Received: 10 October 2019 | Revised: 10 March 2020 | Accepted: 16 April 2020
DOI: 10.1002/smr.2273

RESEARCH ARTICLE - METHODOLOGY

Journal of Software: Evolution and Process

WILEY

Automated generation of *oracled* test cases with regular expressions and combinatorial techniques

Macario Polo[1] | Oscar Pedreira[2] | Ángeles S. Places[2] | Ignacio García Rodríguez de Guzmán[1]

[1]Institute of Technologies and Information Systems, Universidad de Castilla-La Mancha, Ciudad Real, Spain

[2]Centro de Investigación CITIC, Universidade da Coruña, Campus de Elviña s/n, A Coruña, Spain

Correspondence
Oscar Pedreira, Centro de Investigación CITIC, Universidade da Coruña, Campus de Elviña s/n, A Coruña, Spain.
Email: oscar.pedreira@udc.es

Funding information
Grupo de Referencia Competitiva 2017-Xunta de Galicia/FEDER-UE GRC; Centros singulares de Investigación de Galicia-Xunta de Galicia/FEDER-UE CSI, Grant/Award Numbers: ED431C 2017/58, ED431G/01; Xunta de Galicia grants Xunta de Galicia/FEDER-UE, ConectaPeme, Grant/Award Number: IN852A 2018/14; ETOME-RDF3, Grant/Award Number: TIN2015-69951-R; BIZDEVOPS-Global, Grant/Award Numbers: RTI2018-098309-B-C32, RTI2018-098309-B-C31; TESTIMO project, Grant/Award Number: SBPLY/17/180501/000503; Ministerio de Economía, Industria y Competitividad (MINECO) and Fondo Europeo de Desarrollo Regional (FEDER); BIZDEVOPS-Global project

Abstract

One of the main challenges of software testing research is the automated addition of oracles to the generated test cases: Whereas the automated generation of operation sequences (which is one of the essential components of test cases) is in practice a solved problem, the automated addition of the oracle (another indispensable element) is still an important problem and an open research question. This article proposes an approach to get executable test suites composed by complete test cases (i.e., they include the oracle). The core of the method is based on annotated regular expressions. The test generation process, which is supported by a tool, follows three steps: (1) creation of annotated regular expressions, where each regular expression describes a set of sequences of operations to be executed against the system under test; (2) expansion of the regular expressions to get sequences of operations, which still do not have parameter values; and (3) generation of the executable test cases with oracle. In this third step, each test case is generated with the suitable oracle, depending on the conditions specified in the regular expression.

KEYWORDS
oracles, regular expressions, software testing, test case generation

1 | INTRODUCTION

In functional testing, a typical test case is composed of three parts[1,2]:

1. The specification of the initial situation, which puts the system under test (the *SUT*) in the desired starting state, and which is essential to allow the later reproduction of the test case.
2. A set of calls to those operations of the SUT that the tester is interested in exercising.
3. An oracle, which determines whether the test case has or has not found errors in the SUT.

Research in Software Engineering has produced many techniques for automating the generation of test cases, but they are mostly *incomplete* because they lack the inclusion of the oracle. In fact, a test engineer may propose data values for the test cases using classic techniques such as equivalence partitioning or boundary values[3] and, then, apply an algorithm implementing some combinatorial strategy[1] to get sequences of operations that can be executed against the SUT. The construction of the test cases may also be supported and guided by a strategy that iteratively adds more test cases to the test suite until a certain coverage threshold is reached.[2] However, if these test cases have no oracle, they will be still incomplete.

Each one of these test cases requires one or more instructions to compose the oracle; otherwise, the test case will not able to highlight the presence of errors. The automated inclusion of the oracle has been an open research question for many years: In their well-known technical report, Baresi and Young[4] claimed: "In much of the research literature on software test case generation or test set adequacy, the availability of oracles is either explicitly or tacitly assumed, but applicable oracles are not described." These very same authors emphasized the prohibitive cost of creating oracles by hand given the huge quantity of incomplete test cases that can be got through automation. Some years later, in 2007, Bertolino highlighted the oracle automation problem as one of the most important research challenges in software testing and warned that "The test oracles challenge also overlaps the route toward test automation."[5] Barr et al[6] shared the same opinion 8 years later in 2015: "This current open problem represents a significant bottleneck that inhibits greater test automation and uptake of automated testing methods and tools more widely." Pezzè and Zhang[7] pointed out in their "Automated test cases" survey that the large amount of test cases produced by automated tools provides new motivations for researching in test oracle automation.

There is, therefore, a pressing need to develop techniques to overcome this important problem, which prevents the complete automation of the testing process.

This article proposes a method, supported by a tool, that automates the generation of test cases with oracle using annotated regular expressions, which represent the family of execution scenarios the test engineer wants to test. Annotated regular expressions (we call them *phase-1* test cases) are expanded to get sequences of operations (*phase-2* test cases), which are later combined with actual test data to get the executable, *phase-3* test cases. The annotations introduced in the regular expressions are used to determine which oracle must be added to every test case.

As will be seen in Section 2, other authors have also used regular expressions (proceeding from state machines) to generate test cases, although none of the reviewed works enriches them with oracle.

Therefore, the main contribution of this article is the inclusion of the oracle in the final generated, *phase-3* test cases, which is got through the relatively simple annotations that accompany the regular expression at the *phase-1* level.

The paper is organized as follows: Section 2 reviews some relevant related works. Section 3 describes the test generation process with a running example. Then, Section 4 presents the annotations made at the *phase-1* level and illustrates how to deal with them for generating the final *phase-3* tests. Section 5 describes SMACTesting, the tool that implements the complete process. A real application example is presented in Section 6. Finally, we draw our conclusions.

2 | RELATED WORK

The oracle is the fragment or set of fragments in a test case that determines whether the test case has or has no found errors in the SUT. Given an execution scenario, the oracle emits a pass or fail verdict, usually comparing the actual SUT's output with the expected one.

An important obstacle for automating the oracle addition is precisely that the description of its calculus method can be almost as complex as the algorithm that produces the actual output. So, and in the context of oracle automation, it is important to give the tester some mechanism to describe, at the highest possible level, the general rules that must be followed to add to each test case its corresponding oracle.

In this sense, Baresi and Young[4] mentioned the concept of "ideal oracle," which "would satisfy desirable properties of program specifications but avoiding *over-specification*." Bertolino[5] recovered this very same concept, defining the "ideal oracle" as "an engine/heuristic that can emit a pass/fail verdict over the observed test outputs." The problem, in fact, is *over-specification*, which obligates the test engineer to expend a considerable effort for providing a complete description of the SUT.[8–10]

Barr et al[6] review 694 publications related to the oracle problem from 1978 to 2012 and classify them into four categories: (1) *Human* (a person has to create the oracle manually), (2) *Derived* (which distinguish the correct or the incorrect program behavior from a variety of artifacts, properties of the system under test, other versions, etc.), (3) *Implicit* (related to the detection of obvious faults, such as a program crash or null pointer exceptions), and (4) *Specified* oracles (they require some kind of formal specification of the SUT).

Within the last category, state transitions systems have been a very prolific line of research[8,11–14] that, as Utting and Legeard point out,[15] started in the 1950s with Moore machines. A state machine describes the accepted usage protocol of the SUT: transitions are the accepted operations, and states describe the expected invariants that the SUT must fulfill after being stimulated with the precedent sequence of operations.

In fact, it is relatively easy to traverse the state machine to get a set of operations sequences that exercise the SUT: Offutt et al[12] or Weißleder,[16] for example, describe a set of coverage criteria for state machines (visiting all states, all transitions, input/output transition pairs, all paths, MC/DC, all def-use-paths, etc.) that may guide the process of test case generation. Weißleder, moreover, combines OCL-annotated UML state machines and class diagrams to generate test cases. Furthermore, the tour over the state machine can be artificially modified using mutation, as Hierons and Merayo do.[17] Executing operations in an order different than the expected one may discover unknown errors in the SUT, hidden behind an incomplete specification: Salas et al,[18] for example, already faced this problem in a work about cybersecurity. In the context of user interface testing,[19] Belli also generates "faulty interactions," which are sequences of events which are not considered in the state machine.

Obviously, the generation of test cases from a state machine requires this artifact to be a faithful representation of the SUT. However, system models, such as state machines, are usually incomplete, are quite abstract, or simply do not exist. And, as Jahangirova points out in her PhD thesis,[20] “the precision of automatically generated oracles depends on the information used for the generation.” Killincceker et al,[21] for example, a model with a state machine all the possible interactions a user can make on a Graphical User Interface using a tool. Even for a simple GUI, the state machine becomes very complex, with many states, and many transitions that intersect each other.

Other approaches, such as Tuglular et al,[22] propose a testing technique for GUI, based on models to detect data violations, being also the basis for the development of test oracles. In this approach, the user interface is modeled as an “event sequence graph.” The nodes are annotated using design-by-contract, where every contract consists of a set of boolean conditions. Even though these contracts help to add the oracle to test cases, the main difficulty with this approach is the “over-specification” mentioned at the beginning of this section.

Kirani and Tsai[23] proposed to interpret state machines as finite automata. Thus, if the operations labeling the transitions correspond to the automaton's alphabet, its associated regular expression represents the set of all the possible operation sequences the SUT can accept. Polo et al[24] explored the idea of expanding regular expressions to get JUnit test cases, but they had the main problem we are dealing with in the present article: They did not have oracle. Thus, all the test cases were always generated exactly with the same structure, independently of the expected result.

Obviously, the semantic enclosed in a state machine is much greater than a regular expression: Think about the UML metamodel of state machines,[25] which have states, pseudostates, composite and shadow states, submachines, transitions, guards, etc. The tester can get a very faithful representation of the SUT with a state machine but, depending on the circumstances, the effort expended in its modeling may be too high.

