Directed random reduction of combinatorial test suites.

Frédéric Dadeau, Yves Ledru, Lydie du Bousquet.

Abstract

Combinatorial testing consists in generating (possibly large) test suites by combining both the sequencing of several operations, and the selection of test data. The TOBIAS tool is based on this generation technique. The combinatorial part of the approach makes both its strength and its weakness. Indeed, the more tests are produced, the more confidence the user may have in his/her program. Nevertheless, simple patterns may result in millions of test cases due to the combinatorial explosion problem, leading to intractable testsuites. To overcome this weakness, TOBIAS makes it possible to connect the generator to ``selectors" that choose a subset of the test suite based on various techniques or criteria.

This paper presents such a test suite reduction technique, based on a stochastic approach. The selection is improved so a uniform repartition of test cases is performed w.r.t. a given criterion, namely, an abstraction of the test cases based on their ``shape". The approach is implemented into TOBIAS as a generic selector, independent of the system under test. It has been put into practice on a smart card case study, for which the results are reported in this paper.

Frédéric Dadeau, Yves Ledru, Lydie du Bousquet.
Directed random reduction of combinatorial test suites.
RT'07: Proceedings of the 2nd international workshop on Random testing, :18--25, New York, NY, USA, 2007.

Directed Random Reduction of Combinatorial Test Suites

Frédéric Dadeau,Yves Ledru, Lydie Du Bousquet
Laboratoire d'Informatique de Grenoble
B.P. 72
38402 Saint-Martin d'Hères, France
{Frederic.Dadeau,Yves.Ledru,Lydie.du-Bousquet}@imag.fr

ABSTRACT

Combinatorial testing consists in generating (possibly large) test suites by combining both the sequencing of several operations, and the selection of test data. The TOBIAS tool is based on this generation technique. The combinatorial part of the approach makes both its strength and its weakness. Indeed, the more tests are produced, the more confidence the user may have in his/her program. Nevertheless, simple patterns may result in millions of test cases due to the combinatorial explosion problem, leading to intractable testsuites. To overcome this weakness, TOBIAS makes it possible to connect the generator to "selectors" that choose a subset of the test suite based on various techniques or criteria.

This paper presents such a test suite reduction technique, based on a stochastic approach. The selection is improved so a uniform repartition of test cases is performed w.r.t. a given criterion, namely, an abstraction of the test cases based on their "shape". The approach is implemented into TOBIAS as a generic selector, independent of the system under test. It has been put into practice on a smart card case study, for which the results are reported in this paper.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Combinatorial testing, random testing*

Keywords

Combinatorial testing, test suite reduction, TOBIAS

1. INTRODUCTION

Testing takes an important part in the current validation techniques. Of course, testing can only reveal the presence of errors. Showing the absence formally requires exhaustive testing. Still, a practical solution to address this weakness is to produce large test suites, containing numerous test cases that are more likely to find numerous errors. Since writing large test suites is an important and redundant effort, automation, or at least semi-automation of the tests generation is welcome.

Combinatorial testing [4] provides an automated approach for the generation of large test suites of relevant test cases with a minimum of effort. It consists in designing test patterns that will automatically be unfolded. Combinatorial testing unburdens the validation engineer from the clerical tasks of writing repetitive tests and lets him concentrate on the insightful activities of the test design.

A combinatorial approach for testing is of course an advantage, since large test suites are more prone to detect errors, but also a drawback, since it suffers from combinatorial explosion. Nevertheless, several techniques exist to master the combinatorial explosion, such as pairwise coverage of input parameters [20]. Also, test suite reduction techniques [12, 13] consist in selecting a subset of a test suite which presents the same coverage as the complete test suite, w.r.t. a given criterion (code coverage, specification coverage, etc.).

Several tools exist for performing combinatorial testing, such as AETG [3], JMLUnit [2] or Yagg [5]. In this category, we designed several years ago, such a tool, named TOBIAS [16]. This tool has been successfully used in several case studies, including a mini-banking application [7] and a multi-modal fusion engine [8]. These experiments showed that the tool actually improves the productivity of the test engineer: a large test suite can be produced from a few lines of combinatorial description, which give a compact and structured description of the test suite. Other experiments showed that the test suites generated by TOBIAS are generally more complete and detect more errors than manually generated test suites [17]. In 2006, we undertook a complete redesign of the tool, that gave birth to TOBIAS-2. This new version of TOBIAS-2 offers interesting possibilities to plug-in test suite reduction mechanisms.

We present in this paper a test suite reduction technique based on a stochastic selection of the test cases. We define a partition of the test suite using an abstraction function that is based on the structure of the test case. We argue that a purely random selection is, in the large majority of cases, not uniformly distributed over the partition of the test suite.

The abstraction function aims at distinguishing equivalence classes within the test cases. Then, a random selection of the test cases is performed within each of these equivalence classes. We propose an abstraction based on the shapes of the test cases, that balances this incorrect distribution. This approach defines an application-independent way to

RT '07, November 6, 2007, Atlanta, GA, USA

reduce a test suite, without having to involve external tools, such as coverage measurement tools. This principles are put into practice using TOBIAS-2.

