Using TOBIAS for the automatic generation of VDM test cases

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

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

Abstract. TOBIAS is a tool for the automatic generation of a large set of similar test cases from a test pattern. The goal of the tool is to generate larger and more exhaustive testsuites than a classical manual process. This paper presents the principles of TOBIAS and reports on experiments to combine TOBIAS with VDMTools, in order to test implementations against VDM specifications. Our experiments compare the effort of producing tests with TOBIAS with a manually developed testsuite. It shows that the tool detects more errors than the manual testsuite for a similar test design effort.

1 Introduction

Software testing is currently considered as one of the major V&V activities to ensure software quality. Automatic generation of test cases is one of the current challenges of automated software engineering. Due to the complexity of current applications, numerous test cases are needed for a given application in order to increase confidence in the testing process. In recent experiments [5], we have stated that test cases within a given testsuite often feature a high level of similarity. It is often the case that 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 adequate tool support.

This is the goal of the TOBIAS tool[2], developed within the COTE project[1]. Starting from a test pattern and the identification of groups of parameters and operations, TOBIAS generates a large set of similar test cases. The tool was initially build in order to generate test purposes for the TGV tool [8], but the experiments reported in this paper show that it can also be used as a test case generator.

This paper presents a combination of TOBIAS with VDMTools [7]. VDMTools provides a testing infrastructure where an implementation can be compared to some specification by executing it with given test cases. We will show how TOBIAS can be used to systematically generate a very large set of these test cases. We also report on a simple case study which tends to compare the effort required to use our tool with the effort spent to write a VDMTools testsuite.

[1] COTE is a french RNTL project that groups Softeam, France Telecom R&D, Gemplus, IRISA and LSR/IMAG.

2 Testing with VDMTools

VDM [9, 6] is a model-based specification language. Specifications are a combination of invariant definitions for types or variables, and pre- and post-conditions for operations. Such assertions can be compared to executable code: provided these assertions are carefully written, it is possible to evaluate invariants and pre-condition on the entry of an operation, and invariants and post-condition on its exit. This corresponds to conformance testing, where the evaluation of the assertions provides the oracle of the tests.

Conformance testing[3, 10] is a kind of black box testing. In black box testing[1], what happens inside operations is hidden. This is not exactly what happens with VDMTools where assertions are also checked inside operations.

- When a VDM operation is implemented in terms of other VDM operations, the pre- and post-conditions are evaluated for the nested calls also.
- Each modification of a state variable within the body of the operation leads to a new evaluation of the invariant.

VDMTools provides thus a combination of black box and white box testing. Moreover, it offers a measure of the coverage achieved by the testsuite in terms of a percentage of the specification that has been exercised by the testsuite. Each operation is composed of expressions located either in the pre- and post-conditions, or in the actual code that implements the operation. So, the test coverage of an operation is the number of expressions that are evaluated by the test suite divided by the total number of expressions that define the tested operation.

3 A case study

In order to illustrate and evaluate our approach, we have used a simple group management case study. A class of students is divided into small groups. These groups must conform to the following constraints:

- a group is composed of 3 to 5 students; if the class is less than 3 students, they must belong to the same group;
- the largest group has at most one more student than the smallest group;
- each student belongs to one and only one group.

This is expressed in the following VDM specification. First we declare the minimum and maximum sizes of groups as constants (`values`). In this specification we take advantage of the `lim` composite type to enforce a property of these constants. The invariant guarantees that when a group has reached the upper limit it is still possible to split it into two groups of the smallest size to add an new student. For example, a class of 5 students corresponds to a single group, and a class of 6 students to two groups. If the lower limit is set to 4, there is no way to share a class of 6 into valid groups.

```
values limit : lim = mk_lim(3,5)
types
 lim :: min : nat
        max : nat1
 inv mk_lim(min,max) == ((max+1)div 2) >= min;
```

Students are modelled as sequences of characters, group identifiers as strictly positive naturals, and type `table` is defined as a function (map) from students to group identifiers.

```
 student = seq of char;
 gr_id = nat1;
 table = map student to gr_id;
```

Having introduced these types, the state variables can now be defined. There is actually one single state variable `gr` which stores the assignment of students to groups. The state invariant guarantees that:

- the size of the largest group (`size_max_gr`) is lower than the maximum limit;
- the largest group has at most one more element than the smallest one;
- the size of the smallest group is greater than the minimum limit (provided that there are more than one group).

