomizingIbatisDao	0.00	0.00	**1.00**	0.00	0.00	**1.00**	**1.00**
nu.staldal.lagoon.BuildSitemap	0.30	0.33	**0.77**	0.30	**0.69**	**0.79**	**0.99**
client.ClientProperties	0.00	0.00	**1.00**	0.00	0.00	**1.00**	**1.00**
fi.vtt.probeframework.javaclient.api.probe.PFTest	0.79	0.79	0.77	**0.67**	**0.88**	**0.85**	**0.68**
fi.vtt.probeframework.javaclient.protocol.IO	0.67	0.68	0.69	0.61	**0.99**	**0.96**	**0.91**
fr.pingtimeout.jtail.io.index.RomFileIndex	0.08	0.08	**0.99**	0.08	0.08	**0.99**	**1.00**
org.fixsuite.message.parsers.fpl.MainParser	0.25	0.25	**0.87**	0.25	0.25	**0.99**	**1.00**
com.lts.application.repository.ArchiveRepository	0.07	0.07	**0.56**	0.07	0.07	**0.56**	**1.00**
net.sf.xbus.tools.QLoad	0.22	**0.66**	0.22	0.22	0.22	**0.63**	**1.00**
umd.cs.shop.JSUtil	0.28	**0.62**	0.30	0.27	**0.25**	**0.64**	**0.99**
jigl.signal.io.SignalInputStream	0.03	0.03	**0.56**	0.03	0.03	**0.54**	**1.00**
org.jcvi.jillion.internal.core.io.RandomAccessFileInputStream	0.34	0.34	**0.79**	0.34	0.34	**0.80**	**1.00**
Newzgrabber.IniUtility	0.20	0.20	**0.79**	0.20	0.20	**0.79**	**1.00**
Average	0.29	0.35	0.65	0.29	0.37	0.82	0.96

Table 3: Results for RQ2: Average number of unstable tests generated for each of each of the six analyzed configurations. Effect size $\hat{A}_{12}$ is calculated for the "All" configuration compared to the "Base" one. Statistically significant differences at 0.05 level are shown in bold.

