Laboratoire d'Informatique de Grenoble
Unité de recherche UJF, INPG, CNRS, UPMF
UMR 5217
B.P. 72, F-38402 SAINT-MARTIN-D'HÈRES CEDEX, FRANCE

Tobias Notes

Yves Ledru
Taha Triki
German Vega
Lydie du Bousquet

UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS
Laboratoire d'Informatique de Grenoble (LIG), UMR 5217,
F-38041, Grenoble, France
Yves.Ledru@imag.fr

June 2012

Chapter 1

What is Tobias?

Tobias is a combinatorial testing tool which generates a large number of tests by combinatorial unfolding of a test pattern.

Combinatorial testing allows to express a large set of test cases in a few lines. It allows to exhaustively test all combinations of selected parameter values. This brings several benefits to the test engineer:

- It frees the test engineer from clerical copy/paste activities while preparing a large repetitive test suite. It allows him/her to concentrate on insightful activities in the design of the test suite. This increases the productivity in the test production phase.
- Large repetitive test suites increase the confidence of the test engineer in the tested software. They allow to test all possible combinations of selected inputs, providing a systematic character to the tests.
- It eases the maintenance of a test suite. Test patterns are more compact and corrections must only be done once, in most cases.

The Tobias tool is now available through a web interface, with generators for JUnit and JML/JUnit test suites. It is available at http://tobias.liglab.fr and Fig. 1.1 pictures its welcome screen.

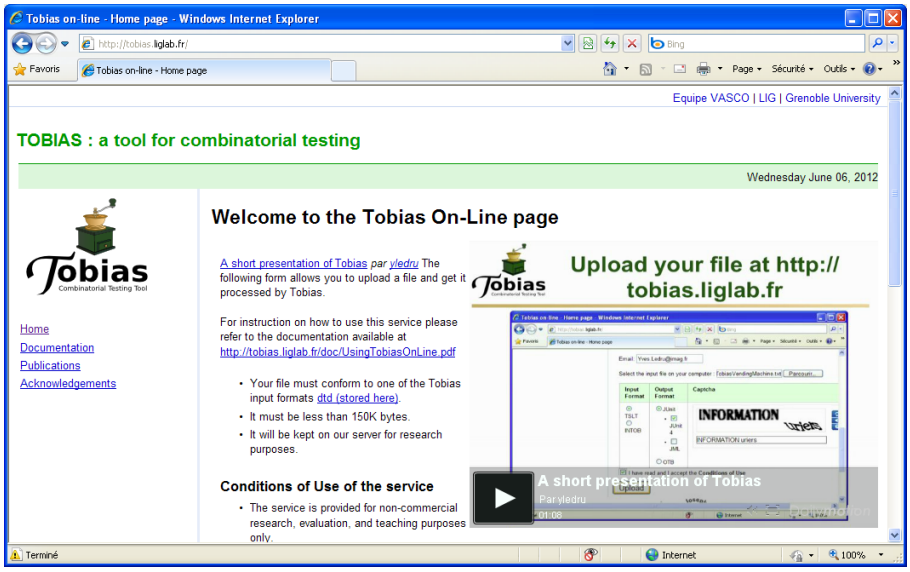

Figure 1.1: The Tobias-on-Line web site

Chapter 2

Tobias-on-Line : getting started

Let us first consider a simple java class `Purse` which handles an electronic purse. It features a constructor, and three operations : `credit(int nb)`, `debit(int nb)` and `getBalance()`. The java code for class `Purse` can be downloaded here and is given at Fig. 2.5, page 9.

In order to test this program, one would like to create an object of class `Purse`, then to credit it with some amount of money, then to debit some other amount and finally check the resulting balance. Tobias allows to generate tests which follow this sequence and combine various predefined amounts.

2.1 The input file

Fig. 2.1 gives the text of the Tobias input file `PurseSeq1.txt` (available here). It is expressed in the TSLT language (Test Schema Language for Tobias) and stored in an ordinary text file. This input file describes a group named `SimpleSequence`. This group is tagged as `us=true` which means that this group will be unfolded by Tobias into a test suite. The group is defined as a sequence of four instructions: a call to the constructor of class `Purse`, followed by calls to methods `credit`, `debit` and `getBalance`. Methods `credit` and `debit` take an integer parameter as input. In group `SimpleSequence`, three values (10, 50, and 0) are defined for the input parameter of `credit`, and two values (5 and 15) for the input parameter of `debit`.

2.2 Unfolding the group

To unfold the group, the input file should be submitted to the Tobias-on-Line web page at the following url http://tobias.liglab.fr/. Figure 2.2 gives the input form for this page. The user must do the following actions:

1. enter his email address
2. select the input file

```
1  group SimpleSequence[us=true] {
2    Purse p = new Purse();
3    p.credit([10, 50, 0]);
4    p.debit([5,15]);
5    p.getBalance();
6  }
```

Figure 2.1: The input file for Tobias : `PurseSeq1.txt`

3. select the types for the input and output files (here TSLT and JUnit)
4. answer the captcha challenge by entering the text corresponding to the distorded image.[1]
5. accept the terms of the licence by checking the corresponding box
6. click on the "upload button"