Kilincceker et al[26] utilize regular expressions for validating circuits implemented in HDL (Hardware Description Language). The source code of the HDL program is automatically analyzed, scanning each line to find particular patterns that represent states, which are added to an FSM (finite state machine). Transitions between states are also extracted from the analysis of the HDL program. The FSM is then processed as a regular expression, generating sequences of symbols that correspond to test sequences. Then, they execute the sequences on the SUT with a simulation environment. Because the sequences generated do not have any oracle, the goodness of their approach is measured in terms of the test execution time for getting different coverage criteria on an example SUT, comparing the time with a random sequence generator. In the afore-mentioned work about GUI testing,[21] these same authors apply a similar approach, also using regular expressions.

Of course, the advantage of using regular expressions (in terms of their ability to identify all possible test scenarios when generating test cases) encloses a disadvantage, related to the cost of exploring the entire test space. For this reason, it is essential to identify and apply good coverage criteria when choosing the optimal test sets to be generated.[27]

Belli and Grosspietsch[28] propose to model the systems by using (1) predicate/transition nets and (2) regular expressions. Whilst the former models the generic behavior of the system using Petri nets, the latter models in detail the sequential behavior of components with a lower level of granularity. This “double strategy” of modeling both at high and low levels perfectly fits with our proposal: Regular expressions specify the behavior of the system components.

A very interesting approach was presented by Liu et al,[29] who propose a notation for writing “extended regular expressions” for generating test cases, together with six modeling rules. Using EREs makes unnecessary to use FSMs. Unlike our approach, the effort to define the ERE seems to be higher than the one needed to define the classical ER. Neither our approach implies losing semantics in terms of the representation of the software behavior. Additionally, our proposal presents a supporting software that fully supports the life cycle of regular expressions, from their creation to their expansion for the generation of test cases with real data, including the generation of the oracles (which is still the most complex part of the testing process).

In their TOTEM methodology, Briand and Labiche[30] use regular expressions to “express in an analyzable and compact form” the sequence diagrams, although they do not explain how they deal with them and, thus, it seems an alternative notation for representing scenarios. TOTEM produces test cases at a system level, but it requires that the SUT behavior is completely specified not only with use case, sequence, and class diagrams but also with OCL annotations (for class invariants and operations' pre- and post-conditions) and “a data dictionary that describes each class, method and attribute.” TOTEM is powerful, but it has the aforementioned problem of *over-specification*.[4]

Finally, we would like to mention other testing approaches that rely on formal theories. For example, it is worthy to mention,[31] where the results of the *Côte de Resyste* project are presented. The most interesting outcome of this project (in the context of this paper) is the TorX tool, which automates the generation, execution, and analysis of test cases. TorX is based on the *ioco* test theory.[32] Although the approach presented in this paper is not formalized at the same level, we think that future work could be improved by introducing some of the concerns, such as the *ioco*-test derivation algorithm.

3 | TEST CASE GENERATION WITH REGULAR EXPRESSIONS

Consider, for the sake of illustration, the simple banking account whose structure and behavior are shown in Figure 1: According to its usage protocol, which is described in the UML state machine shown on the right side of the figure, the account admits calls to *deposit*, *withdraw*, and *transfer*

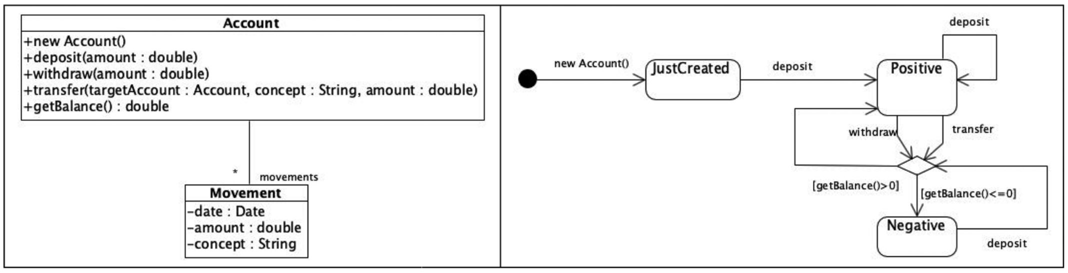


FIGURE 1 Structure and behavior of a simple SUT

after the instance creation. Consider that the tester is, in this example, only interested in testing the behavior of the operations shown in the figure.

The tester could generate test cases for that state machine guiding the process with, for example, the four coverage criteria proposed by Offutt et al[12]:

- With *transitions coverage*, the tester must include tests that cause every transition in the state machine to be taken.
- The *full-predicate* coverage criterion establishes that, for each transition, the tester must include tests triggering the transition reaching MC/DC coverage on the transition's guard condition.
- With *transition-pair*, the tester must include tests in such a way that every possible pair *(input transition, output transition)* is traversed for every state.
- With a *complete sequence*, the tester must include tests that traverse all the “meaningful sequences” of execution on the state machine. According to Offut et al, such sequences are based on the tester's experience, domain knowledge, and other human-based knowledge.

Assuming there are no “meaningful sequences” in the state machine, taking into account that *transition-pair* subsumes *transitions* and that the two transitions with guards (*getBalance* > 0 and *getBalance()* < =0) only have one condition, *transition-pair* would be enough to produce a test suite for this system.

The problem with state machines is that the tester requires a specific tool not only for drawing them and annotating transitions and states (the state annotation is essential to generate the oracle because it holds the SUT expected postcondition after executing each transition) but also for processing it in order to generate the tests. As it was pointed out in Section 2, system models are usually incomplete, and the precision of the generated oracles depends on such precision. In fact, literature evidences that one of the problems of model-based testing is the existing gap between the given SUT specification and its actual behavior.

This approach is inspired by the Kirani and Tsai idea of generating test sequences from the regular expression associated with a state machine, which can be understood as a finite automaton. Thus, it is a form of model-based testing based on state machines that, however, does not need either the complete description of the SUT or of the state machine, but only the regular expressions that represent the test scenarios the tester is interested in exercising. We have decided to make use of regular expressions to specify test cases because we believe they are simpler and easier to write for the test engineer. Belli also appreciates the advantages of applying algebraic methods instead of graphical operations.[19] As we do not need and do not have a state machine modeling the system, we have no algorithms to reach states, transitions or pairs, but we do require an engine to expand the regular expressions and produce test sequences. In terms of regular languages, a test sequence will be a word accepted by the finite automaton associated with the regular expression.

3.1 | Regular expressions as *phase-1 test cases*

In our approach, instead of drawing a state machine, the tester does not need a specific tool to deal with state machines, because these are described as textual regular expressions. From the banking account state machine, we can extract the regular expression of Figure 2 (parameters are omitted for brevity):

$$\boxed{\mathit{new\ Account()}\cdot \mathit{deposit}\cdot(\mathit{deposit}|\mathit{withdraw}|\mathit{transfer})^{*}}$$

FIGURE 2 Regular expression (*phase-1 test case*) proceeding from the state machine in Figure 1

Because a regular expression represents a family of test scenarios and will be progressively converted into the test cases of a test suite, we say that a regular expression is a *phase-1 test case.* Actually, the regular expression requires additional annotations to produce, at the end, executable test cases with oracle. The complete description of a *phase-1* test case is presented in Section 4.

3.2 | Sequences of operations as *phase-2 test cases*

An expansion engine, explained in Section 5, is in charge of expanding the regular expression enclosed in a *phase-1* test case up to a certain length, thus producing a set of sequences, called *phase-2 test cases.* These test cases are not executable because they lack parameter values. A *phase-2* test case represents a sequence of operations that, when combined with actual test data, will likely traverse the system under test in different ways. Some of the authors mentioned in Section 2 call "abstract test cases" or "test sequences" to these specifications.

A length of 4 applied to the *phase-1* test case in Figure 2 produces the *Phase-2* test cases in Figure 3.

3.3 | Executable test cases as *phase-3 test cases*

In order to become executable, the *phase-2 test cases* need parameter values. Given a *phase-2* test case, a combinatorial strategy[1] must be applied to its parameters' values. Suppose the tester selects *{−100, 0, 100, 1000}* as values for the *amount* parameter of *deposit*, *withdraw*, and *transfer*. Depending on the selected combinatorial strategy, the number of executable sequences produced is different: For the 13th sequence of Figure 3, *each choice* would generate only four test cases, AETG[33] 16 test cases and *all combinations* will produce 4 × 4 × 4 = 64 test cases, some of which appear in Figure 4.

The remaining task to convert each executable sequence into an actual case is the addition of the oracle. Thus, for example, it may be expected that all the executable sequences that use −100 or 0 (13.1, 13.2, 13.2, 13.62 in Figure 4) as value for any of the *amount* parameters throw an *NegativeAmountException*, that the executable sequences 13.63 and 13.64 throw an *InsufficientBalanceException*, and that the account balance after 13.48 is 800.

4 | ANNOTATIONS IN *PHASE-1* TO GET *PHASE-3* TEST CASES

Although in Section 3.1, we have assimilated "regular expression" with *phase-1* test case, the truth is that a regular expression needs additional information to become an actual *phase-1* test case. We are behind the "ideal oracle" mentioned by Bertolino's[5] and by Baresi and Young's[4] and, thus, we want to be able to give, at the highest possible abstraction level, the required specifications for generating the "oracled" *phase-3* test cases. In this context, that highest abstraction level is *phase-1* test cases.

As noted in the previous subsection, each *phase-3* test case must be generated with a different oracle, likely according to a different "test template." This template depends (1) on the sequence of operations involved in each test case and (2) on the parameter values assigned to the *phase-3* test case. The left side of Figure 5 shows the test template to be applied to test cases corresponding to negative amounts (executable sequences 13.1, 13.2, 13.2, 13.62 of Figure 4), whilst the right side shows the template to be used when no exceptions are expected (executable sequence 13.48). Obviously, a mechanism is required to decide which test template will be used for writing the code of every *phase-3* test case. This is made using *When* clauses, which will be presented in brief.

