ICSTW 2017
13-17 March 2017
Tokyo, Japan
ICST
10th IEEE International Conference on Software Testing, Verification and Validation Workshops

ICSTW 2017

Conference Sponsors

Platinum Sponsor

Gold Sponsors

Silver Sponsors

Bronze Sponsors

Copper Sponsors

2017 IEEE International Conference on Software Testing, Verification and Validation Workshops

Tokyo, Japan

13-17 March 2017

Conference Information

Copyright Page
Conference Sponsors
Author Index

Papers By Session

Government Sponsorship

Special Sponsorship

Supporters

Pattern-Based Usability Testing
by Fernando Dias, Ana C. R. Paiva

10th IEEE International Conference on Software Testing, Verification and Validation - Posters Track (ICST 2017 Posters)

A Mechanism of Reliable and Standalone Script Generator on Android
by Kuei-Chun Liu, Yu-Yu Lai, Ching-Hong Wu

EarthCube Software Testing and Assessment Framework
by Emily Law

Using Model-Checking for Timing Verification in Industrial System Design
by Laurent Rioux, Rafik Henia, Nicolas Sordon

Challenges of Operationalizing Spectrum-Based Fault Localization from a Data-Centric Perspective
by Mojdeh Golagha, Alexander Pretschner

Towards a Gamified Equivalent Mutants Detection Platform
by Thomas Laurent, Laura Guillot, Motomichi Toyama, Ross Smith, Dan Bean, Anthony Ventresque

Cloud API Testing
by Junyi Wang, Xiaoying Bai, Haoran Ma, Linyi Li, Zhicheng Ji

Automated A/B Testing with Declarative Variability Expressions
by Keisuke Watanabe, Takuya Fukamachi, Naoyasu Ubayashi, Yasutaka Kamei

Weighting for Combinatorial Testing by Bayesian Inference
by Eun-Hye Choi, Tsuyoshi Fujiwara, Osamu Mizuno

Impact of Static and Dynamic Coverage on Test-Case Prioritization: An Empirical Study
by Jianyi Zhou, Dan Hao

BDTest, a System to Test Big Data Frameworks
by Alexandre Langeois, Eduardo Cunha De Almeida, Anthony Ventresque

What You See Is What You Test - Augmenting Software Testing with Computer Vision
by Rudolf Ramler, Thomas Ziebermayr

Framework for Model-Based Design and Verification of Human-in-the-Loop Cyber-Physical Systems
by Filip Cuckov, Grant Rudd, Liam Daly

Automated Test Case Generation from OTS/CafeOBJ Specifications by Specification Translation
by Ryusei Mori, Masaki Nakamura

This site and all contents (unless otherwise noted) are Copyright ©2017 IEEE. All rights reserved.

10th IEEE International Conference on Software Testing, Verification and Validation Workshops

Test case generation with regular expressions and combinatorial techniques

Macario Polo Usaola, Francisco Ruiz Romero, Rosana Rodríguez-Bobada Aranda, Ignacio García Rodríguez
Department of Information Systems and Technologies
University of Castilla-La Mancha
Ciudad Real, Spain

***Abstract*—A test case describes a specific execution scenario of the system under test (SUT). Its goal is to discover errors by means of its oracle, that emits a pass or fail verdict depending on the SUT behavior. The test case has a sequence of calls to SUT's operations with specific test data, which may come from the application of a combinatorial algorithm. This paper describes a method to describe generic test scenarios by means of regular expressions, whose symbols point to a SUT operation. The tester assigns values to each operation's parameter. A further step expands the regular expression and produces a set of operation sequences, which are then passed to a combinatorial algorithm to generate actual test cases. Regular expressions are annotated with a set of *when* clauses, that are processed by the combinatorial algorithm to include the oracle in the generated test cases.**

***Index Terms*— Software testing, Test case generation, Oracles, Regular expressions**

I. Introduction

Test automation involves at least: (1) test data generation, (2) test case generation, (3) test case execution and (4) result analysis. Whilst test case execution and result analysis are both in research and in industrial practice solved problems, only 13% of companies use automation tools to generate test data [1]: the others use anonymized (or not) production data, spreadsheets and, for testing from the user interface, data built on the fly during exploratory testing.

Test cases consist in general of three parts [2][3]: (1) specification of the initial situation, to lead the SUT (the system under test) into the desired starting state, (2) execution of operations, and (3) comparison of the obtained and the expected results with an oracle to check whether the case has or has not found errors in the SUT.

The results returned by several test cases with the same initial situation and sequence of operations depend on the test data. Fig. 1 shows three test cases for a supposed banking *Account* class, all of them including three calls to the same three methods in the same order:

- The first case corresponds to a "normal" situation, where the tester *deposits* 1000, *withdraws* 200 and *transfers* 100. So, no exceptions are expected and the account balance should 700.
- The sequence of operations in the second test case is the same, but now the values are respectively 1000, 400 and 100. Neither we expect exceptions, but the account balance should be 500.
- Operations in the third case are also the same, but the values are now 1000, 200 and 50,000. So, we expect the SUT throws an *InsufficientBalanceException.*

```
@Test
public void testNormal_700() {
  Account account=new Account();
  try {
    account.deposit(1000);
    account.withdraw(200);
    account.transfer(new Account(), "Telephone", 100);
    assertTrue(account.getBalance()==700);
  }
  catch (Exception e) {
    fail("No exception expected");
  }
}
```

```
@Test
public void testNormal_500() {
  Account account=new Account();
  try {
    account.deposit(1000);
    account.withdraw(400);
    account.transfer(new Account(), "Telephone", 100);
    assertTrue(account.getBalance()==500);
  }
  catch (Exception e) {
    fail("No exception expected");
  }
}
```

```
@Test
public void testInsufficientBalance () {
  Account account=new Account();
  try {
    account.deposit(1000);
    account.withdraw(200);
    account.transfer(new Account(), " Telephone ", 50000);
    fail("InsufficientBalanceException expected");
  }
  catch (InsufficientBalanceException e) {}
  catch (Exception e) {
    fail("InsufficientBalanceException expected");
  }
}
```

Fig. 1. Three test cases with the same *initial situation* and *operation calls*, but with different *oracle*

The three test cases in the figure come from the same sequence, but their final structure depends on the test data passed to their operations. Combinatorial techniques resolve the problem of generating the test data, and even the generation of part of the test case code (initial situation and operation sequence). However, they do not resolve the problem of adding the oracle due to the strong dependence of the test data values. This makes that, in industrial practice, test cases are finally most times written by hand.

This article describes a method and a tool to solve the problem described, completely automating test case generation with the inclusion of oracles. The generation method is based on regular expressions (to describe test scenarios), combinatorial algorithms (to combine generic test cases with test data) and test templates. Regular expressions are enriched with a set of *when* clauses that determine the test template that must be applied to each test case.

The article is organized as follows: Section II briefly reviews regular expressions and introduces the idea of expanding them to generate test cases. Section III gives a concrete example of the problem to be solved and depicts the proposed solution. Section IV describes our algorithms to expand regular expressions. Section V, which is the core of the article, explains in parallel the algorithms and the tool. Section VI describes the algorithms and coverage criteria used to reduce and prioritize the test suite. Different examples of use of the tool are presented in Section VIII. Section VIII reviews the related work. Finally, we draw our conclusions and future lines of work.

II. REGULAR EXPRESSIONS

Regular expressions are a very powerful mechanism to describe sequences of words accepted by a regular language. Brookshear [4] gives a recursive definition of a regular expression for an alphabet ∑:

1) ∅ is a regular expression.

2) Each member in ∑ is a regular expression.