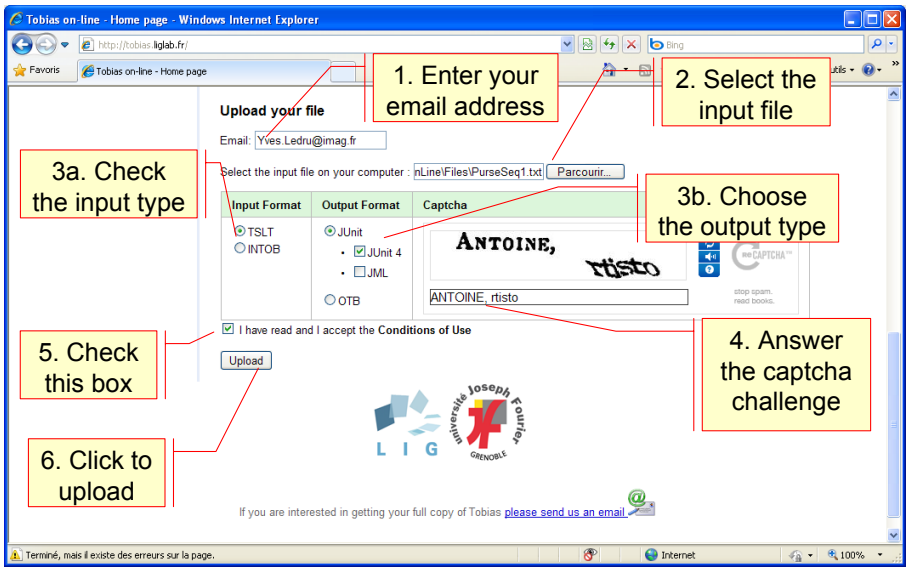

Figure 2.2: The input form for Tobias-on-Line

2.3 The results of Tobias

Fig. 2.3 gives the resulting screen when the input file is correct. This screen tells us that 6 test cases were generated. An email is send to the user with a link to the output file. This output file is a zip file which stores the resulting source code of the JUnit files. It will remain on our web site for a limited amount of time. Use the link to access the file and save it on your computer. You can then extract its contents with your favorite unzip tool.

In this case, the unfolding produces a unique JUnit file, `TS_SimpleSequence.java`, given at Fig. 2.4 at page 8 (after some reformating of the actual output file). This JUnit file includes 6 test cases, corresponding to all combinations of the input parameters given for methods `credit` and `debit`.

2.4 Conclusion

This first chapter has shown how to unfold a simple Tobias input file using Tobias-on-Line. In the next chapters, we will see how the combinatorial constructs of Tobias allow us to produce a large number of tests from a simple input file.

[1] After several succeeded attempts, the captcha challenge will no longer be needed.

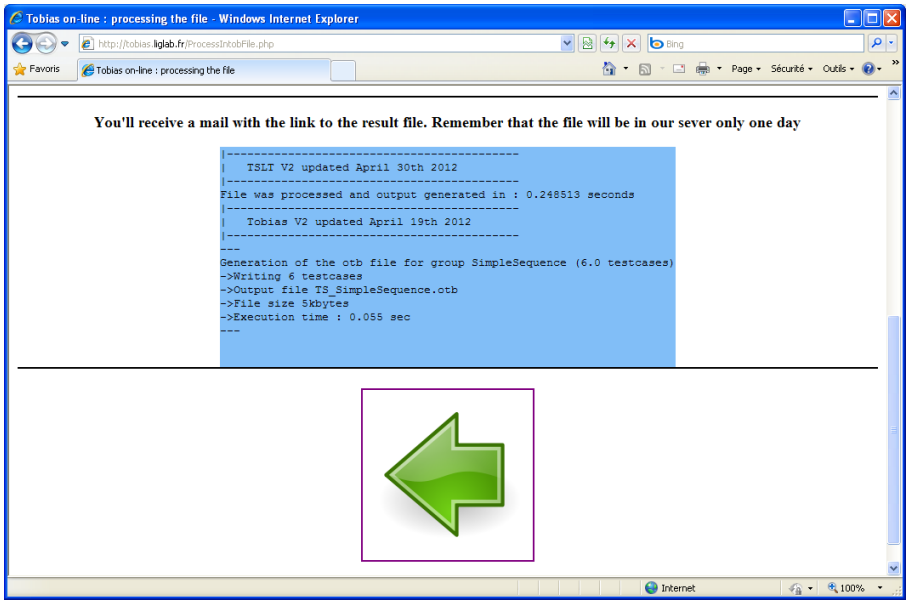

Figure 2.3: The output page corresponding to `PurseSeq1.txt`

```
1  import junit.framework.TestCase;
2
3  public class TS_SimpleSequence extends TestCase {
4
5          public void testSequence_1() throws Exception {
6          try{ Purse p = new Purse() ;
7                   p.credit(10) ; p.debit(5) ; p.getBalance() ;
8         } catch (Exception e) { throw e; }
9          }
10         public void testSequence_2() throws Exception {
11         try{ Purse p = new Purse()  ;
12              p.credit(10) ; p.debit(15) ; p.getBalance() ;
13        } catch (Exception e) { throw e; }
14         }
15         public void testSequence_3() throws Exception {
16         try{ Purse p = new Purse()  ;
17          p.credit(50) ; p.debit(5) ; p.getBalance() ;
18        } catch (Exception e) { throw e; }
19         }
20         public void testSequence_4() throws Exception {
21         try{ Purse p = new Purse()  ;
22          p.credit(50) ; p.debit(15) ; p.getBalance() ;
23        } catch (Exception e) { throw e; }
24         }
25         public void testSequence_5() throws Exception {
26         try{ Purse p = new Purse()  ;
27          p.credit(0) ; p.debit(5) ; p.getBalance() ;
28            } catch (Exception e) { throw e; }
29         }
30         public void testSequence_6() throws Exception {
31         try{ Purse p = new Purse()  ;
32     p.credit(0) ; p.debit(15) ; p.getBalance() ;
33            } catch (Exception e) { throw e; }
34         }
35 }
```