Class	Base	Console	VFS	JVM	Static	All	$\hat{A}_{12}$
org.databene.jdbacl.JDBCDriverInfo	2.72	2.52	2.52	2.49	**0.00**	**0.00**	**0.02**
net.sourceforge.beanbin.reflect.ReflectionShelf	1.14	1.16	1.20	**2.06**	**0.00**	**0.00**	**0.20**
org.jsecurity.realm.ldap.AbstractLdapRealm	2.99	3.04	3.17	3.21	**0.14**	**0.13**	**0.12**
org.jsecurity.codec.Base64	1.27	1.28	1.35	1.74	**0.00**	**0.00**	**0.28**
gui.gl.Camera	2.37	2.36	2.33	2.19	**0.00**	**0.07**	**0.04**
module.MessageFactory	4.20	4.19	4.24	4.15	**1.55**	**0.01**	**0.00**
corina.gui.Bug	1.33	1.40	1.39	**0.00**	1.40	**0.00**	**0.13**
net.sourceforge.schemaspy.Config	0.64	0.75	0.79	0.82	**0.06**	**0.08**	**0.30**
de.huxhorn.lilith.handler.Slf4JHandler	0.84	0.98	0.75	**0.00**	0.77	**0.00**	**0.38**
dk.statsbiblioteket.summa.common.Record	7.16	7.09	6.93	**0.03**	6.94	**0.00**	**0.00**
com.gbshape.dbe.struts.bean.MessageBean	0.61	0.62	0.61	**0.00**	0.62	**0.00**	**0.22**
visu.handball.moves.model.animation.PassLineIterator	2.50	2.52	2.52	2.38	**0.00**	**0.00**	**0.04**
visu.handball.moves.model.player.Player	5.53	5.57	5.47	5.87	**0.00**	**0.00**	**0.10**
fi.vtt.noen.mfw.bundle.probe.plugins.measurement.MeasurementTask	3.73	3.66	3.41	**0.02**	**1.73**	**0.00**	**0.02**
fi.vtt.probeframework.javaclient.api.probe.PFTest	6.86	6.79	7.50	**4.41**	7.05	**0.30**	**0.22**
org.exolab.jms.net.connector.ConnectionContext	1.75	1.73	1.47	1.57	**0.00**	**0.00**	**0.13**
org.exolab.jms.message.Timestamp	1.19	1.23	1.03	**0.00**	1.17	**0.00**	**0.18**
com.lts.util.DateUtil	2.59	2.87	2.55	**0.09**	2.27	**0.11**	**0.08**
com.lts.util.ImprovedRandom	2.49	2.39	2.93	**0.02**	**1.62**	**0.00**	**0.03**
fr.unice.gfarce.identity.Formation	1.91	1.90	1.87	**0.00**	1.77	**0.00**	**0.11**
org.javathena.utiles.ConfigurationManagement	3.96	3.87	**2.40**	3.81	**0.00**	**0.00**	**0.01**
org.javathena.core.utiles.Functions	0.93	0.93	0.78	**0.00**	0.80	**0.03**	**0.23**
net.sf.xbus.protocol.records.RecordTypeMessage	1.82	1.59	1.28	**0.00**	1.55	**0.00**	**0.23**
org.sourceforge.ifx.basetypes.IFXDateTime	3.26	3.67	3.66	**0.00**	3.38	**0.00**	**0.05**
net.virtualinfinity.atrobots.arena.Heading	1.83	1.76	1.79	**0.12**	1.60	**0.16**	**0.09**
org.jcvi.jillion.assembly.consed.ace.WholeAssemblyAceTag	2.13	2.12	2.21	**0.00**	2.19	**0.00**	**0.05**
org.quickserver.net.server.impl.BasicClientHandler	1.91	1.86	2.18	1.79	**0.01**	**0.00**	**0.07**
org.heal.module.oai.provider.basic.BasicResumptionToken	1.63	2.01	1.87	**0.86**	1.22	**0.00**	**0.14**
mygrid.Performance	1.22	1.22	1.22	**0.00**	1.19	**0.00**	**0.00**
net.sourceforge.ext4j.log.Server	0.57	0.54	0.57	0.67	**0.00**	**0.00**	**0.24**
Average	2.43	2.45	2.40	1.27	1.30	0.02	0.12

Some classes (`BuildSitemap`, `ClientProperties` or `RomFileIndex`) not only tried to read from files, but also to write to them (e.g., by dumping an XML document with a description of the object). In those cases the correlation between the "VFS" configuration and the overall improvement is very clear.

Nevertheless, in some classes I/O accesses were combined with changes to the static data (such as `ConfigurationManager`) or calls to `Random` or `Date` (such as `FileUtil`). Here, the combined configuration ("All") achieved the highest coverage.

Interestingly, "Static" by itself led to higher coverage for almost one third of the subjects. A closer inspector revealed that this is mostly due to branches that are very difficult to reach if singleton objects are not reset on a test by test basis (e.g., `HeaderSettings`). Finally, the mocking of `System.in` (the "Console" configuration) was effective in all cases where it occurred (`QLoad`, `JSUtil` and `PasswordReader`).

RQ1: *Controlling the environment can have a large impact on branch coverage, even in the order of +80%/+90%.*

4.2.2 RQ2: Effects on unstable tests

Table 3 shows the results for the second set of experiments on 30 classes resulting in unstable tests. Without any treatment, the average number of failing tests per test suite is in the range of 0.57 to 7.16. Adding console and filesystem mocking has no significant effects on the number of unstable classes, except in one case. This is as expected: When checking for unstable tests, we compiled and ran the JUnit test suites on the same computer and on the same file system. Thus, if any files are read from, this would not be affected. Interestingly, for `ConfigurationManagement` the number of unstable tests nevertheless decreases when mocking the filesystem. This class implements a singleton pattern and attempts to read from a file. If this file does not exist, static default properties are used, which contributes to unstable tests. When mocking the filesystem, EvoSuite creates the file on the virtual filesystem and typically manages to set at least some of the properties, such that fixed rather than static values are used. Consequently, the number of unstable tests is reduced. When resetting the static state, all unstable tests disappear.