3) If p *and* q *are regular expressions, then* (p|q) *is also a regular expression, where | is the union of* p *and* q.

4) If p *and* q *are regular expressions, then the concatenation of* p *and* q (pq) *is also a regular expression.*

5) If p *is a regular expression,* p* *(Kleene closure) is also a regular expression.*

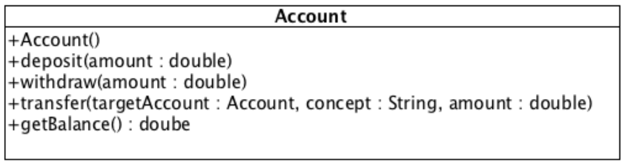
In general, regular expressions are used to check whether a given string matches the enclosed structure. For example, the word *abbabbabb* belongs to the language described by the regular expression *abb(abb)*,* since this one encloses all sequences of at least one *abb* sequence. Although its basic notation is the contained in the Brookshear's definition (mainly union, concatenation and Kleene closure), software environments have extended the possibilities with new operators to exclude, include, define sets of symbols, define maximum lengths, etc. For example, the + operator is commonly used to describe sequences with one or more elements: *(abb)+* is so the same that *abb(abb)**. In the same way, it is usual to accept that *(abb){1,5}* describes all sequences of one to five occurrences of *abb.*

The most common application of regular expressions is the detection of matching patterns: in fact, most programming languages have regular expression libraries to check text patterns, as well as word processors also have engines for searching texts using regular expressions notation.

An opposite approach could try to generate all the words accepted by means of the expansion of the regular expression. Consider for example the regular expression *a(a|b)+a.* Instead of checking whether *abba* or any other word belongs to its accepted language, we could play to generate the infinite number of words it accepts, such as: *aaa, aba, aaaa, aaba, abaa, abba,* etc.

III. DESCRIPTION OF A PROBLEM

Since test cases have a sequence of calls to the SUT's operations, regular expressions are an effective mechanism to describe such sequences. Consider we must test the behavior of the banking *Account* class shown in Fig. 2.



```
Account
+Account()
+deposit(amount : double)
+withdraw(amount : double)
+transfer(targetAccount : Account, concept : String, amount : double)
+getBalance() : doube
```

Fig. 2. A banking *Account* class

Likely, test cases will start by the creation of a new instance followed by an indeterminate number of calls to *deposit, withdraw, transfer* and *getBalance.* The structure of all these test cases will be very similar and, thus, the set of call sequences in test cases could be described by a regular expression like this:

```
new Account·[deposit|withdraw|transfer|getBalance]*
```

The expansion of that regular expression could generate a good number of sequences, such as:

- `new Account()·deposit(double)`
- `new Account()·withdraw(double)`
- `new Account()·deposit(double)·withdraw(double)`
- ...

These sequences are not yet test cases, since they still lack of the test values that must be passed to the parameters (i.e., for the *amount* parameters in *deposit, withdraw* and *transfer,* for *targetAccount* and for *concept).*

Having a sequence coming from a regular expression and a set of parameter values, actual test cases could be generated by applying a combinatorial algorithm to the sequence. For the test template `new Account()·deposit(double)` and the values *{50, 100}*, we can get two test cases with no oracle:

- `new Account()·deposit(50)`
- `new Account()·deposit(100)`

Let us assume the *Account* methods can throw three different types of exceptions:

- *NegativeAmountException,* if any of the *amount* parameters is <=0.
- *InsufficientBalanceException,* if the account's balance is not enough for the *amount* to be withdrawn or transferred.
- *NullTargetAccountException,* if the *targetAccount* parameter of *transfer* is not set.

The first two test cases of Fig. 1 (that correspond to "normal" scenarios) could proceed from a generic test case template for the situations where no exceptions are expected. The top side in Fig. 3 is a possible generic test template for this.

The third test case, which expected an *InsufficientBalanceException,* cannot come from the same template, since its structure is significantly different. So, we would need additional test case templates such as that in the bottom row of Fig. 3. When a test case is to be generated, the test case generation engine will select one or another template depending on the corresponding test data.

```
@Test
public void testNormal_XXX() {
  Account account=new Account();
  try {
    account.deposit(AMOUNT_DEPOSIT);
    account.withdraw(AMOUNT_WITHDRAW);
    account.transfer(new Account(), "Telephone",
                                AMOUNT_TRANSFER);
    assertTrue(account.getBalance()==
       AMOUNT_DEPOSIT-AMOUNT_WITHDRAW-AMOUNT_TRANSFER);
  }
  catch (Exception e) {
    fail("No exception expected");
  }
}
```

```
@Test
public void testInsufficientBalance_XXX() {
  Account account=new Account();
  try {
    account.deposit(AMOUNT_DEPOSIT);
    account.withdraw(AMOUNT_WITHDRAW);
    account.transfer(new Account(), "Telephone",
                                  AMOUNT_TRANSFER);
    fail("InsufficientBalance expected");
  }
  catch (InsufficientBalance e) {}
  catch (Exception e) {
    fail("InsufficientBalance expected");
  }
}
```

Fig. 3. Two possible test case templates for normal (top) and insufficient balance (bottom) scenarios

The template selection will be resolved by the combinatorial algorithm at test case generation time with *when* clauses. A *when* clause groups a set of test data values and points to a test case template. *When* clauses are associated to regular expressions, nor to sequences. For example, a regular expression that calls to *withdraw* or *transfer* before depositing will have associated a test case template that always will expect an exception.

Fig. 4 describes the whole process. In summary:

- In a first step, the tester describes the scenarios under test by means of regular expressions. Symbols in the regular expression correspond to operations, and each operation knows the test data to be used in its parameters. Moreover, the regular expression holds one or more *when* clauses described as a function of the parameters' values.
- Then, an expansion engine expands regular expressions and produces a set of sequences. Each generic test template represents a sequence of messages, but with neither test data nor oracle to determine the expected behavior.
- Finally, a combinatorial algorithm iterates taking each sequence and combining it with the test data. The algorithm decides the test case template to be used for the test data set it receives using the *when* clauses.

IV. Algorithms to Expand Regular Expressions

We will deal with regular expressions having the following operators: union $(p|q)$, concatenation (pq), Kleene closure (p^*), positive closure $(p+)$ and option ($p?$: zero or one occurrences of p).

The expansion of a regular expression is a function that takes two parameters: the self regular expression and the maximum length desired.

expand : RegularExpressions x Natural → {String}

Depending on the regular expression type, the expansion algorithm is different.

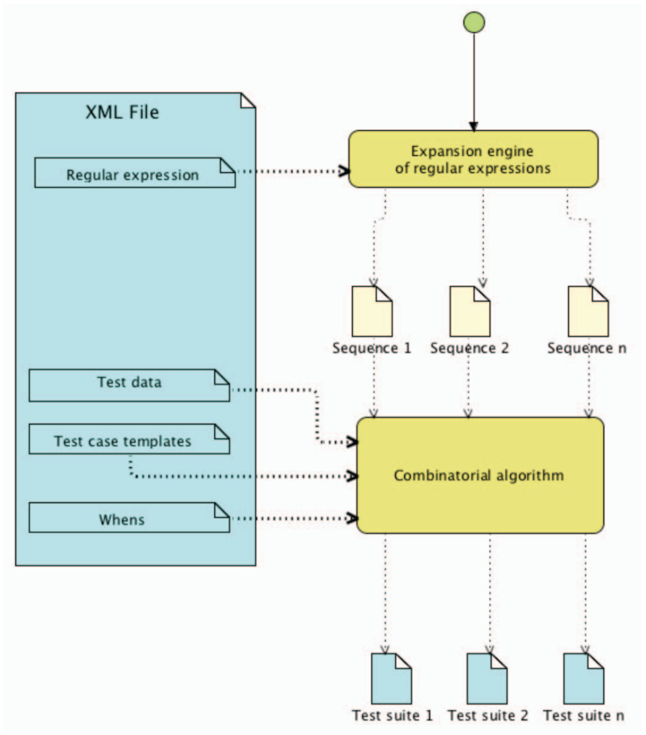


Fig. 4. General view of the test case generation process

A. Expansion of simple symbols

The expansion of a simple alphabet's symbol produces the symbol itself:

expand(symbol, length) = {symbol}

B. Expansion of the concatenation

The expansion of the concatenation of n regular expressions is the concatenation of the expansions of all of them. In the next example we use a dot (·) to represent concatenation:

$$expand(p \cdot q \cdot r, length) = expand(p, length) \cdot expand(q, length) \cdot expand(r, length)$$

