Model-Based Filtering of Combinatorial Test Suites

Taha Triki, Yves Ledru, Lydie Du Bousquet, Frédéric Dadeau, Julien Botella

Model-based filtering of combinatorial test suites

T. Triki[1], Y. Ledru[1], L. du Bousquet[1], F. Dadeau[2], and J. Botella[3]

[1] UJF-Grenoble 1/Grenoble-INP/UPMF-Grenoble 2/CNRS,
LIG UMR 5217, F-38041, Grenoble, France
{Taha.Triki, Yves.Ledru, Lydie.du-Bousquet}@imag.fr
[2] LIFC - INRIA CASSIS Project, 16 route de Gray, 25030 Besançon, FRANCE
frederic.dadeau@lifc.univ-fcomte.fr
[3] Smartesting, Besançon, France
julien.botella@smartesting.com

Abstract. Tobias is a combinatorial test generation tool which can efficiently generate a large number of test cases by unfolding a test pattern and computing all combinations of parameters. In this paper, we first propose a model-based testing approach where Tobias test cases are first run on an executable UML/OCL specification. This animation of test cases on a model allows to filter out invalid test sequences produced by blind enumeration, typically the ones which violate the pre-conditions of operations, and to provide an oracle for the valid ones. We then introduce recent extensions of the Tobias tool which support an incremental unfolding and filtering process, and its associated toolset. This allows to address explosive test patterns featuring a large number of invalid test cases, and only a small number of valid ones. For instance, these new constructs could mandate test cases to satisfy a given predicate at some point or to follow a given behavior. The early detection of invalid test cases improves the calculation time of the whole generation and execution process, and helps fighting combinatorial explosion.

1 Introduction

Combinatorial testing is an efficient way to produce large test suites. In its basic form, combinatorial testing identifies sets of relevant values for each parameter of a function call, and the production of the test suite simply generates all combinations of the values of the parameters to instantiate the function call. JMLUnit [4] is a simple and efficient tool based on this technique, which uses JML assertions as the test oracle. Extended forms of combinatorial testing allow to sequence sets of operations, each operation being associated to a set of relevant parameters values. This produces more elaborate test cases, which are appropriate to test systems with internal memory whose behaviour depends on previous interactions. Tobias [19, 17, 18] is one of these combinatorial generators. It was used successfully on several case studies [3, 9, 10] and has inspired recent combinatorial testing tools, such as the combinatorial facility of the Overture toolset for VDM++[16] or jSynoPSys [8].

Tobias takes as input a test pattern and performs its combinatorial unfolding into a possibly large set of test cases. Each test case usually corresponds to a sequence of test inputs. An additional oracle technology is needed to decide on successful or failed test executions. In the past, we have mainly used the run-time evaluation of JML assertions as a test oracle [3]. But the tool can be used in other contexts than Java/JML. In this paper, we adopt a model-based testing approach where tests are first played on a UML/OCL specification of the system under test. The animation of a sequence of operations on the UML/OCL specification is performed by the Test Designer tool of Smartesting [4] and brings two kinds of answers. First, it reports whether the sequence is valid, i.e. each of its calls satisfies the pre-condition of the corresponding operation and is able to produce an output which verifies the post-condition. Then, it provides the list of intermediate states and operation results after each operation call. This information can be used as test oracle to compare with the actual states and results of the system under test. It must be noted that the model is deterministic, which forces all accepted implementations to produce the same results and intermediate states (if observable). In summary, the model is used (a) to discard invalid sequences, and (b) to provide an oracle for valid ones.

Combinatorial testing naturally leads to combinatorial explosion. This is initially perceived as a strength of such tools: large numbers of tests are produced from a test pattern. This helps to systematically test a system by the exhaustive exploration of all combinations of selected values. The latest version of Tobias has been designed to generate up to 1 million abstract test cases. It is actually only limited by the size of the file system where the generated tests are stored. Unfortunately, the translation of these test cases into a target technology such as JUnit, the compilation of the resulting file and its execution usually require too much computing resources and, in practice, the size of the test suite must be limited between 10 000 and 100 000 test cases.

Several techniques can be adopted to limit combinatorial explosion. The most classical one is the use of pairwise testing techniques [5] which does not cover all combinations of parameter values but simply covers all pairs of parameter values. This technique is very efficient to reduce a large combinatorial test suite to a much smaller number of test cases, but it relies on the hypothesis that faults result from a combination of two parameters. Therefore it may miss faults resulting from a combination of three or more parameters. The technique can be generalized to cover all n-tuples of parameters but it may always miss combinations of $n + 1$ parameters. Another approach is the use of test suite reduction techniques [13] which select a subset of the test suite featuring the same code coverage as the original test suite. This technique has several limitations. First, it requires to play the full test suite in order to collect coverage information. Also, empirical studies have shown that test suite reduction may compromise fault detection [20].