This paper is organized as follows. Section 2 presents the running example that will be used to illustrate the principles describes in this paper. Section 3 describes the TOBIAS tool and a first experiment. Then, Section 4 presents the test suite reduction possibilities and illustrates them on an example. Section 5 presents a smart random reduction of a test suite. Finally Section 6 presents the related work, and Section 7 summarizes the approach and draws the perpectives of this work.

2. RUNNING EXAMPLE

We consider the example of a smart card application, representing an electronic purse (e-purse). This e-purse manages its balance, and two pin codes, one for the bank and one for the holder. Similarily to smart cards, the e-purse has a life cycle, starting with a personalization phase, in which the values of the bank and holder pin code are set. Then, a normal usage phase makes it possible to perform standard operations such as holder authentification (by checking his pin), credit, debit, etc. When the holder fails to authentify himself, the card is invalidated, meaning that the bank has to authenticate to unblock the card. The bank is also limited, and successive failures in the bank authentification attempts make the card return to the personalization phase. Each sequence of operation is performed within sessions, which are initiated through different terminals. All the methods and their signature are given in Table 1.

This example has originally been designed to illustrate access control mechanisms, and it is used as a basis for test generation for access control[1] in the context of the French RNTL POSE project[2].

Access control is introduced by specifying that given operations of the e-purse may only be executed from specific terminals. We distinguish three kind of terminals: administrative terminals, dedicated to personalization, and bank and PDA terminals. An additional constant symbolizes the absence of terminal. Thus, values of the `byte` parameter of `beginSession` have to be chosen amongst constants `ADMIN`, `BANK`, `PDA`, `NONE` from the `Terminal` class. An access rule of the security policy states, for example, that the `checkPin` operation can only be executed from a bank terminal or from a PDA.

The e-purse is described within a Java class, named `Bankcard`, annotated by JML pre- and postconditions [15]. These describe the behaviors of the Java classes. The JML compiler, `jmlc`, produces a Java bytecode enriched with the runtime verification of the JML assertions. This feature is used, in most of the approaches –including this one–, as an oracle [2]. Indeed, the execution of a test case on a `jmlc`-compiled program that does not violate any JML assertion is considered as a success, otherwise it is a failure. The application is designed to be robust, so that every precondition is always true, and tests can not become inconclusive.

This example is used to exercise access control testing methodologies. In this context, our objective is to use a combinatorial approach for the generation of test cases. Nevertheless, this kind of application is related to the smart card industry world, it does not make sense to generate millions of test, because the process of running the tests on the card is highly time-consuming. Hence, a trade-off must be found on the number of test cases. Therefore, we are looking for a way to reduce the size of the produced test suites.

[1]the source code of the application is available at `http://www-lsr.imag.fr/Les.Personnes/Amal.Haddad/AFADL07`

[2]`http://www.rntl-pose.info`

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

Table 1: Methods of the Case Study

In summary, this example was selected here for two reasons. First, the example and the associated test patterns were developed independently of the test suite reduction technique, and are therefore not biased towards the technique. Second, it belongs to an application domain (smart cards) where reducing the size of the test suite definitely makes sense due to the execution time of test suites.

Now, let us focus on how combinatorial testing can be employed to test this application.

3. TOBIAS

TOBIAS is a semi-automated combinatorial test generation tool. Its design started from the observation that a test suite often contains a lot of similar test cases that can be generated by combinatorial unfolding. Thus, the tool proposes to the validation engineer the possibility to express his/her tests in a concise way, to unfold this description into a set of abstract tests cases, and finally to concretize the tests for a target technology (JUnit, CppUnit, etc.). TOBIAS "captures" the know-how and the knowledge of the validation engineer expressed in the test patterns. The TOBIAS test patterns can be assimilated to bounded regular expressions over the alphabet described by the instantiated operations of the system under test. The tool makes it possible to perform combinations that will iterate over pertinent sequences of operations, involving specific parameter values.

In order to illustrate the TOBIAS pattern unfolding, let us consider a simple pattern:

$$\texttt{credit}(200); \{\texttt{credit}(100), \texttt{debit}(\{100, 200, 300\})\}^{1..2}$$

This patterns starts the test with a credit of the e-purse, before repeating, between 1 and 2 times, either a credit, or a debit. Notice that three possible values can be used for the debit. This patterns unfolds into 20 test cases, that are displayed in Figure 1.