Note in both templates the presence of the *#SEQUENCE#* token, which will be substituted by the sequence of operations of this *phase-3* test case, with its parameter values. Note also the *TCNUMBER* token, which will be replaced by an incremental counter.

```
1.  new Account()·deposit(amount)
2.  new Account()·deposit(amount)·deposit(amount)
3.  new Account()·deposit(amount)·deposit(amount)·deposit(amount)
4.  new Account()·deposit(amount)·deposit(amount)·transfer(targetAccount,concept,amount)
5.  new Account()·deposit(amount)·deposit(amount)·withdraw(amount)
6.  new Account()·deposit(amount)·transfer(targetAccount,concept,amount)
7.  new Account()·deposit(amount)·transfer(targetAccount,concept,amount)·deposit(amount)
8.  new
    Account()·deposit(amount)·transfer(targetAccount,concept,amount)·transfer(targetAccount,concept,amount)
9.  new Account()·deposit(amount)·transfer(targetAccount,concept,amount)·withdraw(amount)
10. new Account()·deposit(amount)·withdraw(amount)
11. new Account()·deposit(amount)·withdraw(amount)·deposit(amount)
12. new Account()·deposit(amount)·withdraw(amount)·transfer(targetAccount,concept,amount)
13. new Account()·deposit(amount)·withdraw(amount)·withdraw(amount)
```

FIGURE 3 Some sequences (*phase-2 test cases*) proceeding from the expansion of the regular expression

13.1 new Account()·deposit(-100)·withdraw(-100)·withdraw(-100)
13.2. new Account()·deposit(-100)·withdraw(-100)·withdraw(0)
13.3. new Account()·deposit(-100)·withdraw(-100)·withdraw(100)
…
13.48 new Account()·deposit(1000)·withdraw(100)·withdraw(100)
…
13.62 new Account()·deposit(1000)·withdraw(1000)·withdraw(0)
13.63 new Account()·deposit(1000)·withdraw(1000)·withdraw(100)
13.64 new Account()·deposit(1000)·withdraw(1000)·withdraw(1000)

FIGURE 4 Executable (*phase-3)* test cases

NegativeAmountException **template**

```
public void testTCNUMBER() {
   try {
     #SEQUENCE#
     fail("NegativeAmountException expected");
   }
   catch (NegativeAmountException e) {}
   catch (Exception e) {
    fail("NegativeAmountException expected ");
   }
}
```

Normal **template**

```
public void testTCNUMBER() {
   try {
     #SEQUENCE#
     assertTrue(
      account.getBalance()==expectedBalance);
   }
   catch (Exception e) {
      fail("No exception expected");
   }
}
```

FIGURE 5 Two templates for two different types of test cases

Additionally, the *Normal template* includes a reference to *expectedBalance*, a variable that is used to compare the actual account's balance (returned by *account.getBalance*()) to the expected one. *expectedBalance* has not been defined so far and, therefore, we need to define it at some place.

4.1 | Before, maxLength, and test values annotations

Figure 6 shows some of the annotations added to a regular expression for generating test cases for the account class. It:

1. Includes code in the *Before annotation*, which will be included at the beginning of every test case. Note that this code has: (a) the declaration and construction of *account*, the instance of the class under test; (b) an initial assertion; and (c) the declaration and assignment of an initial value to *expectedBalance*, which is the same variable used in the *Normal template* to check the instance state.
2. Holds the regular expression defined in terms of a set of operation aliases (*D* for *deposit*, for example), that make the writing of the regular expression more comfortable for the tester.
3. Defines the test values for each parameter of each operation.
4. Defines the maximum length of the sequences generated by the expansion of the regular expression.

PHASE-1 TEST CASE

Before annotation

```
Account account=new Account();

assertTrue(account.getBalance()==0);

double expectedBalance=0.0;
```

Regular expression: `D(D|W|T)*`

Usage:

```
D = account.deposit(double amount);
W= account.withdraw(double amount);
T = account.transfer(
      Account targetAccount,
      String concept,
   double amount);
```

Test values:

```
amount = {-100, 0, 100, 1000}
concept = {"Rent", "", null}
targetAccount = { new Account(), this, null}
```

Max length: 6

FIGURE 6 Partial description of an annotated regular expression, a *phase-1 test case*

FIGURE 7 Partial description of a different *phase-1 test case*

PHASE-1 TEST CASE	
Before annotation	**Regular expression**
As in Fig. 6	*As in Fig. 6*

Test values:
`amount = {1000, 5000}   (for deposit)`
`amount = {50, 100}  (for withdraw and transfer)`
 	
`concept = {"Rent" }`
`targetAccount = { new Account() }`
	Max length: 6

The same regular expression can be used for describing several *phase-1* test cases: The only differences of the *phase-1* test case described in Figure 7 with respect to the previous one are in the *Test values* section: Being 6 the maximum length and having those test values, the account instances will only run on "normal" scenarios and should be generated with the *Normal template* of Figure 5.

Executable, *phase-3* test cases proceeding from the *phase-1* test case in Figure 7 must be generated with the *Normal template* shown in Figure 5. However, some of the *phase-3* test cases proceeding from Figure 6 (which had negative values for *amount)* must be generated according to the *Normal template*, others with the *NegativeAmountException* template, and others with other templates, depending on the situation they are describing. For example, the code for the sequence 13.48 of Figure 4 (which corresponded to a normal scenario) appears on the right side of Figure 8.

4.2 | *Precode* and *postcode* annotations

The test case on the right side of Figure 8 is still incomplete because the *expectedBalance* variable is initialized to 0, but its value is not modified during the execution. Thus, at *phase-1* level, operations can be annotated with *Postcode* and maybe with *Precode*, which will be respectively inserted *after* and *before* each operation call (Figure 9).

Now, the test case 13.48 will be generated as in Figure 10.

4.3 | *When* clauses

When clauses are the last annotation required by *phase-1* test cases. A *When* clause holds a condition and points to the template that must be used to generate the *phase-3* test cases that fulfill the condition.

The condition is written as a function on the parameters and their values. Before generating the code of a *phase-3* test case, the parameters are confronted against the condition of each *When* clause: If they match, then the *phase-3* test case is generated according to the template, substituting the *#SEQUENCE#* and *TCNUMBER* tokens by the corresponding values.

There are three types of conditions:

1. *OR*, which are fulfilled when any of the parameter values are included in the values used for defining the condition. For example, "When the amount is 0 or −100, the test case must expect that the SUT throws a NegativeAmountException".

```
public void testTCNUMBER() {
   try {




      #SEQUENCE#



      assertTrue(
       account.getBalance()==expectedBalance);
   }
   catch (Exception e) {
       fail("No exception expected");
   }
}
```

```
public void test13_48() {
   try {
     Account account=new Account();
     assertTrue(account.getBalance()==0);
     double expectedBalance=0.0;

     account.deposit(5000);
     account.withdraw(100);
     account.withdraw(50);

     assertTrue(
      account.getBalance()==expectedBalance);
   }
   catch (Exception e) {
      fail("No exception expected");
   }
}
```

FIGURE 8 The *Normal template* and one of its test cases

PHASE-1 TEST CASE		

Before annotation
`Account account=new Account();`
 	
`assertTrue(account.getBalance()==0);`
 	
`double expectedBalance=0.0;`
		Regular expression: `D(D|W|T)*`
		Usage:
		`D = account.deposit(double amount);`
		`W= account.withdraw(double amount);`
		`T = account.transfer(`
		`Account targetAccount,`
		`String concept,`
		`double amount);`

Precode: --	**Postcode:** 1. `deposit: expectedBalance += #amount#;` 2. `withdraw: expectedBalance -= #amount#;` 3. `transfer: expectedBalance -= #amount#;`	

Test values:
`amount = {-100, 0, 100, 1000}`
`concept = {"Rent", "", null}`
`targetAccount = { new Account(), this, null}`
		Max length: 6

FIGURE 9 Addition of *Precode* and *Postcode* to the *phase-1* test case of Figure 6

2. *AND*, which are fulfilled when all the parameter values are included in those that define the condition. For example, "When the amount of deposit is 1000 and the amount of withdraw is 100, the test case must check that no exception has been thrown and that the account's balance is 900".
3. *ELSE*, which are fulfilled by all those *phase-3* test cases that match neither *OR* nor *AND* conditions.

Figure 11 shows the same *phase-1* test case now enriched with the *Precode* and *Postcode* sections and five *When* clauses. It is worth noting that our experience in the application of SMACTesting to a wide number of different projects with diverse technologies evidences that, in general, *AND* clauses are the preferred ones to generate "positive" test cases, and *OR* clauses are more suitable for "negative" test cases. Those *phase-3* test cases mapped by *ELSE* clauses have test data combinations that the tester has not foreseen either in *AND* or *OR* clauses. SMACTesting gives a default template that throws an exception to inform the tester of this unforeseen test case (Figure 11).

According to the sixth Software Testing Principle of Myers et al,[34] test cases must check that the SUT (1) does what it must do and (2) does not do what it must not do:

