Filtering TOBIAS combinatorial test suites

Y. Ledru, L. du Bousquet, O. Maury, P. Bontron

Laboratoire Logiciels, Systèmes, Réseaux - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex – France
Yves.Ledru, Lydie.du-Bousquet@imag.fr

Abstract

TOBIAS is a combinatorial testing tool, aimed at the production of large test suites. In this paper, TOBIAS is applied to conformance tests for model-based specifications (expressed with assertions, pre and post-conditions) and associated implementations. The tool takes advantage of the executable character of VDM or JML assertions which provide an oracle for the testing process.
Executing large test suites may require a lot of time. This paper shows how assertions can be exploited at generation time to filter the set of test cases, and at execution time to detect inconclusive test cases.

Keywords: combinatorial testing, model-based specifications, VDM, JML

Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron. Filtering TOBIAS combinatorial test suites. In *Proceedings of ETAPS/FASE'04 - Fundamental Approaches to Software Engineering*, pp. 281-294, Vol. 2984 of LNCS, Springer-Verlag, Barcelona, 2004

Filtering TOBIAS combinatorial test suites

Yves Ledru, Lydie du Bousquet, Olivier Maury, Pierre Bontron

Laboratoire Logiciels, Systèmes, Réseaux - IMAG
B.P. 72 - F-38402 - Saint Martin d'Hères Cedex - France
{Yves.Ledru, lydie.du-bousquet}@imag.fr

Abstract. TOBIAS is a combinatorial testing tool, aimed at the production of large test suites. In this paper, TOBIAS is applied to conformance tests for model-based specifications (expressed with assertions, pre and post-conditions) and associated implementations. The tool takes advantage of the executable character of VDM or JML assertions which provide an oracle for the testing process. Executing large test suites may require a lot of time. This paper shows how assertions can be exploited at generation time to filter the set of test cases, and at execution time to detect inconclusive test cases.
Keywords: combinatorial testing, model-based specifications, VDM, JML

1 Introduction

Software testing appears nowadays as one of the major techniques to evaluate the conformance between a specification and some implementation. Some may argue that testing only reveals the presence of errors and that conformance may only be totally guaranteed by formal proof, exhaustive testing or a combination of both techniques. Unfortunately, such techniques are often very difficult to apply. In such cases, testing may contribute to increase the confidence that the implementation conforms to its specification. Confidence may result from coverage measurements, from the principles of the test synthesis or selection technique, from the size of the test suite, or from the expertise of the test engineers.

Industrial experiments [5] have shown that test cases within a large test suite often feature a high level of similarity. Many test cases correspond to the same sequence of method calls, with different parameters. Producing these test cases is a repetitive task that reveals the need for appropriate tool support.

From these observations, we have developed the TOBIAS test generator[1] which is aimed at the production of a large set of similar test cases. TOBIAS starts from a test pattern and a description of its instantiations. The tool then unfolds the pattern into a large set of test cases which can be output according to the format of several test tools: calls to VDM operations [12] for VDMTools,

[1] TOBIAS was developed within the COTE project, with the support of the French RNTL program. The COTE project gathered Softeam, France Telecom R&D, Gemplus, IRISA and LSR/IMAG.

Java test cases for JUnit[9] and JML specifications [8, 10, 11], and test purposes for TGV [7].

TOBIAS is a typical example of combinatorial testing tool. Its originality is to deal with sequences of method calls, instead of only combination of parameter values. This allows to use the tool with systems that require several interactions before reaching some "interesting" states. It also allows to design test cases in terms of the behavior that has to be exercised.

This paper gives an introduction to TOBIAS. Sect. 2 recalls the principles of conformance testing using executable model-based specifications. Then Sect. 3 gives a quick presentation of the tool and reports on its capability to find errors, on the basis of a simple example, and from the results of industrial experiments. The intrinsic limitation of the tool is that it is subject to combinatorial explosion. Sect. 4 presents two kinds of filters that can be used with TOBIAS to help master the size of test suites. Finally, Sect. 5 draws the conclusions and perspectives of this work.