3.1 Combinatorial Testing with TOBIAS

The strength of TOBIAS resides in its combinatorial principles, since it makes it possible to create large test suites without any redundant effort. Nevertheless, this feature also represents its weakness. Indeed, due to its intrinsic principles, TOBIAS suffers from combinatorial explosion. It is

(1)	`credit(200); credit(100)`
(2)	`credit(200); credit(100); credit(100)`
(3)	`credit(200); credit(100); debit(100)`
(4)	`credit(200); credit(100); debit(200)`
(5)	`credit(200); credit(100); debit(300)`
(6)	`credit(200); debit(100)`
(7)	`credit(200); debit(200)`
(8)	`credit(200); debit(300)`
(9)	`credit(200); debit(100); credit(100)`
(10)	`credit(200); debit(200); credit(100)`
(11)	`credit(200); debit(300); credit(100)`
(12)	`credit(200); debit(100); debit(100)`
(13)	`credit(200); debit(100); debit(200)`
(14)	`credit(200); debit(100); debit(300)`
(15)	`credit(200); debit(200); debit(100)`
(16)	`credit(200); debit(200); debit(200)`
(17)	`credit(200); debit(200); debit(300)`
(18)	`credit(200); debit(300); debit(100)`
(19)	`credit(200); debit(300); debit(200)`
(20)	`credit(200); debit(300); debit(300)`

Figure 1: Unfolding of the example test pattern

easy to write patterns that unfold into millions of test cases and whose execution requires unacceptable resources. To avoid combinatorial explosion, we first rely on the intelligence of the validation engineer, who is requested to write patterns that do not explode. Nevertheless, it may happen that writing such patterns may require significant efforts. For this reason, we moved on to the idea of reducing the test suite, and we redesigned TOBIAS in this way.

The last version of TOBIAS builds on the lessons learned from the first one and proposes interesting features, that are the basis of the work presented in this paper:

- a larger expressiveness of the input and output languages, written in the XML format, that makes it possible to express a large variety of compositions;
- low-cost algorithms that make it possible to unfold more than a million of test cases;
- an extensible architecture that offers new mechanisms to master combinatorial explosion, through the writing of plug-ins for filtering test cases, or selecting subsets of test suites.

We now illustrate the use of the combinatorial approach to generate test cases for the example given in Sect. 2.

3.2 Experiments with TOBIAS

We have designed 9 TOBIAS patterns to exercise the access control mechanisms involved in the e-purse implementation. Then, we have experimented our test suites on a mutant detection exercise. Therefore, we have re-used a bench of mutants, used for another experiment on evaluating the efficiency of tests for access control policies. These mutants have been obtained by applying systematic modifications to the reference implementation. We have supposed that no malicious programming error is done, ie., that the programmer does not intentionnally introduce errors into the source code. The mutations were guided by only focusing on the low-level implementation of the access control security policy, namely, the verification of the access conditions. The mutations performed on the source code include: inversion of logical *and*s and *or*s, negation of conditions, replacement of litterals by *true* or *false*, etc. Altogether, 57 mutants have been produced. Each mutant includes a single error and no mutant is equivalent to the original program.

Pattern	# tests	instr. cov.	mutants killed
1	40	228/598 (38%)	26 (45%)
2	50	132/598 (22%)	14 (24%)
3	80	222/598 (37%)	22 (38%)
4	218	191/598 (32%)	18 (31%)
5	250	101/598 (17%)	10 (17%)
6	512	191/598 (32%)	18 (31%)
7	1364	137/598 (23%)	12 (21%)
8	3600	211/598 (35%)	26 (45%)
9	9600	211/598 (35%)	26 (45%)

Table 2: Test suites and efficiency

Pat-tern	% original suite	instruction coverage	mutants killed
4	**3-5-10-20-30**	**160/598 (27%)**	**18/57 (31%)**
5	**3-5-10**	**88/598 (15%)**	**9/57 (15%)**
5	20-30	101/598 (17%)	10/57 (17%)
6	3-5-10-20-30	191/598 (32%)	18/57 (31%)
7	3-5-10-20-30	137/598 (23%)	12/57 (21%)
8	3-5-10-20-30	211/598 (35%)	26/57 (45%)
9	3-5-10-20-30	211/598 (35%)	26/57 (45%)

Table 3: Random Reduction Efficiency

Table 2 summarizes the number of test cases resulting from the unfolding of the different patterns, the code coverage of the test suites, measured with EMMA [9], and finally the number and percentage of mutants killed. Notice that our objective was not to kill all the mutants, we only wanted to establish a basis for evaluating the test suite reduction efficiency.

Large test suites can be an advantage when the execution time on the system under test is small. Nevertheless, even Java test suites run on a desktop computer can possibly take a large amount of time to be compiled, and executed. For example, test suite resulting from pattern 9 sizes about 40 MBytes of JUnit source code, and takes 12 minutes to be compiled, requiring an extension of the JVM heap memory on a recent computer (Intel Centrino 1.4 GHz, 1 GB RAM). In addition, the code coverage tool fails to measure the coverage on-the-fly and requires some tricky mechanisms to be employed with this suite.

Such technical difficulties show that it quickly becomes mandatory to reduce the test suites, when using combinatorial testing. We show, in the next two sections, techniques using a random selection of the test cases within a test suite, and we evaluate on the case study the efficiency of the reduced test suite.