The state variable is initialized to the empty map. Functions `size_max_gr` and `size_min_gr` are specified elsewhere in the specification; for brevity reasons, they are not given here.

```
state groups of
        gr : table
inv mk_groups(gr) ==
        size_max_gr(gr) <= limit.max
        and
        (size_max_gr(gr) - size_min_gr(gr) <= 1)
        and
        if card rng gr > 1 then
                size_min_gr(gr) >= limit.min
        else true
init G == G = mk_groups({|->})
end
```

These types and state definition guarantee the properties stated at the begining of this section.

Several operations are defined in combination with this state.

- `load_table` loads a table in variable `gr`;
- `swap_two_students` exchanges two students, each of them going into the group of the other;

- `add_group` increments the number of groups, keeping the same students and modifying their assignment into groups;
- `add_student` adds a new student to the class; the operation may modify the group assignments of the other students if needed;
- `add_set_students` applies `add_student` to each student of the set given as parameter;
- `delete_student` removes one student from the class, and may modify the group assignments of the other students;
- `change_student` moves a student into a group whose number is given as parameter;
- `mix` and `mix_better` arbitrarily modify the group assignments;
- `print_groups` displays the students of the class as a set of sets of students.

Due to the strong invariant on `gr`, applying most of these operations may result in large modifications of the assignment of students to groups. The specifications are written in a non-deterministic manner to allow various ways of modifying these assignments. Some examples of operation specifications are given in the sequel of this paper.

4 Testing specifications with VDMTools

Let us look into more detail at some of these operations. `load_table` takes a table and loads it into `gr`. The pre-condition states that the table should have the properties expected for a valid share of students into groups.

```
load_table:table ==> ()
load_table(t) ==
(gr := t)
pre size_max_gr(t) <= limite.max
        and
        (size_max_gr(t) - size_min_gr(t) <= 1)
        and
        if card rng t > 1 then
                size_min_gr(t) >= limit.min
        else true
post gr = t
```

`swap_two_students` is an operation that exchanges two students. Each of them goes into the group of the other. The pre-condition states that both students must belong to a group, i.e. they are in the domain of the `gr`. The post-condition is expressed in a declarative style:

- the first conjunct states that the domain of `gr` is the same as the domain of the initial `gr` (`~` is the ASCII version of the VDM "hook" which denotes the initial value of a variable);
- the second and third conjuncts express that e1 and e2 have exchanged their group assignments;

– the last conjunct expresses that `gr` did not change for other domain values than `e1` or `e2` (`<-:` is the domain deletion operator).

The code for this operation applies a double modification to the `gr` map, assigning `e1` into the group of `e2`, and conversely.

```
swap_two_students : student * student ==> ()
swap_two_students(e1,e2)==
(gr := gr ++ {e1 |-> gr(e2), e2 |-> gr(e1)})
pre e1 in set dom gr and e2 in set dom gr
post dom gr = dom gr~ and gr(e1) = gr~(e2) and gr(e2) = gr~(e1)
        and {e1,e2} <-: gr = {e1,e2} <-: gr~
```

These two operations can be tested by the following sequence:

```
"Sequence 1",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","b"),
swap_two_students("a","e"),
swap_two_students("a","a")
```

This sequence first loads a class with two groups, then tries to perform several exchanges. The test coverage measurement of VDMTools reports a full coverage of `swap_two_students` and 97% of `load_table`.

