Test Suite Selection Based on Traceability Annotations Tool demonstration

Yves Ledru, German Vega, Taha Triki, Lydie du Bousquet
UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble2/CNRS
Laboratoire d'Informatique de Grenoble UMR 5217, F-38041, Grenoble, France
Yves.Ledru@imag.fr, German.Vega@imag.fr, Taha.Triki@imag.fr,
Lydie.du-Bousquet@imag.fr

ABSTRACT

This paper describes the Tobias tool. Tobias is a combinatorial test generator which unfolds a test pattern provided by the test engineer, and performs various combinations and repetitions of test parameters and methods. Tobias is available on-line at `tobias.liglab.fr`. This website features recent improvements of the tool including a new input language, a traceability mechanism, and the definition of various "selectors" which achieve test suite reduction.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Verification

Keywords

Combinatorial testing, test suite reduction, Tobias

1. A SHORT PRESENTATION OF TOBIAS

Tobias is a combinatorial test generator. Combinatorial testing performs combinations of selected input parameter values for given operations and given states. This often leads to combinatorial explosion. Hence a lot of research has been devoted to find reduced test suites which cover all combinations of each subset of n parameters. A collection of tools implementing various combination algorithms can be found at [9]. Usually, combinatorial generators feature test cases with a single operation call and many parameters. Tobias adapts combinatorial testing to the generation of *sequences* of operation calls. This allows reaching states that do not correspond to a single call to a constructor and to define tests in terms of behaviours rather than states. Tobias was used successfully on several case studies [4, 5, 3]. It has inspired tools such as the Overture Combinatorial Testing Plug-in [6], jSynoPSys [2], and TL-CAT [11].

One of the difficulties in the automation of testing is the generation of test cases. The approach of Tobias is to capture the knowledge and the know-how of the test engineer in a test pattern, also called test schema, which is a compact description of the set of test cases. This description is abstract and concise, and identifies sets of similar elements (data values, method names, objects, instructions). Tobias then unfolds the pattern into a possibly large set of test cases that can be translated into executable tests in a given target technology (e.g. Java/Junit4). The first version of the tool was used successfully in several case studies, including a mini banking application [3] and a multi-modal fusion engine [4]. These experiments showed that the tool actually improves the productivity of the test engineer: a large test suite can be produced from a few lines of combinatorial description.

Strengths and weaknesses.

Several case studies show that the systematic character of the Tobias test suites helps finding failures [3] and even more than "manual" test suites which feature the same code coverage [4]. Moreover, the tool helps to produce a large amount of test cases with a small definition effort. Several experiments with Master's students, and with other research teams showed that the tool can easily be learned outside our research team. Moreover, the notion of schema helps to structure the test campaign.

The systematic schema unfolding, which is one of the strengths of the tool, is also one of its main weaknesses because it leads to combinatorial explosion. It is easy to write a test schema that results in a million, or even a billion, test cases. Although the current version of the tool has been designed to generate up to one million test cases. Such large test suites are usually incompatible with standard compilers or test drivers. Moreover, the execution of large test suites may require too much time and untractable resources. Therefore, it is crucial to master combinatorial explosion. In the case studies reported in [7, 4], a careful definition of the Tobias schemas allowed to master combinatorial explosion and keep a reasonable size of the test suites (several thousands test cases). But several techniques have been proposed to further master combinatorial explosion.

Mastering combinatorial explosion.

Tobias proposes filtering and selection mechanisms to reduce the size of the resulting test suite. A *filter* is a property expressed by the test engineer that must be fulfilled by the

ASE '12 Essen, Germany

test cases of the test suite. Such a filter should be expressed as a boolean function over the text of the test case, or over its syntax tree. Filters provide an easy way to master the size of a test suite. But, it is the responsibility of the test engineer to design a Boolean function which reduces the size of the test suite while still keeping interesting test cases.