Since *expand* produces a set of strings, the concatenation operator in the above expression needs to be defined. Being S, T two sets of strings:

$S \cdot \varnothing = \varnothing \cdot S = S$

$S \cdot T = \{s_i \cdot t_j, \forall s_i \in S, t_j \in T\}$

Moreover, the expressions whose length is greater than *length* are removed from the result.

C. Expansion of the union

The expansion of n regular expressions related by the union operator is the union of the expansion of the n regular expressions. For example:

$$expand(p \mid q \mid r, length) = expand(p, length) \mid expand(q, length) \mid expand(r, length)$$

Those expressions whose length is greater than *length* are removed from the result.

D. Expansion of positive and Kleene's closure

The only difference between both closures is the inclusion of the empty regular expression (let be λ) in the Kleene closure. This is, being p a regular expression:

$$p^* = \{\lambda\} \mid p+$$

Fig. 5 is a pseudocode with the expansion function of the positive closure. In practice, it calculates *length* times the Cartesian product of the expansion of *re* with itself.

1) The first argument is a regular expression *re* with the positive closure operator (+) applied.

2) Initially, it builds *seqs,* a set of string sequences with the expansion of *re.*

3) The result is flatten. I.e., all the sequences contained in the possibly obtained sets are extracted and put into *aux1*.

4) The algorithm iterates in three nested loops, adding to *result* the concatenation of the strings that are progressively built.

5) After the function, expressions longer than *length* will be discarded.

```
function expand(re+, length) : Set(String)
  // seqs, result, aux1 and aux2 are sets of strings
  seqs = expand(re, length)
  for i=1 to |seqs|
    result = result ∪ $seqs_i$
  next

  aux1 = flat(result)

  for i=1 to length
    aux2 = ∅
    for j=1 to |aux1|
      for k=1 to |seqs|
        aux2 = aux2 ∪ { $aux1_j \cdot seq_k$}  // a·b=ab
      next
      result = result ∪ { flat(aux2) }
      aux1 = aux2
    next
  next

  return result
}
```

Fig. 5. Expansion function for the positive closure

V. Implementation

We have developed a web tool[1] (available at http://alarcosj.esi.uclm.es/CombTestWeb/) implementing the test case generation process described in Section II.

The test engineer uploads a single XML file structured in several sections: *Calls, Regular expressions* and *Test case templates.* When the tool has processed this file, the tester selects a combinatorial algorithm to generate the test cases.

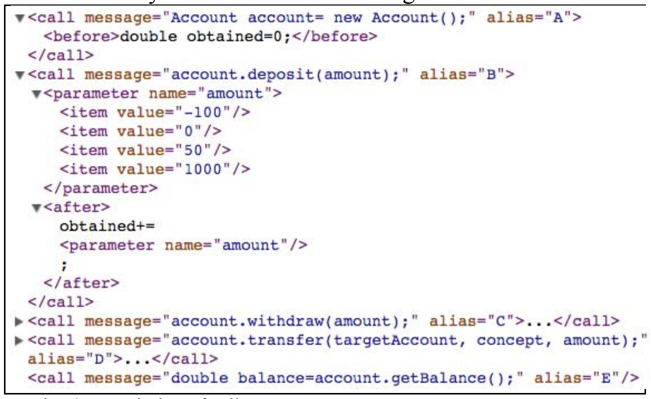
A. Description of the XML file

1) Calls.

In this section, the test engineer describes all the messages' calls that can be included in any test case. A message description consists of a signature, an alias (which is used to write regular expressions more comfortably) and a list of parameters with their values. Additionally, the tester may add text that will be included *before* or *after* the message call.

Code in Fig. 6 shows the complete description of the Account's constructor and its *deposit* method. Regarding the constructor:

- The *message* attribute contains the text that will be written in the test case. Although the code of the example is Java, it could be also C#, Cobol, the structure of a byte array, a SOAP message, or even natural language. The *alias* assigns this message the "A" letter: when the tester writes later the regular expression, s/he will reference this constructor using "A".
- The *before* tag includes code that will be inserted before any call made to this message in the test case.



```
▼<call message="Account account= new Account();" alias="A">
   <before>double obtained=0;</before>
 </call>
▼<call message="account.deposit(amount);" alias="B">
  ▼<parameter name="amount">
     <item value="-100"/>
     <item value="0"/>
     <item value="50"/>
     <item value="1000"/>
   </parameter>
  ▼<after>
     obtained+=
     <parameter name="amount"/>
     ;
   </after>
 </call>
▶<call message="account.withdraw(amount);" alias="C">...</call>
▶<call message="account.transfer(targetAccount, concept, amount);"
 alias="D">...</call>
 <call message="double balance=account.getBalance();" alias="E"/>
```

Fig. 6. Description of calls

So, test cases containing calls to the constructor will have this form:

```
@Test
public void test1() {
   double obtained=0;
   Account account= new Account();
   ...
```

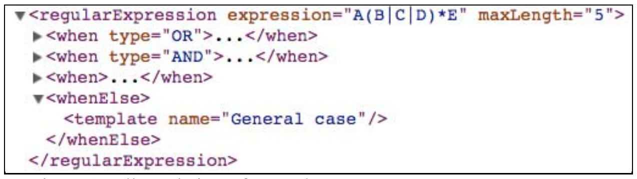
The *deposit* message in Fig. 6 includes also the test data for its *amount* parameter. Furthermore, it has an *after* tag. Inside this *after,* the tester has written the expression *obtained+=<parameter name="amount"/>;.* So, each test case calling *deposit* will include code that will sum the current value of the *amount* parameter to the *obtained* variable. Supposing 50 is the value of *amount* for a test case, the code produced will add the two last lines (***):

```
@Test
public void test1() {
   double obtained=0;
   Account account= new Account();
   account.deposit(50);             ***
   obtained+=50;                    ***
   ...
```

2) Regular expressions

The test engineer writes a regular expression for each sequence of messages s/he desires to generate. A regular expression description holds: (1) the regular expression itself, written in terms of the messages' *aliases,* (2) the *maximum length* of the generated sequences, (3) one or more *when* rules that denote which test case template must be used depending on the parameter values, and (4) a *whenElse* tag, which only contains the name of the test case template to be used when none of the previous *when* tags are applicable. The *whenElse* tag has been specially appreciated by the companies that have so far used the tool.