4. MASTERING COMBINATORIAL EXPLOSION

TOBIAS offers two possibilities for reducing test suites' sizes. They act as plug-ins that are written by the validation engineer and selected to be applied on the test schemas, during the unfolding process.

4.1 TOBIAS Inner Reduction Mechanisms

The first kind of plug-in is the *constraint*, which is invoked for each test case of the test suite that is generated during the unfolding. A constraint is a boolean function that takes as input the test case in XML output format of TOBIAS. This function returns a boolean indicating whether the test case is kept, or not, in the resulting test suite. Such a function is called every time a test case is unfolded, and thus, this test suite reduction mechanism may only be employed when the selection criterion of a test case is independent from the other test cases of the suite. Early examples of such contraints were already given in [16].

For example, we can filter the test cases of Figure 1 in order to reject the ones that lead the balance to be negative at some point during the test. For example, test case 11 has a balance of -100 after the "debit" operation and should be eliminated based on this criterion. Test cases 8, 11, 15, 16, 17, 18, 19, and 20 would be eliminated based on this criterion. The resulting test suite includes only 60% of the original one. Such a technique can be very efficient to reduce the size of the test suite but it requires to carefully select the constraint applied to the test cases. It is the responsibility of the test engineer, based on the requirements documents, to decide that ignoring the test cases that do not fulfill the criterion leads to an "equivalent" test suite. Moreover, the constraints used to filter the test suite are usually dependent of the application (here they interpret the behaviour of "debit" and "credit") and can rarely be reused for a different context.

The second kind of plug-in is the *selector*, which is invoked during the unfolding of a test suite. The selector is a function that, given a set of test cases numbered from 1 to the size of the unfolded suite, returns an array containing the indexes of the tests composing the resulting test suite. Contrary to constraints, selectors make it possible to consider the whole test suite. Thus, a test case can be selected by being compared with the other test cases of the full suite.

TOBIAS is written in the Java programming language. The additional plug-ins are written using Java classes (extending a specific abstract class depending on the type of the plug-in, constraint or selector). The plug-in classes to apply are specified in the input pattern. The TOBIAS engine applies them during the unfolding by systematically invoking the entry-point function of the plug-in, with the correct parameter, the unfolded test case for the constraint, or the folded group for the selector.

4.2 Random Reduction of Test Suites

The TOBIAS selection mechanism is able to reduce a test suite by selecting which tests are kept during the unfolding of the test pattern. We assume that a test suite reduction algorithm is parameterized by the size of the reduced test suite (i.e. the number of test cases that have to be selected from the complete test suite). A simple selector, based on random choice, easily selects a given number of test cases out of a given test suite. Such an algorithm is similar to the function `randomSelect` displayed in Figure 2. This algorithm considers two sets of tests: FullTS (resp. RedTS) is an ordered set (on which usual set operators apply), representing the full test suite (resp. the reduced test suite). Notation $S[i]$ means that we access the i^{th} element of the ordered set S.

We assume that the random number generator, called by function "random" in the algorithm, is uniformly balanced over the considered range of integers. Thus, all the test cases of a full test suite have an equal probability to be chosen from the set. Thus, each test case can be chosen with a probability of $1/card(FullTS)$.

```
RedTS ← randomSelect(FullTS, size)
begin
    if (size > card(FullTS)) then
        return FullTS
    end if
    RedTS = ∅
    while (size ≥ 0) do
        rand = random(0, card(FullTS))
        RedTS = RedTS ∪ {FullTS[rand]}
        FullTS = FullTS \ {FullTS[rand]}
        size = size - 1
    done
    return RedTS
end
```

Figure 2: Random Selection Algorithm

4.3 Experimenting the Random Reduction

We run our selector on the test suite to obtain randomly reduced test suites whose sizes equal 3%, 5%, 10%, 20% and 30% of the original suites' size. We exercise these reduced test suites on the same test bench that was used previously. Table 3 displays the results of the evaluation of the reduced test suites. In order to present reasonable results, we have chosen to reduce only test suites that contained more than 100 test cases (patterns 4–9). Actually, for most patterns, the instruction coverage and number of mutants killed remains constant for sizes ranging from 3 to 30%. Only pattern 5 has lower performances for sizes under 20%.

Due to the design of the test patterns, the random reduction of the suite roughly presents the same code coverage, and mutant detection capabilities, than the full test suite, except in two cases: pattern 5, for a reduction under 20%, and pattern 4 whatever the reduction might be. The reduced pattern 4 only covers 27% of the code, while the full test suite led to a coverage of 32%. Pattern 5, when significantly reduced (3 to 10%) only covers 15% of the code and 15% of the mutants (vs. 17% for the full test suite). Of course, the reduced test suite can obviously not cover more code or kill more mutants than the original suite. But, as we will see in the next section, it is possible to improve the performances for reduced versions of patterns 4 and 5 and overcome this weakness. Therefore, we propose a directed randomized test suite reduction technique.

