Extending the Bacterio tool for web application mutation testing

Macario Polo\textsuperscript{1}, Danilo Caivano\textsuperscript{2}, Pedro Reales\textsuperscript{1}

\textsuperscript{1}Institute of Information Systems and Technologies, U. of Castilla-La Mancha, Spain
macario.polo@uclm.es, pedro.reales@uclm.es
\textsuperscript{2}Dipartimento di Informatica, U. degli Studi di Bari Aldo Moro, Bari, Italy
caivano@di.uniba.it

Abstract. Mutation is a testing technique that, after many years of application in the academic and research environments, has recently started to be applied in industry. The main obstacle for its industrial adoption has been the high costs associated to its three stages: mutant generation, execution of tests cases against mutants and result analysis. In the same way, the techniques that researchers have developed to alleviate these costs are the main reason for its acceptance. In spite of this, the application of mutation is reduced to the testing of the internal layers of systems, and not of the external ones, such as the GUI. Since current trends in software construction mainly involve the development of web and mobile applications, we have extended the Bacterio tool for web application testing using mutation. This paper deals with the integration of the mutant schema technique in Bacterio as a way to efficiently execute mutation testing of web applications. Moreover, a new component has been included to control the execution of the test cases within the web server.

Keywords. Mutation testing, web application testing.

1 Introduction

The V-model for testing [1] describes the several testing levels that may correspond to the different development stages of a software system. Thus, integration testing is carried out after having checked the behavior of individual modules with unit testing techniques. After integration testing, system testing checks the behavior of the system with its environment, what includes its relationships with other systems, functional testing with its users and the testing of non-functional requirements such as performance or security. In this sense, testing of web applications generally consists in two types of tests:

1. Functional testing, which consists in the execution of predefined scenarios on the browser user interface. Often, these scenarios are recorded and later re-executed by a suitable tool, such as Selenium (http://www.seleniumhq.org/). In this kind of tests and tools, the test case’s pass or fail verdict generally depends on the presence or
2. Performance testing, which consists in the simulation of a number of concurrent users which execute different test scenarios against the server system. Tools such as OpenSTA (http://opensta.org/) or JMeter (http://jmeter.apache.org/). In this kind of testing, the test case verdict usually depends on the service time of each request: mean time, quartiles, minimum, maximum, etc.

In this paper we propose a framework to apply mutation to the testing of web applications. Mutation is known to be a very effective but very costly testing technique [2], mainly due to the number of mutants generated, what strongly influences on the required execution time. This problem is even more serious in web applications, where the web application server is a container that, at first glance, should load and unload each mutant. After a short revision of the most important concepts on mutation, the paper describes the proposed process to apply it to web application testing, as well as the tool support. Then, we list some related works and draw our conclusions and future lines of work.

2 Background

Mutation is a testing technique, originally proposed in 1978 by DeMillo et al. [3], which relies on the discovery of the artificial faults which are seeded in the system under test (SUT). These faults are injected in the SUT by means of a set of mutation operators, whose purpose is to imitate the faults that a common programmer may commit. Thus, each mutant is a copy of the program under test, but with a syntactic small change in its code. For most mutants, it is possible to find a test case that leads the mutant and the SUT to exhibit different behaviors: this means that the mutant contains, with respect to the SUT, an actual fault that has been discovered by the test suite. Other times, it is impossible to write a test case to distinguish the behaviors between the SUT and mutant: these types of changes are really optimizations or de-optimizations of the original code and are called “equivalent mutants”. Table 1 shows a very simple function and four of its possible mutants: in mutants 1, 2 and 3, the arithmetic operator “+” has been substituted; in the 4th, a post-increment operator has been added at the end of the statement.

<table>
<thead>
<tr>
<th>Original</th>
<th>Mutant 1</th>
<th>Mutant 2</th>
<th>Mutant 3</th>
<th>Mutant 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{int sum(int a, int b) { return a + b; } )</td>
<td>( \text{int sum(int a, int b) { return a - b; } )</td>
<td>( \text{int sum(int a, int b) { return a * b; } )</td>
<td>( \text{int sum(int a, int b) { return a / b; } )</td>
<td>( \text{int sum(int a, int b) { return a + b++; } )</td>
</tr>
</tbody>
</table>