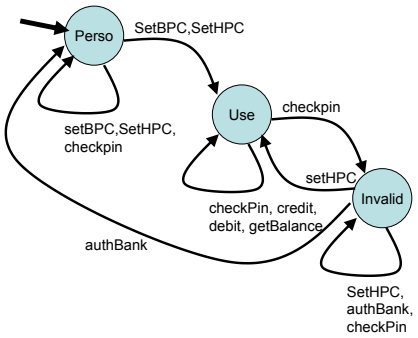
In this paper, we consider the case where combinatorial test suites lead to a large proportion of invalid test cases, i.e. test cases which will not be accepted

[4] http://smartesting.com

by the specification. These invalid test cases must be discarded from the test suite because the specification is unable to provide an oracle for these tests. Discarding these invalid test cases leads to a safe reduction of the test suite. We present a tool which incrementally unfolds a test pattern and discards invalid test cases. The tool is based on an evolution of the Tobias tool where several new constructs have been added to the base language.

Section 2 introduces an illustrative case study. Then Section 3 presents the basic constructs of Tobias, using the case study. Section 4 presents additional constructs which help filter combinatorial test suites. Section 5 presents the toolset which incrementally unfolds the test patterns and filters the resulting test suite. Section 6 reports on several experiments carried with this tool set. Section 7 gives an overview of the research literature related to our work. Finally, Section 8 draws the conclusions of this work.

2 An illustrative case study



Method signature	Informal description
`beginSession(int)`	Opening of session
`endSession()`	Termination of session
`setBpc(int)`	Sets the bank's pin
`setHpc(int)`	Sets the holder's pin
`checkPin(int)`	Identifies the holder
`authBank(int)`	Identifies the bank
`credit(int)`	credit of the purse
`debit(int)`	debit of the purse
`getBalance()`	value of the balance

Fig. 1. The main modes of the bank card and the main operations

We consider the example of a smart card application, representing an electronic purse (e-purse). This purse manages the balance of money stored in the purse, and two pin codes, one for the banker and one for the card holder. Similarly to smart cards, the e-purse has a life cycle (Fig. 1), starting with a *Personalization* phase, in which the values of the banker and holder pin codes are set. Then a *Use* phase makes it possible to perform standard operations such as holder authentication (by checking his pin), crediting, debiting, etc. When the holder fails to authenticate three consecutive times, the card is *invalidated*. Unblocking the card is done by a banker's authentication. Three successive failures in the bank authentication attempts make the card return to the *Personalization* phase. Each sequence of operations is performed within sessions, which are initiated through different terminals. This example has originally been designed to illustrate access control mechanisms, and it is used a basis for test generation for access control[5]. It was already used to illustrate test suite reduction with

[5] the original code of the application (in B and Java/JML) is available at http://membres-liglab.imag.fr/haddad/exemple_site/index.html

Pre-condition:

```
(self.isOpenSess_ = true  and  self.mode_ = Mode::USE and
self.terminal_ = Terminal::PDA  and  self.hptry_ > 0) = true
```

Post-condition:

```
if (pin = self.hpc_) then /**@AIM: HOLDER_AUTHENTICATED */
  self.isHoldAuth_ = true  and  self.hptry_ = self.MAX_TRY
else /**@AIM: HOLDER_IS_NOT_AUTHENTICATED */
  self.hptry_ = self.hptry_@pre - 1 and   self.isHoldAuth_ = false and
  if (self.hptry_ = 0) then /**@AIM: MAX_NUMBER_OF_TRIES_REACHED */
    self.mode_ = Mode::INVALID
  else /**@AIM: MAX_NUMBER_OF_TRIES_IS_NOT_REACHED */
    true
  endif
endif
```

Fig. 2. Pre and post-condition for `checkPin(int)` operation

Tobias [7]. The original example was specified in JML. We have translated this specification into a UML/OCL model for the Smartesting Test Designer tool.

In Test Designer (TD), information about the behaviour of operations is captured in assertions associated to the operations. In the perspective of animation, these assertions must characterize a deterministic behaviour. An example of the pre- and post-conditions of the `checkPin(int)` operation is given Fig. 2. Post-conditions represent the code to be animated by TD if the pre-condition is verified. TD uses an imperative variant of OCL[6], inspired by the B language [1]. The variables appearing in the right hand side of a `=` sign are implicitly taken in their pre-state (usually denoted in OCL by `@pre`). In the model, the conditional branches are tagged with special comments. For example if the pin code is equal to the right one (`self.hpc_`), the tag `@AIM:HOLDER_AUTHENTICATED` will be activated and saved by the animator. After animation of an operation call, TD provides the list of all activated tags. The set of activated tags after an execution represents a behaviour of an operation. For example, the set:
B1 = {`@AIM:HOLDER_IS_NOT_AUTHENTICATED`,`@AIM:MAX_NUMBER_OF_TRIES_REACHED`}
is a behaviour of the checkPin operation leading to `INVALID` mode.