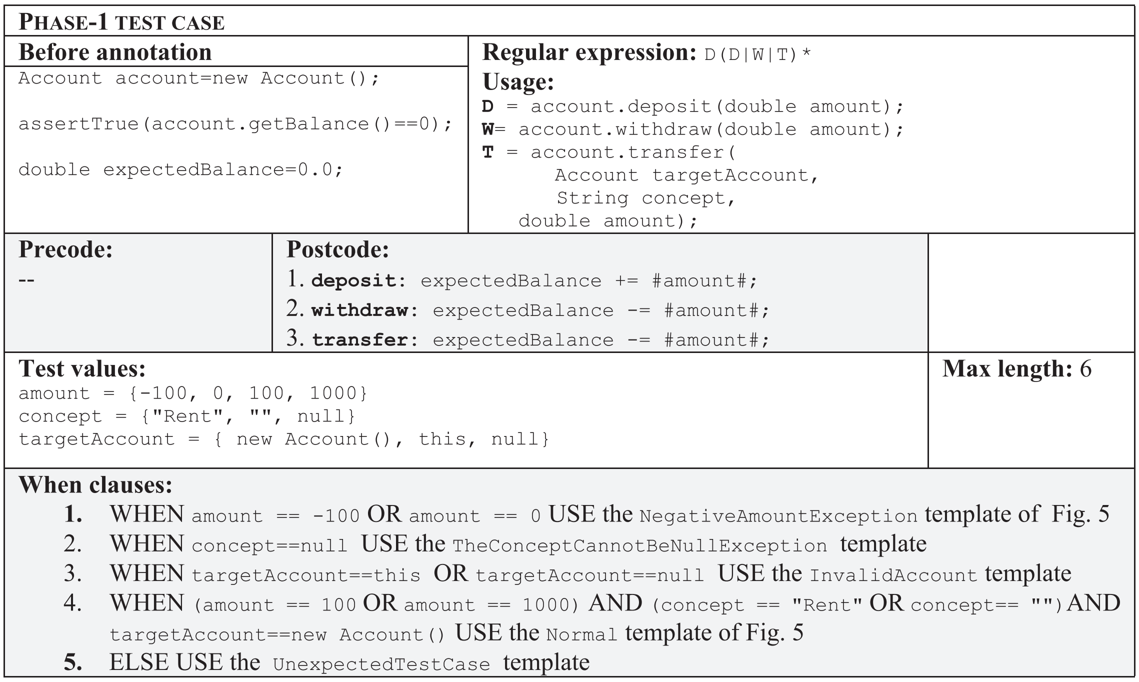
1. The first point ("does what it must do") corresponds to *positive* test cases, which test "normal" scenarios. In a test case that tests a normal scenario, the SUT receives an acceptable sequence of operations and parameters. A positive test case finds an error if the SUT throws an exception and/or reaches an undesired state.
2. The second point ("does not do what it must not do") maps to *negative* test cases. A negative test case tests what in use case design is an exception or error scenario: it is expected that the SUT detects that the received operation sequence and/or the parameter values are not acceptable. Thus, the SUT's right behavior is to throw a certain type of exception and/or to resiliently remain in a coherent, controlled state. A negative test case finds an error if it does not throw the expected exception and/or reaches an incoherent state.

```
public void test13_48() {
   try {
     Account account=new Account();
     assertTrue(account.getBalance()==0);
     double expectedBalance=0.0;

     account.deposit(5000);
     expectedBalance+=5000;
     account.withdraw(100);
     expectedBalance-=100;
     account.withdraw(50);
     expectedBalance-=50;

     assertTrue(
      account.getBalance()==expectedBalance);
   }
   catch (Exception e) {
      fail("No exception expected");
   }
}
```

FIGURE 10 The test case of Figure 8, now with *Postcode*



PHASE-1 TEST CASE

Before annotation

```
Account account=new Account();

assertTrue(account.getBalance()==0);

double expectedBalance=0.0;
```

Regular expression: `D(D|W|T)*`

Usage:

```
D = account.deposit(double amount);
W= account.withdraw(double amount);
T = account.transfer(
      Account targetAccount,
      String concept,
   double amount);
```

Precode:

--

Postcode:

1. **deposit:** `expectedBalance += #amount#;`
2. **withdraw:** `expectedBalance -= #amount#;`
3. **transfer:** `expectedBalance -= #amount#;`

Test values:

```
amount = {-100, 0, 100, 1000}
concept = {"Rent", "", null}
targetAccount = { new Account(), this, null}
```

Max length: 6

When clauses:

1. WHEN `amount == -100` OR `amount == 0` USE the `NegativeAmountException` template of Fig. 5
2. WHEN `concept==null` USE the `TheConceptCannotBeNullException` template
3. WHEN `targetAccount==this` OR `targetAccount==null` USE the `InvalidAccount` template
4. WHEN `(amount == 100` OR `amount == 1000)` AND `(concept == "Rent"` OR `concept== "")` AND `targetAccount==new Account()` USE the `Normal` template of Fig. 5
5. ELSE USE the `UnexpectedTestCase` template

FIGURE 11 Complete description of the *phase-1* test case started in Figure 6 and continued in Figure 9

Table 1 summarizes the conditions under which a positive or negative test case must emit a pass or fail verdict. This table is the basis for creating the *When* clauses.

4.3.1 | Limitations of *When* clauses

An important constraint of *When* clauses is that they can be only defined on parameter values. For example, the sequence of operations shown in Figure 12 makes two withdrawals of 1,000 from an account that only has 100. According to the *When* clauses, this test case corresponds to a *Normal* scenario, because all the values of *amount* are positive. However, and according to the state machine from which the regular expression proceeds, it is expected that the SUT throws some kind of *InsufficientBalanceException*.

To successfully solve this kind of situations, the tester must play with the *phase-1* definitions, considering that she/he:

1. Can distinguish the *amount* parameters of the three operations, naming the *amountDeposit*, *amountWithdraw* and *amountTransfer* instead of simply *amount*.
2. Can use a same regular expression with different test values.
3. Can use a same regular expression with different maximum expansion lengths.
4. Finally, she/he can adjust the condition in the *When* clauses and remove or manually review those test cases matching *Else* clauses.

5 | TOOL SUPPORT

Figure 13 overviews the processing implemented in SMACTesting, a tool that completely automates the process described in the previous sections:

TABLE 1 Verdicts versus positive and negative test cases

	Pass	Fail Verdict
Positive	The SUT does not throw any exception AND reaches the expected coherent state	The SUT throws an unexpected exception OR reaches an undesired state
Negative	The SUT throws the expected exception AND reaches the expected, coherent state	The SUT does not throw any exception OR throws an exception different than the expected one OR reaches an incoherent state

```
Account account=new Account();
assertTrue(account.getBalance()==0);
double expectedBalance=0.0;

account.deposit(100);
expectedBalance += 100;

account.withdraw(1000);
expectedBalance -= 1000;

account.withdraw(1000);
expectedBalance -= 1000;
```

FIGURE 12 A sequence of operations

1. The tester describes, as *phase-1* test cases, the behavior of the system she/he is interested in testing.
2. Then, the regular expressions contained in the *phase-1* test cases are expanded by an *expansion engine* (Section 5.2), that produces *phase-2* test cases.
3. Once the *phase-2* test cases have been produced, a *combinatorial engine* applies the combinatorial strategy selected by the tester to combine each *phase-2* test case with its corresponding parameter values, so producing the final, executable, *phase-3* test cases.

If the SUT is written in Java, SMACTesting analyzes its compiled code using the *reflection* API and automatically assigns a letter (or sequence of letters) to the operations found.

5.1 | Metamodel for test cases

The representation of the different test cases is depicted in Figure 14: As it is seen, a *phase-1* test case produces, via its *generateTestCases()* operation, a collection of *phase-2* test cases. This generation requires the previous execution of *expand* on the *regexp* associated with this *phase-1* test. Details on *expand* are presented in Section 5.2.

Each *phase-2* instance holds a reference to the *phase-1* test case it proceeds from. In this way, when the *generateTestCases()* is executed on a *Phase2TestCase* instance, it looks for which *when* clauses of its corresponding *phase1TestCase* are fulfilled by the current *phase-3* test that the combinatorial engine is generating. The conditions of *When* clauses are represented by the self-type of the instance (*WhenOr* or *WhenAnd*) and the *values* it points to. There is no object representation of *phase-3* test cases because they are a mere transposition to the corresponding *TestTemplate*.

5.2 | Expanding regular expressions

The source code of the regular expressions expansion engine is available at Polo[35] and has been presented in Usaola et al.[36] It deals with regular expressions having the operators shown on the left-hand side of Figure 15. As shown on the right-hand side, there is a supporting class model composed of an abstract *RegularExpression* class that has one specialization per operator, excepting for the *Concatenation*, which is represented in the figure by the *next* association field.

Each specialization helps the RegularExpression's *expand* method to expand the regular expression on which the operation is executed, summarizing the following:

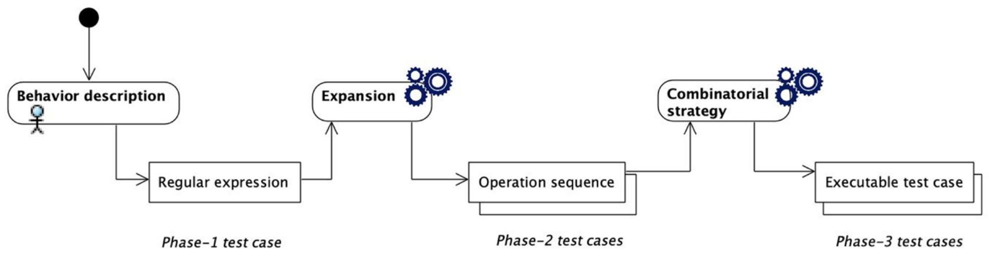


FIGURE 13 The *oracled* test cases generation process

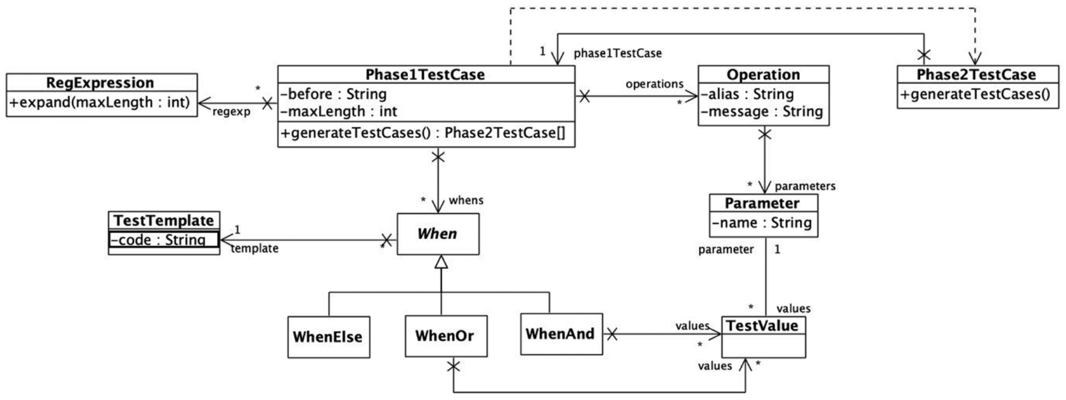


FIGURE 14 Metamodel used for representing *phase-1* and *phase-2* test cases

1. *RESimple*, corresponding to a symbol of the alphabet, produces the symbol itself, independently of the *maxLength* parameter: *expand (symbol, L) = {symbol}*
2. *REBrackets* represents a regular expression enclosed within brackets, such as (a) or (a|b). Note that this subtype holds a *re* attribute, which is the actual regular expression within the brackets. Expanding a *REBracket* is like expanding the *re* attribute: *expand((r), L) = expand(r, L)*
3. *REQuestion* is an optional regular expression that, when expanded, produces the empty regular expression (λ) and the expansion of the optional regular expression (which is represented by the *re* field in the class diagram): *expand(r?, L) = {λ} ∪ expand(r, L)*
4. *REOr* is used to represent the union of regular expressions and, thus, it has two regular expressions: *reLeft* and *reRight* (obviously, the union of 3 or more regular expressions is a *ReOr* regular expression whose *reRightField* is also a *ReOr* regular expression). Expanding a *REOr* instance consists in expanding both *reLeft* and *reRight* and returning all these expansions: *expand(r|s, L) = expand(r, L) ∪ expand(s, L)*