Table 2 shows the data of four possible cases to test the \( \text{sum(a, b)} \) function: each cell saves the result of executing the function version in the row with the test data in the column. Shadowed cells represent returned values which are different in the mutants and in the original: this means that the fault inserted in the mutant has been discovered by the corresponding test case. Blank cells represent test cases that have not been capable of discovering the fault inserted. According to mutation’s terminology,
mutants 1, 2 and 3 are “killed” because their faults have been discovered at least once; in the same way, mutant 4 is “alive” and, moreover, is “equivalent” because the change introduced will never be distinguishable. The test suite quality is evaluated with the mutation score (Fig. 1). A test suite that kills all the non-equivalent mutants is “mutation-adequate”. So, the test suite in Table 2 is mutation-adequate because it kills the three non-equivalent mutants.

\[
MS(P,T) = \frac{K}{M - E}, \text{ where:} \\
P: \text{program under test} \\
T: \text{test suite} \\
K: \text{number of killed mutants} \\
M: \text{number of generated mutants} \\
E: \text{number of equivalent mutants}
\]

![Fig. 1. Mutation score](image)

**Table 2. Results of executing four test cases against the mutants in Table 1**

<table>
<thead>
<tr>
<th>Test data</th>
<th>(1, 1)</th>
<th>(0, 0)</th>
<th>(-1, 0)</th>
<th>(-1, 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
</tr>
<tr>
<td>Mutant 1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>Mutant 2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Mutant 3</td>
<td>1</td>
<td>Error</td>
<td>Error</td>
<td>1</td>
</tr>
<tr>
<td>Mutant 4</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
</tr>
</tbody>
</table>

Mutants are usually obtained by automated tools that apply a set of mutation operators to the statements of the original program. Each operator is specialized in introducing a determined type of change. Two operators are used in the example of Table 1: AOR (Arithmetic Operator Replacement) and UOI (Unary Operator Insertion), although there are many more ([4], [5]). Since almost every program statement may be mutated by at least one operator, the number of mutants that can be obtained for a medium-size program may be very big. And, since test cases must be executed against the mutants, the execution time may be very high. However, current mutation tools [6][7] include different cost-reduction techniques [2] (application of a subset with the best mutation operators; mutant schemata [8], [9]; high order mutation [10], [11]; parallel execution [12]; flexible weak mutation [13]) that are making possible the progressive adoption of mutation in industry [14].

Traditionally, mutation has been only used by researchers as a way of evaluating the quality of test case generation techniques. Moreover, they restricted the application scope almost to unit testing, probably due to the lack of suitable tools for higher levels. A real application consists of a set of modules (for example, classes), libraries resources (images, configuration files), etc. So, a really usable mutation testing tool requires the ability of dealing with not simple applications. Suppose an application composed by a set of libraries, resources and classes \( A, B, \ldots \ Z \). A mutation tool will generate a set of mutants for each class: this is illustrated on the left side of Fig. 2, which shows \( n \) mutants for \( A \), \( m \) for \( B \), \ldots and \( p \) mutants for the \( Z \) class. Since the test cases cannot be launched against the isolated mutants (this is, with independence of the remaining application elements, libraries, etc.), we require a second step that compose or links each mutant with the remaining elements. If we have, for example,
1000 mutants for all the classes and want to make a $1^{st}$-order mutation analysis (this is, one change per mutant), we will need to mount 1000 versions. Fig. 2 illustrates the process of mounting versions of the SUT with two faults per versions.

![Diagram](image-url)

**Fig. 2. Mounting of a mutated version of an actual application**

When the set of mutated versions is available, the test cases are executed against them. There exist different execution algorithms, two of which are illustrated in Fig. 3, that represents a system with 7 mutated versions and 6 test cases.

- In the left side, all the test cases are executed against all the mutated versions. So, there maybe versions that are “killed more than once”, such as $mv2$, which is killed by $tc1$, $tc2$ and $tc5$. With an algorithm such as this, there are $7 \times 6 = 42$ executions. Algorithms like this are useful to do further analysis of test case effectiveness or test suite reduction [15]: for example, $\{tc2, tc3, tc4\}$ gets the same mutation score than the original suite.

- In the right side, the algorithm executes test cases only against the mutated versions remaining alive. So, since $mv1$ is killed by $tc1$, no other test case is executed against $mv1$. This algorithm significantly reduces the required execution time.