Figure 2.4: The JUnit file `TS_SimpleSequence.java` corresponding to the unfolding of `PurseSeq1.txt`

```
1  public class Purse {
2      private int bal;
3
4      public Purse(){
5          bal = 0;
6      }
7
8      public int credit(int nb){
9          int newBalance = getBalance()+nb;
10         if (nb >= 0 && newBalance <= 100 ){
11                 bal=bal+nb;
12                 return nb;
13         }
14         else {return 0;}
15     }
16
17     public int debit(int nb){
18         int newBalance = getBalance()-nb;
19         if (nb >= 0 && newBalance >= 0 ){
20                 bal=bal-nb;
21                 return nb;
22         }
23         else {return 0;}
24     }
25
26     public int getBalance(){
27         return bal;
28     }
29
30
31
32 }
```

Figure 2.5: The java code for `Purse.java`

Chapter 3

The Tobias constructs in TSLT

TSLT is a convenient text format for Tobias input files. Its detailed syntax is presented in appendix A.

3.1 Basic form for a tobias group

A Tobias input file gives a list of "groups" of the form:

```
group GroupName [ GroupQualifier = value , ...]  {
      Sequence of instructions
      OR Set of instructions
      OR Set of values
}
```

- Each group is characterized by a unique *GroupName* that is used to refer to the group in other groups, and provides the basis for the filename of the resulting test suite (e.g. the JUnit file).
- The group is associated to several *GroupQualifiers* associated to a given value. At this stage, the most useful qualifier is `us`, which defines the "unfold status". When `us` is equal to `true`, Tobias will produce a test suite corresponding to this group. When `us` is equal to `false`, it means that the group is not aimed to produce a test suite, but it may be used in the definition of another group.
- The body of the group is either a sequence of instructions, using a syntax close to the Java syntax, or a set of instructions or a set of values. These cases will be described in the next sections.

3.2 Group featuring a sequence of instructions

In its simplest form, a group includes a sequence of instructions which does not include other groups, and will not lead to combinatorial unfolding. If the group is marked as `us = true`, and its processing will produce a testsuite with a single test case, corresponding to the sequence of instructions. For example, group `SimpleSequence001`, available here, is defined as

```
1  group SimpleSequence001[us=true] {
2    Purse p = new Purse();
3    p.credit(10);
4    p.debit(5);
5    int x = p.getBalance();
6    assertTrue(x >= 0);
7    assertTrue(100 >= x);
8  }
9
```

Note that the sequence of instructions may involve variable declarations, assignments, method calls, including assert statements of JUnit. It may not include loops or conditional statements, which would break the linear sequence of instructions. In this example, the test will create a purse, credit it with the amount of 10, debit the amount of 5, assign the current balance to a new variable `x`, and finally check that the current balance, stored in `x`, is in the interval `0..100`.

It is unfolded into the following class:

```
1  import junit.framework.TestCase;
2
3  public class TS_SimpleSequence001 extends TestCase {
4
5      public void testSequence_1() throws Exception {
6      try{
7          Purse p = new Purse()  ;
8          p.credit(10) ;
9          p.debit(5) ;
10         int x = p.getBalance() ;
11         assertTrue(x>=0) ;
12         assertTrue(100>=x) ;
13        }
14        catch (Exception e) {
15         throw e;
16        }
17     }
18 }
```

3.3 Adding groups of values

The goal of Tobias is to provide a compact form to express a large number of similar test cases. This is achieved by replacing single values by a group of values, or instructions by a group of instructions. Let us first consider groups of values. File `PurseSeq002.txt`, available here, defines three groups.

```
1  group SimpleSequence002[us=true] {
2    Purse p = new Purse();
3    p.credit(@AmountsToCredit);
4    p.debit(@AmountsToDebit);
5    int x = p.getBalance();
6    assertTrue(x >= 0) ;
7    assertTrue(100 >= x) ;
8  }
9
10 group AmountsToCredit {
11 values = [10, 50, 0];
12 }
13
14 group AmountsToDebit {
15 values = [5, 15];
16 }
```

The first group is a variant of group `SimpleSequence001` where the parameters of `credit` and `debit` now refer to groups of values. Groups `AmountsToCredit` and `AmountsToDebit` are not marked with qualifier `us`, therefore they will not result in test suites. They are meant to be used in other groups, like `SimpleSequence002`. The prefix `@` is used in order to refer to such groups.

Processing this file with Tobias will result in a single file `TS_SimpleSequence002.java` which includes 6 test cases corresponding to all combinations of the three values of `AmountsToCredit` with the two values of `AmountsToDebit`.