2 Conformance testing with model-based specifications

2.1 Checking conformance with model-based specifications

Model-based specifications describe a system in terms of invariant properties, pre- and postconditions. Some model-based languages, e.g. VDM and JML, have an executable character. It is thus possible to use invariant assertions, as well as pre- and postconditions as oracle for a conformance testing process. VDM assertions can be evaluated in the VDMTools environment against the VDM version of the specified code [6], or compiled into C++ [1]. JML specifications are translated into Java, added to the code of the specified program, and checked against it. The executable assertions are thus executed before, during and after the execution of a given operation (Fig. 1).

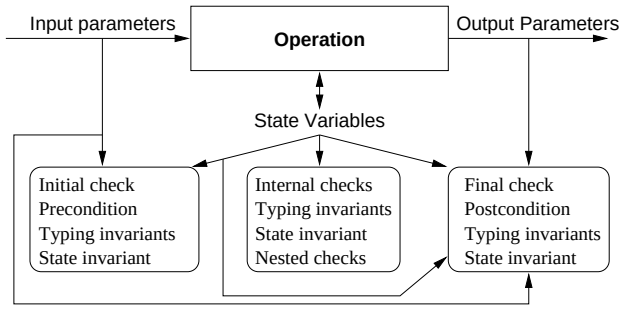

Fig. 1. Dynamic checks associated to operation execution

One should note that the specification invariants are not exactly checked at the same instants in JML and VDM. In VDM, invariants are evaluated after each statement. In JML, invariants are properties that have to hold in all *visible states*. A visible state roughly corresponds to the initial and final states of any method invocation [8].

When an operation is executed in one of those environments, three cases may happen (Fig. 1):

- All checks succeed: the behavior of the operation conforms with the specification for these input values and initial state. The test delivers a PASS verdict.
- An intermediate or final check fails: this reveals an inconsistency between the behavior of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict.
- An initial check fails: in this case, performing the whole test will not bring useful information because it is performed outside the specified behavior. This test delivers an INCONCLUSIVE verdict.
  For example, $\sqrt{x}$ has a precondition that x has to be positive. Therefore, a test of a square root method with -1 leads to an INCONCLUSIVE verdict.

2.2 A small example in VDM and JML

Let us study a simple example of buffer system (Fig. 2). This system is composed of three buffers. The specification models only the number of elements present in the buffers. A buffer is then modeled with an integer value, which indicates the number of elements in it. The system state is given by the three variables `b1`, `b2` and `b3`.

The maximum size of the system is 40 elements. The system has to distribute the elements amongst the buffers so that: buffer `b1` is smaller than `b2`, which is smaller than `b3`. The difference between `b1` and `b3` should not exceed 15 elements. These constraints leave some freedom on the way to share the elements between buffers. For example, 30 elements can be stored as `b1=5 b2=10 b3=15` or as `b1=8 b2=10 b3=12`.

Three methods are set to modify the systems:

- `Init` resets all buffers to zero.
- `Add(x)` increases the total number of elements of the system of a strictly positive number (`x`) (i.e. it adds `x` elements to the buffers; these elements are distributed in `b1`, `b2`, and `b3`).
- `Remove(x)` decreases the total number of elements in the system of a strictly positive number (`x`) (i.e. it removes `x` elements from the buffers).