3 Basic Tobias test patterns

To generate test cases, Tobias unfolds a test pattern (also called "test schema"). The textual Tobias input language (TSLT) contains several types of constructs allowing the definition of complex system scenarios. The key concept in the Tobias input language is the **group** concept which defines a set of values or sequences of instructions. The group concept is subject to combinatorial unfolding. Some other concepts can be applied to instructions like iteration or choice. To illustrate these constructs, let us consider the following pattern:

[6] The example presented in this paper follows the standard OCL syntax.

```
group EPurseSchema1 [us=true, type=instruction] {
   @IUT;  @Personalize; @AuthenticateHolder; @Transaction{1,3};}
group IUT [type=instruction] { EPurse ep = new EPurse(); }
group Personalize [type=instruction] {
   ep.beginSession(Terminal.ADMIN);  ep.setBpc(@BankPinValue);
   ep.setHpc(@UserDebitValue);  ep.endSession(); }
group AuthenticateHolder{
   ep.beginSession(Terminal.PDA); ep.checkPin(@UserPinValue){1,4}; }
group Transaction [type=instruction] {
    (ep.credit(@Amounts) | ep.debit(@Amounts)); }
group BankPinValue [type=value] {values = [12,45];}
group UserPinValue [type=value] {values = [56,89];}
group Amounts [type=value] { values = [-1,0,50]; }
```

EPurseSchema1 is a group of instructions (`type = instruction`), and the flag **us** indicates whether the group will be unfolded (`=true`) or not. This group is a sequence of 4 other groups: IUT, Personalize, AuthenticateHolder and Transaction. This last group will be repeated one to three times in the sequence. The IUT group defines a new instance of class EPurse. Then, the Personalize group opens a new ADMIN session, sets the banker and the holder PIN codes, and finally closes the session. The AuthenticateHolder group authenticates the holder, and finally the Transaction group allows to do transactions. We use groups of values in some operation calls. For instance, the parameter of the `setBpc` method has 2 possible values.

The iteration construct $\{m,n\}$ repeats an instruction, or a sequence of instructions, from m to n times or exactly m times. For example, in group AuthenticateHolder, the `checkPin` operation is iterated 1 to 4 times (to check all possible sequences of correct/incorrect user authentication.

The Transaction group illustrates the choice construct. It consists of an exclusive choice between the two operation calls `Debit` or `Credit`. Each of them can be instantiated by three different amounts.

The `EPurseSchema1` pattern is unfolded into 30 960 test cases :
1 * (2*2) * $(2^1+2^2+2^3+2^4)$ * $((3*2)^1+(3*2)^2+(3*2)^3)$. Only 2776 test cases are valid ones (i.e. satisfy the pre-conditions). In Fig. 3, examples of test cases unfolded from `EPurseSchema1` are given, `TC3` is valid, contrary to `TC26835` (which executes 4 consecutive calls to the checkPin operation with the wrong Pin code) and `TC30960` (which executes a debit operation but never credits).

If we put the maximum iteration bound of Transaction to 10, it would result into 8 707 129 200 test cases and would cause combinatorial explosion. In the next sections, we will see how the new Tobias constructs make it possible to take such explosive test patterns into account.

4 New Tobias constructs

Here, we introduce three new constructs for the Tobias input language. These constructs support new techniques for filtering test cases. This allows to control the size of the produced test suite, and incrementally pilot the combinatorial

```
...
TC3: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(12);  ep.setHpc(56); ep.endSession();
     ep.beginSession(PDA); ep.checkPin(56); ep.credit(50)
...
TC26835: EPurse ep = new EPurse(); ep.beginSession(ADMIN);
       ep.setBpc(45); ep.setHpc(89); ep.endSession();
       ep.beginSession(PDA); ep.checkPin(56); ep.checkPin(56);
       ep.checkPin(56); ep.checkPin(56); ep.credit(50)
...
TC30960: EPurse ep = new EPurse(); ep.beginSession(Terminal.ADMIN);
     ep.setBpc(45);  ep.setHpc(89); ep.endSession();
     ep.beginSession(PDA); ep.checkPin(89); ep.checkPin(89);
     ep.checkPin(89); ep.checkPin(89); ep.debit(50); ep.debit(50);
     ep.debit(50)
```

Fig. 3. Examples of test cases unfolded from `EPurseSchema1`

unfolding process. These constructs are inspired by the jSynoPSys scenario language [8] and are syntactically and semantically adjusted to meet our needs.