Tobias *selectors* provide a second way to reduce the size of a test suite. While a filter takes a single test case as argument, a selector takes the whole test suite as argument, and returns a subset of the test suite satisfying a given criterion. Simple selectors use random sampling techniques. More complex ones exploit the code of the test cases, or could even connect to the code or the specification of the system under test to measure some coverage.

Other techniques to master combinatorial explosion require to execute test cases [8], or to simulate them on a model [10]. These techniques are very efficient to rule out of the resulting test suite a large number of test cases which fail to meet predefined conditions or behaviours.

Recent evolutions of Tobias.

Tobias was demonstrated at ASE'07 [7]. This year's demo will focus on recent evolutions of the tool.

- The tool can now be accessed on-line at the following url: `http://tobias.liglab.fr/`
- Traceability annotations can now be added to the Tobias input file and are propagated to the output files.
- New selectors have been designed which perform various test suite reductions, including reduction based on traceability annotations.

This article will now briefly review these three advances and illustrate them on a simple vending machine example.

2. THE TOBIAS WEB SITE

Fig. 1 gives the main page of the Tobias web site[1]. The user of the web site must provide his email address, select the format of the input and output files, and upload a Tobias input file. The web site processes the file, which may result in several output files. These are grouped in a zip file, whose url is sent by email to the user. The user can then retrieve the zip file at the given url. The zip file is kept on the web site for a limited time.

The input file may be expressed in two languages. TSLT (Test Schema Language for Tobias) is a new input language for Tobias. The language is textual, compact and includes much of the Java syntax. It focuses on the most frequently used constructs of Tobias. As shown in Fig. 2, TSLT is a preprocessor for the input syntax of Tobias, expressed in XML. The XML syntax of Tobias, named INTOB, includes additional constructs (e.g. filters), but focuses on the abstract syntax of the Tobias schemas, without syntactic sugar, which makes it more difficult to use.

The TSLT or INTOB file is processed by Tobias which generates one or several test suites, stored in XML files. These XML files are translated in a target technology using XSL translators. Currently, target technologies include JUnit (versions 3 and 4), potentially adapted for a JML oracle.

The on-line version of Tobias ensures that the most recent version of the tool is available to its users. Still, it suffers

[1] `http://tobias.liglab.fr/`

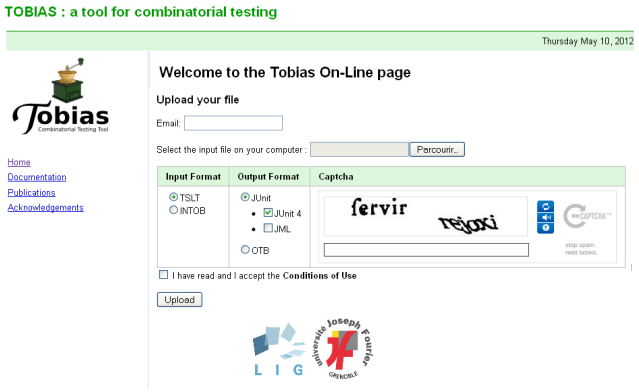

Figure 1: Main page of the Tobias Web site

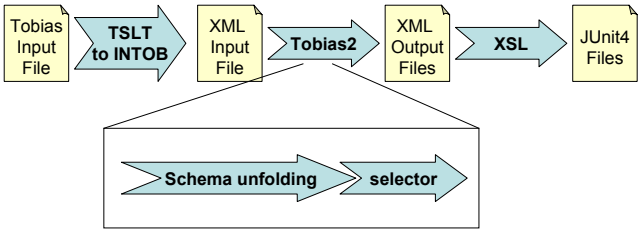

Figure 2: Main steps of test generation

some limitations, compared to the full executable. These limitations are motivated by obvious security reasons.

- The sizes of the input and output files are limited, in order to avoid saturation of the server's disk space.
- The user cannot upload his own filters and selectors, which would allow him to execute arbitrary code on the server. A choice of general purpose selectors is available instead.