5 Finding errors

Let us now have a look at a variant of `swap_two_students`. The `swap_two_studentsF` operation has the same specification (pre- and post-condition) but a slightly different code: a local variable `g` is declared to temporarily store the group of `e1`; then the group of `e1` is modified and finally the group of `e2` is assigned to g.

```
swap_two_studentsF : student * student ==> ()
swap_two_studentsF(e1,e2)==
(dcl g : gr_id := gr(e1); gr(e1) := gr(e2) ; gr(e2) := g)
pre e1 in set dom gr and e2 in set dom gr
post dom gr = dom gr~ and gr(e1) = gr~(e2) and gr(e2) = gr~(e1)
        and {e1,e2} <-: gr = {e1,e2} <-: gr~
```

This code is wrong because the invariant may be false after the modification of `gr(e1)`. The following test case exhibits this error.

```
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_studentsF("f","a")
```

Here, `"f"` belongs to the smallest group. After `gr(e1) := gr(e2)`, group 1 has 5 elements and group 2 only 3, which breaks the invariant.

It is interesting to notice that sequence 1 in section 4 has 100% coverage but does not reveal this error, because `swap_two_students("a","e")` first moves the element of the largest group.

This rather trivial example shows that blindly trusting the test coverage information leads to incomplete testsuites. Moreover, when these testsuites are written "by hand", they tend to be small. When specifications and implementations get bigger, there is a definite risk to overlook some useful test cases. This is where TOBIAS is of interest, by helping the software engineer write a large and more exhaustive testsuite.

6 TOBIAS

TOBIAS is a tool for automatic generation of test cases from a given test pattern. Writing test cases is a very tedious and repetitive task, especially when we need a large set of test cases. This is where TOBIAS helps produce a large set of similar test cases. We have experimented that many test cases feature the same sequence of operations but with different parameter values. Other sequences may also differ by exchanging an operation with a similar one.

TOBIAS allows the user to define a set of relevant values for each operation parameter or to identify sets of similar operations (named "groups" in TOBIAS). These form the basis for the definition of test patterns (named "test schemas" in TOBIAS). A test schema is a bounded regular expression over operations and groups. Test schemas are then unfolded by TOBIAS into a large set of test cases.

We expect that TOBIAS will help test engineers generate more tests cases and in a more systematic way for about the same effort than "manually" written test cases. Generating more tests may increase the confidence in the testsuite. Generating these more systematically will help cover more behaviours of the system, including situations that could be overlooked or forgotten by the test engineer. So we may reasonably expect that TOBIAS increases the chance of detecting errors.

6.1 Iteration over parameter values

Let us now illustrate the use of TOBIAS on some small examples. First we start with the test of `swap_two_students`.

<table>
<tr><th>Group Name</th><th>Operation Name</th><th colspan="4">Parameter(s)</th></tr>
<tr><td>LoadStudents</td><td>load_table</td><td>t</td><td colspan="3">{"a" | -> 1, "b" | -> 1, "c" | -> 1,
"d" | -> 1, "e" | -> 2, "f" | -> 2,
"g" | -> 2}</td></tr>
<tr><td rowspan="4">SwapStudents</td><td rowspan="4">swap_two_students</td><td rowspan="4">e1</td><td rowspan="4">"a"
"f"</td><td rowspan="4">e2</td><td>"a"</td></tr>
<tr><td>"b"</td></tr>
<tr><td>"e"</td></tr>
<tr><td>"f"</td></tr>
</table>

The starting point is the identification of groups.

1. We define a `LoadStudents` group, composed of the `load_table` operation with the parameter value `{"a"|->1,"b"|->1,"c"|->1,"d"|->2,"e"|->2, "f"|->2,"g"|->2}`. This group actually includes a single element.
2. We define a `SwapStudents` group, composed of the `swap_two_students` operation with the following parameter values: `"a", "f"` for `e1` and `"a", "b", "e", "f"` for `e2`. This second group has 8 elements.