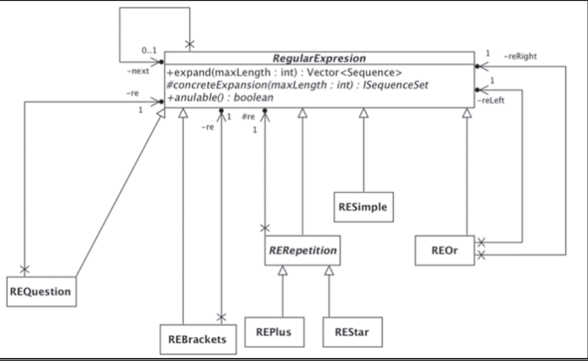
Union: $p\|q$ **Concatenation**: pq **Kleen closure**: p^* **Positive closure**: $p+$ (at least one occurrence of p) **Option operator**:	 RegularExpresion +expand(maxLength : int) : Vector<Sequence> #concreteExpansion(maxLength : int) : ISequenceSet +anulable() : boolean 0..1 -next -re #re -reRight -reLeft RESimple RERepetition REOr REQuestion REBrackets REPlus REStar
$p?$ (zero or one occurrences of p)	

FIGURE 15 Supported operators and the corresponding class model

```
function expand(r+, L) : Set(String)
   expansions : Set(String) = expand(r, L)

   result : Set(String) = expansions

   for i=1 to L
      result = result ∪ (result X expansions)
   next

   remove elements from result with length>L

   return result
}
```

FIGURE 16 Expansion function for the positive closure (+)

5. *REStar* is a regular expression receiving the Kleen closure operator (*). It returns the empty regular expression and the result of applying the "+" operator (*REPlus*, see below) to the regular expression: $expand(r^*) = \{\lambda\} \cup expand(r+, L)$
6. *REPlus* represents a regular expression with the positive closure (+). Expanding it consists in expanding the closured regular expression up to the maximum length. Actually, this is the only operator that increments the length of the expression. So, this version of *expand* is slightly more difficult. A simplified pseudocode appears on Figure 16 (note that the parameter received is a regular expression *r* with the + operator applied):
   a Initially, the closured expression (*r*, not *r+*) is expanded by calling the suitable *expand* (line 1) version according to the type of *r*. This expansion is saved in *expansions* (also in line 1) and a copy in *result* (line 2). In the loop of lines 3–5, *result* will progressively accumulate the expansions. It is also the variable with the return value.
   b Then, the algorithm iterates so many times as the desired length (*L*). In each iteration (line 4), *result* accumulates the cartesian product of the current value of *result* with the original value of *expansions*.
   c Finally, line 6 removes those sequences longer than *L* (line 6) and returns the *result*.

7. Concatenation: As said before, concatenation is represented by the *next* field hold by the *RegularExpression* superclass: after executing the concrete version of *expand* corresponding to the actual regular expression which is being processed, if *next* is not *null*, then the expansion of *next* is concatenated to the results previously obtained.

We want to expand the expression *(ab|c)** up to length 6:

$$\mathrm{expand}((\mathrm{ab}|\mathrm{c})*, 6).$$

Because the regular expression is an instance of *REStar*, the function described in the fifth place is applied:

$$\mathrm{expand}((\mathrm{ab}|\mathrm{c})*, 6) = \{\lambda\} \cup \mathrm{expand}((\mathrm{ab}|\mathrm{c})+, 6).$$

Consider now *(ab|c)+*, which is an instance of *REPlus*. The algorithm of Figure 16 proceeds as follows:

1. First, it expands *(ab|c)*, which is an instance of *REOr*, composed in turn by an instance of a *RESimple* (*a*) whose *next* field is another *RESimple* (*b*), and another instance of *RESimple* (*c*): expand((ab| c), 6) = expand(ab, 6) ∪ expand(c,6) = {ab} ∪ {c} = {ab, c}
2. The set obtained is saved in *expansions*, and a copy in *result*.
3. Then, because *L* = *6*, the loop makes six iterations. In the first one, *result* accumulates the cartesian product of *result* by *expansions*. Note that the cartesian product concatenates the elements of both sets: result = result ∪ (result × expansions) = {ab, c} ∪ ({ab, c} × {ab, c}) = {ab, c, abab, abc, cab, cc}
4. In the second iteration, *result* against accumulates the cartesian product of itself by the original *expansions*:result = {ab, c, abab, abc, cab, cc} ∪ ({ab, c, abab, abc, cab, cc} × {ab, c}) = {ab, c, abab, abc, cab, cc, ababab, ababc, abcab, abcc, cabab, cabc, ccab, ccc}
5. The function continues in this very same way. At the end, *result* will hold a number of sequences. The statement in line 6 cleans *result*, removing those sequences longer than 6.

5.3 | Combining test data values in *phase-2* test cases

SMACTesting provides three combinatorial algorithms[1] to combine test data values with *phase-2* test cases to get the final result:

TABLE 2 Partial description of a *phase-1* test case for the call data records system (I)

Phase-1 Test Case	
Before annotation	**Regular expression:**`AB (CDE) + F`

`double expectedTotal = 0.0;`	**Usage:**
	A = `Call call = new Call();`
	B = `call.setUserId(callUserId);`
	C = `call.setDate(date);`
	D = `call.setCost(cost);`
	E = `registerCall(call);`
	F = `double total = getBill(userId, start, end)`

- *Each choice* generates a test suite where each test value of each parameter is used at least once. It produces as many test cases as the number of test values we have in the parameter with the highest number of test values.
- *AETG*, an algorithm by Cohen et al[33] that gets a test suite where the test cases cover all values pairs of any two parameters.
- *All combinations*, that computes the cartesian product of all parameters. For a given set of parameters and values, this algorithm generates all the possible test cases, so being the technique with more ability to find errors in the SUT. However, many of the test cases generated are redundant from any coverage criterion point of view (i.e., they visit exactly the same areas of the program).

5.4 | Algorithmic complexity of the process

Computationally speaking, the test generation process may find two bottlenecks: the first one, in the computational cost of the *expand* function in the *REPlus* subclass of Figure 15; the second, if the combinatorial strategy for converting *phase-2* into *phase-3* is *all combinations*.

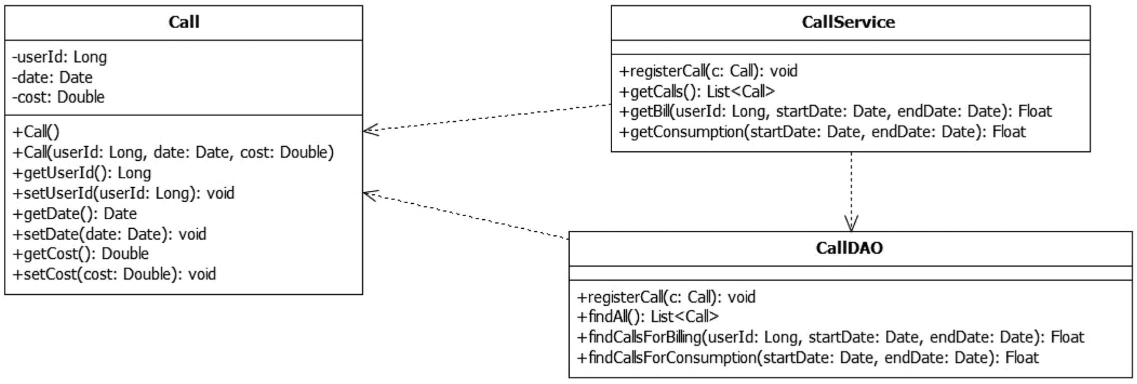
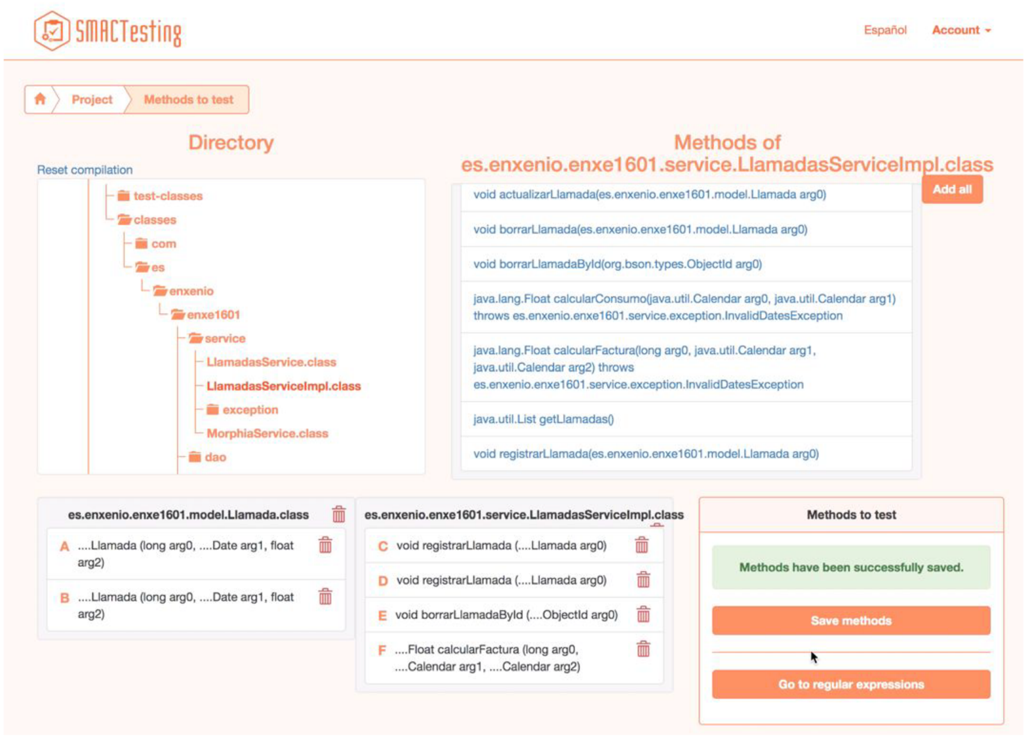