3.4 Using unnamed groups of values

Sometimes, a group of values will only be used once. TSLT provides a simple form of group definition where the group is defined where it is used. Group `SimpleSequence003`, available here, gives the same result as `SimpleSequence002` but it only requires to define a single group which refers to groups of values represented between square brackets.

```
1  group SimpleSequence003[us=true] {
2    Purse p = new Purse();
3    p.credit([10, 50, 0]);
4    p.debit([5,15]);
5    int x = p.getBalance();
6    assertTrue(x >= 0);
7    assertTrue(100 >= x);
8  }
```

3.5 Groups of instructions

Tobias also allows to define groups of instructions. For example, the following file defines group `Modify` which includes one call to `credit(10)` and one call to `debit(5)`. Group `SimpleSequence004`, available here, makes two consecutive calls to this group. Unfolding this file gives a single testsuite with 4 test cases corresponding to all possible sequences of two calls to group `Modify`.

```
1  group SimpleSequence004[us=true] {
2    Purse p = new Purse();
3    @Modify;
4    @Modify;
5    int x = p.getBalance();
6    assertTrue(x >= 0);
7    assertTrue(100 >= x);
8  }
9
10 group Modify {
11  p.credit(10)| p.debit(5)
12 }
```

Defining groups of instructions is a convenient way to define tests which differ by some of their treatments. It is also very useful to define variants of the initialisation process in a test suite.

3.6 Groups using groups

A group of values, or a group of instructions may itself refer to other groups.

In the following example, available here,, the group of instructions `Modify` refers to the two groups `AmountsToCredit` and `AmountsToDebit`. As a result, `Modify` includes 5 elements, and group `SimpleSequence005` unfolds into 25 test cases.

```
1  /* Group of instructions */
2
3  group SimpleSequence005[us=true] {
```

```
4     Purse p = new Purse();
5     @Modify;
6     @Modify;
7     int x = p.getBalance();
8     assertTrue(x >= 0) ;
9     assertTrue(100 >= x ) ;
10  }
11
12  group AmountsToCredit {
13  values = [10, 50, 0];
14  }
15
16  group AmountsToDebit {
17  values = [5, 15];
18  }
19
20  group Modify {
21   p.credit(@AmountsToCredit)| p.debit(@AmountsToDebit)
22  }
```

A group of values may also refer to other groups of values. In file `PurseSeq006.txt`, available here, group `Amounts` is defined as the union of groups `AmountsToCredit` and `AmountsToDebit`. As a result, `Amounts` includes 5 elements, and group `SimpleSequence006` unfolds into 25 test cases.

```
1   group SimpleSequence006[us=true] {
2     Purse p = new Purse();
3     p.credit(@Amounts);
4     p.debit(@Amounts);
5     int x = p.getBalance();
6     assertTrue(x >= 0);
7     assertTrue(100 >= x);
8   }
9
10  group AmountsToCredit {
11  values = [10, 50, 0];
12  }
13
14  group AmountsToDebit {
15  values = [5, 15];
16  }
17
18  group Amounts {
19  values = [@AmountsToCredit, @AmountsToDebit ];
20  }
```

3.7 Iterations

Similarity in test cases may also result from the fact that the same sequence of instructions is performed a variable number of times. The iteration construct of Tobias allows to repeat a sequence of instructions. The number of iterations is defined by an interval of values. The unfolding will provide test cases corresponding to each number of iterations included in the interval.

For example, group `SimpleSequence007`, available here, includes an iteration on `p.credit(10)` which is repeated one to three times. The iteration is denoted by enclosing the lower and upper bounds of the interval, separated by a coma, and enclosed into braces: { *lowerbound* , *upperbound* }

. When the upperbound is equal to the lower bound which corresponds to a fix number of iterations, a shortcut is { *nbOfIterations* }

```
1  group SimpleSequence007[us=true] {
2    Purse p = new Purse();
3    p.credit(10){1,3};
4    int x = p.getBalance();
5    assertTrue(x >= 0);
6    assertTrue(100 >= x);
7  }
```

The resulting test suite includes thus three test cases corresponding to 1, 2, and 3 iterations of `p.credit(10)` . Here is the code of the corresponding JUnit file.

```
1  import junit.framework.TestCase;
2
3  public class TS_SimpleSequence007 extends TestCase {
4      public void testSequence_1() throws Exception {
5      try{
6          Purse p = new Purse()  ;
7          p.credit(10) ;
8          int x = p.getBalance() ;
9          assertTrue(x>=0) ;
10         assertTrue(100>=x) ;
11        }
12     catch (Exception e) {
13        throw e;
14       }
15     }
16
17     public void testSequence_2() throws Exception {
18     try{
19         Purse p = new Purse()  ;
20         p.credit(10) ;
21         p.credit(10) ;
22         int x = p.getBalance() ;
23         assertTrue(x>=0) ;
24         assertTrue(100>=x) ;
25        }
26     catch (Exception e) {
27        throw e;
28       }
29    }
30
31     public void testSequence_3() throws Exception {
32     try{
33         Purse p = new Purse()  ;
34         p.credit(10) ;
35         p.credit(10) ;
36         p.credit(10) ;
37         int x = p.getBalance() ;
38         assertTrue(x>=0) ;
39         assertTrue(100>=x) ;
40        }
41     catch (Exception e) {
```

```
42          throw e;
43         }
44      }
45  }
```

3.8 Combining these constructs

The following examples combine several group definitions, involving groups of values, groups of instructions, and even groups of objects.