These groups can be exploited by the following test schema:

```
LoadStudents;SwapStudents^1..2
```

This schema means that TOBIAS will generate all the test sequences that are: an operation call from the `LoadStudent` group followed by one to two operation calls from the `SwapStudent` group. This schema unfolds into 72 (1*(8 + 8*8)) test cases. They have the same coverage than the manually written test sequence, but if applied to `swap_two_studentsF`, they now include tests that detect the error described in section 5.

```
"Test case 1",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","a"),
...
"Test case 2",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
echanger_deux_etudiants("f","a"),
"Test case 3",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","b"),
...
"Test case 5",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","e"),
...
"Test case 35",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("f","b"),
swap_two_students("a","b"),
...
```

```
"Test case 72",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("f","f"),
swap_two_students("f","f")
```

Test cases 1, 3, and 5 correspond to the test cases of section 4. Test case 2 corresponds to the test presented in section 5 that exhibits the error of `swap_two_studentsF`.

More complete testsuites can be generated by defining several values for parameter `t` of `load_table`. This would allow to perform these tests with an empty class of students, or with various combinations of groups. Such a testsuite allows the error to manifest itself in several ways: while the above mentioned test case broke the second conjunct (difference between groups) of the invariant, other test cases also break the first conjunct (maximum number) resulting in a group of six students. This makes the testsuite more robust towards evolutions of the specification.

6.2 Iteration over similar operations

The `change_student` operation takes two parameters : `e`, a student, and `g`, a group number. It assigns the student to the group and performs the necessary adjustments to keep the invariant. It is specified as follows.

- The pre-condition ensures that the student and the group already exist.
- The post-condition states that the domain of `gr` does not change, i.e. the students of the class remain the same, and that the student given as parameter is assigned to the mentioned group. Everything else (number of groups, group numbers, other assignments) may change.

The proposed code picks up `e2`, one of the students of group `g`, and swaps him with `e`.

```
change_student:student * gr_id ==> ()
change_student(e,g) ==
(let e2 in set dom(gr :> {g})
in swap_two_students(e2,e)
)
pre e in set dom gr and g in set rng gr
post dom gr = dom gr~ and gr(e) = g
```

The following test sequence was produced for `change_student`. Once again, although very short, it offers 100% coverage of the operation.

```
"Sequence 2",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
change_student("a",1),
change_student("a",2),
change_student("f",1),
```

There are obvious similarities between `change_student` and `swap_two_students`. Not only is the one implemented on basis of the other, but also because they both result in some student being assigned to another group, this group being specified explicitly or implicitly. This similarity also appears in the test sequences, both test sequences test the transfer into the other group and in the same group.

TOBIAS makes it possible for the tester to express this similarity by extending the definition of group `SwapStudents`:

Group Name	Operation Name	Parameter(s)			
LoadStudents	load_table	t	{"a" | -> 1, "b" | -> 1, "c" | -> 1, "d" | -> 1, "e" | -> 2, "f" | -> 2, "g" | -> 2}		
SwapStudents	swap_two_students	e1	"a"	e2	"a"
			"f"		"b"
					"e"
					"f"
	change_student	e	"a"	g	1
			"f"		2

Now `SwapStudents` groups 8 instanciated calls to `swap_two_students` with 4 instanciated calls to `change_student`. We can reuse the already defined test schema:

```
LoadStudents;SwapStudent^1..2
```

TOBIAS unfolds this schema into 156 (1*(12+144)) test cases.

```
"Test case 1",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","a"),
"Test case 2",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("f","a"),
...
```

```
"Test case 30",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("f","a"),
swap_two_students("f","e"),
...
"Test case 45",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
swap_two_students("a","b"),
change_student("a",1),
...
"Test case 142",
load_table({"a"|->1,"b"|->1,"c"|->1,"d"|->1,"e"|->2,"f"|->2,
"g"|->2}),
change_student("a",2),
change_student("f",1),
...
```

These test cases include the manually defined tests, but apply these in a more exhaustive manner. Moreover, the grouping mechanism of TOBIAS generates combinations of both operations. Finally, it also worth mentioning that the effort of defining groups and test schema is similar to the effort of manually writing the two short sequences of tests.