<table>
<thead>
<tr>
<th>tc1</th>
<th>tc2</th>
<th>tc3</th>
<th>tc4</th>
<th>tc5</th>
<th>tc6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mv1$</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv2$</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv3$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv4$</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv5$</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv6$</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv7$</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>tc1</th>
<th>tc2</th>
<th>tc3</th>
<th>tc4</th>
<th>tc5</th>
<th>tc6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$mv1$</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv2$</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv3$</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv4$</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>$mv5$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>$mv6$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$mv7$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fig. 3. Two different ways of executing test cases**
Fig. 4 shows two different implementations of an execution algorithm to get the results as in the right side of Fig. 3:

- The algorithm in the left side loads a mutated version and starts executing test cases against it until the version is killed or there are no more test cases in the suite.
- The algorithm in the right side takes a test case and executes it against all the alive mutated versions.

<table>
<thead>
<tr>
<th>Let be:</th>
<th>Let be:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MV) the set of mutated versions</td>
<td>(MV) the set of mutated versions</td>
</tr>
<tr>
<td>TS the test suite</td>
<td>TS the test suite</td>
</tr>
<tr>
<td>for each (mv) in (MV)</td>
<td>for each (tc) in (TS)</td>
</tr>
<tr>
<td>(tc = TS.firstElement())</td>
<td>(mv = MV.firstElement())</td>
</tr>
<tr>
<td>while (mv.isAlive()) and (TS.hasMoreElements())</td>
<td>for each (mv) in (MV)</td>
</tr>
<tr>
<td>(tc = TS.nextElement())</td>
<td>if (mv.isAlive())</td>
</tr>
<tr>
<td>execute (tc) against (mv)</td>
<td>execute (tc) against (mv)</td>
</tr>
<tr>
<td>end</td>
<td>end</td>
</tr>
<tr>
<td>next</td>
<td>next</td>
</tr>
</tbody>
</table>

Fig. 4. Two test case execution algorithms

Since loading a mutated version is a relatively costly operation, in general is more efficient the algorithm in the left side of the figure. In the context of web applications, the web container imposes a series of additional limitations: the process of loading and unloading a web application is really costly. So, in order to extend mutation testing to this kind of systems requires additional efforts.

3 A process for mutation testing of web applications

This section describes the process for applying mutation to the testing of web applications. Firstly, we give a very brief overview of web application architecture and present a simple application example. Then we present the technique of Mutant Schema, which is specially suitable for this kind of mutation testing.

3.1 Architecture of web applications

A web application is a client-server system that uses http as the basic protocol to communicate both sides. The first web clients were mere renderizers of the HTML code received from the server (left side of Fig. 5): all the business logic remained in the server and, when a request arrived, the server processed it and composed the view in the form of HTML code which was returned back to the client.

The advance of web technologies allows to move part of such logic to the client (right side): with a design like this, the client holds a reduced representation of the server’s business layer. Whilst the server side may be implemented in a variety of programming languages, the client side is usually composed of three kinds of elements:
- HTML to specify the data.
- CSS to specify the visual aspect.
- JavaScript to define the behavior.

In modern browsers, HTML5 is tightly integrated with JavaScript: each HTML5 tag has a corresponding JavaScript object. So, the structure of the client side is specified with HTML and with an object model in JavaScript. JavaScript introduces new elements to program the client, such as client-side storages, websockets or webworkers.

In spite of the enriched experiences that modern web applications give to the client side, most important operations require to send a message to the server, what is made through asynchronous requests that the client side manages as XMLHttpRequest but that, in the server side, are managed as common http requests.

![Diagram of two web architectural styles](image)

**Fig. 5.** Two web architectural styles (taken from [16])

So, the server side continues having the most important responsibilities, and it is here where we focus our testing effort: mutants will be generated for the code in the server side, although the test cases will be executed from the client.

### 3.2 A small web application as proof of concept

In order to exemplify, we present the small web system\(^1\) depicted in **Fig. 6**. It consists of a simple web page in the client and a Converter class in the server. Through the web page, the user sends measure conversion requests between different units, which are processed in the server.

As it is seen, Converter only offers the client two public operations (execute and convert), which forward the request to some of the existing private operations. As it happens with other types of applications, functional tests may not be enough to run over all the program code, being interesting to make some kind of coverage analysis. For this we propose to mix functional testing and mutation analysis, counting the mutation score reached on the server by the test cases executed in the client.