Group `SimpleSequence008`, available here, unfolds into a test suite with 10 test cases. It defines an unnamed group of objects and uses `Modify` as a group of method calls, instead of a group of instructions.

```
1   group SimpleSequence008[us=true] {
2     @InitPurses;
3     [p1,p2].@Modify;
4   }
5
6   group AmountsToCredit {
7   values = [10, 50, 0];
8   }
9
10  group AmountsToDebit {
11  values = [5, 15];
12  }
13
14  group Modify {
15   credit(@AmountsToCredit)| debit(@AmountsToDebit)
16  }
17
18  group InitPurses {
19    Purse p1 = new Purse();
20    Purse p2 = new Purse();
21  }
```

Group `SimpleSequence009`, available here, is a compact version of the previous one, using only two groups

```
1   group SimpleSequence009[us=true] {
2     Purse p1 = new Purse();
3     Purse p2 = new Purse();
4     [p1,p2].@Modify;
5   }
6
7   group Modify {
8    credit([10, 50, 0])| debit([5, 15])
9   }
```

Group `SimpleSequence010`, available here, is also written in a compact manner. It performs various combinations involving calls to `debit` and `credit` and results into 155 test cases.

```
1   group SimpleSequence010[us=true] {
2     Purse p = new Purse();
3     (p.credit([10, 50, 0])| p.debit([5, 15])){1,3};
4     int x = p.getBalance();
```

```
5     assertTrue(x >= 0);
6     assertTrue(100 >= x );
7   }
```

3.9 Conclusion

This chapter has presented the basic constructs of Tobias which are available in the on-line version of the tool. It is not meant to cover exhaustively the constructs of TSLT or Tobias but focuses on the ones used most often.

Chapter 4

Tobias headers and JUnit specificities

When producing JUnit test cases, it might be needed to declare :

- a list of imported packages
- a target package
- SetUp and TearDown methods
- specific treatment for exceptions

These are supported by the headers mechanism of Tobias. A header is actually a structured comment, made of pairs (attribute, value). Headers can be associated to the whole project, or to each individual group. Section 4.1 will discuss headers associated to the whole project, and their specific translation in java. Group headers, which provide a traceability mechanism independently of the target language, are discussed in section 4.2. To illustrate the headers mechanism, we use a modified version of J. Offut's vending machine, available here [1] .

4.1 Project headers

Project headers are referred to as "headers".

4.1.1 Uninterpreted headers

In its simplest form, a header is just a comment on the test suite.

The file `TobiasVendingMachine00.txt`, available here, includes a single group `Add2Coins` which includes two consecutive calls to the `coin` method. 6 values are provided as input to `coin`. The group will unfold into 36 test cases.

```
group Add2Coins[us=true] {
  VendingMachine vm = new VendingMachine();
      vm.coin([10,20,25,50,100,200]){2};
  }
```

We can add a header at the beginning of the file. Here the header is just a simple comment.

```
header { myComment = #This file includes a single
                      group, named Add2Coins.# }
```

[1] The original file by J. Offutt is available at `http://cs.gmu.edu/~offutt/softwaretest/ms-level-syllabus/assigns/vendingMachine.java`

```
group Add2Coins[us=true] {
  VendingMachine vm = new VendingMachine();
      vm.coin([10,20,25,50,100,200]){2};
  }
```

This file produces the following JUnit file. Note that the contents of the header appears as a comment at the beginning of the file.

```
/* myComment == This file includes a single
                      group, named Add2Coins.
        */

import org.junit.*; // JUnit4

public class TS_Add2Coins {

  @Test
  public void testSequence_1() {
    VendingMachine vm = new VendingMachine()  ;
    vm.coin(10) ;
    vm.coin(10) ;
    }
...
```

4.1.2 Handling java packages

The VendingMachine class, used by this test suite, is included into package `vendingMachine`. Therefore, we need to import that package in the JUnit file. This is declared by adding a "header" at the beginning of the file:

```
header { import = #vendingMachine.*# }
```

Since we work with packages, we'd also like the resulting test suite to be included in a package named `test`. This is declared by completing our header as follows:

```
header { package = #test#,
         import = #vendingMachine.*# }
```

This example is available here The file produced by Tobias looks like the following lines:

```
package test;
import vendingMachine.*;

import org.junit.*; // JUnit4

public class TS_Add2Coins
 {
    @Test
   public void testSequence_1() {
      VendingMachine vm = new VendingMachine()  ;
       vm.coin(10) ;
       vm.coin(10) ;
   }

   ...
```

4.1.3 Other project headers

Global headers of a project may also be used to declare systematic exception handling in the test suite. Here we would like to catch exceptions raised by any of our tests. This is achieved by adding the following header:

```
header { package = #test#,
         import = #vendingMachine.*#,
         globalCatch = #catch(Exception global_e){ /* process exception */ }# }
```

This example is available here The file produced by Tobias is given below. Note that every test case now includes a `try...catch` statement.

```
package test;
import vendingMachine.*;

import org.junit.*; // JUnit4

public class TS_Add2Coins {

  @Test
  public void testSequence_1() {
    try{
        VendingMachine vm = new VendingMachine()  ;
        vm.coin(10) ;
        vm.coin(10) ;
      } catch(Exception global_e){ /* process exception */ }
   }
```

There are several predefined headers in Tobias. They can be used to catch exceptions at every step of the test case (`localCatch`), to declare global variables in the test suite (`globalVar`), or to declare methods which are executed before or after every test (`before`, `after`, `setUp`, `tearDown`), or at the beginning or end of the test suite (`beforeClass`, `afterClass`). They are used alltogether, just for illustration purposes, in the following example, available here

```
header {
  package = #test#,
  import = #vendingMachine.*#,
  globalCatch = #catch(Exception global_e){ }# ,
  localCatch = #catch(Exception local_e){ }# ,
  globalVar = #int global_x = 0;#,
  setUp = #System.out.println("executing setup");#,
  tearDown = #System.out.println("executing teardown");#,
  before = #System.out.println("executing before processing");#,
  after = #System.out.println("executing after processing");#,
  beforeClass = #System.out.println("executing before class processing");#,
  afterClass = #System.out.println("executing after class processing");#
  }
```