7 Using brute force

7.1 Groups and test schema

The next experiment is an attempt to test the whole specification. Figure 1 shows the definition of several groups. They are used by the following test schema:

```
LoadTable; AddStudents^0..1; Mix^0..1; Swap; Delete
```

This schema aims at covering most of the specification in a single testsuite: it loads a table, adds several students, performs some modifications of the group assignments (mixing or swapping), and finally deletes some students. The underlying idea of this schema is to exploit "brute force": the schema unfolds in 4320 test cases that exercise various configurations of the initial table and of the operations. Moreover, the schema also takes advantage of nested calls: many operations of the specification are defined in terms of other operations (as we saw for `change_student`).

Group Name	Operation Name	Parameter(s)			
LoadTable	load_table	t	{"a" | -> 1}		
			{"a" | -> 1, "b" | -> 1, "c" | -> 1}		
			{"a" | -> 1, "b" | -> 1, "c" | -> 1, "d" | -> 1, "e" | -> 1}		
			{"a" | -> 1, "b" | -> 1, "c" | -> 1, "d" | -> 1, "e" | -> 2, "f" | -> 2, "g" | -> 2}		
AddStudents	add_set_of_students	ee	{"z"}		
			{"z", "y"}		
			{"z", "y", "x", "w"}		
			{"z", "y", "x", "w", "v"}		
			{"z", "y", "x", "w", "v", "a"}		
Delete	delete_student	e	"a"		
			"z"		
			"e"		
Swap	swap_two_students	e1	"a"	e2	"a"
			"f"		"z"
			"w"		
	swap_two_studentsF	e1	"a"	e2	"a"
			"f"		"z"
			"w"		
	change_student	e	"a"	g	1
			"v"		2
Mix	mix				
	mix_better				

Fig. 1. Groups to test the whole specification

7.2 Coverage

Figure 2 compares this testsuite to a manually developed test suite. The manually developed testsuite is about 45 lines and achieves 85% of coverage of the specification. A closer look shows that this coverage ranges from 94 to 100% for the operations exercised by the testsuite. Some operations or functions were not tested because they correspond to dead code or less interesting operations. The TOBIAS testsuite reaches similar coverage, both globally and individually, but there is a significant difference in the #calls column, which reveals that the specification has been extensively tested.

7.3 Errors found

The TOBIAS testsuite finds two more errors than the manual testsuite. These two errors appear in the `add_student` operation. It also revealed the error of `swap_two_studentsF` through two different ways, as already explained in section 6.1.

`add_student` adds a student to the class. It can add it to one of the existing groups or compute a new share of students into groups. It is specified as follows:

- there is no pre-condition: any student may be added at any time;
- the post-condition states that the student has been added to the domain of the map, and that the output parameter is the group assigned to the student.

	Manual		TOBIAS	
	#Calls	**Coverage**	**#Calls**	**Coverage**
print_groups	1	√	0	0%
add_set_students	1	√	3600	√
add_student	9	√	8208	√
add_group	2	√	1872	√
change_student	3	√	1080	√
load_table	16	√	4320	√
swap_two_students	12	√	4662	√
swap_two_studentsF	4	√	1620	√
delete_student	5	√	4320	√
maximum	316	95%	80515	95%
mix	1	√	1440	√
mix_better	0	0%	1440	48%
minimum	302	95%	75366	95%
max_nb_of_gr	2	√	1872	√
min_nb_of_gr	0	0%	0	0%
table2partition	379	√	123974	√
table2table-inverse	0	0%	0	0%
size-max-gr	194	94%	65375	94%
size-min-gr	183	94%	58599	94%
Total Coverage		**85%**		**85%**