In general, either resetting the static state or mocking nondeterministic JVM calls removes the unstable tests in almost all cases. Out of those cases fixed by the JVM-related instrumentation, the main source for unstable tests is time: E.g., usage of `Date` (e.g. `Bug`), `System.currentTimeMillis` (e.g. `PFTest`, `DateUtil`) or `GregorianCalendar` (e.g. `Formation`). Some classes use random numbers (`Functions`, `Heading`), and `Performance` accesses information about the available memory on the system from `Runtime`. In total, out of the 30 classes our inspection revealed that 13 are affected exclusively by non-deterministic calls to the JVM, another 12 are affected strictly by changes to the static state, and the remaining 5 are affected simultaneously by both.

There are several classes where no individual technique manages to remove all unstable tests, but only a combination of techniques succeeds. For example, `MessageFactory` is a singleton that serves as a factory of objects that access the current time. When just mocking JVM calls, the created objects will still be influenced by the static state and lead to unstable tests; when just handling static state, there is a reduction in the number of unstable tests, but the created objects will still refer to the current time. Used together, the unstable tests disappear. Conversely, for `PFTest` or `BasicResumptionToken` it is the other way round: Replacing JVM calls achieves some reduction, but only in conjunction with handling static state.

Interestingly, there are several classes where unstable tests happen even for the "All" configuration, although rarely (e.g., `PFTest`). These cases exhibit corner cases we have not handled yet. For example, `PFTest` defines its own `hashCode` method that builds the hash code from a range of different objects, and is a method which EvoSuite tries to cover. Every now and then EvoSuite manages to find specific cases not handled by our instrumentation (e.g., EvoSuite's own mock objects in the experiments underlying the presented data). It is an ongoing engineering effort to cover all corner cases.

RQ2: *Controlling the environment removes unstable tests for intended sources of non-determinism.*

4.2.3 RQ3: Generalization of results

When run on all the 11,219 classes of the SF100 corpus, the default version of EvoSuite obtains 76.5% branch coverage. When using all the environment techniques discussed in this paper (the "All" configuration), then the coverage is increased to 77.9%, covering 2,803 additional branches. For each class we calculated the $\hat{A}_{12}$ effect size, and then checked if the resulting 11,219 are symmetric or not to 0.5 (i.e., if there are more classes in which "All" provides better results than worse). The improvement is statistically significant, as the resulting p-value is very close to zero.

Although the overall coverage increase is significant, why does EvoSuite still not achieve 100% coverage? To some extent, this is simply due to the large number and variety of classes in SF100—any specific improvement will only apply to a fraction of the classes, and thus only slightly increase coverage. The environmental dependencies we handled in our experiments are only some of many remaining issues in achieving high code coverage (e.g., multi-threading, GUI code, networking, databases, etc.).

Furthermore, based on our observations (cf. **RQ1**) we conjecture that the I/O related coverage increase could be larger if EvoSuite were more efficient at generating relevant *content* for these files. Although this is already possible in principle with the techniques presented in this paper, three minutes of regular search may simply not be enough to optimise complex strings representing file content. Optimisations such as the integration of DSE [13] for the specific task of optimising mocked data will be helpful in achieving this goal, but requires additional engineering effort.

For each class, we checked if, in any of the five runs per configuration, there were any generated test cases that were unstable. On one hand, the default "Base" EvoSuite led to 1,452 classes with unstable tests. On the other hand, the techniques discussed in this paper reduced the number of classes with unstable tests to 671, i.e., less than half.

Sampling the classes with unstable tests revealed some common cases that still need to be handled:

- GUI related code is notoriously non-deterministic. For example, AWT and Swing components have names and IDs that change with every execution. In our implementation, we have not addressed mocking of GUI components yet.
- Multi-threading is a well-known source of non-determinism. EvoSuite so far does not address multi-threading explicitly, but many classes in SF100 spawn threads.
- Hashcode calculation on objects not handled by mocking can lead to nondeterminism, when the hashcode values are used in computations (which may happen in the CUT or the test).