The specifications of `Add` and `Remove` keep some implementation freedom: the buffer in which the elements have to be added/removed is not set. For example, if the current state is `8 10 12`, and if 2 elements have to be added, the final state could be `8 10 14`, `8 12 12` , but also `6 12 14`.

```
------------------ VDM ------------------
state buffers of
b1 : nat
b2 : nat
b3 : nat

inv mk_buffers(b1,b2,b3) ==
b1+b2+b3<=40 and 0<=b1 and b1<=b2 and b2<=b3 and b3-b1<=15
init B == B = mk_buffers(0,0,0)
end

operations
Init:() ==> ()
Init() == ...
post b1+b2+b3=0
;
Add: nat ==> ()
Add(x) == ...
pre x<=5 and b1+b2+b3+x<=40
post b1+b2+b3 = b1~+b2~+b3~+x
;
Remove: nat ==> ()
Remove(x) == ...
pre x<=5 and x<=b1+b2+b3
post b1+b2+b3 = b1~+b2~+b3~-x
;
------------------ JML ------------------
public class Buffer{
    public int b1;
    public int b2;
    public int b3;

    /*@ public invariant
      @ b1+b2+b3<=40 && 0<=b1 && b1<=b2 && b2<=b3 && b3-b1<=15; */

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0; */
    public Buffer(){}

    /*@ requires true;
      @ modifies b1, b2, b3;
      @ ensures b1==0 && b2==0 && b3==0;   */
    public void Init(){...}

    /*@ requires x<=5 && b1+b2+b3+x<=40 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)+x;
    */
    public void Add(int x){...}

    /*@ requires x<=5 && x<=b1+b2+b3 && x>=0;
      @ modifies b1, b2, b3;
      @ ensures b1+b2+b3==\old(b1+b2+b3)-x;  */
    public void Remove(int x){...}
}
```

Fig. 2. Buffer example specification in VDM and JML

2.3 Test cases

We define a test case as a sequence of operation calls. For example, the following test case initializes the buffer system, adds two elements and removes one of them.

```
TC1 : Init() ; Add(2) ; Remove(1)
```

Each operation call may lead to a PASS, FAIL or INCONCLUSIVE verdict. As soon as a FAIL or INCONCLUSIVE verdict happens, we choose to stop the test case execution and mark it with this verdict. A test case that is carried out completely receives a PASS verdict.

For example, in the context of the above specification, the test cases TC2 and TC3 should produce an INCONCLUSIVE verdict. If test TC4 is executed against a "correct" implementation, it should produce a PASS.

```
TC2 : Init() ; Add(-1)
TC3 : Init() ; Add(2) ; Remove(3)
TC4 : Init() ; Add(3) ; Remove(2) ; Remove(1)
```

3 TOBIAS

TOBIAS is a test generator based on combinatorial testing [4]. Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit [3] generates test cases which consist of a single call to a class constructor, followed by a single call to one of the methods. Each test case corresponds to a combination of the parameters of the constructor and a combination of the parameters of the method.

3.1 Principles of TOBIAS

TOBIAS adapts combinatorial testing to the generation of sequences of operation calls. This allows to reach states that do not correspond to a single call to a constructor. It also allows to design tests in terms of behaviors rather than states. For example, in the specification of the buffers, the initial state is fixed (`0 0 0`), and it is not possible to add more than 5 elements at a time. Therefore a rather long sequence is needed (at least 8 operations) to test the behavior of the system at its limits (40 elements).

The input of TOBIAS is composed of a test pattern (also called test schema) which defines a set of test cases. A pattern is a bounded regular expression involving the operations of the VDM or JML specification. TOBIAS unfolds the pattern into a set of sequences, and then computes all combinations of the input parameters for all operations of the pattern.

The patterns may be expressed in terms of *groups*, which are structuring facilities that associate a method, or a set of methods to typical values. For example, let us consider schema `S1`:

$$\begin{cases} \texttt{Init() ; Add\_Gr} \\ \text{with } \texttt{Add\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

`Add_Gr` is a set of 5 instantiations of calls to the method `Add`. The pattern `S1` is unfolded into 5 test sequences:

```
S1-TC1 : Init() ; Add(1)
S1-TC2 : Init() ; Add(2)
...
S1-TC5 : Init() ; Add(5)
```

Groups may also involve several operations. Let `S2` and `S2'` be two other examples of schemas:

$$\begin{cases} \texttt{S2} = \texttt{Init() ; Modify\_Gr\^{}\{1..2\}} \\ \texttt{S2'} = \texttt{Init() ; Add(2) ; Modify\_Gr\^{}\{1..2\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

`Modify_Gr` is a set of (5+3)=8 instantiations. The expression `^{1..2}` means that the group is repeated 1 to 2 times. The patterns `S2` and `S2'` are unfolded into 8+(8*8)=72 test sequences:

```
S2-TC1 : Init() ; Add(1)
  ...
S2-TC8 : Init() ; Remove(5)
S2-TC9 : Init() ; Add(1) ; Add(1)
  ...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
          ----------------------
S2'-TC1 : Init() ; Add(2); Add(1)
  ...
S2'-TC72 : Init() ; Add(2); Remove(5) ; Remove(5)
```

Group definitions may be reused in several schemas, leading to some level of modular construction.