The State predicate construct inserts an OCL predicate in the test sequence. The predicate expresses that a property is expected to hold at this stage of the test sequence. Tests whose animations do not satisfy this OCL predicate should be discarded from the test suite. It allows the tester to select a subset of the unfolded test suite featuring a given property at execution time. For example, not all AuthenticateHolder animations succeed. Therefore, we define a state predicate to select the tests which succeed the authentication. The pattern is defined as follows:

```
group EPurseSchema5 [us=true, type=instruction] {  @IUT; @Personalize;
@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true); @Transaction; }
```

AuthenticateHolder performs checkPin one to four times. Then the pattern selects those sequences which end up with a successful authentication. The TSLT construct takes the form `⇝(set of targets , OCL predicate)`, where the set of targets identifies the objects which correspond to `self` in the OCL predicate. Here the set of targets is the singleton including `ep`.

The behaviours construct is another way to filter tests. It applies to an operation and keeps the tests whose animation covers a given behaviour, expressed as a set of tags. For example, in the previous pattern (`EPurseSchema5`), instead of using a state predicate to keep tests that succeed the holder authentication, we select the authentication sequences whose last call to `checkPin` covers the tag `@AIM:HOLDER_AUTHENTICATED`.

```
group EPurseSchema6 [us=true, type=instruction] {
@IUT; @Personalize; @AuthenticateHolder2; @Transaction; }

group AuthenticateHolder2 {
ep.beginSession(Terminal.PDA); ep.checkPin(@UserPinValue){0,3};
ep.checkPin(@UserPinValue)/w{set(@AIM:HOLDER_AUTHENTICATED)}; }
```

After the last call to checkPin, we put the symbol `/w` and we define a set of tags that must be activated after the operation execution. Here, when the pin code is correct, the tag `@AIM:HOLDER_AUTHENTICATED` is covered in the post-condition of `checkPin` (see Sect. 2).

The Filtering key is called after an instruction. It allows to accept a set of succeeded tests at some position and to discard the others. Then it will select some of the succeeded tests. TSLT provides four filtering keys (_ONE, _ALL, _n, _%n) to keep one, all, n or n% of the valid prologues. If we want to accept all of them we use _ALL, just one we use _ONE and _n (resp. _%n) randomly selects n (resp. n% of the) test cases amongst the valid ones. For example consider `EPurseSchema7`. The prologue group leads the purse to a state where the holder is authenticated. If the test engineer simply wants to keep one of these, he can add keyword `_ONE` after the prologue :

```
group EPurseSchema7 [us=true, type=instruction] {
@Prologue_ONE; @Transactions; }
group Prologue [us=true, type=instruction] {
@IUT;@Personalize; @AuthenticateHolder2; }
```

5 The incremental unfolding and filtering process

5.1 Standard unfolding and filtering process

Before introducing the mechanism of incremental unfolding, we begin by presenting the process of generation, animation and filtering of test cases by coupling the Tobias and Test Designer tools (Fig. 4). The starting point is a schema file including a test pattern written in TSLT. Three steps are automatically involved to produce the test evaluation results:

1. The schema file is unfolded by the Tobias tool which generates one or several test suite files written in the XML output language of the tool (outob file). For each group marked in TSLT as `us=true`, Tobias produces an outob file. This file contains all abstract test cases generated by the combinatorial unfolding of the corresponding group.
2. The outob files are translated into JUnit test suites including all necessary information to animate test cases. Each JUnit test case interacts with the API of TD to animate the model. We take advantage of the JUnit framework and the Java API of TD to animate the tests in a popular and familar tool for engineers, and to benefit from the JUnit structure of test suites.
3. JUnit executes the test suites. Each test case is animated on the TD model through the TD API. The animation process allows to identify and filter out invalid test cases, i.e. the ones which:
   - include some operation call that violates its precondition,
   - include some operation call that violates its postcondition,
   - do not fulfill some state predicate or

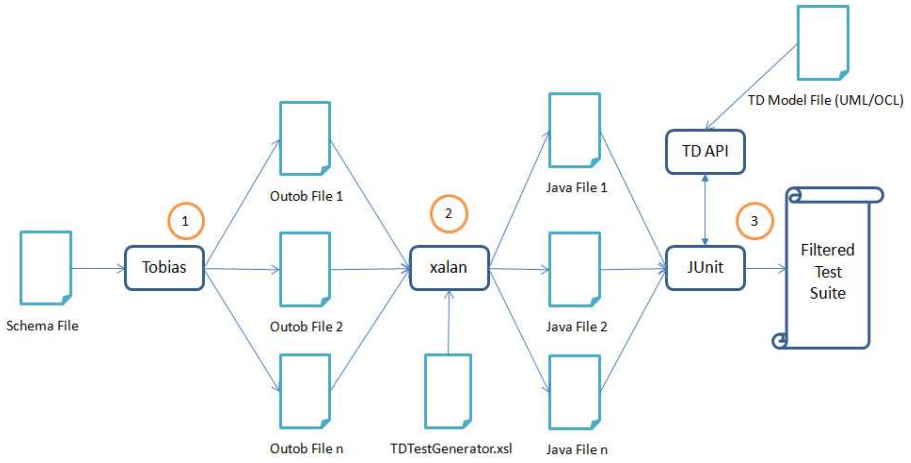


Fig. 4. The process of generation and filtering test cases (standard process)

– include some operation call that fails to activate its associated behaviours.