---

\(^1\) Available at [http://alarcosj.esi.uclm.es/ConversionMedidas/](http://alarcosj.esi.uclm.es/ConversionMedidas/)
3.3 Mutant schema

A very effective technique for reducing the costs of test case execution is Mutant Schema [8], which is based on program schema [17]. This technique allows to compose some different programs in the same source code. Thus, only one compilation is required to obtain the executable of each program. Since all programs are included in a single file, it is necessary to include a configuration parameter to determine which program included in the schema must be executed. Fig. 7 shows an example of two programs (add and sub) composed into a single schema. Note the presence of the operator parameter, which is used to determine which of the two original programs must be executed.

```
Programs

int add(int a, int b){
    return a+b;
}

int sub(int a, int b){
    return a-b;
}

Program schema

int addsub (int operator, int a, int b){
    if (operator == 0){
        return a+b;
    } else if (operator == 1){
        return a-b;
    }
}
```

Fig. 7. Two programs mixed in a program schema

The same idea may be applied to both the program and the two mutants that appear in Fig. 8: the schema groups the two mutants produced by the application of the AOR operator to line 5.

```
Original

1 class ClassA{
2     public int foo (int a, int b){
3         for(b<10){
4             a++;;
5             b = b+2;
6         }
7         return a;
8     }

Mutant schema

1 class ClassA{
2     public int foo(int a, int b){
3         for(b<10){
4             a++;;
5         }
6         if (exec(m1)){
7             b = b-2;
8         }
9     }
```
Fig. 8. An original program and two mutants, combined in a mutant schema

With mutant schema, the web server only requires to load once the application under test. An additional component guides the execution, providing a value to the configuration parameter used to decide the version to be executed.

In a previous work [2], we outlined the structure of the generic mutation testing process depicted in Fig. 9: taking into account that testing is a destructive task that tries to find so many errors as possible in the system under test, the tester firstly executes the test suite $T$ against the original program ($P$). If she finds errors, the development team fixes them and returns the new version to the tester, who again launches the test cases. When no errors are found, the tester generates mutants with a tool: now, the idea is to find in the program areas that have remained unexplored by the test cases in the suite. So, she executes the test suite against the mutants: if all of the them were killed, this means that the test suite is a very good test suite (it finds all the faults) and, moreover, the program under test is almost correct (such a good test suite does not find any fault on it). If there are alive mutants, the test suite must be enriched in order to kill them: then, it is possible to find errors in the areas now explored. In this case, such errors should be reported, fixed and a new testing cycle would start.

3.4 Mutant generation

Bacterio [6] is a tool that gives a complete support to mutation testing. It includes many cost-reduction techniques, both for mutant generation as for test case execution, including mutant schema. This tool is intended for testing desktop Java applications. The mutants that Bacterio produces are copies of the original program with: (1) one or more artificial faults inserted (depending on the tester preferences) and (2) additional code that saves the state of the objects involved in the executed scenario. For this, the object state is described in terms of the object fields values.
Fig. 9. Structure of the mutation testing process

For the measure conversion system, the application of the mutation operators AOR (Arithmetic Operator Replacement), ROR (Relational Operator Replacement), ABS (Absolute Value insertion), UOI (Unary Operator Insertion) and LCR (Logical Connector Replacement), which are the most powerful operators [4], produces 461 mutants. By means of Mutant schema, all of them are grouped in the same code, although the tools is capable of showing the location of the introduced change (Fig. 10).

Fig. 10. A mutant (right) generated by Bacterio for the source code in the left
3.5 Test case design

As shown in Fig. 9, the test suite is, together to the mutant set, the second important element for mutation testing. The first goal of the test suite is to find errors in the SUT. Only when no errors are found, it is executed against the mutants to discover the artificial faults: we can have an evaluation of the test suite quality by calculating the mutation score. If it is under the desired threshold, more test cases must be added to the suite to increase the mutation score.

Since the test execution step will be an automatic task, test cases must be provided as an automated script. For this, we save Selenium scripts which are launched against the original web application. A Selenium script is a composed by a series of commands, each with up to three parts: command, name locator pattern. Commands may be actions (such as opening a URL, writing a text or pressing a button), accessors (read and save variables related to the system state) and assertions (that make some kind of checking and which are used to write the oracle test case). Along the user navigation session, Selenium evolves saving the actions performed by the tester on the browser window (Fig. 11).

Since Bacterio deals with JUnit-style test cases, we use the export functionality of Selenium to save the test cases in JUnit format (Fig. 12).

