From Formal Models to JUnit Tests: A Proof Of the Concept Tool

[1] *Maxim Gromov <maxim.leo.gromov@gmail.com>*
[1] *Natalia Shabaldina <nataliamailbox@mail.ru>*
[1] *Tomsk State University,*
634050, Russia, Tomsk, Lenin av., 36.

Abstract. In this paper, we present a tool which automates conversion of a test suit into a JUnit test class. We do some experiments with the tool in order to see if the concept of considering a Java class as a finite state machine (FSM) or an extended finite state machine (EFDM) has any worth when talking about testing.

Keywords: Finite State Machine, JUnit, Testing a Java Class, Proof of the Concept Tool

1. Introduction

This paper is devoted to testing of Java class. The idea of using formal models in test generation is not new (see, for example, [1–3]) since it is well-known that only in this case we can guarantee the fault coverage. When we have the description of the program (class) as a Finite State Machine [4] we can derive a complete test suit (w.r.t. the fault model) using one of the well-known methods: W-, HSI- or H-method [5]. These methods were implemented in our scientific group as a part of a tool FSMTest-1.0 [6]. If we want to apply the derived test suit to an implementation under test, we need to arrange, somehow, an experiment. And of course, it would be preferable to use some tool, which would automate the experiment. One of the most popular tools for testing Java programs is JUnit [7]. However, in order to use JUnit we need to convert test suit that is derived based on a formal model, into JUnit format. For the best of our knowledge, there is no tool that supports the whole process from modeling Java class to JUnit test, so in this work we present the tool that partly covered this process.

Since the question of formal model extraction from a description of a program is very important, we discuss it here. There are works that are devoted to the so-called "automata-based programming" [8]. In the work [9] the authors introduce an extension of Java that allows to implement Java program based on the given automaton. In the work [10] the authors describe different tools for converting of an FSM into a program, for example, they mentioned a tool UniMod [11] that allows converting a transition diagram into a Java or C++ program. The authors point out some disadvantages of this tool and they work on constructing their own tool for converting a transition diagram into a program. At the first glance, such a tool can be useful for us since if the program is the result of converting from the transition diagram then we have a strict correspondence between the program and the model and it's easy to derive a test for the given model. However, this strict correspondence is not really good for testing, since in this case we are testing not the implementation but the tool that converts the diagram into the program (whether this tool provides a good converting or not). In our case, in order to test a conformance relation, we suppose that there is a specification written by somebody (as a formal model) and there is a program that was developed by another programmer or engineer. So at this step of our investigations we do not have any tools for extraction of formal models; actually we extract a model from a given specification by hands.

Due to the space limitation, we skip the part of preliminaries, and suggest the reader to see the works [4, 5], where the definitions of an FSM, a fault model and a complete test suit are given.

The rest of the paper is organized as follows. In Section 2 we describe our tool. Section 3 contains the description of our experiments with a Java class that implements an SMTP client [12]. Section 4 concludes the paper.

2. A Proof Of The Concept Tool

We implemented a converter from an FSMTest-1.0 test suit to a JUnit test class as an Eclipse-Kepler [13] plug-in. The implementation is quite straightforward and simple, but still allows to do real testing.

To describe the work of the plug-in, let suppose we would like to test Java class named `MyClass`

```
class MyClass {
  void foo1(int a) {…}
  double foo2() {…}
  PrintStream foo3(int a, String b) {…}
}
```

When engaged, the procedure of JUnit test build creates a JUnit class which contains predefined fields:

```
<T> SUT;
byte byteResult;
char charResult;
double doubleResult;
float floatResult;
int intResult;
long longResult;
short shortResult;
```

```
boolean booleanResult;
String StringResult;
Object ObjectResult;
```

The type `T` of the field `SUT` is the class under test (SUT – system under test) and for our example it equals `MyClass`. This field is used to call methods defined by a test case. The JUnit class will create a new exemplar of the class `T` before start of every test case and put the link to it in `SUT`. When the test case is over plug-in will assign to **`null`** this field.