3. ILLUSTRATIVE EXAMPLE

Chocolate vending machine.

In order to illustrate the use of Tobias, we consider a classical vending machine example, expressed in Java by J. Offutt[2]. This Java class includes the following methods. Method `addChoc` refills the vending machine with one chocolate. The machine stores up to 10 chocolates. Method `coin` (Fig. 3) inputs a coin into the machine. The machine only accepts three kinds of coins: dimes (10 cents), quarters (25 cents) and dollars (100 cents). The other kinds of coins are ignored. The machine accepts valid coins unless the amount exceeds the price of one chocolate (90 cents). Additional coins are ignored. Finally, `getChoc` (Fig. 4) checks that there is a sufficient amount of coins, and that the machine stores at least one chocolate. It returns the change, and stores the delivered chocolate in the input object.

A first schema.

Group `AddTwoCoins` shows a first Tobias pattern.

[2] `http://cs.gmu.edu/~offutt/softwaretest/ms-level-syllabus/assigns/vendingMachine.java`

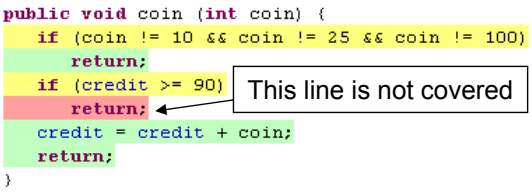

Figure 3: Coverage of the Coin Method by schema strictRobustness_buyChoc

```
group AddTwoCoins[us=true] {
  VendingMachine vm = new VendingMachine();
  vm.coin([10,20,25,50,100,200]){2}; }
```

This pattern creates a new vending machine object, and calls the `coin` method twice. The pattern expresses a set of possible values for the parameter of `coin`. This set is expressed as a list of integers, between square brackets. In this pattern, the test engineer has chosen a mix of valid and invalid coins. The `coin` method will be called twice, as expressed by `{2}`.

Tobias unfolds this compact description into 36 test cases, corresponding to two successive calls to `coin` with all possible combinations of the input parameters. Here is one of these test cases:

```
@Test
public void testSequence_17() {
  VendingMachine vm = new VendingMachine()  ;
  vm.coin(25) ;
  vm.coin(100) ; }
```

Such a test case actually corresponds to a test input. It should be combined with some oracle technology. In the past we have used JML executable assertions as oracle [3]. It is also possible to include some JUnit assert statements.

In order to evaluate how the generated test suite covers the program under test, we used the eclipse plug-in of EMMA[3], a structural coverage measurement tool. This tool shows that the 36 test cases of `AddTwoCoins` cover 100% of the code of the `coin` method. They cover valid and invalid coins, and the case where an excessive amount of valid coins is provided.

The `AddTwoCoins` pattern shows that Tobias allows to express a large number of tests in a compact description. It is easy to increase the number of values, and the number of iterations of the `coin` method, and so produce hundreds, thousands or even millions of test cases.

4. TRACEABILITY ANNOTATIONS

Tobias can also use structured test patterns. In `AddTwoCoins`, the parameters of `coin` correspond to both valid and invalid coins. Tobias allows to distinguish valid and invalid coins in specific groups, then to merge these into a single group, `Coins`, used in the `AddTwoCoins` pattern.

```
group validCoins { values = [10,25,100]; }
group invalidCoins { values = [20,50,200]; }
group Coins { values = [@validCoins, @invalidCoins]; }

group AddTwoCoins[us=true] {
  VendingMachine vm = new VendingMachine();
  vm.coin(@Coins){2}; }
```

[3]http://www.eclemma.org/

This definition is less compact but more structured than the previous one, and it generates the same 36 test cases. Unfortunately, the resulting test suite does not reflect the structure of the initial pattern. All tests are mixed, without distinction between valid and invalid coins.