[1] You can log into the tool using *reviewer@reviewer.com* as login and *reviewer* as password.



```
▼<regularExpression expression="A(B|C|D)*E" maxLength="5">
  ▶<when type="OR">...</when>
  ▶<when type="AND">...</when>
  ▶<when>...</when>
  ▼<whenElse>
     <template name="General case"/>
   </whenElse>
 </regularExpression>
```

Fig. 7. A collapsed view of a *regularExpression* tag

Fig. 7 is an example of a regular expression (`A(B|C|D)*E`) corresponding to sequences containing up to 5 messages (`maxLength=5`) starting by a call to the constructor (alias *A*), ending with a call to *getBalance* (alias *E*), and with intermediate, arbitrarily sorted calls to *deposit, withdraw* and *transfer* (respectively *B, C* and *D,* according to Fig. 6).

Let us suppose test cases proceeding from this regular expression must: (1) throw a *NegativeAmountException* when any of the *amount* parameters passed to any operation is -100 or zero, (2) check an “All positive” oracle if all parameters are positive, (3) throw a *NullPointerException* when the *targetAccount* is *null* in *transfer* calls, and (4) check a “General case” oracle otherwise.

Figures 10 and 11 show two of the *when* clauses:

(1) The first one (Fig. 8) is a *when* of type *OR* for the *NegativeAmountException* situation. I.e., we are saying: “when the value of any of the *amount* parameters of *deposit, withdraw* or *transfer* is -100 or 0, use the test case template called *NegativeAmountException expected”*.

(2) The second clause (Fig. 9) is a *when* of *AND* type. It says: Wwhen the *amount* parameter of *deposit, withdraw* and *transfer* are positive, use the *All positive* test case template*”*.

The *whenElse* clause only holds a pointer to the name of test case template that must be used when the test case parameter values do not fit with any of the other *when* clauses. Test cases coming from the *whenElse* tag correspond to situations that the tester has not foreseen and, therefore, the pointed test template should emit a fault verdict to report the tester of these cases.

```
<regularExpression expression="A(B|C|D)*E" maxLength="5">
  <when type="OR">
    <parameter operationName="account.deposit(amount);"
          operationAlias="B" parameterName="amount">
       <item value="-100"/>
       <item value="0"/>
    </parameter>
    <parameter operationName="account.withdraw(amount);"
          operationAlias="C" parameterName="amount">
       <item value="-100"/>
       <item value="0"/>
    </parameter>
    <parameter
     operationName="account.transfer(targetAccount,
                                     concept,amount);"
      operationAlias="D" parameterName="amount">
       <item value="-100"/>
       <item value="0"/>
    </parameter>
    <template name="NegativeAmountException expected"/>
  </when>
   ...
```

Fig. 8. A *when* of the *OR* type

```
<when type="AND">
  <parameter operationName="account.deposit(amount);"
         operationAlias="B" parameterName="amount">
      <item value="50"/>
      <item value="1000"/>
  </parameter>
  <parameter operationName="account.withdraw(amount);"
        operationAlias="C" parameterName="amount">
      <item value="50"/>
  </parameter>
  <parameter operationName="account.transfer(targetAccount, concept,
              amount);" operationAlias="D" parameterName="amount">
      <item value="50"/>
  </parameter>
  <template name="All positive"/>
</when>
 ...
```

Fig. 9. An example of a *when* of the *AND* type

3) *Test case templates*

A test case template has an XML attribute with its name and likely a piece of plain text with a <*sequence*> tag (Fig. 10).

```
<template name="NegativeAmountException expected">
   @Test
   public void testTCNUMBER() {
      try {
         <sequence separator="\n"/>
         fail("NegativeAmountException expected");
      }
      catch (NegativeAmountException e) {}
      catch (Exception e) {
         fail("NegativeAmountException expected");
      }
   }
</template>
```

Fig. 10. An example of a test case template

The test case template in Fig. 10 (whose name is *NegativeAmountException expected*) is the one pointed by the *when* clause in Fig. 8. When a test case is generated with this template, all the code between <*template*> and <*/template*> is directly copied into the test case code, although the *sequence* tag is substituted by the sequence of calls produced by this regular expression. Note the presence of the *TCNUMBER* token in the head of the test case code: the generation engine will substitute it by the number corresponding to the test case order. Although the text in the test case template used in this example is Java code, it could be any kind of programming language or other structure.

For example, one of the possible sequences proceeding from the *A(B|C|D)*E* regular expression is *ABCE,* which corresponds to *Account·deposit·withdraw·getBalance.* Supposing *amount* takes the values -100 and 50 for *deposit* and *withdraw,* the *NegativeAmountException expected* test case template must be used: for this combination, the tool copies the test template code and substitutes the *sequence* tag by the actual calls, taking into account the possible presence of *before* and *after* tags into the *call* sections (Fig. 11).

```
@Test
// Generated from A(B|C|D)*E, maxLength=5
public void test1() {
  try {
    double obtained=0;      // Proceeds from A's before
    Account account= new Account();    // From A
    account.deposit(-100);             // From B
    obtained+=-100;                    // From B's after
    account.withdraw(50);              // From C
    obtained-=50;                      // From C's after
    double balance=account.getBalance();  // From E
    fail("NegativeAmountException expected");
  }
  catch (NegativeAmountException e) {}
  catch (Exception e) {
    fail("NegativeAmountException expected");
  }
}
```

Fig. 11. A test case

B. Obtaining combinatorial tables

When the test engineer uploads the XML file, the server expands the regular expressions and produces sequences. From each sequence, a combinatorial table (described in XML) is built, which will be processed by the combinatorial algorithm.

The code in Fig. 12 corresponds to the combinatorial table coming from the *ABCE* sequence (i.e., *new Account, deposit, withdraw* and *getBalance*). A combinatorial table holds:

(1) The definition of the parameters corresponding to this sequence. In this example, since *B* and *C* are the only operations in the sequence with parameters, and both of them have just one, two parameters are declared within the *parameters* section: *A,* which is the *deposit's amount,* and *B,* which is the *withdraw's amount*. Each *parameter* tag includes the parameter name (*A*), its actual name (*amount*), the *operation name* (*deposit*) and the *operation alias* (*B*): `<parameter name="A" actualName="amount" operationName="account.deposit(amount);" ...>`,

(2) The test data required by each parameter, inside every *parameter* tag. In this case: *{-100, 0, 50, 1000}* for the *amount* parameter in both *deposit* and *withdraw*.

(3) The *when* clauses, but now particularized for this sequence.

(4) The test templates in the regular expressions file, but now particularized for this sequence: the *<sequence/>* tag is replaced by the code that annotates each operation in the sequence. This code includes the text within the *before* tag, the value of the *message* attribute and the text within the *after* tag. Note also that the parameter values in the code templates are referenced by special tokens *#[A], #[B]*, etc. The combinatorial algorithm will substitute these tokens by the values of the corresponding parameter declared in the *parameters* section: for example, since the *#[A]* token corresponds to the declaration `<parameter name="A" actualName="amount" operationName="account.deposit(amount);" ...>`, which has four *item* tags associated with the values *{-100, 0, 50, 1000},* the combinatorial algorithm will replace *#[A]* by -100, 0, 50 or 100.

```
<table>
  <parameters>
    <parameter name="A" actualName="amount"
      operationName="account.deposit(amount);" operationAlias="B">
        <item value="-100"/>
        <item value="0"/>
        <item value="50"/>
        <item value="1000"/>
    </parameter>
    <parameter name="B" actualName="amount"
      operationName="account.withdraw(amount);"
      operationAlias="C">
        <item value="-100"/>
        <item value="0"/>
        <item value="50"/>
    </parameter>
  </parameters>
  <when type="OR">
    <parameter name="A">
      <item value="-100"/>
      <item value="0"/>
    </parameter>
    <template name="NegativeAmountException expected"/>
  </when>
  <when type="OR">
    <parameter name="B">
      <item value="-100"/>
      <item value="0"/>
    </parameter>
    <template name="NegativeAmountException expected"/>
  </when>
  <whenElse>
    <template name="General case"/>
  </whenElse>
  <template name="NegativeAmountException expected">
@Test
public void testTCNUMBER() {
  try {
    double obtained=0;
    Account account= new Account();

    account.deposit(#[A]);
    obtained+=#[A];

    account.withdraw(#[B]);
    obtained-=#[B];

    double balance=account.getBalance();

    fail("NegativeAmountException expected");
  }
  catch (NegativeAmountException e) {}
  catch (Exception e) {
    fail("NegativeAmountException expected");
  }
}
  </template>
  <template name="NullTargetAccountException expected">
    ...
</table>
```

Fig. 12. An excerpt of a combinatorial table for *new·deposit·withdraw·getBalance*

C. Test case generation

When the combinatorial tables have been obtained, the tester selects one of the several combinatorial algorithms the tool offers (All combinations, several implementations of pairwise, and Each choice).

