Generating Unit Tests with Structured System Interactions

Nikolas Havrikov, Alessio Gambi, and Andreas Zeller
Saarland University
Germany
{havrikov, gambi, zeller}@cs.uni-saarland.de

Andrea Arcuri
Westerdals Oslo ACT, Norway, and
the University of Luxembourg
arcand@westerdals.no

Juan Pablo Galeotti
University of Buenos Aires,
Argentina
jgaleotti@dc.uba.ar

***Abstract*—There is a large body of work in the literature about automatic unit tests generation, and many successful results have been reported so far. However, current approaches target *library* classes, but not full applications.**

A major obstacle for testing full applications is that they interact with the environment. For example, they establish connections to remote servers. Thoroughly testing such applications requires tests that completely control the interactions between the application and its environment.

Recent techniques based on mocking enable the generation of tests which include environment interactions; however, generating the right type of interactions is still an open problem.

In this paper, we describe a novel approach which addresses this problem by enhancing search-based testing with complex test data generation. Experiments on an artificial system show that the proposed approach can generate effective unit tests. Compared with current techniques based on mocking, we generate more robust unit tests which achieve higher coverage and are, arguably, easier to read and understand.

I. Introduction

Throughout the years, many successful techniques and tools to automatically generate unit tests for object-oriented systems have been developed [1]. Unfortunately, it is insufficient to just generate sequences of function calls on the class under test (CUT) and its parameters to achieve thorough unit testing. In fact, when a CUT interacts with the *environment*, its behavior becomes dependent on exogenous factors. For example, classes that open connections to remote servers might behave differently depending on how the servers respond to their requests.

When writing unit tests for such classes, developers have the following options to deal with the environment: 1) interact with the environment, for example by connecting to remote servers; 2) stub out the environment resources, for example by implementing a local test server; 3) isolate the unit tests from the environment by encapsulating interactions with resources in specific classes which *mock* their behavior [2].

Modern unit test generation tools like EvoSuite [3] can generate tests using any of these techniques [4]–[6]. While this reduces the burden of writing tests for the developers, it might introduce issues which limit the applicability of each technique. In fact, in many cases, developers must manually inspect all tests to ensure their correctness.

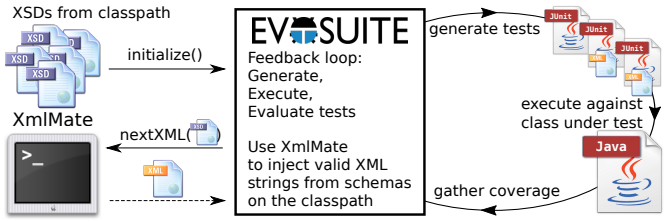

Figure 1. Overview of the integration of EvoSuite and XMLMate for generating unit tests with realistic environmental test data

Interacting with the real environment makes the test unreliable as its results become dependent on factors that are not under the control of the developers.

Mocking libraries allow to control every aspect of the behavior of the mocked objects, but provide no guidance in doing so; as a consequence, the test generation tools might create executions that would be impossible to obtain in practice like configuring a mocked object to return `null` despite the real object preventing this. This leads to frequent generation of unit tests resulting in false positives and false negatives alike.

The ability to mock the environment also introduces issues because tools can stub environmental resources only randomly. This means that when complex inputs, like XML messages, are required by the classes under test, it is unlikely that the test generation tools provide syntactically correct values.

There are tools like XMLMate [7] that can effectively generate samples of complex test data for well defined grammars like XML Schema definitions (XSD).

In this paper, we explore the possibility of using such tools to enhance automatic unit test generation. Specifically, we integrate EvoSuite, a modern unit tests generation tool, and XMLMate into a prototype that can generate string data representing XML files with arbitrary content. For a non-trivial artificial system, our prototype generated a set of unit tests that, compared to competitive solutions, are more robust, achieve higher coverage, and result in no false positives.