3.2 Finding errors with Tobias

Let us consider the buffer problem specification. We have proposed an implementation, containing one error: the `Remove` operation can set one of the three buffers to a negative value while keeping the total number of elements positive, which is forbidden by the specification invariant. This solution was implemented in VDM and Java. We executed the tests corresponding to the schemas `S2`, `S2'`, `S3`, and `S4`.

$$\begin{cases} \texttt{S3} = \texttt{Init() ; Add(5)\^{}7 ; Modify\_Gr\^{}\{1..2\}} \\ \texttt{S4} = \texttt{Init() ; Add\_Gr ; Modify\_Gr\^{}\{1..3\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \\ \text{and } \texttt{Add\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \end{cases}$$

The schema `S2'` was introduced in order to decrease the number of inconclusive verdicts of schema `S2`. The schema `S3` aims at testing the behavior of

the application at the "limits", i.e. when the buffer system is quite full. The schema `S4` was built to produce lots of test sequences (some kind of "brute force approach"). The following table gives the verdicts of the various test cases.

Schema	Test cases	Pass	Inconclusive	Fail
`S2`	72	39	33	0
`S2'`	72	48	22	2
`S3`	72	57	15	0
`S4`	2920	1887	773	260

As expected, the error is detected (by schemas `S2'` and `S4`). `S3` is aimed at testing full buffers and can not reveal the error; `S2` is a small test suite with a lot of inconclusive test cases, which does not achieve enough exhaustiveness to find the error.

This example shows that TOBIAS test suites are able to find errors. Here the error was not straightforward, and small test suites such as `S2` are not sufficient to detect it (actually, the error may only happen if two `Add` operations have been performed). Longer test sequences are needed, such as the ones generated by `S4`.

We have carried out several experiments with TOBIAS. In [12], we report on a VDM case study. This case study showed that the development of a TOBIAS test suite requires the same amount of effort as a simple manual test suite. It also shows that since TOBIAS test suites achieve more exhaustiveness (by exercising all combinations in the schema), they reveal some errors that are often overlooked by manual test suites.

3.3 Industrial case study

Two experiments were also carried out on an industrial case study provided by Gemplus (a smart card manufacturer). The case study is a banking application which deals with money transfers. It has been produced by Gemplus Research Labs and is somehow representative of java applications connected to smart cards. The application user (i.e. the customer) can consult his accounts and make some money transfers from one account to another. The user can also record some "transfer rules", in order to schedule regular transfers. These transfer rules can be either saving or spending rules.

The case study is actually a simplified version of an application already used in the real world. The code length is 500 lines. The specification was given in JML. Most preconditions are set to true. Since the application deals with money, and since some users may have malicious behaviors, the application is expected to have defensive mechanisms. Thus, it is supposed to accept any entry, but it should return error messages or raise exceptions if the inputs are not those expected for a nominal behavior. It is a typical example of *defensive programming* style. This means that test cases do not produce INCONCLUSIVE verdicts.

Two testing experiments with TOBIAS were carried out from this case study. The first one was carried out by a Gemplus team. They have first used their

internal testing methodology to elaborate an informal test plan. It includes 40 nominal "scenarios" (a scenario is an informal description of a test case). It was possible to abstract those scenarios and express them with only 5 TOBIAS schemas, which were unfolded into 1900 executable test cases. This experiment shows that TOBIAS schemas are more compact than test cases. Moreover, by abstracting test cases into schemas, we ended up with more general schemas than the original scenarios, resulting into 50 times more test cases.

This experiment was considered as a success by our industrial partner. On the one hand, TOBIAS schemas were perceived as an interesting structuring mechanism for the design of tests. On the other hand, the tool allows to complete the original test suites by achieving some kind of exhaustiveness.