The animation of test cases proceeds sequentially. If an instruction fails because of one of these four reasons, the animation of the test stops and the test case is declared as failed and discarded from the test suite. The valid ones are saved to a repository.

5.2 Incremental unfolding and filtering process

Algorithm The standard process requires to completely unfold the test patterns and to animate each test case of each test suite. At this stage, we did not take advantage of filtering keys (_ONE, _ALL, _n, _n%). These filtering keys can be applied on the resulting test suite to select the relevant test cases. In this section, we will see that the early application of filtering keys may lead to significant optimisations of (a) the unfolding process and (b) the animation of the test suite.

The incremental process is defined for the unfolding of a single pattern `p`. It can be generalized to unfold multiple patterns. Its algorithm is given in Fig. 5 and performs the following steps:

– At each iteration, pattern p is divided into a prefix, located before the first filtering key, and a postfix, located after it.
– The standard unfolding and filtering process of Sect. 5.1 is applied to the prefix. It results into a group of valid unfolded prefixes.
– A subset of this group is selected according to the filtering key.
– This subset of valid unfolded prefixes is concatenated with the postfix to form the new value of p.
– The process iterates until all filtering keys are processed in the pattern.
– A last unfolding is applied to the resulting pattern stored in p.

```
algorithm Incremental_Generation_And_Execution_Process (p):
  while( p contains at least one filtering key )
    Let (prefix _1stKey ; postfix) match p in
    validPrefixes := apply_Standard_Process(prefix);
    validPrefixesSubset := Select_Subset_Of_
                           Valid_Prefixes_According_To(1stKey);
    p := (validPrefixesSubset ; postfix);
  end while
  result := apply_Standard_Process(p);
end
```

Fig. 5. The incremental unfolding algorithm

Example To illustrate this incremental process, consider the following pattern:
`group EPurseSchema9 [us=true, type=instruction] { @IUT; @Personalize;`
`@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true)_ONE; @Transactions;}`
Before calling `@Transactions`, we would like to choose just one (`_ONE`) sequence of operations that succeeds holder authentication.
The prefix of this pattern is:
`group EPurseSchema9pre [us=true, type=instruction] { @IUT; @Personalize;`
`@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true); }`
This prefix is then unfolded using the standard process. The three steps are executed to generate, animate and filter test cases. It unfolds into 120 tests, where only 56 are valid. A valid test is choosen randomly between them and inserted as a prefix in the new pattern:
`group EPurseSchema9b [us=true, type=instruction] {`
`(ep.beginSession(ADMIN) ;  ep.setBpc(45) ;  ep.setHpc(56) ;`
`ep.endSession() ; ep.beginSession(PDA) ; ep.checkPin(89) ;`
`ep.checkPin(56) ; ep.checkPin(56) ;);   @Transactions; }`
Since there is no remaining filtering key, the whole pattern will be unfolded to generate the final test cases. This unfolding leads to 6 test cases, where only 3 are valid. The final number of valid test cases may depend on the prefix that will be chosen randomly. These test cases will be animated to discard the invalid ones, and then produce the filtered test suite. This process is clearly optimized since only 126 test cases were completely unfolded, instead of 720 in the standard process. In the next section, we present experimental results on more complex examples.

6 Some experimental results

Let us consider the following example:
`group EPurseExample [us=true, type=instruction] {@IUT; @Personalize;`
`@AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);@Transactions{4};}`
The `EPurseExample` is unfolded into 155 520 test cases. Our tools succeed to achieve steps 1 and 2 (translation into TSLT and production of an outob file). Unfortunately, the translation of the outob XML file into a JUnit file crashes due to a lack of memory (we used up to 1.5Gb of RAM). If this had succeeded, we

presume that the compilation of the JUnit file would also crash. These technical problems can be overcome by decomposing our files into smaller ones, but still the whole process would take time and computing resources. Other group definitions can rapidly reach over 1 million test cases which may require untractable time and memory resources. We redefine the pattern by introducing filtering keys:

```
group EPurseExampleUsingKeys [us=true, type=instruction] {
@IUT; @Personalize; @AuthenticateHolder⇝({ep} , self.isHoldAuth_ = true);
@Transactions_ALL; @Transactions_ALL; @Transactions_ALL; @Transactions; }
```

This pattern will produce the same valid test cases as the previous one, since we used the `_ALL` key. Using the incremental process, we need four iterations to remove the three filtering keys and unfold the resulting pattern. The pattern is completely unfolded and animated in 175 seconds as given in Fig. 6. As a

Iteration	Nb of tests unfolded	Nb of tests accepted
1	720	168
2	1008	560
3	3360	1904
4	11424	6496

Fig. 6. Results of `EPurseExampleUsingKeys` unfolding

result our 155 520 test cases only include 6496 valid ones. To identify these, our incremental process needs four iterations but only unfolds and plays 16512 test cases. In this case, it performed the selection process using 10% of the resources needed for the standard one, and kept the test suites small enough to avoid tool crashes.