![Selenium test script within the Selenium environment](image)

3.6 Test case execution

Once the mutants and the test suite are available, the next step is to execute the test cases against them. For executing a web application, when the server is started, it
loads and deploys the applications it holds installed in its public directory. Then, it is ready to receive and process requests from the clients.

Going back to the execution model seen on the right side of Fig. 3, the server must load the web application, receive the test cases (which actually implement http requests) and register the results for the further analysis.

```java
public class TestSelenium {
    private Selenium selenium;
    @Before
    public void setUp() throws Exception {
        selenium = new DefaultSelenium("localhost", 4444, "*chrome",
                                         "http://localhost:8000/");
        selenium.start();
    }

    @Test
    public void test1() throws Exception {
        selenium.open("/ConversionMedidasWeb/index.jsp");
        selenium.type("name=unidadesOrigen", "-300");
        selenium.click("css=input[type="submit"]");
        selenium.waitForPageToLoad("30000");
        assertTrue(selenium.isTextPresent("There was"));
    }

    @Test
    public void test2() throws Exception {
        selenium.open("/ConversionMedidasWeb/index.jsp");
        selenium.type("name=unidadesOrigen", "0");
        selenium.click("css=input[type="submit"]");
        selenium.waitForPageToLoad("30000");
        assertEquals("0.0", selenium.getValue("name=unidadesDestino"));
    }
    ...
}
```

**Fig. 12.** Translation into JavaScript of the Selenium script in Fig. 11

Bacterio has been extended with a new component that controls the execution process. Since all class mutants are contained in a single file in the form of a program schema, Bacterio is in charge of telling the web server which mutated version must be executed (Fig. 13).

Thus, Bacterio launches the web server (currently it works with Apache Tomcat) and iterates on each mutated version, progressively launching the test suite against them. The progress bar in the right side of Fig. 14 advances according to the mutated version currently being executed.

The test suite under execution is a set of JUnit test cases which proceed from a Selenium script, such as that in Fig. 12. During the execution of the test cases, the instrumented code of the application under test leaves a trace that will be later used by Bacterio to show the results. These ones consist in a matrix showing which mutated versions each mutant kills (Fig. 15).
Fig. 13. Bacterio controls the execution of test cases and mutated versions

![Image](image13.png)

**Fig. 14.** A moment of the execution

<table>
<thead>
<tr>
<th>Killing matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
</tr>
<tr>
<td>version109</td>
</tr>
<tr>
<td>version108</td>
</tr>
<tr>
<td>version107</td>
</tr>
<tr>
<td>version106</td>
</tr>
<tr>
<td>version11</td>
</tr>
</tbody>
</table>

**Fig. 15.** A fragment of the killing matrix obtained for the example

As in traditional mutation, the tester must complete the test suite with new test cases until the desired mutation score threshold is reached.
4 Related work

As we have said, mutation has been widely applied in research, mainly for validating the goodness of test case generation strategies. Studies about its application to big-sized software are very recent and are mainly focused on the testing of the internal layers of the application. So, the number of studies relating mutation with web application testing is very low.

In this kind of applications, mutation can be applied either to the client or to the server side. The very few existing studies deal with mutation in the client side: Praphamontripong and Offutt [18] propose mutation operators for modifying the HTML and the JSP code of Java web applications. In the same line, Nishiura et al. [19] describe mutation operators for JavaScript. Applying mutation in the client side is obviously interesting for detecting faults in the user interface; however, if the browser sends requests that may modify the state of the objects in the web server, then mutation must be applied at the server side: otherwise, errors on the server may pass unattended, even though this one is the most critical side of this kind of client-server applications.

To our best knowledge, this is the first work dealing with mutation testing on the server side of web applications.

5 Conclusions and future work

This paper has presented a novel approach for mutation testing of web applications. The idea consists in generating mutants on the server side of the application and exercising the mutated versions with test cases generated for the client side. Thus, the business layer of the system, whose core stays in the server, is tested with test cases written or generated for the browser, which is the tool that users employ to interact.

Up to now, the approach has been implemented in Bacterio, a general-purpose mutation testing tool that applies mutant schemata to reduce the execution cost and that interacts with the web server to decide the mutated version to be executed. The test cases proceed from Selenium scripts and, thus, they are functional test cases that check the fulfillment of the different application functionalities. Since performance testing is also important for web applications, we are currently working on the inclusion of JMeter test scripts for extending the application scope.

6 References