In addition, we found several cases where reproduction of the unstable tests observed on the cluster (which runs Linux) were difficult on our development machines, which are Macs. Consequently, there are some remaining platform specific details not fully covered by our infrastructure.

RQ3: *Coverage on* SF100 *increased significantly, and the number of unstable tests was reduced by half.*

4.2.4 RQ4: Effects on unsafe operations

Some environmental interactions can not only influence coverage or stability of tests, they can be harmful. For example, in our past experiments on SF100 [8] we have observed classes that create files named using string inputs, leading to the disk being cluttered with thousands of randomly named files; we even had incidents where we lost all of our experimental data and SF100 files due to unsafe interactions. As a consequence, EVOSUITE uses a conservative security manager that prohibits any potentially unsafe operations.

When we run EVOSUITE with a virtual file system, it could still happen that some CUTs access the real file system, as not all Java API classes are mocked. In these cases, the default sandbox will prevent harmfull operations, but it is of interest to verify how much of the I/O issues are solved. When run with its default configurations, there are 3,436 CUTs that do access the file system at least once in a potentially harmful way. When run with the virtual file system introduced in this paper, there are only 599 cases left.

A manual verification of some of those 599 CUTs point to some I/O related classes that were not mocked, like `ZipFile` and `JarFile`. Furthermore, there are quite a few cases in SF100 in which the CUTs try to start non-Java processes. These result in *execute* permission requests on the executable files, which are denied by our sandbox (also in the presence of the virtual filesystem).

RQ4: *Our implementation covers a large share of I/O, but fully covering the Java standard library is ongoing engineering effort.*

4.3 Threats to Validity

Threats to *internal validity* might come from how the empirical study was carried out. EVOSUITE and the extensions are heavily tested ($>$ 1600 tests), although ultimately there is no guarantee against faults. As we employed randomized algorithms, all experiments were repeated (100 times for the smaller samples of classes; five times on the entire SF100 corpus).

To cope with possible threats to *external validity*, the SF100 corpus was employed as case study. SF100 is a collection of 100 Java projects randomly selected from SourceForge [8], and thus provides high confidence in the possibility to generalize our results to other open source software as well. Although we ran EVOSUITE on different operating systems during testing and development, the experiments were run only on Linux. Consequently, there remains the question of whether results also hold on other operating systems, or across different operating systems. In general, Java should be platform independent, but we have observed different environmental behavior occasionally.

Threats to *construct validity* are on how the performance of a technique is defined. We focused on code coverage and unstable tests in our experiments. However, in practice, an increase in coverage may not be desirable if it results in degraded readability of the test cases. To ensure validity of our measurement of unstable tests, all tests were compiled and executed outside of the EVOSUITE framework. However, some cases of unstable tests may escape this check. For example, an assertion that checks the current date may only fail when executed on a different day. Nevertheless, our experiments show that the reduction is significant, and point out areas that need to be addressed to further increase test stability.

5. CONCLUSIONS

When generating unit tests, isolating the unit under test from its environment is important to achieve higher code coverage. To obtain reliable regression suites, it is also important to produce test cases that are stable and deterministic.

In this paper, we have presented a fully automated approach where we replace API calls and classes related to the environment with mocked versions, and allow EVOSUITE to explicitly set the state of the environment and its inputs as part of the tests it generates. In particular, with the techniques introduced in this paper we are able to handle inputs from the console, the file system, the static state of all classes loaded in the JVM, and, finally, we handle many the of non-deterministic functionalities of the JVM.

Experiments on selected classes exhibiting these problems show the large potential of improvement resulting from this approach. In some cases, improvements were in the order of +80%/+90% branch coverage, and the number of unstable tests is greatly reduced. To study how these results generalize and to see how common these environment problems are for practitioners, we also carried out an empirical study on the SF100 corpus, which contains 11,219 Java classes consisting of more than two million lines of code.