Support for a brute force approach. Let us consider another explosive pattern, based on Fig. 1. The aim of this pattern is to find test sequences where the purse goes back to Personalisation mode, before being set in Use mode. The only way to reach this goal is to start from $Perso$ mode, go into Use and $Invalid$ modes, before getting back to $Perso$ and finally to Use. These major steps are captured in the state predicates of the following pattern:

```
group EPurseSchema18op [us=true, type=instruction] {
@IUT;
@ALLOps{4}⇝({ep}, self.mode_ = Mode::USE);
@ALLOps{5}⇝({ep}, self.mode_ = Mode::INVALID);
@ALLOps{5}⇝({ep}, self.mode_ = Mode::PERSO);
@ALLOps{4}⇝({ep}, self.mode_ = Mode::USE); }
```

In order to change states, we adopt a brute force approach where a single group has been defined for all operations offered by the card. Group `ALLOps` can be unfolded in 19 elements.

```
group ALLOps { ep.beginSession(@TerminalValue) | ep.endSession() |
ep.setBpc(@BankPinValue) | ep.setHpc(@UserDebitValue) |
ep.authBank(@BankPinValue) | ep.checkPin(@UserDebitValue) |
ep.credit(@Amounts) | ep.debit(@Amounts); }
```

EPurseSchema18op repeats all operations 4 times, until it reaches the Use mode. Finding that it requires 4 iterations can result from a trial and error process, or from a careful study of the specification. Since we adopt a brute force approach, let us consider that the engineer has attempted to reach the Use mode in one to three steps, without success, and finally found that four steps were sufficient (session opening, setting the Holder and Bank codes, and session close). Similarly he found that 5 steps are the minimum to reach state $Invalid$ (session opening, three unsuccessful attempts to checkPin and session close), and to then reach state $Perso$ (session opening, three unsuccessful attempts to authBank and session close). As a result, to find a valid sequence reaching the Use mode and returning to the same mode after visiting the other modes, we need to call at least 18 operations (4+5+5+4). `EPurseSchema18op` represents 19^{18} test cases (about 10^{23} test cases), and thus cannot be directly unfolded. Because of the brute force approach, and because we inserted filtering predicates, a large number of these test cases will be invalid. This enables us to call the incremental process.

We redefine `EPurseSchema18op` using the filtering key `_ALL` to keep all valid prefixes.

```
group EPurseSchema18opWFilteringKey [us=true, type=instruction] {
@IUT;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::USE)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::INVALID)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::PERSO)_ALL;
@ALLOps_ALL; @ALLOps_ALL; @ALLOps_ALL;
@ALLOps⇝({ep}, self.mode_ = Mode::USE); }
```

`EPurseSchema18opWFilteringKey` is unfolded incrementally in 18 iterations. Fig. 7 shows the number of unfolded and accepted tests at each iteration. Steps 9 and 14 show how filtering predicates dramatically decrease the number of accepted tests. Fig. 7 shows that the number of test cases animated at each step remains small enough to be handled with reasonable time and computing resources, and to avoid tool crashes. As a result, we unfolded and animated a total of 85424 test cases for the 18 iterations in less than 17 minutes, instead of 19^{18} in the standard process. We finally found all 640 valid test cases hidden into this huge amount of potential test cases. This second example shows that this incremental technique is efficient to find complex test cases hidden in a huge search space. The key to success is to make sure that the use of filtering keys will effectively reduce or limit the number of test cases at each iteration. Therefore, one should prefer a specification and a test pattern which help identify invalid test cases as soon as possible.

Improvements with respect to our previous work. In the past [17, 18], we have proposed two techniques to master combinatorial explosion with Tobias: test filtering at execution time, and test selection at generation time. Filtering at

Iteration	Nb of tests unfolded	Nb of tests accepted	Iteration	Nb of tests unfolded	Nb of tests accepted
1	19	3	10	1064	72
2	57	7	11	1368	184
3	133	29	12	3496	376
4	551	24	13	7144	1160
5	456	40	14	22040	64
6	760	104	15	1216	80
7	1976	312	16	1520	224
8	5928	1136	17	4256	624
9	21584	56	18	11856	640