This example unfolds into the following lines:

```
package test;
import vendingMachine.*;

import org.junit.*; // JUnit4
```

```
public class TS_Add2Coins  {

int global_x = 0;

@Before
public void setUp(){
System.out.println("executing setup");
}

@After
public void tearDown(){
System.out.println("executing teardown");
}

@Before
public void setUp(){
System.out.println("executing before processing");
}

@After
public void tearDown(){
System.out.println("executing after processing");
}

@BeforeClass
public static void beforeClass_method(){
System.out.println("executing before class processing");
}

@AfterClass
public static void afterClass_method(){
System.out.println("executing after class processing");
}

  @Test
  public void testSequence_1() {
  try{
      try{VendingMachine vm = new VendingMachine()  ;
         }catch(Exception local_e){ }
      try{ vm.coin(10) ;
         }catch(Exception local_e){ }
      try{ vm.coin(10) ;
         }catch(Exception local_e){ }
      } catch(Exception global_e){ }
   }
   ...
```

4.2 Group headers

Headers can also be associated to group definitions. They are part of a traceability mechanism which marks every test case with the group headers corresponding to the groups involved in the synthesis of the test case.

Let us illustrate this on our example. Group `Add2Coins` makes use of group `Coins`, which itself includes a group of valid coins and a group of invalid ones. Group headers are used to tag groups

validCoins and invalidCoins. Valid coins are tagged as NominalValues and invalid ones as RobustnessValues. This Tobias input file can be retrieved here.

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

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

group Coins { values = [@validCoins, @invalidCoins]; }

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

Group AddTwoCoins generates the same 36 test cases as Add2Coins. But the tests are now marked with the annotations of the groups used in their synthesis. For example, the following test case first uses a valid coin, then an invalid one. It includes tags NominalValues and RobustnessValues.

```
  @Test
  public void testSequence_4() {
        /* testCategory == NominalValues
        */
        /* testCategory == RobustnessValues
        */
     VendingMachine vm = new VendingMachine()  ;
     vm.coin(10) ;
     vm.coin(20) ;
  }
```

This traceability mechanism can help in two kinds of activities.

- Understanding the meaning of a test: annotations provide some documentation on how the test was synthesized.
- Test selection: Tobias provides a construct named “selectors” which helps to extract a subset of tests out of a test suite. Annotations can be used as a criterion for this extraction. See chapter 5 for more information on the selection mechanism.

Chapter 5

Selectors

5.1 Definition

A selector is an operator which takes as input a testsuite resulting from the unfolding of a group definition, and returns a subset of the test cases of this test suite.

A selector does not invent new test cases but simply chooses several test cases from the initial test suite. The criteria for this choice depend on the selector. The choice can be random, guided by the form of the test cases, or by the annotations associated to each test case.

A selector is usually written in Java. Executing a selector is thus executing a piece of java code. For obvious security reasons, the on-line version of Tobias only supports the use of predefined selectors, which are already uploaded on the web site. This chapter details the use of these predefined selectors.

The following examples can be found in file `TobiasVendingMachine04.txt` available here.

The use of a selector proceeds in two steps. First, an instance of the selector must be created. It actually corresponds to a call to the constructor of the selector. Then this instance is used to process a given group. This two-step process is illustrated in section 5.2 with the random selector.

5.2 Random selector

The random selector selects a subset of the testsuite on a random basis.

The constructor of this selector takes three parameters; The first two parameters define the expected size of the result. This size can be specified explicitly using the `nb` parameter, or as a percentage of the original testsuite using the `percent` parameter. By default, the percentage is used, unless it is assigned to an invalid value (e.g. 101), in which case the number is used. The third parameter gives a seed for the random generator. This seed guarantees that the selection process can be reproduced.

- `nb` of type `int` is the number of tests to select
- `percent` of type `int` is the percentage of tests to select
- `seed` of type `long` is a seed for the random generator

The following line declares an instance of the random selector which chooses 8 tests. The percentage is fixed to an invalid value, so that `nb` defines the number of expected tests. The selector declaration refers to the name of the corresponding java class, specifying its class file, and its language.

```
// If percent is out of the range [0..100],
// then the number of tests (nb) takes precedence.
selector random8CasesSelection  (int nb=8, int percent=101, long seed=12345)
                                [lang=java,file=SelectorRandom.class]
```

This instance of the selector is then combined with group `AddTwoCoins`. The result is a new group definition, declares with "`us=true`", which means that the group will be unfolded.

```
groupheader {comment = #This is a random reduction
                        of AddTwoCoins into 8 test cases. #}
selectorgroup random8CasesSelection_Add2C
              [groupid=AddTwoCoins, selectorid=random8CasesSelection, us=true]
```

The resulting test suite includes 8 test cases taken from `AddTwoCoins`.