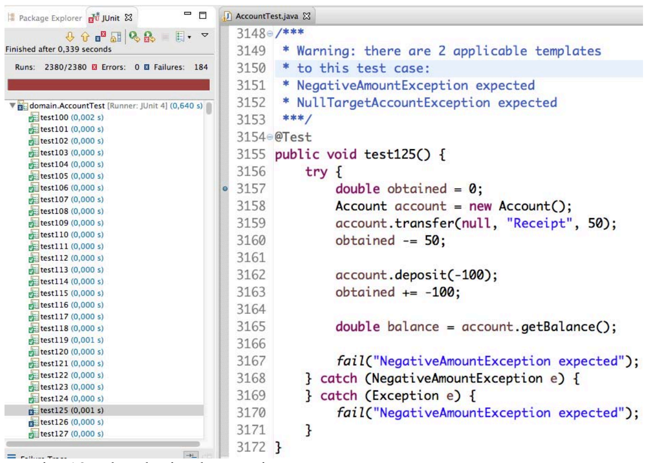
These combinatorial algorithms take a combinatorial table (such as that in Fig. 12) as input and produce a directly usable test case file. Sometimes, the file may need some format or indentation change to make it more legible. But, if the test templates are adequately written, they are completely legible and executable.

If a test case fits with more than one test template, the tool picks up the first template applicable, but adds a warning comment to the code produced. This is the case of the test case shown in Fig. 13, that matches in the first time with the *NegativeAmountException,* but that actually produces a *NullTargetAccountException.*

VI. Reducing and Prioritizing the Test Suite

The expansion of the regular expressions and the subsequent combinatorial explosion may lead to a huge, unmanageable number of test cases, many of which will be moreover redundant with respect to others.

A test case, *t*, is redundant with respect to a set of test cases, *T*, if the set of test-case requirements covered by *t* is a subset of the set of test-case requirements covered by *T-{t}* [5]. In general, these test-case requirements are coverage criteria. For the well known Myer's Triangle-type determination problem [6], for example, two test cases exercising an equilateral triangle will be likely redundant for any coverage criterion.



```
Package Explorer   JUnit
Finished after 0,339 seconds
Runs: 2380/2380   Errors: 0   Failures: 184
domain.AccountTest [Runner: JUnit 4] (0,640 s)
  test100 (0,002 s)
  test101 (0,000 s)
  test102 (0,000 s)
  test103 (0,000 s)
  test104 (0,000 s)
  test105 (0,000 s)
  test106 (0,000 s)
  test107 (0,000 s)
  test108 (0,000 s)
  test109 (0,000 s)
  test110 (0,000 s)
  test111 (0,000 s)
  test112 (0,000 s)
  test113 (0,000 s)
  test114 (0,000 s)
  test115 (0,000 s)
  test116 (0,000 s)
  test117 (0,000 s)
  test118 (0,000 s)
  test119 (0,001 s)
  test120 (0,000 s)
  test121 (0,000 s)
  test122 (0,000 s)
  test123 (0,000 s)
  test124 (0,000 s)
  test125 (0,001 s)
  test126 (0,000 s)
  test127 (0,000 s)
```

```
AccountTest.java
/***
 * Warning: there are 2 applicable templates
 * to this test case:
 * NegativeAmountException expected
 * NullTargetAccountException expected
 ***/
@Test
public void test125() {
    try {
        double obtained = 0;
        Account account = new Account();
        account.transfer(null, "Receipt", 50);
        obtained -= 50;

        account.deposit(-100);
        obtained += -100;

        double balance = account.getBalance();

        fail("NegativeAmountException expected");
    } catch (NegativeAmountException e) {
    } catch (Exception e) {
        fail("NegativeAmountException expected");
    }
}
```

Fig. 13. The obtained test suite

In this context, *test suite reduction* identifies "a reduced test suite that provides the same coverage of the software as the original test suite", whereas *test suite prioritization* identifies "an ordering of the test suite according to some criteria" [5].

The method and the tool presented in this article work in two steps (expansion of the regular expression and application of a combinatorial algorithm), and in both steps the tester can limit and prioritize the test cases generated:

- In the first step (expansion of regular expressions), the tester may select those sequences running "more deeply" the regular expression using coverage criteria for regular expressions (see below epigraph A).
- In the second step (combinatorial algorithms), the tester selects the most appropriate combination technique. If s/he uses all combinations, s/he will probably obtain redundant test cases. The tool provides other algorithms: each choice [7], AETG [8] (a greedy pairwise algorithm with lineal cost), and one exponential-cost pairwise version that, however, guarantees the minimum number of visits to pairs.

A. *Coverage criteria for regular expressions*

Mariani et al. [9] have proposed specific coverage criteria for regular expressions that take into account their constitutive elements:

- *Alphabet coverage*: a test suite *TS* satisfies this criterion if for each symbol *a* in the alphabet, there is at least a test case *tc* in *TS* that contains *a*.
- *Operator coverage:* a test suite *TS* satisfies this criterion if: (1) for each union operator (|) occurring in the regular expression, *TS* includes at least one test case *TCa* that contains the first operand and one test case *TCb* that contains the second operand; (2) for each Kleene operator (*), *TS* includes at least three test cases: *TCa* that corresponds to no iterations of the operand, *TCb* that corresponds to exactly one iteration and *TCc* that corresponds to more than one iteration of the operand.
- *Expression coverage:* a test suite *TS* satisfies this criterion if for each choice of operators that results in a sentence *S* up to consecutive iterations of the Kleene operator, it contains at least one test case *TC* corresponding to *S*.

Mariani et al. do not mention the *concatenation operator*. An additional coverage criterion for it is: a test suite *TS* satisfies the concatenation operator if it contains at least a test case that includes the sequence of the concatenated operands.

Besides the Kleene operator, the union and the concatenation, our tool includes two additional, commonly used operators: the positive closure (+) and the option (?). So, the *Operator coverage* can be extended for these ones:

- For the *positive closure*, a test suite *TS* must include at least a test case that corresponds to one iteration and another test case that corresponds to more than one iteration.
- For the *option operator,* it is required that the test suite includes a test case corresponding to non-using the operand and another test case that uses the operand.

Considering the equivalence between regular expressions and finite automate, the idea of these coverage criteria is that test cases must reach all the final states, using all symbols, but avoiding redundant transition sequences.

1) *Prioritization and reduction of the test suite*

Consider the regular expression *A(B|CD?)+*. According to the afore mentioned coverage criteria:

- With respect to the alphabet, we need test cases containing *A, B, C* and *D.*
- W.r.t. *(CD?),* we need: *CD* and *C.*
- W.r.t. *(B|CD?),* we need: *B, CD* and *C.*
- W.r.t. *(B|CD?)+*, we need: *B, C, CD, BB, BCD, BC, CDB, CDCD, CDC, CB, CCD* and *CC*

If we expand the regular expression with a maximum length of, for example, 7, our expansion engine produces 287 sequences. If we later combine each sequence with test data, the number of test cases will be huge.