5. DIRECTED RANDOM REDUCTION

We noticed the fact that a purely random selection is not uniformly spread amongst the interesting cases. Indeed, the essence of combinatorial testing resides in the fact that it captures, by means of test patterns –ie. specific method sequencing– the know-how of the test engineer. A purely random selection of the test cases may, in several cases, leave frequently apart method sequencing that have even be designed by the validation engineer.

We define a "shape" of a test case as the sequence of

method names involved in the test case. For example,

$$\texttt{credit(100)}; \texttt{debit(300)}; \texttt{credit(100)}$$

abstracts to the following shape:

$$\texttt{credit}; \texttt{debit}; \texttt{credit}$$

Intuitively, two test cases with the same shape are more similar than two test cases with different shapes. Hence, we will use the shapes of the test cases to group them into "equivalence" classes.

Figure 1 shows the equivalence classes of the unfolded test suite. The largest class lists test cases 12 to 20, while the smallest classes correspond to a single test (test cases 1 and 2 correspond to these classes). There is thus 9 times more chances that a random selection chooses a test case from the largest class than from the smallest one.

TOBIAS test suites often feature such a difference between the sizes of equivalence classes. As in this example, longer test cases offer more opportunities to perform combinations of parameters. When the unfolding of a pattern leads to test cases of various lengths, the longer shapes have often more representative than the shorter ones as shown in Figure 1.

5.1 When random selection is not sufficient ...

To illustrate this fact on our example, consider test pattern 4, from the case study. This pattern is defined by the following groups:

$$\mathrm{Gr}_{\mathrm{Perso}}; \mathrm{Gr}_{\mathrm{CheckPinOK}}; \mathrm{Gr}_{\mathrm{Modify}}$$

which are defined by:

$$\begin{aligned}
\mathrm{Gr}_{\mathrm{Perso}} &= \texttt{beginSession(ADMIN); setBpc(1234);} \\
&\quad \texttt{setHpc(7187); endSession();} \\
\mathrm{Gr}_{\mathrm{CheckPinOK}} &= \texttt{beginSession(PDA); checkPin(7187)} \\
\mathrm{Gr}_{\mathrm{Modify}} &= \{\texttt{credit(100)}^3, \\
&\quad \texttt{debit}(\mathrm{Gr}_{\mathrm{ValDebit}})^3, \\
&\quad \texttt{getBalance()}\} \\
\mathrm{Gr}_{\mathrm{ValDebit}} &= \{0, -1, 1, 1000, 32768, 3000\}
\end{aligned}$$

This pattern starts with a personalization phase, which sets the pin codes for the bank and for the holder. Then it correctly identifies the holder, and it proceeds to the test itself, namely, either crediting three times the purse, or debiting the purse with 6 possible values three times in a row, or getting the value of the balance. The experienced reader will have noticed that this pattern is not necessarily meaningful; it is only designed for didactical purposes, in order to illustrate this paper. However, the eight other patterns have been designed independently from this kind of consideration.

This pattern is unfolded into 218 test cases (1 for the credits + 6^3 for the debits + 1 for the balance) corresponding to the unfolding of group $\mathrm{Gr}_{\mathrm{Modify}}$. If we only consider the "shapes" of the test cases in terms of method sequencing, we have the following probabilities for each shape to be selected during the random reduction of the test suite:

Prefix; `credit`; `credit`; `credit`	1/218
Prefix; `debit`; `debit`; `debit`	216/218
Prefix; `getBalance`	1/218

with

$$\begin{aligned}
\text{Prefix} &= \texttt{beginSession}; \texttt{setBpc}; \texttt{setHpc}; \texttt{setBpc}; \\
&\quad \texttt{endSession}; \texttt{beginSession}; \texttt{checkPin}
\end{aligned}$$

When randomly selecting a subset of the test suite, the second sequence is more likely to be chosen, and the first and last ones would, most of the time, not be selected. Statistically the selection algorithm has to be run $218/n$ times (n being the size of the reduced test suite) so that the first and last shapes appear at least in one test case.

Our opinion is that if the validation engineer has chosen to produce this sequence of method calls, then (s)he did it on purpose, and the test suite reduction should take this fact into account. Leaving these shapes appart would intuitively reduce the potential coverage of the test suite, and decrease its error detection capabilities. In order to increase the scores of code coverage and mutant killing, we propose to guide the selection of the test cases that will compose the reduced suite.

5.2 A Shape-based Random Reduction

The objective is to guide the selection of the test cases during the reduction so that all the shapes are covered within the reduced test suite. This technique is similar to test suite reduction techniques based on the preservation of a given coverage criterion, as explained in [13]. Here the coverage criterion is defined on the text of the test cases instead of the structure of the code.

The shape-based reduction selector works in two parts. In a first part, a tree representation of the test cases is drawn, in order to identify and balance the shapes. As an illustration, consider again this simple pattern:

$$\texttt{credit}(200); \{\texttt{credit}(100), \texttt{debit}(\{100, 200, 300\})\}^{1..2}$$

This pattern is unfolded into 20 test cases that are balanced amongst the different shapes as illustrated by the tree depicted in Figure 3.