The second experiment was carried out by our research team. From the informal requirements, we deduced 17 TOBIAS schemas, mainly to simulate malicious behaviors. They were unfolded into 1100 test sequences, representing 40 000 Java code lines (for JUnit). It took 6 person-day to analyze the specification, produce the abstract scenarios, execute the tests and analyze the traces. (The test suite execution time by itself takes only 1 hour.) The execution of the test cases revealed 16 errors, in either the Java code or in the corresponding JML specification. A discussion with the Gemplus team after the experiment showed that we discovered most of the errors in the code. The only remaining error was impossible to detect because the JML specification did not address this feature of the system.

3.4 Conclusion

TOBIAS is a combinatorial tool that instantiates a large set of test sequences from an abstract description. It aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool can also be used in order to express existing test sequences in a more abstract way, which helps the test engineer to structure his test suite.

Several experiments have shown that it is well-suited for a conformance testing activity, in conjunction with executable model-based specification. These experiments include both research and industrial case studies. In all these case studies, the tool allowed to detect errors in the implementation under test.

4 Handling large test suites

The major strength of TOBIAS is also its main weakness. The combinatorial approach allows to produce large test suites, whose systematic character helps to detect errors. But the size of the test suite may also become a problem when too many resources are needed to run the tests and analyze their results. The first way to avoid combinatorial explosion is to design test schemas with great care. By avoiding useless calls in the schema and by keeping the possible values of a parameter to a minimum, we were able to control the number of generated test

cases in the experiments we led so far. Nevertheless, two additional mechanisms have proved to be useful to reduce the amount of tests. They filter the set of test cases either at execution or at generation time.

The typical size of a TOBIAS test suite ranges from hundreds to thousands of tests. Today, the largest test suite generated by the tool counts about 40 000 test cases. Several experiments have shown that such test suites include a large number of inconclusive test cases. For instance, schema `S4` leads to 773 INCONCLUSIVE test cases. Although these are useful to test preconditions, their execution may require a significant amount of time, and it makes sense to try to eliminate some of them.

This section will discuss two techniques, based on predicates, that are used to control or cope with the size of TOBIAS test suites.

- Filtering at the execution time: a test driver takes into account the results of the oracle and filters out test cases with a prefix that has already failed one of the checks.
- Filtering at the generation time: the test case generator can take into account a predicate which filters out test cases whose input parameters do not fulfill the predicate.

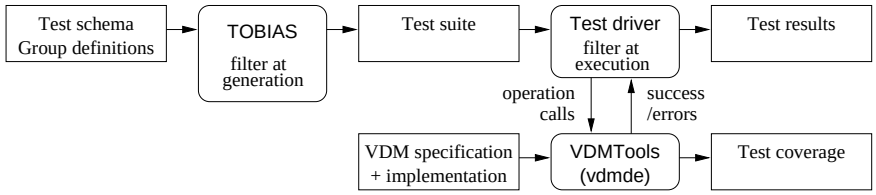

Fig. 3. Exploitation of TOBIAS generated tests in the VDM environment

4.1 Filtering at execution time

By construction, TOBIAS test suites are made up of similar test cases. One of the possible similarities is that several test cases may share a common prefix. For example, schema `S2'` includes 9 test cases which start with prefix `init() ; Add(2) ; Remove(5)`. If the execution of the prefix is erroneous (or INCONCLUSIVE) for any of the 9 test cases, and if the implementation is deterministic, the 8 remaining test cases will also exhibit an erroneous (or INCONCLUSIVE) prefix. It is useless to execute the test cases with this prefix.

Therefore, we have developped test drivers that take this property into account. Every erroneous prefix is stored during the execution of the test suite.

Before playing a new test case, it is compared to the stored erroneous prefixes and discarded if it matches any of them.

The nice property of this filtering scheme is that it takes advantage of the executable VDM/JML assertions (mainly preconditions) to help filter the test suite. It does not require additional input from the user. Of course, this filtering scheme is better suited to specifications that include strong preconditions. It will not provide any benefit for specifications which adopt a defensive programming style, with all preconditions set to true.