The following example shows an alternate way to use the random selector, specifying a percentage of the original test suite. Here the percentage is 25%, which corresponds to 9 test cases when applied to group `AddTwoCoins`.

```
// If percent is out of the range [0..100],
// then the number of tests (nb) takes precedence.
selector random25PercentSelection (int nb=8, int percent=25, long seed=12345)
                                  [lang=java,file=SelectorRandom.class]

groupheader {comment = #This is a random reduction of AddTwoCoins
                        into 9 test cases (25%). #}
selectorgroup random25PercentSelection_Add2C
              [groupid=AddTwoCoins, selectorid=random25PercentSelection, us=true]
```

Please note that the TSLT syntax requires that all selector declarations are grouped, and that group definitions using a selector are also grouped (see file `TobiasVendingMachine04.txt` for an example.

The random selector is one of the simplest forms of selectors. Clearly, it does not take into account the semantics of test cases in its selection. But It may still perform a decent job when the number of test cases is fixed by some outside factors, like limited resouces.

In this example, the full 36 test cases of AddTwoCoins cover 100% of the code of the `coin` method[1]. The randomly selected test suite with 8 test cases covers 95,5% of the `coin` method, and the 25% test suite, with 9 test cases, covers 100% of the `coin` method. This shows that random sampling can provide a representative subset of the initial test suite.

The random selector has also a default constructor, without parameters, which is instantiated to a 10% selection. In this example, it can be called as

```
selector randomSelection [lang=java,file=SelectorRandom.class]

groupheader {comment = #This is a random reduction of 10% of AddTwoCoins
                       into 4 test cases.#}
selectorgroup randomSelection_Add2C
              [groupid=AddTwoCoins, selectorid=randomSelection, us=true]
```

5.3 Periodic selector

Like the random selector, the periodic selector does not take into account the semantics of test cases. Like the random selector, it allows to specify the expected number of test cases or a percentage of the initial test suite. What differs is that test cases are selected on the basis of their position in the test suite. The selector takes test cases located at a regular interval, starting with the first test case and ending with the last one.

For example, the following declarations will extract 8 test cases out of the 36 test cases of group AddTwoCoins.

```
selector periodicSelection (int nb=8, int percent=101)
                           [lang=java,file=PeriodicSelector.class]

groupheader {comment = #This is a periodic reduction
```

[1] Using the eclemma coverage tool to measure instruction coverage.

```
                        of AddTwoCoins into 8 test cases. #}
selectorgroup periodicSelection_Add2C
              [groupid=AddTwoCoins, selectorid=periodicSelection, us=true]
```

The resulting test suite will include the 1st, 6th, 11th, 16th, 21st, 26th, 31st, and 36th test cases of the original test suite. (every selected test cases is 5 positions after the previous one; when the period is not an exact integer, these positions are rounded).

The underlying idea of the periodic selector is that sampling test cases on a regular basis will provide a representative subset of the test suite. In this example, the 8 periodic test cases cover 95,5% of the `coin` method.

5.4 Selectors based on annotations

The random and periodic selector don't take into account the semantics of test cases. The selectors presented in this section exploit the additional semantics given by the group headers. They were initially presented in [LVTdB12].

There are two kinds of selections:

- Selections which aim to find a representative subset of the test suite, corresponding to some test suite reduction.
- Selections which target a specific characteristics of some tests in the test suite, i.e. which filter the test suite with respect to a specific property.

5.4.1 Filtering on the basis of test category

The selectors presented in this section correspond to the second kind of selectors, i.e. filters which keep the test cases exhibiting a given property. The `TestCategorySelector` only focuses on the headers refering to the "`testCategory`" tag. The first example selects the test cases which only refer to some test category including the string "Nominal".

```
groupheader {comment = #This selector keeps only the test cases
                        which include only nominal tags.#}
selector keepPureNominalTests (String criterion=OnlyNominal)
                              [lang=java,file=TestCategorySelector.class]

groupheader {comment = #A subset of the testsuite with only nominal tags#}
selectorgroup pureNominal_Add2C
              [groupid=AddTwoCoins, selectorid=keepPureNominalTests, us=true]
```

The filtered test suite includes 9 test cases, corresponding to all combinations of two valid coins.

The `TestCategorySelector` has a single parameter `criterion` of type `String` which can take the following values : "OnlyNominal", "OnlyRobustness", "OneOrMoreNominal", "OneOrMoreRobustness", "OneNominal", "OneRobustness" as defined in the following table.

"OnlyNominal"	keeps the test cases whose `testCategory` annotations all include the word "nominal" in upper or lowercases
"OnlyRobustness"	keeps the test cases whose `testCategory` annotations all include the word "robustness" in upper or lowercases
"OneOrMoreNominal"	keeps the test cases whose `testCategory` annotations include at least once the word "nominal" in upper or lowercases
"OneOrMoreRobustness"	keeps the test cases whose `testCategory` annotations include at least once the word "robustness" in upper or lowercases
"OneNominal"	keeps the test cases whose `testCategory` annotations include only once the word "nominal" in upper or lowercases
"OneRobustness"	keeps the test cases whose `testCategory` annotations include only once the word "robustness" in upper or lowercases

For example, another group which now only includes robustness tags, and also unfolds into 9 test cases.

```
// This selector keeps only the test cases which include only robustness tags.
selector keepStrictRobustnessTests (String criterion=OnlyRobustness)
                                   [lang=java,file=TestCategorySelector.class]

groupheader {comment = #A subset of the testsuite with only robustness tags#}
selectorgroup strictRobustness_Add2C
              [groupid=AddTwoCoins, selectorid=keepStrictRobustnessTests, us=true]
```

5.4.2 Test suite reduction based on annotations

Annotations can also be used to guide the reduction of the test suite. `HeaderCombinationsSelector` selects a reduced test suite which covers all combinations of annotations.

```
selector tagCombAllSelection [lang=java,file=HeaderCombinationsSelector.class]

selectorgroup tagsCombAllSelection_Add2C
              [groupid=AddTwoCoins, selectorid=tagCombAllSelection, us=true]
```

The resulting test suite includes only 4 test cases (covering all combinations of two tags), and covers 95,5% of the `coin` method.