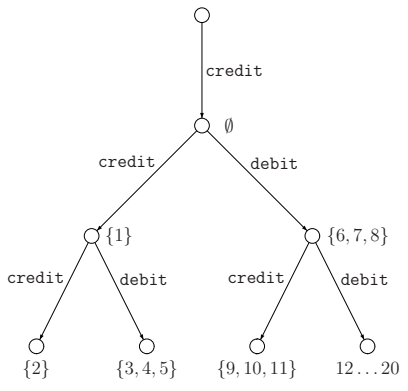

Figure 3: Tree Representing the Shapes

This tree reads as follows. The number in braces represent the identifiers of the unfolded test cases that compose the test suite. It is thus possible to compute the different

probabilities for each shape to appear. In this example:

`credit;credit`	1/20	($\{1\}$)
`credit;credit;credit`	1/20	($\{2\}$)
`credit;credit;debit`	3/20	($\{3,4,5\}$)
`credit;debit`	3/20	($\{6,7,8\}$)
`credit;debit;credit`	3/20	($\{9,10,11\}$)
`credit;debit;debit`	9/20	(12..20)

The second part of the selector is the selection of the test cases. This selection is performed using the algorithm given by Figure 4.

The principle of the algorithm is the following. First, the full test suite (FullTS) is decomposed into a set of test suites, using a partitioning function φ, that decomposes the full test suite into subsets. PartTS is the set of sets resulting from this partition. Each subset corresponds to a shape (or an equivalence class). Then, within each subset, a given number of test cases is randomly selected, using the `randomSelect` function described in Figure 2. This algorithm strives to uniformly balance the number of selected test cases within each subset. Since this selection may require more test cases than the subset contains, a remainder is managed. When all the partitions have been covered, a random selection is performed over the remaining unselected tests.

Notice that the algorithm guarantees that if the size of the reduced test suite is greater than the number of shapes, then each shape will appear at least once in the reduced test suite.

Table 4 displays the results of the evaluation of the shaped-based reduced test suites from patterns 4–9. The results are the same as for the corresponding full test suites. The experimentation of the shape-based reduction technique lets us draw the following conclusions.

1. It is efficient on compliant patterns. The shape-based reduction of pattern 4 makes it possible to increase its coverage, by selecting, on purpose, non-frequent shapes. This is not surprising since pattern 4 was built specifically for this study and is biased towards our technique. Nevertheless it shows that the technique works where expected.
2. It is also efficient on general patterns. Pattern 5 was defined independently of this study and table 3 has shown that it lost some of its detection capabilities for reductions to 3, 5 and 10% of the original size. The shape-based reduction of pattern 5 makes it possible, even for strong reductions (between 3 and 10 percent) to reach the highest possible code coverage rate, as well as the maximal possible efficiency on error detections.
3. It costs more than a simple random reduction. Indeed, this reduction technique requires the test pattern to be unfolded so that the computation of the partitioning may happen. On the contrary, a random selection only unfolds the selected test cases, and not the complete test set.
4. It does not loose efficiency. In this case study, the results obtained for the reduced test suites are never worse than those obtained by applying a random reduction, whatever the size of the test suite is.

```
RedTS ← directedRandomSelect(FullTS, size)
begin
    if (size > card(FullTS)) then
        return FullTS
    end if
    RedTS = ∅
    PartTS = φ(FullTS)
    nb = size div card(PartTS)
    rem = size div card(PartTS)
    foreach (Part ∈ PartTS) do
        RedTS = RedTS ∪ randomSelect(Part,nb)
        if nb > card(Part) then
            rem = rem + (nb - card(Part))
        end if
    done
    if rem > 0 then
        Red2 = randomSelect(FullTS\RedTS,rem)
        RedTS = RedTS ∪ Red2
    end if
    return RedTS
end
```

Figure 4: Directed Random Selection Algorithm

The directed selection algorithm performs an uniform selection over the test cases based on the subsets of the full test suite. These latter are computed using an abstraction function, in our study, a function that decomposes the test cases according to their shapes, ie., the sequence of methods they display, abstracting the parameters. This abstraction based on the shapes is not the only one that can be done, although it is the only one implemented so far in TOBIAS. We now present how our test suite reduction technique can be employed in a more generic way.

5.3 A Generic Reduction Mechanism

This selection of the test cases relies on two distinct elements. The first one is an abstraction function that decomposes the test suite into partitions. The second one is a random selection algorithm that aims at exhibiting at least one test case from each set of the partition of the test suite.

These two elements are independent. If the algorithm that performs the selection itself is always the same (given in Figure 4), the abstraction φ function may vary at will. In order to guarantee the uniformity of the selection, the abstraction function has to fulfill several requirements.

Properties for the Abstraction Function.