II. Motivating Example

We introduce the challenges of automatically generating unit tests with realistic environmental data and the limitations of current approaches with a motivating example.

```java
@XmlRootElement
@XmlAccessorType(XmlAccessType.FIELD)
public class Dto {
 @XmlElement(required = true)
 private Boolean a;
 @XmlElement(required = true)
 private Boolean b;
 /* getters and setters follow */
}
```

Figure 2. Source code of the `Dto` class

```java
public class Retriever {

final String httpUrl = "http://www.
    someExternalService.org/dto.xml";
final String schemaLocation = "/DtoSchema.xsd";

public Dto retrieveDto(){
  return fromXML(getXmlContent());
 }

private String getXmlContent() throws Exception{
    /* return string read from httpUrl */
 }

private Dto fromXML(String xml) throws Exception{
    /* return xml unmarshalled according to
    schema */
 }
}
```

Figure 3. Source code of the `Retriever` class

Consider the Java classes described in Figures 2, 3, and 4. These three classes implement a system which interacts with a remote Web service and elaborate the values returned by it:

- `Dto` is a POJO (Plain Old Java Object) which encapsulates the data transferred from the remote Web service to the application and implements the data transfer object pattern [8].
- `Retriever` handles the HTTP connections, contacts the remote Web service, and reads back the data. Additionally, this class converts the server data, i.e. the content of an XML file, into complex Java objects which it then returns to its caller. We must note that `Retriever` expects the XML data to adhere to a specific XML Schema Definition (XSD) defined by the remote Web service.
- `Checker` takes as input a boolean `val` and a reference to a `Retriever`, uses the `Retriever` to read data from the remote Web service, and computes a result based on some properties of two variables: the boolean `val`, which is passed as input, and a `Dto` instance, which is returned by the `Retriever`.

From the point of view of unit testing, those three classes pose quite different challenges which relate to accessing external resources and dealing with dependencies.

Challenges in Class `Dto`

The class `Dto` is a simple POJO with a default constructor, and setters and getters for all its fields. This makes it the easiest class to test. Modern tools like EvoSuite can easily generate tests which cover all the code of `Dto`.

```java
public class Checker {

 static boolean check(Boolean val, Retriever r) {
  Dto dto = r.retrieveDto();
  if(val && dto.getA() && ! dto.getB())
   return true;
  else
   return false;
 }
}
```

Figure 4. Source code of the `Checker` class

```java
@Test(timeout = 4000)
public void test1() throws Throwable  {
 EvoSuiteURL evoSuiteURL0 = new EvoSuiteURL("http
    ://www.someExternalService.org/dto.xml");
 NetworkHandling.createRemoteTextFile(
    evoSuiteURL0, "J*&#efc!g*");
 Retriever retriever0 = new Retriever();
 Dto dto0 = retriever0.retrieveDto();
 assertNull(dto0);
}
```

Figure 5. Testing `Retriever` with environmental mocking and random test data.

Challenges in Class `Retriever`

The class `Retriever` does not require any special initialization and its sole public method `retrieveDto()` takes no arguments. In theory, generating tests for `Retriever` should not be problematic; in practice, however, this task is challenging as the execution of `retrieveDto()` involves some environment interactions which include:

- creating a TCP connection toward an external server;
- making an HTTP request;
- retrieving the corresponding HTTP response;
- elaborating its payload.

For this method, any automated test generation approach must control the availability of the environmental resources. Additionally, it must also control the data inside the payload to avoid generating *flaky* tests.

In order to deal with environmental dependencies while achieving high code coverage, EvoSuite employs a technique called *Environment Mocking*. This technique allows EvoSuite to intercept the interactions with the external resources and override the values they return, such that unit tests are isolated from the environment. This way during execution the tests do not directly interact with any external resource, which improves their robustness. For example, the execution of `getInputStream()` in the class `Retriever` is mocked by EvoSuite, which replaces the byte stream with values generated during EvoSuite's search.

Although EvoSuite is effective in mocking external resources and Web services, the problem of generating **meaningful** output data remains. Take as example the test case for the `Retriever` class in Figure 5. EvoSuite generated a random string instead of a valid XML input. As a consequence,

```
1  @Test(timeout = 4000)
2  public void test4() throws Throwable  {
3   Boolean boolean0 = Boolean.TRUE;
4   Dto dto0 = mock(Dto.class, new
       ViolatedAssumptionAnswer());
5   doReturn((Boolean) null).when(dto0).getA();
6   Retriever retriever0 = mock(Retriever.class, new
        ViolatedAssumptionAnswer());
7   doReturn(dto0).when(retriever0).retrieveDto();
8   try {
9    Checker.check(boolean0, retriever0);
10   fail("Expecting exception: NullPointerException
       ");
11  } catch(NullPointerException e) {
12  }
13 }
```

Figure 6. An example of a test for `Checker` which employs functional mocking, but results in a false positive.

the class `Retriever` only is exercised when returning `null` values, and the achieved code coverage is far from optimal. To achieve higher coverage, the test generator should not only generate a valid XML string, but an XML string which satisfies the XSD of the connecting Web service.