So, we provide a method to prioritize the sequence suite and, if the tester considers its convenience, to reduce its size. This method sorts the sequence suite according to three criteria:

1) The *first criterion* calculates the coverage of a node according to its type, in a similar way to how a regular expression is expanded (section IV), but now selecting the minimal set of acceptable sequences that fulfill the coverage criterion corresponding to the node type, independently of the maximum length given to the regular expression expansion. For example, for *(A|B)+* (that requires 1 and 2 iterations) we would select *A, B, AA, AB, BA, BB,* but no AABB.

For the regular expression *A(B|CD?)+,* 12 sequences are required to fulfill the afore mentioned coverage criteria:

AB, ACD, AC, ABC, ABB, ABCD, ACDB, ACDCD, ACDC, ACB, ACCD, ACC

In order to prioritize later this set of sequences, we assign them a *weight* of *1*.

2) With the *second criterion* we extract a subset of the sequences with *weight=1.* The selected sequences are those whose starting symbols completely contain any other of the previously selected sequences. We assign them *weight=2.* For example, since *AB* is the beginning of *ABC, ABC* is weighted to 2. In a next iteration it is seen that *ABC* is the beginning of *ABCD:* thus, *ABCD* is weighted to 2 and the *ABC* weight is returned to 1.

In this way, we have the following two sets for the running example after this step:

Weight2 = {ABB, ABCD, ACDB, ACDCD, ACB, ACCD}

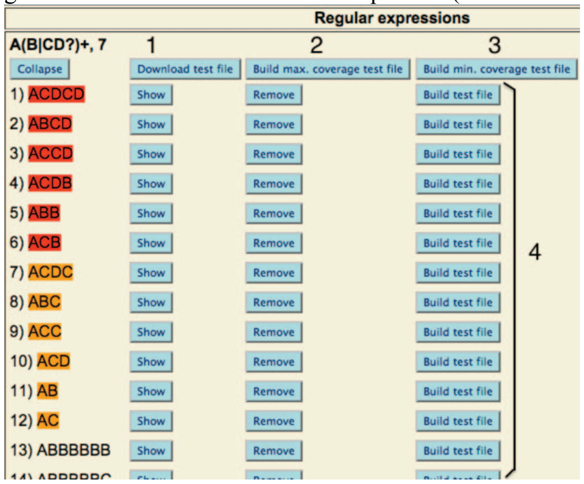
Weight1 = {AB, ACD, AC, ABC, ACDC, ACC}

3) The *last criterion* sorts the sequences in every set by its lengths. This criterion is inspired in the work of Fraser and Gargantini [10], who have analyzed the relationship between the test case length with respect to its fault-detection ability, coverage (according to several criteria) and execution cost. From their experimentation, they conclude that it is preferable to have fewer longer test cases instead of many short test cases, even thought long test cases are less understandable than short ones.

Our tool highlights the required sequences to fulfill the coverage of the regular expression according to these three criteria (Fig. 14):

1) Sequences with *weight=2,* ordered from greater to minor length. The tool highlights them in red.
2) Sequences with *weight=1*, ordered from greater to minor length. They are highlighted in orange.
3) Sequences with *weight=0,* also ordered from greater to minor length. These sequences are not highlighted.

Note the presence of three buttons on the top: the button labeled with "1" generates the whole test file (i.e., containing all the sequences); button "2" generates the test file for sequences with *weight>0.* The third one (button "3") only considers sequences with *weight=2.* There are additional buttons to generate test cases for individual sequences (buttons with "4").


Regular expressions

A(B|CD?)+, 7	1	2	3
Collapse	Download test file	Build max. coverage test file	Build min. coverage test file
1) ACDCD	Show	Remove	Build test file
2) ABCD	Show	Remove	Build test file
3) ACCD	Show	Remove	Build test file
4) ACDB	Show	Remove	Build test file
5) ABB	Show	Remove	Build test file
6) ACB	Show	Remove	Build test file
7) ACDC	Show	Remove	Build test file
8) ABC	Show	Remove	Build test file
9) ACC	Show	Remove	Build test file
10) ACD	Show	Remove	Build test file
11) AB	Show	Remove	Build test file
12) AC	Show	Remove	Build test file
13) ABBBBBB	Show	Remove	Build test file

Fig. 14. The tool highlights the sequences required to completely cover the regular expression

VII. Uses of CTWeb

This section presents two examples of application of the tool.

A. *Testing a simple login functionality*

CTWeb offers a single login functionality in a single html form with two text boxes for writing the user's email and password, and a login button. So, there are three messages for testing this use case.

Two of these messages receive parameters for writing the login and the password. For this example, we have created a valid user *(pepe@pepe.com,* with password "*pepe"*). We will use an inexistent user too: j*ohn.updike@writers.org.*

Since the tool is offered through the web, test cases will be generated for the Selenium testing framework (http://www.seleniumhq.org), in its Java variant. A Selenium test case holds at least one reference to a *WebDriver,* which is an object representing a wrapper to manage the navigator screen. There are different *WebDriver* implementations for several common navigators (Firefox, Chrome, etc.).

Thus, a reference to the *WebDriver* is created and instanced to the required subtype. The *WebDriver* offers a *findElement* method that returns a *WebElement*, an interface for accessing the different widgets in the web page. *findElement* may recover widgets using several criteria: *id, name, xpath, CSS class, tag name,* etc. In turn, *WebElement* has several methods for writing (*sendKeys)*, clicking (*click)*, etc.

The three main widgets in the login form appear in Fig. 15: the two text boxes can be collected by their *name* attribute; the button, through an *xpath* string.

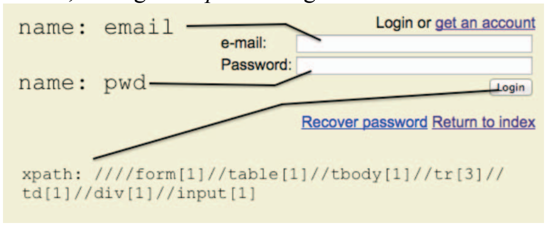

name: email
Login or get an account
e-mail:
Password:
name: pwd
Login
Recover password Return to index
xpath: ////form[1]//table[1]//tbody[1]//tr[3]//td[1]//div[1]//input[1]

Fig. 15. Important elements in the login form

Therefore, the structure of the test cases will consist in a simple sequence of calls to *A, B, C.* Depending on the parameters' combination, test cases will map either to the *Bad login* or the *Right login* templates. The description of the messages and the regular expression appear in Fig. 16: note the presence of a *when* of *AND* type, that drives to the *Right login* test case template only when the LOGIN and PASSWORD parameters respectively are *pepe@pepe.com* and *pepe.* Otherwise, the *Bad login* template is applied thanks to the *whenElse* clause.