Other fields, as one could deduce from their names, are used to collect results of the called methods. For example, for our case, a result of a call of the method `foo1` won't be collected, since the return type is **`void`**, a result of a call of the method `foo2` will be collected to the variable `doubleResult` and a result of a call of the method `foo3` will be collected to the variable `ObjectResult`. Collected results can be checked then.

To engage the procedure of JUnit test build a user should select a class he or she wants to test in the Eclipse project explorer and press right button of the mouse. In the drop-down menu there is the menu item called "FSM Test to JUnit" using which the user starts the procedure. When started, the procedure at the first stage opens a dialog window, where the user should specify a name of a JUnit class which is being built and four files: a file with description of input stimuli, a file with description of checks for output reactions, a file defining the class (field `SUT`) initialization and extra parameters and a file with a test suit. All files are text files.

The file with the test suit describes a set of test cases and each test case is a sequence of input-output pairs. Test cases are arranged by lines: the first line of the file is the first test case, the second line – the second test case etc. Pairs in the test cases are separated by spaces and are given in the form `inp/outp`, where `inp` is an index (non-negative integer number) of an input stimulus and `outp` is an index of an expected reaction. For our example a file with a test suit may look like as follows:

```
0/1 1/2
1/2 2/0 3/1
```

This test suit example contains two test cases: the first one contains two input-output pairs, the second test case – three pairs.

The file with stimuli description is also arranged by lines. Each line corresponds to certain stimulus. Indexing comes from top to bottom, i.e. the line number 1 describes the stimulus with index 0, the line number 2 describes the stimulus with index 1 etc. Descriptions of the stimuli are given in the form

```
return_type method(parameters)
```

or

```
return_type method(parameters); extra_code
```

Here `return_type` is either one of the basic types (**`byte`**, **`char`**, etc.) or `String`, or `Object`. Depending on `return_type`, the plug-in will use corresponding field to collect a result: for **`byte`** it will use `byteResult`, for **`char`** – `charResult` etc. To make the method call the plug-in just concatenates string "`return_typeResult = SUT.`" with selected line except return type and adding "`;`" at the end. For example, if we have the following input stimuli description

```
void foo1(-5)
double foo2(); doubleResult -= 800
Object foo3(8, "Hello")
Object foo3(-90, nullString)
```

Then input stimulus 0 will appear as

```
SUT.foo1(-5);
```

input stimulus 1 will appear as

```
doubleResult = SUT.foo2(); doubleResult -= 800;
```

and so on. Each such group is surrounded with **`try-catch`** directives since a result of a call can be an exception and this exception may be or may be not expected, therefore it should be checked.

Each line of the reactions' check file is also corresponds to certain index of an output reaction (the first line corresponds to the index 0 etc.) and describes a check of a result. Two first lines are predefined: the first line (the reaction with the index 0) should be **`true`**, the second line (the reaction with the index 1) must be `Exception`. The reaction with the index 0 means, that no check is done and any reaction, but exception is suitable. The reaction with the index 1 means, that the only suitable reaction for a method call is an exception. The rest lines should contain Java valid Boolean expressions (one expression per line). These expressions may use any predefined (like `byteResult` etc.) or defined by the user (see below) fields, or special group of symbols: `@r@`. This group of symbols will be substituted with the certain result-field depending on the type, defined for the method call. The expressions are then used in *`assertTrue`* JUnit call. For example, we have the following lines:

```
true
Exception
@r@ > 0
@r@ != null && @r@ instanceof PrintStream
```

Then input-output pair `1/2` will bring us the following source code

```
try {
  doubleResult = SUT.foo2(); doubleResult -= 800;
  assertTrue("message", doubleResult > 0);
} catch (Exception e) {
  assertTrue("Exception" + e.getMessage(), false);
}
```

And so on for every input-output pair of a test case. Of course, if an expected result is an exception, then catch block will contain nothing, and try block will contain `assertTrue("No exception but should be", false)`.

And the last file to prepare – file with initialization code. This file contains three blocks. The blocks are separated by the group of symbols `#!`. The blocks may be empty. No block may contain group of symbols `#!` since this will break right split of the file into blocks. The last requirement can be seen as a drawback. Each block should be a valid Java source code and will be placed at certain part of the resulting JUnit class source code.