Call
-userId: Long
-date: Date
-cost: Double
+Call()
+Call(userId: Long, date: Date, cost: Double)
+getUserId(): Long
+setUserId(userId: Long): void
+getDate(): Date
+setDate(date: Date): void
+getCost(): Double
+setCost(cost: Double): void
CallService
+registerCall(c: Call): void
+getCalls(): List<Call>
+getBill(userId: Long, startDate: Date, endDate: Date): Float
+getConsumption(startDate: Date, endDate: Date): Float
CallDAO
+registerCall(c: Call): void
+findAll(): List<Call>
+findCallsForBilling(userId: Long, startDate: Date, endDate: Date): Float
+findCallsForConsumption(startDate: Date, endDate: Date): Float

FIGURE 17 Service and data access classes in the sample application



Directory

Reset compilation

- test-classes
- classes
  - com
  - es
    - enxenio
      - enxe1601
        - service
          - LlamadasService.class
          - LlamadasServiceImpl.class
          - exception
          - MorphiaService.class
        - dao

Methods of es.enxenio.enxe1601.service.LlamadasServiceImpl.class

Add all

- void actualizarLlamada(es.enxenio.enxe1601.model.Llamada arg0)
- void borrarLlamada(es.enxenio.enxe1601.model.Llamada arg0)
- void borrarLlamadaById(org.bson.types.ObjectId arg0)
- java.lang.Float calcularConsumo(java.util.Calendar arg0, java.util.Calendar arg1) throws es.enxenio.enxe1601.service.exception.InvalidDatesException
- java.lang.Float calcularFactura(long arg0, java.util.Calendar arg1, java.util.Calendar arg2) throws es.enxenio.enxe1601.service.exception.InvalidDatesException
- java.util.List getLlamadas()
- void registrarLlamada(es.enxenio.enxe1601.model.Llamada arg0)

es.enxenio.enxe1601.model.Llamada.class

A ....Llamada (long arg0, ...Date arg1, float arg2)
B ....Llamada (long arg0, ...Date arg1, float arg2)

es.enxenio.enxe1601.service.LlamadasServiceImpl.class

C void registrarLlamada (....Llamada arg0)
D void registrarLlamada (....Llamada arg0)
E void borrarLlamadaById (....ObjectId arg0)
F ....Float calcularFactura (long arg0, ...Calendar arg1, ...Calendar arg2)

Methods to test

Methods have been successfully saved.

Save methods

Go to regular expressions

FIGURE 18 Selection of methods under test in SMACTesting

- For calculating *all combinations*, SMACTesting computes, for each *phase-2* test case, the cartesian product of all the parameters involved in the sequence of operations. If $\{p_1, p_2, \ldots p_n\}$ are the parameters involved in the *phase-2* test case and $\{v_1, v_2, \ldots v_n\}$ are the number of values of the parameters, the algorithm starts calculating $p_1 \times p_2$, which has a cost of $v_1 \cdot v_2$. In the next iteration, the result (a set composed of $v_1 \cdot v_2$ pairs) is multiplied by v_3, which requires $(v_1 \cdot v_2) \cdot v_3$ multiplications. In the end, the required number of multiplications, and the computational cost of this strategy, is $\sum_{i=1}^{|p|} \prod_{j=1}^{i} v_j$, being $|p|$ the cardinal of the parameters set.
- Regarding the *expand* function of the positive closure, the loop of lines 3–5 of Figure 16 also calculates L times the cartesian product of the expansions of the closured regular expression. Depending on the type of regular expression, the cost of the *expand* function may become exponential.

In the common practice, the design of the *phase-1* test cases (overall the maximum length of the regular expression, the number of closure operators and the number of parameter values) does not jeopardize the test generation time. Anyway, the maximum time of a test generation can be parametrized, being stopped when it is reached.

6 | APPLICATION EXAMPLE

This section illustrates the test case generation method applied to one real project. First, we briefly describe the SMACTesting platform, a tool that supports the test generation process described in the previous sections. After presenting the system under test we used in this application example, we describe how to specify *phase-1* test cases for this application. Finally, we show how we can define a small set of oracles that covers a larger set of test cases, thus making the generation of complete test cases much simpler.

6.1 | The SMACTesting platform

To apply the process described in the previous section in an application example, we implemented a test generation platform called SMACTesting. The tool supports all the steps of the test generation process: loading the SUT code using the reflection engine, selecting the methods to be used in the tests, annotating them, defining test values for the method's parameters, defining the regular expressions, expanding them, defining the oracles using the when clauses, and, finally, generating and running the test cases.

The platform is called SMACTesting because it also provides modules for automating testing of SMACT (Social, Mobile, Analytics, Cloud, and IoT) applications. However, those modules and functionalities exceed the scope of this article.

6.2 | System under test

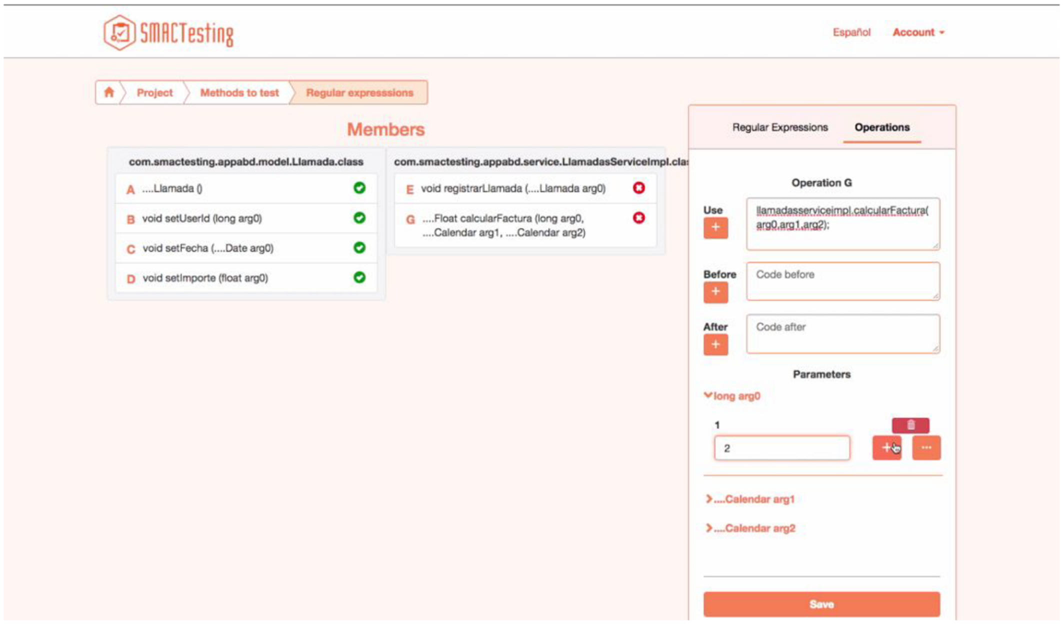
The application is based on a real scenario, but it has been intentionally simplified because its only purpose is to serve as a test object for the application example presented in this section. The application allows storing telephone *call data records* (CDR). For our testing purpose, we will only consider the *date* when the phone call took place, the *id* of the user that made the call, and the *cost* to be billed to the user. The application allows the user to insert new CDRs in the database and to compute the amount to bill to a given user on a given period. The application is developed as a web application with a model-view-controller architecture using Java, Spring, and JSP, and uses MongoDB to store the data.

The next figure shows the three classes of the model layer of the application that are relevant for the example. The *Call* class implements the Value-Object pattern,[37] and it allows to represent a CDR with its three attributes: *user id*, *date*, and *cost* of the call. The class *CallDAO* implements the data access object pattern for the entity *Call*. It implements four methods, that allows to insert a new CDR in the database (*registerCall*), to obtain all the CDRs in the database (*findAll*), to find the CDRs corresponding to a given user in a period (*findCallsForBilling*), and to find all the CDRs of any user in a period (*findCallsForConsumption*). The class *CallService* implements the model of the application and provides methods for getting all the calls, computing the bill for a given user on a period, and to compute the global consumption of all users on a period.

6.3 | Specification of *phase-1* test cases

A reasonable test case for this application would imply inserting some calls into the database, and then computing the amount to bill to a given user on a certain period. Table 2 gives a partial description of a *phase-1* test case.

The selected regular expression starts with the creation of an instance of *Call* (operation *A*) made by a certain user (operation *B*). Then, it assigns it the *date* and *cost* (operations *C* and *D*) and registers it in the database (*E*). *CDE* can be executed several times. Finally, the *F* operation



com.smactesting.appabd.model.Llamada.class

- A ....Llamada ()
- B void setUserId (long arg0)
- C void setFecha (....Date arg0)
- D void setImporte (float arg0)

com.smactesting.appabd.service.LlamadasServiceImpl.cla

- E void registrarLlamada (....Llamada arg0)
- G ....Float calcularFactura (long arg0, ...Calendar arg1, ...Calendar arg2)

Use: llamadasserviceimpl.calcularFactura(arg0,arg1,arg2);

Before: Code before

After: Code after

long arg0

1: 2

....Calendar arg1

....Calendar arg2

FIGURE 19 Interface of the test generation tool—selection of methods and specification of test values

represents a call to the *getBill* method: note, in this case, that its usage assigns the result to a variable called *total*. Note also the declared *expectedTotal* variable in the *Before annotation* section.