Buffer case study

The following table shows the execution time for the 4 test suites of the buffer problem. The tests were executed on a Pentium III/500MHz/128Mb linux machine.

					VDM		JML	
Schema	Test cases	Pass	Incon-clusive	Fail	Exec. Time	Exec. Time with filtering	Exec. Time[2]	Exec. Time with filtering
S2	72	39	33	0	2 s.	1.205 s.	0.709 s.	0.134 s.
S2'	72	48	22	2	2 s.	1.674 s.	0.524 s.	0.157 s.
S3	72	57	15	0	6 s.	4.596 s.	0.400 s.	0.269 s.
S4	2920	1887	773	260	2min 05 s.	9.930 s.	14.307 s.	1.352 s.

The tests were executed with filtering and non-filtering test drivers. As expected, the optimized drivers execute the test suites quicker than the original test drivers. The speed up is more important when there are many INCONCLUSIVE (or FAIL) verdicts. Both kinds of drivers reveal the implementation error.

Banking application

The banking application is a typical example of defensive programming. The preconditions of operations are usually set to `true`, in order to face all kinds of unexpected inputs. With such applications, the test cases never end up with an INCONCLUSIVE verdict. Therefore, filtering at execution time can only take into account the prefixes which lead to a FAIL verdict. In the banking application, this corresponds to a small number of tests (at most 1%). Hence, filtering at execution time does not lead to a significant speed-up.

The following experiment was led to make sure that the filtering mechanism did not slow down execution significantly when there are no INCONCLUSIVE test cases. We have executed the tests with both JUnit and our driver for Java, on a Pentium III/500MHz/128Mb Windows machine. This one has some limitations. For instance, it is not possible to set several instantiations for a constructor method in the same test suite (a test suite here is the set of test cases derived from one schema). As a result, some the test suites were not executable with our

[2] With JML-Junit environment.

driver. The following table shows the execution time for the tests. As it can be noticed, the execution time with our driver is shorter than with JUnit.

Schema	nb of tests	with Junit	with our driver	Speedup
One account creation	162	0.671 s.	0.410 s.	0.39
Several account creations	96	0.401 s.	0.030 s.	0.93
One account deletion	30	0.160 s.	0.060 s.	0.63
Several account deletions	512	2.553 s.	0.641 s.	0.75
Several transfers	1	0.050 s.	0.060 s.	-0.20
Incorrect transfer (1)	16	0.251 s.	0.240 s.	0.04
Incorrect transfers (2)	60	0.520 s.	0.601 s.	-0.16
Incorrect transfers (3)	2	0.040 s.	0.030 s.	0.25
Use of infinity values	12	0.141 s.	0.110 s.	0.22
12 digit numbers	4	0.070 s.	0.060 s.	0.14
Transfers and account deletion	12	0.140 s.	0.080 s.	0.43
Transfer rules (1)	120	2.163 s.	2.204 s.	-0.02
Transfer rules (2)	96	1.733 s.	1.592 s.	0.08
Transfer rules (3)	12	0.341 s.	0.180 s.	0.47
Transfer rules (4)	8	0.301 s.	0.110 s.	0.63
Saving rule and account deletion	3	0.591 s.	0.050 s.	0.91
Spending rule and account deletion	3	0.711 s.	0.080 s.	0.87

Our experiments (the banking application and the buffers) show that our test driver is faster than JUnit. There are several reasons:

- algorithmic reasons: when a large number of tests have the same prefix, and when this prefix leads to a FAIL or an INCONCLUSIVE verdict, these tests (which are amongs the longest of the test suite) are not executed with our driver. For example, in the `S4` schema, 760 tests are discarded, which corresponds to a quater of the tests.
- technical reasons: JUnit has a generic character and uses introspection/reflection facilities to discover th tests stored in a class. Our test driver is directly compiled from the test suite and does not have to find this information. Moreover, we suspect that the graphical interface of JUnit (which were used during our tests) also slows down the execution. The banking experiment, which never leads to INCONCLUSIVE verdicts, shows that these technical reasons alone result in significant speedups.