The first block may contain required imports and will be placed at import part of the resulting source code. The second block may contain definitions of required fields and will be placed bellow the predefined definitions. The last block may contain an initialization block, where the user must define `SUT` field and may define other fields. If the last block is empty, then the plug-in will define `SUT` by calling the default constructor. The initialization block is placed within **`try-catch`** directives.

3. The Experiments With an SMTP Client Java Class

3.1 The First Round Of the Experiments

To check how our plug-in copes with the real life Java classes we decided to hold some experiments. We downloaded the source code for `sun.net.smtp.SmtpClient` Java class from [12]. Obviously, this class implements an SMTP [14] client.

The FSM behavior of the class, as we see it, is shown in Fig. 1. It can be noticed that the given FSM does not allow several transactions within one connection to a server. It is so because of the implementation: the finishing of a message to send is concatenated with the closing the connection to the server.

The input stimuli are given by the following lines (mind hyphenations):

```
void openServer("goodserver.org", 25);
        SUT.issueCommand("helo" +
        InetAddress.getLocalHost().getHostName() +
        "\r\n", 250)
```

```
void openServer("nonexistingserver.org", 25)
void from("gooduser@mail.com")
void from("abc%")
void to("gooduser@mail.com")
void to("abc%")
Object startMessage();
       ((PrintStream) ObjectResult).print("Hello")
Object startMessage();
       ((PrintStream) ObjectResult).print(nullString)
void closeServer()
```

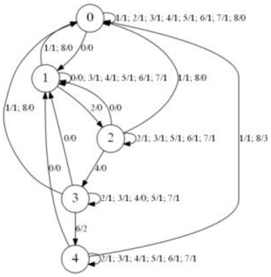

Fig. 1. *The FSM description of the class* `sun.net.smtp.SmtpClient`

The expected output reactions are checked by the following expressions:

```
true
Exception
@r@ != null && (@r@ instanceof PrintStream)
true
```

One may notice that the expression with the index 3 is the same (**`true`**) as the default expression with the index 0. This is done to keep the state 4 non-equivalent to the state 0. The logics here is that at the state 4 `closeServer` method does the actual finishing of the message to send with closing the connection to the server, whilst at any other state this method just closes the connection and no message is send.

Using the FSM we generated a test suit by HSI method [5] using the tool FSMTest-1.0 [6]. This test contains 105 test cases and the total length of the test suit (together with the resets) is 536. The plug-in successfully wrapped the test suit into a JUnit test. The run of this JUnit test showed one issue. It turned out, that the SMTP client implementation does not keep the state of a transaction, and it just sends commands to a server checking the response. The client throws an exception only if the server responses with an error. All in all it is OK, until it comes to the command `DATA`. After the command any other command send by the client (let say "`MAIL FROM:`" issued by the method `from`) is considered by the server as just a text. And the server waits when the message will be finished (by the dot in the empty line) while the client waits a response from the server blocking a thread it works within. If there were no timeout, set by the server, it would be definitely a deadlock. Luckily, in our case the timeout was set (6 minutes). Timeout expiration is an error and the client throws an exception. However, here comes the problem. Timeout expiration makes the server to close the connection and no other communication is possible. If the client kept the state of a transaction and formed exceptions not only according to the server's responses but also according to the kept state, it could overcome the described issue.

3.2 The Second Round Of the Experiments

Keeping in the mind the issue described in the Section 3.1, we have revised the FSM (Fig. 2). Now it contains less states.

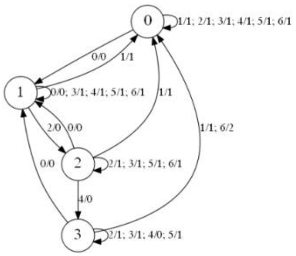

Fig. 2. Revision of the FSM

We have revised inputs and outputs as well. Inputs are as follows:

```
void openServer("goodserver.org", 25);
        SUT.issueCommand("helo" +
        InetAddress.getLocalHost().getHostName() +
        "\r\n", 250)
void openServer("nonexistingserver.org", 25)
void from("gooduser@mail.com")
void from("abc%")
void to("gooduser@mail.com")
void to("abc%")
Object startMessage();
        ((PrintStream) ObjectResult).print("Hello");
        SUT.closeServer()
```

and outputs are as follows:

```
true
Exception
```

```
@r@ != null && (@r@ instanceof PrintStream)
```

A test suit again was built with HSI method implementation from the tool FSMTest-1.0. This test suit contains 51 test cases and the total length is 240.