First, it takes as an input a set of test cases and returns of set of non-empty sets of test cases. Let Γ be the set of the tests cases, the signature of φ has to be the following.

$$\varphi : \Gamma \rightarrow 2^{\Gamma} \setminus \emptyset.$$

Second, the subsets of the full test suite have to be a partition of it, meaning that they are pairwise disjoint and their union is equal to the full test suite. Let TS be a test suite, and let φ be the abstraction function, then:

$$\forall s_1, s_2 . s_1 \in \varphi(TS) \wedge s_2 \in \varphi(TS) \wedge s_1 \neq s_2 \Rightarrow s_1 \cap s_2 = \emptyset$$

$$\wedge \bigcup_{s \in \varphi(TS)} s = TS.$$

Pattern	# Test Cases	# shapes	% original size	instruction coverage	mutants killed
4	218	3	3-5-10-20-30	191/598 (32%)	18/57 (31%)
5	250	4	3-5-10-20-30	101/598 (17%)	10/57 (17%)
6	512	27	3-5-10-20-30	191/598 (32%)	18/57 (31%)
7	1364	5	3-5-10-20-30	137/598 (23%)	12/57 (21%)
8	3600	162	3-5-10-20-30	211/598 (35%)	26/57 (45%)
9	9600	150	3-5-10-20-30	211/598 (35%)	26/57 (45%)

Table 4: Directed Random Reduction Efficiency

Some Examples of Abstraction Functions.

In the case study, the abstraction function that has been chosen is based on the "behaviors" identified by the shapes of the test sequences. It has been motivated by the fact that the shapes described by the validation engineer have to be considered during the reduction process. Nevertheless, other abstractions can be chosen. Here is a small overview of pertinent abstractions that can be performed.

- *Parameter-based abstraction.* Each partition of the full test suite contains test cases that cover specific values of the inputs (boundary values, etc.).
- *Coverage-based abstraction.* Each partition of the full test suite contains test cases that cover the same parts of the program/specification.
- *Mutation-based abstraction.* Each partition of the full test suite contains test cases that kill a specific mutant.

Notice that these abstraction functions are more or less independent from the application under test, and notice also, that several may require to execute the test suite at least once, which is not always possible.

6. RELATED WORK

Since rare test cases are not or poorly covered with statistical testing, Thévenod-Fosse and Waeselynck proposed a testing method combining statistical strategies with structural or functional coverages [21, 22]. The idea is to build a specific probability distribution on the input domain with respect to a given coverage criterion. The distribution maximizes the weakest probability for an element to be activated by an execution. For instance, considering the "all statements" coverage criterion, the input distribution built will avoid a too weak probability for any statement to be exercised. Authors have reported several experiments, which led to the conclusion that their approach has a better fault detection ability than uniform random testing and deterministic testing based on classical coverage criteria. A. Denise *et Al.* have proposed an extension of these works, which is more efficient [6]. It results in a tool named AuGuSTe [11].

On the other hand, several works in random testing consist in improving the input selection by observing the execution result of a test case, before producing a new one, such as Adaptive Random Testing [1, 14, 18] or feedback-directed random test generation [19]. An important point that differentiates our approach is that our test set is computed before any execution of test cases, and so, the test suite can be built before the code is developed. In a recent work [10], the authors introduce a random test generation technique based on a uniform dstribution for only a subset of paths, called Path-Oriented Random Testing. This approach is, in its essence, similar to ours, since both works aim at covering left apart cases.

7. CONCLUSION AND FUTURE WORK

This paper has presented an approach for automatically reducing combinatorial test suites. We have introduced the notion of test cases "shapes", that are resulting of the application of abstraction functions on the test cases, in order to identify a partition of the test cases. We have seen that a purely random selection was not relevant for covering all sets of the partition. Thus, we have proposed a guided algorithm that uniformly balances the selection of the test cases over the different shapes. Although the technique can be applied to all kinds of test suites, it is particularly interesting for combinatorial test suites where some shapes are more represented than others due to combinatorial explosion. This technique has been implemented using the TOBIAS selector mechanism, and put into practice on a realistic application, developed independently of this study. Indeed, the application we used was already existing, so were the mutants used for the evaluation of the test suite reduction mechanism. Since the study was carried out independently from the technique, only one of the existing test pattern did clearly benefit from it. A second pattern, built specifically for this experiment, also showed clear benefits as expected. For the other patterns the technique produces equivalent results than random selection. The results have shown that, in practice, the directed reduction gives equivalent or better results for the abstraction that we have proposed.

Nevertheless, the principles of the directed test suite reduction are generic. These principles rely on two elements, namely an abstraction function, provided by the user, and a distribution algorithm, independent from the application under test, which guarantees that at least one test case will be selected for each test shape. This principle applies whatever the abstraction function might be.

We have noticed that the relevance of the abstraction function proposed, based on the contents of the test cases, highly depends on the shape of the original test pattern. One interesting point to investigate in the future is to identify and isolate different shapes for the test patterns and to associate, for each one, a specific abstraction function. The experience has shown that, technically, it is easily feasible, due to the large possiblities of TOBIAS and its selectors.