4.2 Filtering test cases at generation time

The previous section has shown that preconditions and other assertions could filter a lot of INCONCLUSIVE test cases at execution time. TOBIAS provides an other mechanism which allows to eliminate some test cases at generation time, using a VDM predicate as a filter.

Let us consider again the schemas `S2` to `S4`. A lot of the inconclusive verdicts are due to the fact that the total number of removed elements is greater than the

total number of added elements. One idea is to complete the schema definitions with a constraint on the combination of parameters. Schema `S2` was defined as:

$$\begin{cases} \texttt{S2} = \texttt{Init() ; Modify\_Gr\^{}\{1..2\}} \\ \text{with } \texttt{Modify\_Gr} = \{\texttt{Add}(x) | x \in \{1,2,3,4,5\}\} \cup \{\texttt{Remove}(y) | y \in \{1,3,5\}\} \end{cases}$$

and unfolds into 72 test cases:

```
S2-TC1 : Init() ; Add(1)
...
S2-TC72 : Init() ; Remove(5) ; Remove(5)
```

Let `add1` be the sequence of `x` parameters associated to each call to `Add` in a given test case, and `del1` be the sequence of `y` parameters associated to each call to `Remove`. For example, test case `S2-TC1` corresponds to `add1 = [1]` and `del1 = []`, and test case `S2-TC72` corresponds to `add1 = []` and `del1 = [5,5]`. The following VDM constraint expresses that the sum of the elements of `add1` is greater than or equal to the sum of elements of `del1`.

```
S2_constraint : () ==> bool
S2_constraint() == (
  dcl sommeAdd : nat:=0;
  dcl sommeDel  : nat:=0;
  for a in add1 do (sommeAdd:= sommeAdd+a;);
  for d in del1 do (sommeDel:= sommeDel+d;);
return(sommeAdd>=sommeDel) )
```

TOBIAS has been extended to generate sequences `add1` and `del1` for each unfolded test case, and then pass them to a VDM interpreter which evaluates the constraint. Test cases which fail to verify the constraint are discarded from the generated test suite.

With schema `S2` and `S2_constraint`, the resulting test suite only features 48 test cases, among wich only 9 lead to INCONCLUSIVE verdict (instead of 33).

This first example has shown that constraints can get rid of INCONCLUSIVE tests at generation time. But this technique requires the test engineer to write the constraint, while filtering at execution time took advantage of the existing preconditions. Still, filtering at generation time is an interesting mechanism, because constraints can be motivated by other concerns than simply ruling out INCONCLUSIVE tests, as will be shown in the following example.

Application to the banking problem

It was already mentioned that the banking problem does not lead to INCONCLUSIVE verdicts. Still, constraints can be used in this case study to master combinatorial explosion by adding further test hypotheses.

One of the 17 schemas is named "Several account deletions". It is unfolded into 512 test cases, which is actually the highest number of test cases in this experiment. This schema is defined as follows:

```
Create^{2..2}; Delete^{3..3}; CreateErr; Delete
```

where `Create` has only one instance, `CreateErr` has two instances and `Delete` has four instances corresponding to four possible values of its only integer parameter. This schema is unfolded thus into 1*1*4*4*4*2*4= 512 test cases.

In order to reduce this size, one may wish to express additional test hypotheses. For example, `Delete` can be instantiated as `Del(10)`, `Del(11)`, `Del(12)`, or `Del(13)`. A first test hypothesis may be that the order of the first three instances of `Delete` is not significant. Therefore the following test sequences are equivalent for the tester:

```
Create; Create; Del(10); Del(11); Del(12); CreateErr; Del(10)
Create; Create; Del(12); Del(11); Del(10); CreateErr; Del(10)
```

Let `del1` be the sequence of parameters associated to the first three calls to `Delete`, the following constraint expresses that only the sequence where the parameters appear in ascending order will be kept:

```
forall val1, val2 in set inds del1 &
              val1<val2 => del1(val1)<=del1(val2)
```

Another test hypothesis (here a regularity hypothesis) is that it does not make sense to try to delete the same account more than twice. This hypothesis can be enforced if the four `Delete` calls refer to at least three different accounts. Let `del2` be the single element sequence corresponding to the fourth call to `Delete`, this constraint can be expressed as:

```
card(elems(del1) union elems(del2))>=3
```