The test suit was successfully wrapped into a JUnit class and was run with no problems.

4. Conclusions

In this paper we presented a tool that helps to organize model-based testing of Java class using JUnit. We extract an FSM from a description of a system (for example, an RFC) by hands and it seems that it is impossible to do it automatically. Then we use the tool FSMTest-1.0 to construct complete test suit (we have chosen HSI method). And then comes the time of the presented tool: it converts FSMTest-1.0 test suit into a JUnit test class.

We took an SMTP-client Java class from [12] in order to see if the tool in particular and the concept in general are any good. Our experiments with this Java class were successful; moreover, we found a weak place in the implementation.

5. Acknowledgements

This work is partially supported by the grant of the Russian Scientific Fund No. 16-49-03012.

The authors also would like to thank Tatiana Bondareva for the help in searching of a Java class for the experiments.

References

[1]. Ermakov A.D., Yevtushenko N.V. [Deriving Test Suites with the Guaranteed Fault Coverage for Extended Finite State Machines], Metod sinteza testov s garantirovannoy polnotoy po modeli rasshirennogo avtomata. [Modeling and Analysis of Information Systems], Modelirovaniye i analiz informatsionnyh system, 2016, vol. 23, No. 6, pp. 729-740 (in Russian).

[2]. Kushik N., Kolomeez A., Cavalli A., Yevtushenko N. Extended Finite State Machine Based Test Derivation Strategies for Telecommunication Protocols. Proceedings of SYRCOSE, 2014, pp. 108- 113.

[3]. Kushik N., Forostyanova M., Prokopenko S., Yevtushenko N. Studying the optimal height of the EFSM equivalent for testing telecommunication protocols. Proceedings of CCIT, 2014.

[4]. Gill A. Introduction to the Theory of Finite-state Machines. McGraw-Hill, 207 p., 1962.

[5]. Dorofeeva R., El-Fakih K., Maag S., Cavalli A., Yevtushenko N. FSM-based Conformance Testing Methods: a Survey Annotated With Experimental Evaluation. Information and Software Technology, vol. 52, No. 12, pp. 1286-1297, 2010.

[6]. Shabaldina N., Gromov M. FSMTest-1.0: a Manual For Researches. Proceedings of IEEE East-West Design & Test Symposium (EWDTS'2015). Ukraine, Kharkov: SCITEPRESS, 2015, pp. 216-219.

[7]. JUnit Home Page: JUnit 4, documentation. http://junit.org/junit4/, 04.07.2017.

[8]. Shalyto A. [A Techology of Automata-Based Programming], Tehnologiya avtomatnogo programmirovaniya. http://is.ifmo.ru, 04.07.2017

[9]. Korneev G.A., Shamgunov N.N., Shalyto A.A. [State Machine – Java Language Extantion for Effective Implementation of Finite State Machines], Iazyk State Machine – rasshirenie iazyka Java dlia effektivnoi realizatsii avtomatov. [Information and Control Systems], Informatsionno-upravliaiuschie sistemy. Vol. 1, No. 14, pp. 16-24, 2005.

[10]. Polikarpova N.I., Shalyto A.A. [Automata-based programming], Avtomatnoe programmirovanie. Saint-Petersburg: Piter, 167 p., 2008, in Russian.

[11]. [Unimod-projects], Unimod-proekty. http://is.ifmo.ru/unimod-projects, 04.07.2017.

[12]. http://grepcode.com/search?query=sun.net.smtp.SmtpClient&start=0&entity=type&n=, 04.07.2017.

[13]. Eclipse. https://eclipse.org, 04.07.2017.

[14]. Postel J. Simple Mail Transfer Protocol. Information Sciences Institute, University of Southern California. https://tools.ietf.org/html/rfc821, August, 1982.