Figure 18 shows the screen of SMACTesting where the test engineer selects the operations to be included in the test cases. At this point, the reflective engine of the tool has analyzed the project structure and shows a tree with all the operations contained in all its.*class* files. In this example, the right-hand side shows the methods contained in the class selected in the tree (*CallServiceImpl*, that corresponds to *CallService* in

TABLE 3 Partial description of a Phase-1 test case for the call data records system (II)

Phase-1 Test Case	

Before annotation
`double expectedTotal = 0.0;`
	Regular expression: `AB (CDE) + F`
	Usage:
	`A = Call call = new Call ();`
	`B = call.setUserId (callUserId);`
	`C = call.setDate (date);`
	`D = call.setCost (cost);`
	`E = registerCall (call);`
	`F = double total = call.getBill (userId, start, end)`

Test values:
`callUserId = {1}`
`date = {"2017/01/31", "2018/01/1", "2018/01/15", "2018/01/17", "2018/01/31", "2018/02/01"}`
`cost = {1}`
`userId = {1, 2} (for getBill)`
`start = {"2018/01/01", "2018/01/15"}`
`end = {"2018/01/15", "2018/01/31"}`
	Max length: 12

```
Call call=new Call();
call.setUserId(1);
call.setDate(2017/01/31);
call.setCost(1);
call.registerCall(call);
call.setDate(2017/01/31);
call.setCost(1);
call.registerCall(call);
double total=getBill(1, 2018/01/01, 2018/01/15);
```

FIGURE 20 Actual sequence of calls corresponding to the first data combination of Table 5

TABLE 4 Number of test cases produced by each combinatorial technique

	Each Choice	AETG	All Combinations
ABCDEF	6	13	48
ABCDECDEF	6	36	2,304
ABCDECDECDEF	6	39	110,592
Total	18	88	112,944

Figure 17). The operations the user has already selected appear on the bottom side. Note that the tool assigns a consecutive and different letter to each method.

In Figure 19, the user may add *precode* and *postcode* to each operation, as well as to assign values to the parameters: In the figure, the test engineer is assigning the values 1 and 2 to the first argument of the *getBill* operation (*getBill* in Figure 17).

Table 3 now includes the test values used for the parameters of the operations involved in the test scenario. Note that

- *setUser* takes only the *id* of the user that performs a telephone call. The test engineer assigns 1 to this value, which means that this will be the only user that, for his/her testing goal, will make calls.
- *setDate* assigns a *date* to a telephone call. The test engineer has assigned six possible dates.
- *getBill* takes three parameters: for *userId*, the test engineer has assigned 1 (that corresponds to the calling user) and 2 (a user that has no made calls); for *start* and *end*, that respectively represent the start and end billing periods, the user has assigned two values, which are closely related to the *setDate's date* parameter.

At *phase-1* level, the regular expression **AB (CDE) + F**, together to the test values in Table 3, represents all the scenarios where the user with *id = 1* performs a set of calls in the specified *dates* and with the specified *cost*. Every *call* is inserted in the database. Finally, the *total* due is calculated with the *getBill* method, in several periods (given by *start* and *end*) and for both users *1* and *2*.

Setting 12 as the maximum length for the expansion of the regular expression produces three operation sequences that respectively corre-

TABLE 5 Data combinations generated by *Each choice*

Sequence: ABCDEF	
1/6	1, 2017/01/31, 1, 1, 2018/01/01, 2018/01/15
2/6	1, 2018/01/01, 1, 2, 2018/01/15, 2018/01/31
3/6	1, 2018/01/15, 1, 1, 2018/01/01, 2018/01/15
4/6	1, 2018/01/17, 1, 2, 2018/01/15, 2018/01/31
5/6	1, 2018/01/31, 1, 1, 2018/01/15, 2018/01/15
6/6	1, 2018/02/01, 1, 1, 2018/01/01, 2018/01/31
Sequence: ABCDECDEF	
1/6	1, 2017/01/31, 1, 2017/01/31, 1, 1, 2018/01/01, 2018/01/15
2/6	1, 2018/01/01, 1, 2018/01/01, 1, 2, 2018/01/15, 2018/01/31
3/6	1, 2018/01/15, 1, 2018/01/15, 1, 2, 2018/01/15, 2018/01/31
4/6	1, 2018/01/17, 1, 2018/01/17, 1, 2, 2018/01/15, 2018/01/31
5/6	1, 2018/01/31, 1, 2018/01/31, 1, 2, 2018/01/15, 2018/01/15
6/6	1, 2018/02/01, 1, 2018/02/01, 1, 2, 2018/01/15, 2018/01/31
Sequence: ABCDECDECDEF	
1/6	1, 2017/01/31, 1, 2017/01/31, 1, 2017/01/31, 1, 1, 2018/01/01, 2018/01/15
2/6	1, 2018/01/01, 1, 2018/01/01, 1, 2018/01/01, 1, 2, 2018/01/15, 2018/01/31
3/6	1, 2018/01/15, 1, 2018/01/15, 1, 2018/01/15, 1, 2, 2018/01/01, 2018/01/15
4/6	1, 2018/01/17, 1, 2018/01/17, 1, 2018/01/17, 1, 2, 2018/01/01, 2018/01/31
5/6	1, 2018/01/31, 1, 2018/01/31, 1, 2018/01/31, 1, 2, 2018/01/15, 2018/01/15
6/6	1, 2018/02/01, 1, 2018/02/01, 1, 2018/02/01, 1, 1, 2018/01/01, 2018/01/31

TABLE 6 Description of a phase-1 test case for the call data records system

Phase-1 Test Case	

Before annotation
double expectedTotal = 0.0;
	Regular expression:`A (BCDE) + F`
	Usage:
	A = call call = new call();
	B = call.setUserId (callUserId);
	C = call.setDate (date);
	D = call.setCost (cost);
	E = registerCall (call);
	F = double total = getBill (userId, start, end)

Precode: --	**Postcode:** 1. **`setCost:`** `expectedTotal += (#date# > = #start# && #date# < = #end#? #cost#: 0);`

Test values:
`callUserId = {1}`
`date = {"2017/01/31", "2018/01/1", "2018/01/15", "2018/01/17", "2018/01/31", "2018/02/01"}`
`cost = {1}`
`userId = {1, 2} (for getBill)`
`start = {"2018/01/01", "2018/01/15"}`
`end = {"2018/01/15", "2018/01/31"}`
	Max length: 12

When clauses: **1.** WHEN `userId == 2` USE the `ZeroExpected` template **2.** ELSE USE the `NormalTemplate` template	

spond to 1, 2 and 3 telephone calls, plus the calculus of the bill which is made in the *F* method: **ABCDEF**, **ABCDECDEF,** and **ABCDECDECDEF**. These sequences could correspond to *phase-2* test cases.

For the moment there are no oracles defined. As we have said in Section 5.3, SMACTesting implements three combinatorial strategies to get *phase-3* test cases from *phase-2* ones (*Each choice*, *All combinations* and AETG). Table 4 shows the number of test cases produced by each combinatorial strategy with the test values of Table 3.

For example, the 18 test value combinations that *Each choice* produces are those in Table 4.

The sequence of operations corresponding to the first data combination of the previous table appears in Figure 20 (for the sake of clarity we assume that the *Date* values can be processed as they are written, with no quotation marks nor the construction of a *Date* or *Calendar* object). Note that, even for the 18 test cases of *Each choice*, calculating the oracle by hand may be costly and is fault-prone.

6.4 | Oracle definition

The oracle definition is made at *phase-1* level. The special situations for this system are the following ones:

1. If the user for whom the bill is being calculated (*userId)* is *2*, then the *total* variable (which saves the result of *getBill)* must be *0*, no matter the billing period.
2. If *userId = 1*, the *total* variable must accumulate those calls that took place between the *start* and the end *dates*.

All these situations are described in the highlighted cells of the *phase-1* test case given in Table 6:

1. On the *Postcode* section we establish that, after executing *setCost*, the *expectedTotal* variable must be incremented only when the *date* value of *setDate* is within the limits of the billing period (variables *start* and *end).*

TABLE 7 Test templates

ZeroExpected	NormalTemplate

`public void testTCNUMBER() {`
`#SEQUENCE#`
`assertTrue (total == 0.0);`
`}`
	`public void testTCNUMBER() {`
	`#SEQUENCE#`
	`assertTrue (total == expectedTotal);`
	`}`

2. The *When clauses* section establishes that, when the *userId* = 2, test cases must be generated using the *ZeroExpected* template. Otherwise, the *NormalTemplate* must be used.

6.5 | Test cases

Table 7 shows the two templates mentioned in the *When clauses* section of Table 6. Both of them are quite similar, only differing in the expression used to build the assertion.