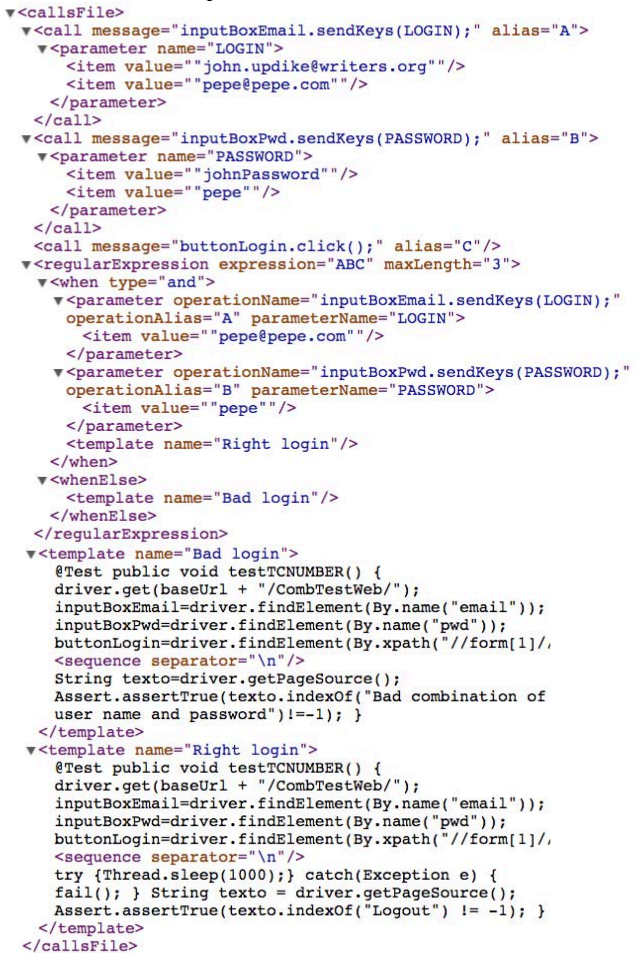
The two test case templates appear in the bottom side of the same figure: both of them instantiate the driver, point it to the required URL, find the three interesting widgets*,* insert the operations sequence and, finally, check the obtained result: if the login and password combination is incorrect, the web page must show the "Bad combination of user name and password" message; otherwise, it will include a link in the page for logging out.

Fig. 17 shows two of the four test cases produced: the first one (*test3)* corresponds to an unauthorized access; the second (*test4)* has applied the *Right login* template since it matches with the *when AND* described in the XML file.

B. *Industrial experience*

We have compared, in several companies, the time expended to the construction of test cases both by hand and by the tool. In general, CTWeb requires about 10% of the time devoted, although we have got savings greater than 95%.

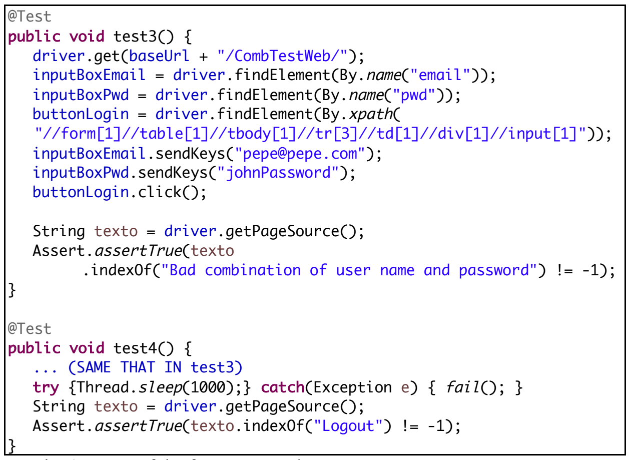
As an example, a software factory required three business days to write 36 manual test cases for 35 requirements (this is, only one requirement was tested by more than one test case). These test cases described the manual steps that testers should execute on the web application of an insurance company. The number of different parameters was close to 40.



```
▼<callsFile>
  ▼<call message="inputBoxEmail.sendKeys(LOGIN);" alias="A">
    ▼<parameter name="LOGIN">
        <item value=""john.updike@writers.org""/>
        <item value=""pepe@pepe.com""/>
      </parameter>
    </call>
  ▼<call message="inputBoxPwd.sendKeys(PASSWORD);" alias="B">
    ▼<parameter name="PASSWORD">
        <item value=""johnPassword""/>
        <item value=""pepe""/>
      </parameter>
    </call>
    <call message="buttonLogin.click();" alias="C"/>
  ▼<regularExpression expression="ABC" maxLength="3">
    ▼<when type="and">
      ▼<parameter operationName="inputBoxEmail.sendKeys(LOGIN);"
        operationAlias="A" parameterName="LOGIN">
          <item value=""pepe@pepe.com""/>
        </parameter>
      ▼<parameter operationName="inputBoxPwd.sendKeys(PASSWORD);"
        operationAlias="B" parameterName="PASSWORD">
          <item value=""pepe""/>
        </parameter>
        <template name="Right login"/>
      </when>
    ▼<whenElse>
        <template name="Bad login"/>
      </whenElse>
    </regularExpression>
  ▼<template name="Bad login">
      @Test public void testTCNUMBER() {
      driver.get(baseUrl + "/CombTestWeb/");
      inputBoxEmail=driver.findElement(By.name("email"));
      inputBoxPwd=driver.findElement(By.name("pwd"));
      buttonLogin=driver.findElement(By.xpath("//form[1]/
      <sequence separator="\n"/>
      String texto=driver.getPageSource();
      Assert.assertTrue(texto.indexOf("Bad combination of
      user name and password")!=-1); }
    </template>
  ▼<template name="Right login">
      @Test public void testTCNUMBER() {
      driver.get(baseUrl + "/CombTestWeb/");
      inputBoxEmail=driver.findElement(By.name("email"));
      inputBoxPwd=driver.findElement(By.name("pwd"));
      buttonLogin=driver.findElement(By.xpath("//form[1]/
      <sequence separator="\n"/>
      try {Thread.sleep(1000);} catch(Exception e) {
      fail(); } String texto = driver.getPageSource();
      Assert.assertTrue(texto.indexOf("Logout") != -1); }
    </template>
  </callsFile>
```

Fig. 16. Calls, regular expressions and test templates for the Login use case

The complete process of designing the requirements with regular expressions required less than 40 minutes. Then, CTWeb generated 9219 "oracled" test cases with *All combinations*, 19 with pairwise and 6 with *Each choice* in less than 2 seconds.



```
@Test
public void test3() {
   driver.get(baseUrl + "/CombTestWeb/");
   inputBoxEmail = driver.findElement(By.name("email"));
   inputBoxPwd = driver.findElement(By.name("pwd"));
   buttonLogin = driver.findElement(By.xpath(
   "//form[1]//table[1]//tbody[1]//tr[3]//td[1]//div[1]//input[1]"));
   inputBoxEmail.sendKeys("pepe@pepe.com");
   inputBoxPwd.sendKeys("johnPassword");
   buttonLogin.click();

   String texto = driver.getPageSource();
   Assert.assertTrue(texto
         .indexOf("Bad combination of user name and password") != -1);
}

@Test
public void test4() {
   ... (SAME THAT IN test3)
   try {Thread.sleep(1000);} catch(Exception e) { fail(); }
   String texto = driver.getPageSource();
   Assert.assertTrue(texto.indexOf("Logout") != -1);
}
```

Fig. 17. Two of the four generated test cases

VIII. RELATED WORK

Automated test case generation is a prolific research line in software engineering: as an example, a simple search in Scopus for publications with the terms “Test case generation” in the title throws 233 conference papers and 78 articles, just from 2011 to 2016 (included). In [11], several authors review test case generation techniques and classify them into five categories: (1) symbolic execution and program structural coverage; (2) model-based test (MBT) case generation; (3) combinatorial testing; (4) adaptive random testing (ART); and (5) search-based software testing (SBST).