These hypotheses are then grouped into the following constraint.

```
Delete_C : () ==> bool
Delete_C () == (
 return(
        (forall val1, val2 in set inds del1 &
              val1<val2 => del1(val1)<=del1(val2))
        and
         card(elems(del1) union elems(del2))>=3
        )
)
```

When TOBIAS takes this constraint into account, the number of unfolded test cases is reduced from 512 to 80. From a test engineer point of view, this reduction may be interesting since it results in a better balance of the whole test suite. Thus this test schema no longer appears as the most significant one.

5 Conclusion

This paper has presented TOBIAS, a test case generator based on the combinatorial unfolding of test schemas. It has shown how the tool can be combined with executable model-based specifications in a conformance testing process. TOBIAS aims to be a simple and easy to use tool for combinatorial testing which supports and amplifies the creative work of a test engineer. The tool has proved to be useful to detect errors in several case studies, including an industrial experiment where Java code was tested against a JML specification.

Other tools adopt a combinatorial testing approach in combination with model-based specifications. Korat [2] and JML-JUnit [3] generate combinations of a call to a constructor followed by a single call to one of the methods of the class. Korat uses an elaborate generator to cover a wide range of calls to the constructor. TOBIAS adds the possibility to express a sequence of method calls in the test schema, allowing to reach states that cannot be created with the constructor and to express tests on the basis of a behavior.

This paper has also presented filters that help master the size of the generated test suites. Filtering at execution time is an interesting feature because it does not require additional inputs from the test engineer. It allows to filter a significant percentage of the tests for specifications with strong preconditions.

Filtering at generation time requires that the test engineers express some constraints on the schema parameters. But it is a more flexible filtering mechanism and allows to translate test hypotheses into filtering constraints.

Perspectives Several improvements may be considered when filtering at generation time. On the one hand, several typical constraints could be added as primitives of the schema language. For example, a variant of iteration of a method could mandate parameter values to be all different, or to appear in ascending order. On the other hand, a library of constraints could be developed to express frequently used testing constraints. Moreover, since constraints translate test hypotheses, the library could be structured in terms of these higher level concerns.

Still, improvements in filtering capabilities should not prevent the test engineers from handling combinatorial explosion by a careful design of their test schemas. Further methodological advances are needed to guide the elaboration of test schemas. We expect that further experiments with TOBIAS with help us to progress in that direction.

References

1. B.K. Aichernig. Automated black-box testing with abstract VDM oracles. In M. Felici, K. Kanoun, and A. Pasquini, editors, *SAFECOMP'99*, LNCS 1698, pages 250–259. Springer, 1999.

2. C. Boyapati, S. Khurshid, and D. Marinov. Korat: Automated testing based on java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis*, Rome, 22–24 July 2002. IEEE.
3. Y. Cheon and G.T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In B. Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming, 16th European Conference, Malaga, Spain, Proceedings*, LNCS 2474, pages 231–255. Springer, 2002.
4. D.M. Cohen, S.R. Dalal, J. Parelius, and G.C. Patton. The combinatorial design approach to automatic test generation. *IEEE Software*, 13(5):83–88, 1996.
5. L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance Testing from UML specifications, Experience Report. In Gesellschaft für Informatik, editor, *p-UML workshop, Lecture Notes in Informatics*, volume P-7, pages 43–56, Toronto, 2001.
6. The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.
7. T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*, LNCS 1633. Springer, 1999.
8. The Java Modeling Language (JML) Home Page. http://www.cs.iastate.edu/ leavens/JML.html.
9. JUnit. http://www.junit.org.
10. G.T. Leavens, A.L. Baker, and C. Ruby. JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
11. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06q, Iowa State University, Department of Computer Science, June 2002.
12. O. Maury, Y. Ledru, P. Bontron, and L. du Bousquet. Using TOBIAS for the automatic generation of VDM test cases. In *Third VDM Workshop (in conjunction with FME'02)*, july 2002.