Challenges in class `Checker`

The class `Checker` contains only a very simple predicate (line 5); however, covering all of its branches depends on `Retriever` getting a valid XML input from the remote Web service, thereby returning `non-null` instances of `Dto`. In other words, even if the class `Checker` does not directly access the remote Web service, its unit tests require a suitable setup of the environment.

Although the `Checker` class does not directly depend on the environment, it has dependencies on the other two classes, `Retriever` and `Dto`. A common practice to generate unit tests for classes like `Checker` is to rely on mocking [2] and programmatically define interesting behaviors for the dependency classes.

EvoSuite applies a technique called *Functional Mocking*[1] to address those cases. This technique allows the test generator to automatically synthesize mock objects that are fed to the target class. Figure 6 presents a test case which relies on functional mocking for exercising the target class `Checker`.

Although thanks to functional mocking EvoSuite achieves full code coverage of the class `Checker`, problems still remain. For example, Figure 6 shows a test case that mocks an instance of `Dto` that forces `Checker` to throw a null pointer exception (NPE). However, after inspecting the class `Retriever` it becomes clear that none of the instances of `Dto` which can be created in practice would lead to a NPE. In fact, in the case of null values the XSD validation step for `Dto` would fail before reaching the point of execution that throws the NPE. This results in *false positives*, i.e. test executions that would be impossible in the original, non-mocked code.

In summary, by relying solely on environmental and functional mocking, tools like EvoSuite might not be able to test all the code of classes like `Retriever` that interact with the environment, and they might not be able to generate effective test suites for classes like `Checker` that depend on other classes and have indirect environment interactions.

To overcome those problems and automatically generate unit tests which achieve higher coverage, are stable, and result in less false positives, we propose to integrate modern test generation tools, like EvoSuite, with state-of-art generators of complex data, like XMLMate.

III. Integrating EvoSuite and XMLMate

We enable the automatic generation of unit tests that require realistic environmental data by extending EvoSuite with the ability to call the generator of structural data XMLMate during the search. We depict this approach in Figure 1.

Specifically, we extend the current implementation of EvoSuite's environmental mocking to query XMLMate with some configurable probability whenever the class under test interacts with an external resource. As a response, XMLMate generates a random, yet syntactically valid XML string which is used as test data.

From EvoSuite's perspective a string representing a valid XML is similar to any other (random) string. As a consequence, given the nature of EvoSuite's search to reuse data, XML strings might end up in places where they are not expected, for example inside the `URL` constructor, possibly causing exceptions, like `java.net.MalformedURLException`. Despite this, we must note that tools like EvoSuite are robust against this type of behavior as, given enough time, they will eventually rule out misplaced XML strings, thus generating meaningful unit tests.

To generate valid XML content, XMLMate needs an XML Schema which specifies what type of XML must be created. In fact, each XSD describes concrete XML formats by specifying the *elements* (also known as *tags*) and their *attributes* that are allowed to be part of the document. Additionally, XSD supports a type system, which can assign not only primitive types like `boolean`, `string`, or `int`, but also *complex types*, which allow to compose multiple elements into larger trees by means of concatenating or alternating child elements. When given an element definition to instantiate, XMLMate looks up its type and recurses for any children until it reaches types that can be instantiated and for which random values can be generated in a conventional manner. Therefore, EvoSuite must first understand what XSD files are available and then decide which one to use for each environmental interaction.

Java programs are commonly delivered as packages that contain all the resources necessary for their execution. This means that if XSD files belong to the program, they will be located somewhere on its *classpath*. Therefore, we force EvoSuite to scan the program classpath for collecting the available XSD files before starting the search.

During the search, EvoSuite randomly selects which of the available XSD files to use for creating XML content of different types. As before, different XML content might be expected in different parts of the code, hence this approach might result

[1] A term used to differentiate from environment mocking.

in exceptions during the search. Once again, we rely on the ability of the search to try out the various possibilities and eventually find the correct schema to use for each injection site.

As we show in the next section, this technique does not impair the search and lets EVOSUITE generate effective unit tests. Moreover, by employing this technique we can deal with two counter-intuitive cases that are common with XSD files, which improves the robustness of our solution. On one hand, not all the XSD files correspond to concrete types, therefore, such XSD files cannot be used to generate any valid XML content. During the search, EVOSUITE must, and eventually does, get rid of them. On the other hand, some XSD files contain multiple top-level elements, hence, the same XSD files can be used to generate XML content of different types. During the search, EVOSUITE eventually figures out which type can be used where.