Fig. 2. A comparison of manually generated vs automatically generated testsuite

The proposed implementation tests that there is room in one of the groups. If all groups have reached the upper limit, `add_group` is called. This operation increases the number of groups; for example, it transforms two groups of 5 students, into one group of 4 and two groups of 3. Then, `add_student` assigns the new student to one of the groups which has the lowest number of students.

```
add_student:student ==> gr_id
add_student(e) ==
(if size_min_gr(gr) = limit.max then add_group();
 let gid in set rng gr
     be st card dom (gr :> {gid}) = size_min_gr(gr)
          in (gr := gr ++ {e |-> gid}; return(gid)))
```

```
post dom gr = (dom gr~ union {e}) and RESULT = gr(e)
```

The TOBIAS testsuite revealed two errors in this implementation, namely when there are no groups, and when there is a single group of 5 students. If no group exists, it is not possible to add the student to the smallest group. If there is a single group of five students, it is not possible to split it into two groups before adding the student.

These two errors were not detected by the manual testsuite, because the tests of `add_student` were applied to a configuration of at least two groups. It is obvious that the systematic character of the tool reduces the risk to overlook such test cases. Actually, one of these two errors was first detected by reading the specification after the manual testsuite had been developed. Therefore, we expected TOBIAS to detect it also. But the second error was unknown to us and we discovered it by executing the TOBIAS testsuite!

8 Conclusion and perspectives

8.1 Summary

This paper has presented TOBIAS, and how it can be combined with VDMTools. TOBIAS aims to be a simple and easy to use tool which supports and amplifies the creative work of a test engineer. The experiments presented in this paper show that the effort of defining groups and test schemas is similar to the effort of producing a small manual testsuite. Of course, the tool does not help the test engineer finding the right tests, but it provides support by performing the repetitive tasks to produce a more exhaustive testsuite. This allows the test engineer to concentrate on the difficult task, i.e. define the most appropriate test schemas. In this paper, the largest experiment shows that good results can already be obtained using a "brute force" approach. Still, the complexity of real applications requires a more disciplined way of testing to ensure quality. In such cases, brute force can only be considered as a complementary approach to increase confidence in the already performed test process.

This paper has reported on an experiment which showed that

- The testsuite generated by TOBIAS discovered more errors than a manual testsuite. It also exercised some known errors in several different ways, making the testsuite more robust towards evolutions of the specification.
- Writing TOBIAS test schemas requires a similar effort than writing a small manual testsuite.

8.2 VDMTools

Strictly speaking, TOBIAS does not produce "test cases" but simply "test data". To turn test data into a test case, you need to add the oracle. VDMTools provides a way to add the oracle to test data dynamically by evaluating the assertions

of the specification. The combination of both tools allows us to experiment and better understand TOBIAS. Another interesting feature of VDMTools is the test coverage information, provided the tester is aware that full coverage does not necessarily mean perfect testsuite.

Our experiments were carried out with version 3.6 of VDMTools in a Unix environment. In this version, a large testsuite must be executed in command-line mode, using a ".arg" file. This file is the concatenation of the generated test cases. In this mode, the result of the execution of the test cases appears on stdout, while error messages appear on stderr. This separation of results in two distinct streams makes it difficult to relate an error message to the execution of a given test case. This problem is easy to fix but reveals a necessary feature for executing large testsuites with VDMTools.

The size of the TOBIAS testsuite also led to two kinds of execution problems. First, execution time becomes a concern. Executing the brute force testsuite takes about 90 minutes, while the manual testsuite takes only 3 minutes. In particular, invariant assertions are executed a very large number of times. This encourages to write "efficient" invariants, i.e. assertions that are optimized for execution. Of course, such optimization may conflict with the readability of the specification. Second, we experimented some memory problems and had to cut our test file into files of about 1000 test cases.