Fig. 7. Results of `EPurseSchema18opWFilteringKey` unfolding

execution time is based on a simple idea: if the prefix of a test case fails, then all test cases sharing the same prefix will fail. In [17], we have proposed an intelligent test driver which remembers the failed prefixes, and avoids to execute a test case starting with a prefix which previously failed. This idea is close to the one presented in this paper. Still, there are significant advances in the new technique proposed here. First, the original technique required to produce the full test suite. Every test was examined to check if it included a failing prefix. Our new incremental process does not generate the full test suite, it incrementally builds and filters the prefixes by alternating between unfolding and animation activities. Because we avoid the full unfolding of the test suite, we are able to consider test patterns corresponding to huge numbers of test cases (19^{18} in the last example). Another contribution of this paper is the definition of new constructs for test patterns (state predicates, behaviours, filtering keys), which help invalidate earlier the useless test cases in the unfolding process. Selection at generation time is another technique, where one selects a subset of the test suite based on some criterion. This selection takes place during the unfolding process and does not require to execute or animate test cases. In [18] we filtered the elements of the test suite whose text did not fulfill a given predicate. This predicate is freely chosen by the test engineer and does not prevent to filter out useful test cases. For example, one could filter out all test cases whose length was longer than a given threshold. In [7, 18], we investigated the use of random selection techniques. These techniques are by essence unable to distinguish between valid and invalid test cases, but they are able to reduce the number of test cases to an arbitrary number whatever be the size of the initial test suite. Compared to these selection techniques, our incremental process does not discard valid test cases, but makes the assumption that the number of valid test cases is small enough to remain tractable.

7 Related work

In [15], authors propose to study test reduction in the context of bounded-exhaustive testing, which could be described as a variation of combinatorial testing. Three techniques are proposed to reduce test generation, execution time

and diagnosis. In particular, the *Sparse Test Generation* skips some tests at the execution to reduce the time to the first failing test. Unlike ours, this approach does not rely on a model to perform test generation and reduction. Our approach allows filtering large combinatorial test suites by animating them on an UML/OCL model. It eliminates tests which don't verify the pre and post conditions of operations and/or given predicates or states. Generating tests or simply checking their correctness, is a classical approach when a model is available (principle of the "model-based testing" approaches). For instance, in [12], authors generate automatically a combinatorial test suite, that satisfies both the specification and coverage criteria (among which pairwise coverage of parameter values). The generation engine is based on a constraint solver and the specification is expressed in Spec#. In [2], authors also propose to automate test case generation with a constraint solver, but the specification is expressed as contracts extended with state machines. For both, the objective of the work is to generate a test suite which fulfills a coverage criterion on a model. In our approach, the test schema gives a supplementary selection criterion for test generation. In [6], specification is expressed as IOLTS and generation is done with respect to a test purpose, for conformance testing. In some way, our test schema can be compared to a sort of test purpose, but contrary to this work where only one test is generated for each test purpose, we aim at generating all the test cases satisfying the test purpose.

The problem of test suite reduction is to provide a shorter test suite while maintaining the fault detection power. The approach presented in [14] generates test suites from a model and traps properties corresponding to structural coverage criteria. An algorithm is then executed on the resulting test suite to generate a reduced test suite having the same coverage than the original one. The original and the reduced test suites are animated on a faulty model to compare their fault localization capabilities. In [11], authors propose an approach where test cases created thanks to model-checker are transformed such that redundancy within the test-suite is avoided, and the overall size is reduced. Our approach differs from test suite reduction techniques in two points. First, we don't need to execute all tests of the original test suite to perform the reduction, and second, we consider that all valid test cases are equivalent when performing reduction with the filtering keys. YETI[7] is a random test generation tool which generates test cases from program bytecode. The number of generated test cases is limited by the available time. The report shows in a real time GUI the bugs found sofar, the coverage percentage according to a classical coverage criteria, the number of system calls and the number of variables used in the system to carry out the test generation and execution. Compared to our toolset, our approach performs the generation of tests from a UML/OCL model and not from a program. The choice of the methods under test, the length of the call sequences and the parameter values is not done randomly as in the YETI tool but according to a careful test schema defined by the user. Contrary to our tool, the notion of filtering

[7] Tool website : http://www.yetitest.org/

against specific states or behaviors and the incremental unfolding do not exist in the YETI tool whose purpose is to maximize bugs detection and structural coverage.

8 Conclusion and perspectives

In this paper, we address the problem of filtering a large combinatorial test suite with respect to a UML/OCL Model. The whole approach relies on three main steps. First the set of tests to generate has to be *defined* in terms of a test pattern, expressed in a textual language called TSLT. Second, this schema is *unfolded* to produce abstract test cases that are *animated* within Smartesting Test Designer tool. This animation allows to identify and remove invalid test cases. The process of unfolding and filtering can be done incrementally so that potential combinatorial explosion can be mastered. Several examples have been presented. They show that the incremental process is able to generate all valid test cases scattered in a huge search space, provided that the number of valid test cases remains small enough. This paper has presented several new constructs which help the test engineer to express more precise test patterns and to filter out invalid test cases at early stages of the unfolding process. From a methodological point of view, this requires to augment the test pattern with state predicates, behaviour selectors, and filtering keys, which keep the incremental process within acceptable bounds. This approach has been defined in the context of the ANR TASCCC Project. We intend to apply it soon to the Global Platform case study provided by Gemalto, a last-generation smart card operating system[8]. This model presents 3 billions of possible atomic instantiated operation calls, due to combination of operations parameters values. A large proportion of these latters represent erroneous situations that should not be considered.