This has motivated the definition of annotations in Tobias. Each group may be associated to one or several annotations, named "headers", which feature pairs of attribute names and values. These headers are propagated to the output of Tobias where each test case is associated to comments tracing which groups have contributed to the test case.

```
groupheader {testCategory = #NominalValues#}
group validCoins { values = [10,25,100]; }

groupheader {testCategory = #RobustnessValues#}
group invalidCoins { values = [20,50,200]; }
```

For example, valid coins are tagged as "NominalValues", and invalid ones as "RobustnessValues". These two tags are propagated to the following test case, which uses valid (10 cents) and invalid (50 cents) values.

```
@Test
public void testSequence_5() {
/* // testCategory == NominalValues
   // testCategory == RobustnessValues */
   VendingMachine vm = new VendingMachine()  ;
   vm.coin(10) ;
   vm.coin(50) ; }
```

Such traceability annotations are useful to characterize the nature of a generated test case. In the next section, we will see how selectors can exploit this information to extract specific test cases from a test suite or reduce the test suite while keeping some diversity in the tests.

5. TEST CASE SELECTION

Selectors are applied as the last step of the generation process. They return a subset of the test suite satisfying a given criterion. A variety of selectors have been developed for the on-line version of Tobias.

The `random` selector randomly chooses n elements of the testsuite. The `randomTree` selector performs a random selection which keeps a coverage of the structures of the tests as described in [1]. The `periodic` selector keeps test cases located at position 1, i+1, 2*i+1,.... These three selectors take as parameter the number of elements of the reduced test suite. Of course, they have a poor understanding of the semantics of the test suite.

The `testCategory` selector exploits the values of the `testCategory` annotations. Depending on its input parameter,

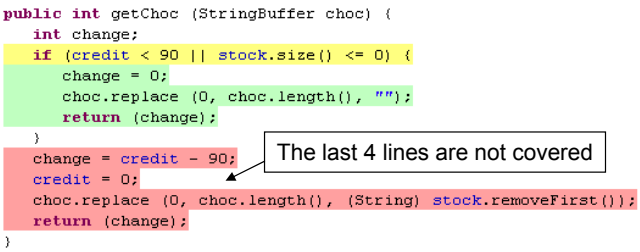

Figure 4: Coverage of the getChoc Method by schema strictRobustness_buyChoc

it can keep all tests which are only annotated as "Nominal", or as "Robustness", or the ones which have exactly one or at least one "Nominal"/"Robustness" annotation. To illustrate this selector, we defined a pattern which performs various refills of the vending machine, then adds several coins, and finally buys a chocolate.

```
group buyChocolate[us=true] {
  VendingMachine vm = new VendingMachine();
  @Refill;
  @addCoins;
  StringBuffer res = new StringBuffer("xx");
  vm.getChoc(res); }
```

Several groups are defined to perform valid and invalid refills. They are tagged as "Nominal" or "Robustness", and other annotations, marked as "comments", denote the empty of full character of the vending machine.

```
groupheader {testCategory = #NominalOps#}
group validRefill{ vm.addChoc(["Belgian","Swiss"]); }

groupheader {comment = #Empty vending machine#}
group invalidRefill0{ vm.getStock(); }

groupheader {comment = #Full vending machine#}
group invalidRefill11{ vm.addChoc(["Belgian"]){11}; }

groupheader {testCategory = #RobustnessOps#}
group invalidRefill{@invalidRefill0 | @invalidRefill11 }

group Refill {@validRefill | @invalidRefill }
```

Group `addCoins` covers the cases where the amount is sufficient or not. Tags are defined accordingly.

```
groupheader {comment = #InsufficientAmount#,
        testCategory = #RobustnessOps#}
group addLessThan90 { vm.coin([10,25]){0,3}; }

groupheader {comment = #SufficientAmount#,
        testCategory = #NominalOps#}
group addAtLeast90 { vm.coin(25){3}; vm.coin(10);
                     vm.coin(@validCoins); }

groupheader {comment = #SufficientAmount#,
        testCategory = #RobustnessOps#}
group add90AndMore {vm.coin(100); vm.coin(@validCoins);}

group addCoins { vm.coin(@Coins){2} | @addLessThan90 |
                 @addAtLeast90 | @add90AndMore }
```