Finally, we noticed an interesting problem while interpreting the error messages related to pre-conditions. In our brute force testsuite, many test cases call operations outside their pre-condition. Such tests don't reveal errors in the implementation and these pre-condition errors should be interpreted as "inconclusive" verdicts. Still pre-condition errors may also reveal errors in the implementation, when they correspond to nested calls. For example, the `add_student` operation calls `add_group` outside of its pre-condition when there is a single group of five students. This makes it necessary to carefully analyze the errors reported while executing the testsuite, in order to distinguish inconclusive tests from failed tests.

8.3 Evolution of TOBIAS

These experiments also helped us evaluate TOBIAS, contributed to its debugging and to improvements of its user interface. The biggest challenge faced by TOBIAS is to handle combinatorial explosion which may lead to very large testsuites. Unfolding test schemas may easily lead to millions of test cases. While it may be interesting to have large testsuites, very large testsuites are no longer of any help in the testing process, because they usually require untractable resources to be generated and executed.

Therefore, our current research focuses on reducing the number of automatically generated test cases on the basis of information collected from the specification. This is where the formal character of specifications helps. Information extracted from the specification helps detecting inconclusive test cases and may also contribute to identify sets of "equivalent" test cases. Further research will investigate the exploitation of information from pre- and post-conditions, or from UML diagrams associated to the specification, to filter out inconclusive

test cases. Detecting equivalent test cases requires to formulate test hypotheses on the parameters of operations. Here again, using a formal and executable specification language like VDM to express these hypotheses is of definite interest.

Another major challenge of using TOBIAS corresponds to the test data selection. With the current version of TOBIAS, test data are selected "manually" by the test engineer. This may lead to two kinds of problems:

- the test engineer may overlook an interesting value, so the test suite may miss some kinds of errors;
- he may also select redundant values, resulting in a larger but not more useful test suite.

Therefore, it makes sense to combine TOBIAS with test data generation tools such as those based on the ideas of Dick and Faivre [4]. In the coming months, we expect to establish a link with the CASTING tool [11].

Today, TOBIAS already provides an effective solution for the generation of large test suites. In further work, we expect to take advantage of formal techniques to generate more efficient and pertinent test suites.

References

1. B. Beizer. *Software Testing Techniques.* Van Nostrand Reinhold, 1990.
2. P. Bontron, O. Maury, L. du Bousquet, Y. Ledru, C. Oriat, and M.-L. Potet. TOBIAS : un environnement pour la création d'objectifs de tests à partir de schémas de tests. In J.C. Rault, editor, *ICSSEA 2001*, decembre 2001.
3. E. Brinksma. A theory for derivation of tests. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII.* North Holland, 1988.
4. J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications . In *FME'93: Industrial-Strength Formal Methods.* LNCS 760, Springer-Verlag : 268-284, April 1993.
5. L. du Bousquet, H. Martin, and J.-M. Jézéquel. Conformance Testing from UML specificationd, Experience Report. In Gesellschaft für Informatik (GI), editor, *p-UML workshop, Lecture Notes in Informatics (LNI)*, volume P-7, pages 43–56, Toronto, Canada, 2001.
6. John Fitzgerald and Peter Gorm Larsen. *Modelling Systems – Practical Tools and Techniques in Software Development.* Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998. ISBN 0-521-62348-0.
7. The VDM Tool Group. VDM-SL Toolbox User Manual. Technical report, IFAD, October 2000. ftp://ftp.ifad.dk/pub/vdmtools/doc/userman_letter.pdf.
8. T. Jéron and P. Morel. Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV).* LNCS 1633, Springer-Verlag, 1999.
9. C. B. Jones. *Systematic Software Development Using VDM (Second Edition).* Prentice-Hall, London, 1990.
10. J. Tretmans. *A formal approach to conformance testing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.
11. L. Van Aertryck, M. Benveniste, and D. Le Métayer. Casting: A formally based software test generation method. In *The 1st Int. Conf. on Formal Engineering Methods, IEEE, ICFEM'97*, Hiroshima, 1997.