Our method probably fits between MBT and Combinatorial testing. In [11], Grieskamp reviews MBT generation techniques and emphasizes research challenges still remain open: dealing with non-determinism in models; the gap between the abstraction level of models and code; repeatability of test cases, overall in distributed and cloud environments; difficulties for tracing from requirements to models; and lack of use of Model-Driven Engineering in industry. In this sense, our method and tool decrease the gap between models and code (test templates can be written in any format: a programming language, XML, JSON, natural language...), what completely decouples the models (described as XML files) from the final execution platform. Moreover, several users in different companies consider the self regular expression as the representation of all the different scenarios in the requirement, what makes easy the tracing that Grieskamp highlights.

In the same article, Cohen is in charge of the combinatorial testing methods revision. According to her, some fruitful future directions of combinatorial testing include “automated model extraction, adapting to model evolution, and developing techniques that re-use or share information between different test runs”. The simplicity of our models helps in preserving the model evolution (changes in the system are translated into changes in the model). Moreover, since the same file may contain more than one regular expression, it is possible to describe scenarios with a subset of the test data of another one: for example, a regular expression *A* may be preceded by other regular expression *B*. *B* may hold specific test data representing one or more scenarios that are prerequisite for executing *A*. This is also a contribution in the sense of reusing and sharing information between different test runs.

A. *The oracle problem*

A very recent survey of Barr et al. [12] reviews almost forty years of research about the oracle problem. According to these authors, the oracle is a “current open problem representing a significant bottleneck that inhibits greater test automation and uptake of automated testing methods and tools more widely”. In fact, as well as the generation of test cases may be a problem relatively solved, the addition of oracles to each test case remains practically unsolved. Bertolino also emphasized the automation of the oracle generation as one of the main challenges of software testing research [13]. For her, an “ideal oracle” should provide “the expected outputs for each given test case”. Such ideal oracle would be an “engine/heuristic that can emit a pass/fail verdict over the observed test outputs”. Baresi and Young [14] also mentioned the concept of “ideal oracle”, which “would satisfy desirable properties of program specifications such as being complete but avoiding over-specification, while also being efficiently checkable". Formal methods or complete system models are currently the most suitable artifact for automatically developing oracles, but they need the "over-specification" mentioned by Baresi and Young. Pérez et al. [15], for example, derive *oracled* test cases from UML sequence models in Software Product Lines. Their approach requires, in fact, a quite precise specification of the test scenarios with their pre and postconditions, messages, parameters, etc. The proposal is economically viable because it is highly reusable in the SPL context within it is framed, but it is too expensive for the development of isolated developments.

But, as Barr et al. pointed out, "for many systems and most testing as currently practiced in industry, the tester does not have the luxury of formal specifications or assertions, or automated partial test oracles. […] The tester faces the daunting task of manually checking the system's behavior for all test cases".

For generating the oracle, our approach applies the same method a tester employs: after having inspected the initial state of the SUT, s/he observes the calls in the test case, checks the parameter values and decides about the corresponding oracle. This is exactly what our method does.

B. *Test case generation in compilers testing*

The expansion of regular expressions may remind some algorithms used for testing compiler parsers. The most relevant is that of Purdom [16], which iterates over the production rules of the grammar for using each production rule at least once, but preserving the length of the language strings produced in a minimum. Several authors have modified the original algorithm ([17][18]), although the idea behind these proposals remains the same. The difference with our approach is quite significant, because we deal with regular expressions (i.e., with finite automate), whilst compilers deal with stack automata.

C. *Protocol testing*

Finite state machines have been also extensively used in protocol testing. In many papers (e.g., [19]), Bochmann and others have described different algorithms for deriving test sequences for finite state machines. But to our best knowledge, expanding regular expressions to get sequences of events or messages, as we do in this article, has been never used in software testing.

IX. Conclusions and future work

This article has presented a method and a tool (with around 600 users registered) to generate test cases from systems' specifications using regular expressions.

The implemented tool accepts XML files containing the complete description of the set of interesting scenarios by means of annotated regular expressions, that are later expanded by a specific engine. Each regular expression produces a number of test sequences, which are combined with test data to get executable test cases. Oracles are also described in the XML file using generic templates and a simple set of tags. Combinatorial algorithms are in charge of combining test templates, test data and generic oracles to produce test cases with the desired structure and the adequate oracle.

We are currently modifying CTWeb to describe the scenarios with annotated finite automata instead of with regular expressions (remind they are equivalent). This change has been suggested by several companies that have used the tool: actually, a model is more visual and understandable than an XML file.

X. Acknowledgments

This work has been developed within the GINSENG and the SEQUOIA projects (respectively TIN2015-70259-C2-1-R and TIN2015-63502-C3-1-R), Fondo Europeo de Desarrollo Regional and Ministerio de Economía y Competitividad.

References

[1] Capgemini, HP, and Sogeti, "World Quality Report 2015-2016," *Sogeti*. [Online]. Available: http://www.sogeti.com/explore/reports/world-quality-report-2015-2016/. [Accessed: 02-Nov-2015].

[2] IEEE Computer Society, "IEEE Standard 829 for Software and System Test Documentation," 2008.

[3] K. Beck and E. Gamma, "Test-infected: programmers love writing tests," in *More Java Gems*, Cambridge University Press, 2000.

[4] J. Glenn Brookshear, *Theory of Computation: Formal Languages, Automata, and Complexity*. Benjamin-Cummings Publishing Co., Inc., 1989.

[5] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Transactions on Software Engineering*, vol. 29, no. 3, pp. 195–209, Mar. 2003.

[6] G. J. Myers, *The Art of Software Testing, Second Edition*, 2nd ed. Wiley, 2004.

[7] P. Ammann and J. Offutt, "Using formal methods to derive test frames in category-partition testing," in *Proceedings of the Ninth Annual Conference on Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security*, 1994, pp. 69–79.

[8] D. Cohen, I. C. Society, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach to Testing Based on Combinatorial Design," *IEEE Transactions on Software Engineering*, vol. 23, pp. 437–444, 1997.

[9] L. Mariani, M. Pezzè, and D. Willmor, "Generation of Integration Tests for Self-Testing Components," in *Applying Formal Methods: Testing, Performance, and M/E-Commerce*, M. Núñez, Z. Maamar, F. L. Pelayo, K. Pousttchi, and F. Rubio, Eds. Springer Berlin Heidelberg, 2004, pp. 337–350.

[10] G. Fraser and A. Gargantini, "Experiments on the test case length in specification based test case generation," in *ICSE Workshop on Automation of Software Test, 2009. AST '09*, 2009, pp. 18–26.

[11] S. Anand *et al.*, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, Aug. 2013.

[12] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 507–525, May 2015.

[13] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," in *Future of Software Engineering, 2007. FOSE '07*, 2007, pp. 85–103.

[14] Luciano Baresi and Michal Young, "Test oracles, Technical Report CIS-TR-01-02." University of Oregon, Dept. of Computer and Information Science, 2001.

[15] B. Pérez Lamancha, M. Polo, D. Caivano, M. Piattini, and G. Visaggio, "Automated generation of test oracles using a model-driven approach," *Information and Software Technology*, vol. 55, no. 2, pp. 301–319, Feb. 2013.

[16] P. Purdom, "A sentence generator for testing parsers," *BIT*, vol. 12, no. 3, pp. 366–375, Sep. 1972.

[17] B. A. Malloy, "An interpretation of Purdom's algorithm for automatic generation of test cases," in *In 1st Annual International Conference on Computer and Information Science*, 2001, pp. 3–5.

[18] F. Bazzichi and I. Spadafora, "An Automatic Generator for Compiler Testing," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 4, pp. 343–353, Jul. 1982.

[19] S. Fujiwara, G. v Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi, "Test selection based on finite state models," *IEEE Transactions on Software Engineering*, vol. 17, no. 6, pp. 591–603, Jun. 1991.