This pattern unfolds into 228 test cases covering 100% of the vending machine code. The `testCategory` selector with parameter "OnlyRobustness" returns the 48 test cases which only include robustness tags. Figures 3 and 4 show how these tests cover two of the vending machine methods. They show that the nominal behaviour of `getChoc` was not covered, which was the purpose of this selection.

The `testCategory` selector aims to cover a subset of functionalities. Annotations can also be used to select a reduced test suite covering all functionalities. For example, the `HeaderCombinations` selector keeps a single test case for each combination of all tags, including "comment" tags. In our example, it selects 21 test cases which cover 100% of the code, and hence provides a reduced version of the test suite featuring the diversity of test cases expressed by the tags.

6. CONCLUSION

The on-line version of Tobias features a new textual input language and adds traceability and selection capabilities. Traceability annotations can be exploited to keep track of the pattern structure in the final test suite. Combined with selectors, they provide a powerful test suite reduction or selection tool, using the intended semantics of test patterns.

The original vision of Tobias is to capture the test engineer's knowledge in a compact description, leaving the clerical and repetitive unfolding tasks to the tool. Annotations provide an additional way to express this knowledge, and selectors based on annotations allow him to exploit it further.

7. ACKNOWLEDGMENTS

This work is supported by the ANR TASCCC Project under grant ANR-09-SEGI-014. We also thank Iñaki Iralde for his work on the initial version of the web site.

8. REFERENCES

[1] F. Dadeau, Y. Ledru, and L. D. Bousquet. Directed random reduction of combinatorial test suites. In *Random Testing '07*. ACM, 2007.

[2] F. Dadeau and R. Tissot. jSynoPSys – a scenario-based testing tool based on the symbolic animation of B machines. In *MBT'09*, volume 253-2 of *ENTCS*, pages 117–132, 2009.

[3] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. Reusing a JML specification dedicated to verification for testing, and vice-versa: case studies. *J. of Automated Reasoning, Springer*, 45(4), 2010.

[4] S. Dupuy-Chessa, L. du Bousquet, J. Bouchet, and Y. Ledru. Test of the ICARE platform fusion mechanism. In *DSVIS'05*, LNCS 3941. Springer, 2006.

[5] L. Ferro, L. Pierre, Y. Ledru, and L. du Bousquet. Generation of test programs for the assertion-based verification of TLM models. In *3rd Int. Design and Test Workshop, IDT 2008.* IEEE, 2008.

[6] K. Lausdahl, H. K. A. Lintrup, and P. G. Larsen. Connecting UML and VDM++ with open tool support. In *FM 2009: Formal Methods, Second World Congress*, LNCS 5850. Springer, 2009.

[7] Y. Ledru, F. Dadeau, L. du Bousquet, S. Ville, and E. Rose. Mastering combinatorial explosion with the Tobias-2 test generator. In *IEEE/ACM Int. Conf. on Automated Software Engineering.* ACM, 2007. Demo.

[8] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE'04)*, LNCS 2984. Springer, 2004.

[9] M. Polo and B. Pérez. A framework and a web implementation for combinatorial testing. 2010. White paper of the Alarcos Research Group. Available at http://alarcosj.esi.uclm.es/CTWeb.

[10] T. Triki, Y. Ledru, L. du Bousquet, F. Dadeau, and J. Botella. Model-based filtering of combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE'12)*, LNCS 7212. Springer, 2012.

[11] Trusted Labs. *TL CAT Testing Tools*, 2010. http://www.trusted-labs.com/IMG/pdf/TL_CAT.pdf.