IV. CASE STUDY

We conducted a preliminary evaluation by running our extension of EVOSUITE with XMLMATE on the motivating example with the aim of assessing how beneficial the proposed approach is.

We compared our prototype (in which EVOSUITE uses XMLMATE with a probability of 0.5) to the following EVOSUITE configurations:

- *Basic*, in which EVOSUITE does not employ any of the available techniques for mocking;
- *Environmental Mock*, in which EVOSUITE mocks the interactions with the network and the file system;
- *Functional Mock*, in which EVOSUITE mocks objects with a probability of 0.5.

Since the purpose of this evaluation is mostly illustrative, i.e. we are not performing a thorough empirical analysis on the subject, we conducted a limited number of repetitions (i.e. 5 repetitions) using a low budget (i.e. 30 seconds).

We report the averaged results of the experiments in Table I where the labels D, R, and C, identify the results that the various configurations of EvoSuite achieved for the classes `Dto`, `Retriever`, and `Checker` respectively. We report the number of tests generated (*# Tests*)[2] as well as the code coverage achieved by the generated test suite (*Coverage*).

By extending EVOSUITE with XMLMATE, we achieved almost full coverage (above 95% on average) on all the classes in the motivating example by means of a test suite of reasonable size (ca. 16 tests in total). Compared to the other techniques, we achieved the same code coverage on `Dto`, but higher code coverage on classes `Retriever` and `Checker`. It is worth noting that we did so with only a few additional tests.

V. CONCLUSIONS

In this paper, we have presented a novel technique that enhances search-based unit test generation with the ability to generate realistic environmental test data.

[2]EVOSUITE performs test suite minimization to remove redundant tests.

Table I
COMPARISON OF RESULTS ACHIEVED BY EVOSUITE IN DIFFERENT CONFIGURATIONS

Configuration	Coverage			# Tests		
	D	R	C	D	R	C
Basic	100%	39%	33%	6	1	3
Environment Mock	100%	88%	69%	6	2	4
Functional Mock	100%	39%	100%	6	1	6
XMLMATE	100%	94%	100%	6	3	7

The table reports the average results for the classes `Dto` (col. D), `Retriever` (col. R), and `Checker` (col. C)

As a proof of concept, we extended the EVOSUITE unit test generation tool to use XMLMATE and focused on the generation of string data in XML format, which is a typical case when systems interact with external Web Services.

Our experiments on an illustrative set of three artificial, albeit non-trivial, classes show that our technique outperforms the current state-of-the-art techniques in automatic unit test generation. Not only higher code coverage is achieved, but also no false-positive tests are created. Our prototype of EVOSUITE and XMLMATE, the motivating example, and the instructions for reproducing our preliminary study, are publicly available for download at:

https://www.st.cs.uni-saarland.de/testing/ast2017/

ACKNOWLEDGMENTS.

This work was supported by the ERC Advanced Grant SPECMATE, and by the National Research Fund, Luxembourg (FNR/P10/03).

REFERENCES

[1] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," in *Proceedings of the 2014 IEEE 25th International Symposium on Software Reliability Engineering*, ser. ISSRE '14, 2014, pp. 201–211.

[2] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes, "Mock Roles, Not Objects," in *Proceedings of the International Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '04, 2004, pp. 236–246.

[3] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276–291, 2013.

[4] A. Arcuri, G. Fraser, and J. P. Galeotti, "Generating TCP/UDP network data for automated unit test generation," in *10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 155–165.

[5] ——, "Automated Unit Test Generation for Classes with Environment Dependencies," in *IEEE/ACM Int. Conference on Automated Software Engineering (ASE)*. New York, NY, USA: ACM, 2014, pp. 79–90.

[6] A. Arcuri, G. Fraser, and R. Just, "Private API Access and Functional Mocking in Automated Unit Test Generation," in *Proceedings of the International Conference on Software Testing, Verification and Validation*, ser. ICST '17, 2017 to appear.

[7] N. Havrikov, M. Höschele, J. P. Galeotti, and A. Zeller, "XMLMate: Evolutionary XML Test Generation," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. ACM, 2014, pp. 719–722.

[8] M. Fowler, "The data transfert object pattern," https://martinfowler.com/eaaCatalog/dataTransferObject.html.