Other techniques allow to reduce the size of a test suite. Some of these techniques, such as pairwise combinatorial generation [3, 20], or reduction on the basis of code coverage measurement [13] are not yet implemented in TOBIAS. An experimental comparison with these techniques is therefore left as future work. Other techniques [16] are based on predicate filtering. A first technique learns, from previous

executions, the prefix sequences that led to false preconditions. This technique is not relevant for this case study, which uses a defensive programming style were all preconditions are true. The second technique was briefly presented in the beginning of Sect. 4.1. It filters the testsuite based on some property expressed by the test engineer. Selecting and expressing this property relies on the skills of the test engineer and depends on each pattern and each case study, while the technique presented here is fully automated, general and independent of the case study. We did not try to use property filtering on this example and this experiment is also left as future work.

At short term, we plan to employ this in the French RNTL POSE project, which aims at combining different techniques for the certifying access control mechanisms in embedded smart-card software. As already mentioned, these test suite reduction techniques are welcome in the context of smart cards where the execution times of the tests suite on the test bench remains a crucial point.

Acknowledgements We would like to thank the anonymous referees of the RT 2007 workshop for their useful comments on this paper. Part of this work was performed within the POSE project (ANR-05-RNTL-01001), sponsored by the french National Research Agency (ANR).

8. REFERENCES

[1] T. Y. Chen and F.-C. Kuo. Is Adaptive Random Testing really better than Random Testing. In *the 1st International Workshop on Random Testing (RT 2006)*, pages 64–69, Portland, Maine, USA, July 2006. ACM.

[2] 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, Máalaga, Spain, Proceedings*, volume 2374 of *LNCS*, pages 231–255, Berlin, June 2002. Springer-Verlag.

[3] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG System: An Approach to Testing Based on Combinatorial Design. *IEEE Trans. Softw. Eng.*, 23(7):437–444, 1997.

[4] D. M. Cohen, S. R. Dalal, J. Parelius, and G. C. Patton. The Combinatorial Design Approach to Automatic Test Generation. *IEEE Softw.*, 13(5):83–88, 1996.

[5] D. Coppit and J. Lian. yagg: An Easy-To-Use Generator for Structured Test Inputs. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 356–359, New York, NY, USA, 2005. ACM Press.

[6] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A Generic Method for Statistical Testing. In *15th International Symposium on Software Reliability Engineering (ISSRE 2004)*, pages 25–34, Saint-Malo, France, November 2004. IEEE Computer Society.

[7] L. du Bousquet, Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet. A Case Study in JML-Based Software Validation. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE'04)*, pages 294–297, Washington, DC, USA, 2004. IEEE Computer Society.

[8] S. Dupuy-Chessa, L. du Bousquet, J. Bouchet, and Y. Ledru. Test of the ICARE Platform Fusion Mechanism. In *DSVIS 2005, Newcastle upon Tyne, UK*, volume 3941 of *LNCS*. Springer, 2006.

[9] The EMMA web site. `http://emma.sourceforge.net/`, 2006.

[10] A. Gotlieb and M. Petit. Path-oriented random testing. In J. Mayer and R. G. Merkel, editors, *Proceedings of the 1st International Workshop on Random Testing, RT 2006*, pages 28–35, Portland, Maine, July 2006. ACM.

[11] S.-D. Gouraud. AuGuSTe: a Tool for Statistical Testing (Experimental results). Research Report RR–1400, LRI, 2005.

[12] M. J. Harrold, R. Gupta, and M. L. Soffa. A Methodology for Controlling the Size of a Test Suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[13] J. A. Jones and M. J. Harrold. Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. *IEEE Trans. Software Eng.*, 29(3):195–209, 2003.

[14] F.-C. Kuo, T. Y. Chen, H. Liu, and W. K. Chan. Enhancing adaptive random testing in high dimensional input domains. In *2007 ACM Symposium on Applied Computing (SAC)*, pages 1467–1472, Seoul, Korea, March 2007. ACM.

[15] G. Leavens, A. 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.

[16] 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*, volume 2984 of *LNCS*, pages 281–294, Barcelona, Spain, 2004. Springer-Verlag.

[17] 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)*, 2002.

[18] J. Mayer, T. Y. Chen, and D. Huang. Adaptive random testing through iterative partitioning revisited. In *Third International Workshop on Software Quality Assurance, SOQUA 2006*, pages 22–29, Portland, Oregon, USA, November 2006. ACM.

[19] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-Directed Random Test Generation. In *29th International Conference on Software Engineering (ICSE 2007)*, pages 75–84, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[20] K. C. Tai and Y. Lie. A Test Generation Strategy for Pairwise Testing. *IEEE Trans. Softw. Eng.*, 28(1):109–111, 2002.

[21] P. Thévenod-Fosse and H. Waeselynck. An Investigation of Statistical Software Testing. *Softw. Test., Verif. Reliab.*, 1(2):5–25, 1991.

[22] P. Thévenod-Fosse and H. Waeselynck. STATEMATE Applied to Statistical Software Testing. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 99–109, Cambridge, MA, USA, june 1993.