Acknowledgment This research is supported by the ANR TASCCC Project under grant ANR-09-SEGI-014.

References

1. Abrial, J.R.: The B Book - Assigning Programs to Meanings. Cambridge University Press (Aug 1996)
2. Belhaouari, H., Peschanski, F.: A constraint logic programming approach to automated testing. In: The 24th Int. Conf. on Logic Programming. pp. 754–758. ICLP '08, Springer (2008)
3. du Bousquet, L., Ledru, Y., Maury, O., Oriat, C., Lanet, J.L.: Reusing a JML specification dedicated to verification for testing, and vice-versa: case studies. Journal of Automated Reasoning, Springer 45(4) (2010)
4. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: ECOOP'02. No. 2374 in LNCS, Springer (2002)
5. Cohen, D.M., Dalal, S.R., Parelius, J., Patton, G.C.: The combinatorial design approach to automatic test generation. IEEE Softw. 13(5), 83–88 (1996)

[8] http://www.globalplatform.org/specifications.asp

6. Constant, C., Jeannet, B., Jéron, T.: Automatic test generation from interprocedural specifications. In: Testing of Software and Communicating Systems, LNCS, vol. 4581, pp. 41–57. Springer (2007)
7. Dadeau, F., Ledru, Y., Bousquet, L.D.: Directed random reduction of combinatorial test suites. In: Random Testing '07. pp. 18–25. ACM (2007)
8. Dadeau, F., Tissot, R.: jSynoPSys – a scenario-based testing tool based on the symbolic animation of B machines. In: Finkbeiner, B., Gurevich, Y., Petrenko, A. (eds.) MBT'09 proceedings. ENTCS, vol. 253-2, pp. 117–132 (2009)
9. Dupuy-Chessa, S., du Bousquet, L., Bouchet, J., Ledru, Y.: Test of the ICARE platform fusion mechanism. In: DSVIS'05. LNCS, vol. 3941. Springer (2006)
10. Ferro, L., Pierre, L., Ledru, Y., du Bousquet, L.: Generation of test programs for the assertion-based verification of TLM models. In: Design and Test Workshop, 2008. IDT 2008. 3rd International. pp. 237–242. IEEE (dec 2008)
11. Fraser, G., Wotawa, F.: Redundancy based test-suite reduction. In: 10th Int. Conf. on Fundamental Approaches to Software Engineering. pp. 291–305. FASE, Springer (2007)
12. Grieskamp, W., Qu, X., Wei, X., Kicillof, N., Cohen, M.B.: Interaction coverage meets path coverage by SMT constraint solving. In: The 21st IFIP WG 6.1 Int. Conf. on Testing of Software and Communication Systems and 9th Int. FATES Workshop. pp. 97–112. (TESTCOM/FATES), Springer (2009)
13. Harrold, M.J., Gupta, R., Soffa, M.L.: A methodology for controlling the size of a test suite. ACM Trans. Softw. Eng. Methodol. 2(3), 270–285 (1993)
14. Heimdahl, M., George, D.: On the effect of test-suite reduction on automatically generated model-based tests. Automated Software Engineering 14, 37–57 (2007)
15. Jagannath, V.and Lee, Y., Daniel, B., Marinov, D.: Reducing the costs of bounded-exhaustive testing. In: Fundamental Approaches to Software Engineering. LNCS, vol. 5503, pp. 171–185. Springer (2009)
16. Lausdahl, K., Lintrup, H.K.A., Larsen, P.G.: Connecting UML and VDM++ with open tool support. In: Cavalcanti, A., Dams, D. (eds.) FM 2009: Formal Methods, Second World Congress. LNCS, vol. 5850, pp. 563–578. Springer (2009)
17. Ledru, Y., du Bousquet, L., Maury, O., Bontron, P.: Filtering TOBIAS combinatorial test suites. In: ETAPS/FASE'04 – Fundamental Approaches to Software Engineering. LNCS, vol. 2984, pp. 281–294. Springer (2004)
18. Ledru, Y., Dadeau, F., du Bousquet, L., Ville, S., Rose, E.: Mastering combinatorial explosion with the Tobias-2 test generator. In: IEEE/ACM Int. Conf. on Automated Software Engineering. pp. 535–536. ACM (2007), demonstration
19. Maury, O., Ledru, Y., Bontron, P., du Bousquet, L.: Using Tobias for the automatic generation of VDM test cases. In: 3rd VDM Workshop (in conjunction with FME'02) (2002)
20. Rothermel, G., Harrold, M.J., Ostrin, J., Hong, C.: An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Int. Conf. on Software Maintenance. pp. 34–43. IEEE (1998)