By default, it considers all types of tags. It is also possible to provide a list of kinds of tags considered for the reduction, like in the following example where we restrict tags to `testCategory` and `otherTag`.

```
selector tagCombListSelection (String attr_names="testCategory,otherTag")
                              [lang=java,file=HeaderCombinationsSelector.class]


groupheader {comment = #This will select a subset of the test suite
                        on the basis of the tags. Two tags are considered
                        in this case: testCategory and otherTag.#}
selectorgroup tagsCombListSelection_Add2C
              [groupid=AddTwoCoins, selectorid=tagCombListSelection, us=true]
```

Since `otherTag` does not appear in the original group, this gives the same result as for considering `testCategory` only.

It can be expected that this selector can be rather efficient to provide a reduced test suite, provided that enough information is given in the annotations.

5.5 Selectors based on the contents of the test cases

5.5.1 Directed random selection

`SelectorTreeRandom` is a random selector which implements the algorithm proposed in [DLdB07].

This selector performs a random selection, but makes sure that all "shapes" of test cases are represented in the final test suite. In order to illustrate this selector, we need a new group definition, available here.

```
group AddThreeCoins[us=true] {
  VendingMachine vm = new VendingMachine();
  vm.coin(@Coins){1,3};
  }
```

In this example, we perform one to three calls to `coin`, with the same definition for group `Coins` as in `AddTwoCoins`. The unfolding of this group produces 258 test cases, corresponding to:

- 6 test cases with a single call to `coin` (2%).
- 36 test cases with two calls to `coin` (14%).
- 216 test cases with three calls to `coin` (84%).

If we perform a random selection on this group, there is little chance that we will select a test case with a single call to `coin`. For example, the following definitions produce a group of 10 test cases, chosen randomly, as presented in section 5.2.

```
selector randomSelection (int nb=10, int percent=101, long seed=12345)
                         [lang=java,file=SelectorRandom.class]

roupheader {comment = #This is a random reduction of 10 test cases of AddThreeCoins.#}
selectorgroup random10casesSelection_Add3C
              [groupid=AddThreeCoins, selectorid=randomSelection, us=true]
```

The resulting test suite includes 2 test cases with two calls to `coin` and 8 test cases with three calls. As expected, there is little chance to select a test case with a single call to `coin`.

`SelectorTreeRandom` guarantees that each "shape" is represented in the selection. The shape of a test case corresponds to the sequence of method calls while forgetting the parameters. In our example, we have three shapes, corresponding to the number of calls to `coin` in the test case. The selector tries to give an equal number of test cases for each shape, unless it covers all elements of a given shape. The following lines correspond to a call to `SelectorTreeRandom`, in order to select 10 test cases.

```
// Selects 10 test cases randomly but preserving samples of each test shape.
selector randomTreeSelection (int nb=10, int percent=101, long seed=12345)
                             [lang=java,file=SelectorTreeRandom.class]

groupheader {comment = #This is a random reduction of 10 test cases of
                       AddThreeCoins preserving the "shape" of the pattern.#}
selectorgroup random10casesTreeSelection_Add3C
              [groupid=AddThreeCoins, selectorid=randomTreeSelection, us=true]
```

The resulting test suite includes 3 test cases with three calls, four test cases with two calls, and 3 test cases with a single call. This example shows that `SelectorTreeRandom` keeps all shapes in the resulting test suite. For more information on this selector, see [DLdB07].

5.6 Other selectors

Recent work on Tobias will soon allow to combine selectors, i.e. to use selectors at any stage of the unfolding process. This will give rise to new selectors, applied not to test cases, but to intermediate constructs such as cartesian product of values. These new selectors will be announced as soon as they will be available.

Chapter 6

Limits of Tobias-on-line

We have put some restrictions on the size of the input and output files of Tobias.

- The input file is limited to 150 kBytes. This should not be a problem in most cases since Tobias schemas are usually compact.
- The output file (actually, the size of an intermediate output file) is limited to 4 MBytes. It is actually very easy to write an "explosive" test pattern which will result in millions or billions of test cases. In order to avoid saturation of our disk space and CPU time, we have put this limit. In such cases, the tool will produce a result but truncate it.

If you feel that these limits are an obstacle to your evaluation of Tobias, please contact Yves.Ledru@imag.fr.

Bibliography

[DLdB07] Frdric Dadeau, Yves Ledru, and Lydie du Bousquet. Directed random reduction of combinatorial test suites. In *RT'07: Proceedings of the 2nd international workshop on Random testing*, pages 18–25, New York, NY, USA, 2007. ACM.

[LVTdB12] Yves Ledru, German Vega, Taha Triki, and Lydie du Bousquet. Test Suite Selection Based on Traceability Annotations - tool demonstration. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, Essen, Germany, sep 2012. ACM.

Appendix A

TSLT Syntax

The full BNF of the TSLT Syntax is available at http://tobias.liglab.fr/doc/TSLTSyntax/tsl.html