Larger experiments on the SF100 corpus of classes confirmed those improvements, showing that the techniques presented in this paper have practical value for software engineers. These novel techniques are implemented in the EVOSUITE tool, which is freely available to download from *www.evosuite.org*.

However, these experiments also show that there is further work to be done. First, there is a range of engineering tasks necessary to extend the support of environmental isolation:

- As witnessed by the remaining unstable tests, there are further sources of non-determinism we have not yet handled in our implementation, for example caused by multi-threading and GUI code.
- As witnessed by the remaining file-related security exceptions, we have only mocked a subset of the file-related API of Java. There are other APIs (e.g., `nio` and `nio2`) that require further engineering work to be fully mocked.
- There are other environmental interactions we have not addressed yet, such as networking (e.g., sockets) and databases.

Besides these engineering issues, there are more fundamental issues that require more in-depth research:

- Our approach so far enables the search to optimize the environmental state and inputs, but there is a need to improve the ways in which these inputs are optimized. For example, the search should be able to detect whether a file should contain binary or XML content, and then advanced search techniques are required specifically for each type of input.
- Even if the search is fully able to optimize complex inputs, including the binary input of a file explicitly in the test case may not be ideal, as test cases may need to be read and understood by developers during maintenance or test oracle generation. Consequently, future work needs to consider ways to improve the readability of the generated test cases.

Acknowledgments. This work is supported the EPSRC project "EXOGEN" (EP/K030353/1), a Google Focused Research Award on "Test Amplification", the Norwegian Research Council, the ERC grant SPECMATE and EU FP7 grant 295261 (MEALS).

6. REFERENCES

[1] A. Arcuri and L. Briand. 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.

[2] A. Arcuri, M. Z. Iqbal, and L. Briand. Black-box system testing of real-time embedded systems using random and search-based testing. In *IFIP International Conference on Testing Software and Systems (ICTSS)*, pages 95–110, 2010.

[3] A. Arcuri, M. Z. Iqbal, and L. Briand. Random testing: Theoretical results and practical implications. *IEEE Transactions on Software Engineering (TSE)*, 38(2):258–277, 2012.

[4] J. Bell and G. Kaiser. Unit test virtualization with VMVM. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 550–561, New York, NY, USA, 2014. ACM.

[5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Softw. Pract. Exper.*, 34:1025–1050, 2004.

[6] J. de Halleux and N. Tillmann. Moles: tool-assisted environment isolation with closures. In *Objects, Models, Components, Patterns*, pages 253–270. Springer, 2010.

[7] G. Fraser and A. Arcuri. EvoSuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[8] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

[9] G. Fraser and A. Arcuri. 1600 faults in 100 projects: Automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering (EMSE)*, 2013. To appear.

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

[11] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering (EMSE)*, 2014. To appear.

[12] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)*, 28(2):278–292, 2012.

[13] J. P. Galeotti, G. Fraser, and A. Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *IEEE International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[14] S. J. Galler, A. Maller, and F. Wotawa. Automatically extracting mock object behavior from Design by Contract specification for test data generation. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 43–50, 2010.

[15] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI'05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223. ACM, 2005.

[16] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *International Workshop on Dynamic Analysis (WODA)*, pages 26–31, 2010.

[17] M. R. Marri, T. Xie, N. Tillmann, J. De Halleux, and W. Schulte. An empirical study of testing file-system-dependent software with mock objects. In *Automation of Software Test, 2009. AST'09. ICSE Workshop on*, pages 149–153, 2009.

[18] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[19] D. Saff and M. D. Ernst. Mock object creation for test factoring. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 49–51, 2004.

[20] K. Taneja, Y. Zhang, and T. Xie. Moda: Automated test generation for database applications via mock objects. In *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*, pages 289–292, 2010.

[21] N. Tillmann and W. Schulte. Parameterized unit tests. In *Proc. of the 10th European Software Engineering Conference and 13th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 253–262, New York, NY, USA, 2005. ACM.