TABLE 8 Test data combinations corresponding to the **ABCDECDEF** sequence

Sequence: ABCDECDEF		
#	**Data Combination**	**Template**
1/36	*1,”2017/01/31”,1,”2017/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
2/36	*1,”2018/01/01”,1,”2018/01/01”,1,2,”2018/01/15”,”2018/01/31”*	*Zero*
3/36	*1,”2018/01/15”,1,”2018/01/15”,1,1,”2018/01/15”,”2018/01/15”*	*Normal*
4/36	*1,”2018/01/17”,1,”2018/01/17”,1,1,”2018/01/01”,”2018/01/31”*	*Normal*
5/36	*1,”2018/01/31”,1,”2018/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
6/36	*1,”2018/02/01”,1,”2018/02/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
7/36	*1,”2017/01/31”,1,”2018/01/15”,1,2,”2018/01/01”,”2018/01/31”*	*Zero*
8/36	*1,”2018/01/15”,1,”2017/01/31”,1,2,”2018/01/01”,”2018/01/31”*	*Zero*
9/36	*1,”2018/01/01”,1,”2017/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
10/36	*1,”2018/01/17”,1,”2017/01/31”,1,2,”2018/01/15”,”2018/01/15”*	*Zero*
11/36	*1,”2018/01/31”,1,”2017/01/31”,1,2,”2018/01/15”,”2018/01/31”*	*Zero*
12/36	*1,”2018/02/01”,1,”2017/01/31”,1,2,”2018/01/15”,”2018/01/31”*	*Zero*
13/36	*1,”2017/01/31”,1,”2018/01/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
14/36	*1,”2017/01/31”,1,”2018/01/17”,1,2,”2018/01/15”,”2018/01/15”*	*Zero*
15/36	*1,”2017/01/31”,1,”2018/01/31”,1,2,”2018/01/15”,”2018/01/31”*	*Zero*
16/36	*1,”2017/01/31”,1,”2018/02/01”,1,2,”2018/01/15”,”2018/01/31”*	*Zero*
17/36	*1,”2018/01/01”,1,”2018/01/15”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
18/36	*1,”2018/01/15”,1,”2018/01/01”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
19/36	*1,”2018/01/17”,1,”2018/01/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
20/36	*1,”2018/01/31”,1,”2018/01/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
21/36	*1,”2018/02/01”,1,”2018/01/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
22/36	*1,”2018/01/01”,1,”2018/01/17”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
23/36	*1,”2018/01/01”,1,”2018/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
24/36	*1,”2018/01/01”,1,”2018/02/01”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
25/36	*1,”2018/01/15”,1,”2018/01/17”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
26/36	*1,”2018/01/17”,1,”2018/01/15”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
27/36	*1,”2018/01/31”,1,”2018/01/15”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
28/36	*1,”2018/02/01”,1,”2018/01/15”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
29/36	*1,”2018/01/15”,1,”2018/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
30/36	*1,”2018/01/15”,1,”2018/02/01”,1,1,”2018/01/01”,”2018/01/15”*	*Normal*
31/36	*1,”2018/01/17”,1,”2018/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
32/36	*1,”2018/01/31”,1,”2018/01/17”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
33/36	*1,”2018/02/01”,1,”2018/01/17”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
34/36	*1,”2018/01/17”,1,”2018/02/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
35/36	*1,”2018/01/31”,1,”2018/02/01”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*
36/36	*1,”2018/02/01”,1,”2018/01/31”,1,1,”2018/01/01”,”2018/01/15”*	*Zero*

```
public void test1() {
  double expectedTotal = 0.0;

  Call call=new Call();
  call.setUserId(1);

  call.setDate("2017/01/31");
  call.setCost(1);

  expectedTotal += ("2017/01/31">="2018/01/01" &&
          "2017/01/31"<="2018/01/15" ? 1 : 0);
  // Adds 0 to expectedTotal

  call.registerCall(call);


  call.setDate("2017/01/31");
  call.setCost(1);

  expectedTotal += ("2017/01/31">="2018/01/01" &&
          "2017/01/31"<="2018/01/15" ? 1 : 0);
  // Adds 0 to expectedTotal

  call.registerCall(call);

  double total=
    call.getBill(1, "2018/01/01", "2018/01/15");

  assertTrue(total == expectedTotal);
}
```

```
public void test2() {
  double expectedTotal = 0.0;

  Call call=new Call();
  call.setUserId(1);

  call.setDate("2018/01/01");
  call.setCost(1);

  expectedTotal += ("2018/01/01">="2018/01/15" &&
          "2018/01/01"<="2018/01/31" ? 1 : 0);
  // Adds 0 to expectedTotal

  call.registerCall(call);


  call.setDate("2018/01/01");
  call.setCost(1);

  expectedTotal += ("2018/01/01">="2018/01/15" &&
          "2018/01/01"<="2018/01/31" ? 1 : 0);
  // Adds 0 to expectedTotal

  call.registerCall(call);

  double total=
    call.getBill(2, "2018/01/15", "2018/01/31");

  assertTrue(total == 0);
}
```

```
public void test3() {
  double expectedTotal = 0.0;

  Call call=new Call();
  call.setUserId(1);

  call.setDate("2018/01/15");
  call.setCost(1);

  expectedTotal += ("2018/01/15">="2018/01/15" &&
          "2018/01/15"<="2018/01/15" ? 1 : 0);
  // Adds 1 to expectedTotal

  call.registerCall(call);


  call.setDate("2018/01/15");
  call.setCost(1);

  expectedTotal += ("2018/01/15">="2018/01/05" &&
          "2018/01/15"<="2018/01/15" ? 1 : 0);
  // Adds 1 to expectedTotal

  call.registerCall(call);

  double total=
    call.getBill(1, "2018/01/15", "2018/01/15");

  assertTrue(total == expectedTotal);
}
```

```
public void test22() {
  double expectedTotal = 0.0;

  Call call=new Call();
  call.setUserId(1);

  call.setDate("2018/01/01");
  call.setCost(1);

  expectedTotal += ("2018/01/01">="2018/01/01" &&
          "2018/01/01"<="2018/01/15" ? 1 : 0);
  // Adds 1 to expectedTotal

  call.registerCall(call);


  call.setDate("2018/01/17");
  call.setCost(1);

  expectedTotal += ("2018/01/17">="2018/15/01" &&
          "2018/01/17"<="2018/01/15" ? 1 : 0);
  // Adds 0 to expectedTotal

  call.registerCall(call);

  double total=
    call.getBill(1, "2018/01/01", "2018/01/15");

  assertTrue(total == expectedTotal);
}
```

FIGURE 21 Four of the Phase-3 test cases generated

Applying AETG,[33] SMACTesting generates 88 data combinations (see Table 4). As an example, Table 8 shows the 36 data combinations corresponding to ABCDECDEF: that is., two telephone calls are made and recorded, and finally, the bill is calculated. The corresponding template of each combination is on the last column:

- The first combination corresponds to two telephone calls made on 2017/January/31 by *userId = 1*. The billing period goes from 2018/January/01 to 2018/January/15: thus, the *NormalTemplate* must be applied, although the *expectedTotal* is 0.
- The second combination registers two calls of the *userId = 1*, but calculates the bill for the customer with *userId = 2*: thus, the corresponding template is also the *ZeroTemplate.* Moreover, both calls were made out of the billing period.
- The 3rd combination corresponds to two telephone calls made on January 1, 2018, by *userId = 1*. In this special case, the billing period takes only 1 day (January 15) and, therefore, both telephone calls must be charged to the customer. The corresponding phase-3 test case will have to be generated with the *NormalTemplate.*
- Taking as the last example the 22nd combination, it corresponds to two calls made on January 1 and 17 by *userId = 1*. The bill is being calculated for this very same user, from January 1 to 15. The test case must be generated with the *Normal* template and must check that the amount due is 1 €.

To conclude this example, Figure 21 shows the four test cases we have described now. Note that, for simplicity, the dates are written as if they were strings.

The generation time of the test file with the 88 *phase-3*, executable, and *oracled* test cases is 28 s.

7 | CONCLUSIONS

More than generating test cases, one of the most challenging issues on the automation of software testing is the (automatic) generation of the oracle of test cases. The solving of such problem is not only an academic issue but also an industrial challenge which is demanding software testing improvements to deal with continuous testing.

In this paper, we propose a testing approach based on the specification of regular expressions which highly automate the generation of test cases. Particularly, the approach provides the tester with a very clear workflow where the test cases are built in several steps:

1. Definition of regular expressions (or *phase-1* test cases), where the tester specifies a regular expression that depicts a general execution scenario. The tester annotates each regular expression with: (i) test data, (ii) "before instructions", (iii) precode and postcode operations, and (iv) "when clauses" (which in turn represent the future oracles).
2. Expansion of the regular expression (or *phase-2* test cases), where the regular expressions are expanded according to a length established by the tester. The main idea of this second stage is to produce, from the initial regular expression (which draws a general execution scenario) a set of scenarios (instances of the regular expression) that could occur during an interaction with the SUT. At this point, obviously, these *phase-2* test cases are not already executable.
3. Executable test cases (or *phase-3* test cases), where all the instances obtained from the regular expression expansion are populated with test data provided by the tester (in the first stage, when phase-1 tests were defined), and the executable code is automatically generated. It is very important to point out that in this stage, "when clauses" are also populated in order to generate the code of the oracle for each test case. Once the code of all the *phase-3* test cases has been generated, the test suite (that groups all the test cases intended to test the execution scenario described in the first stage), has been finally created and can be launched against the SUT.

In order to support the aforesaid workflow, we have built a supporting tool, which implements a metamodel to describe all the elements involved in the process. Each instance of the metamodel holds all the information required to generate (at the end of the process) a complete test suite to test the functionality described by the regular expression designed by the tester. The tool is being used in several software factories of a big, multinational, company.

In this article, the approach has been illustrated by conducting the test on a subset of an industrial system. This case study shows, step by step, how the different stages are carried out and which is the information that the tester must configure in order to generate the resulting test suite.

ACKNOWLEDGEMENTS

This work has been partially funded by (for M. Polo and I. García-Rodríguez) the BIZDEVOPS-Global project (RTI2018-098309-B-C31), Ministerio de Economía, Industria y Competitividad (MINECO) and Fondo Europeo de Desarrollo Regional (FEDER), and the TESTIMO project (Consejería de Educación, Cultura y Deportes de la Junta de Comunidades de Castilla La Mancha, y Fondo Europeo de Desarrollo Regional FEDER, SBPLY/17/180501/000503); and (for O. Pedreira and A. S. Places) BIZDEVOPS-Global (Ministerio de Ciencia e Innovación MICINN-AEI/FEDER-UE BIZDEVOPS: RTI2018-098309-B-C32), ETOME-RDF3 (Ministerio de Economía grant MINECO-AEI/FEDER-UE ETOME-RDFD3: TIN2015-69951-R), and Xunta de Galicia grants Xunta de Galicia/FEDER-UE, ConectaPeme, GEMA: IN852A 2018/14, Centros singulares de Investigación de Galicia-Xunta de Galicia/FEDER-UE CSI: ED431G/01 and Grupo de Referencia Competitiva 2017-Xunta de Galicia/FEDER-UE GRC: ED431C 2017/58.

ORCID

Macario Polo https://orcid.org/0000-0001-6519-6196
Oscar Pedreira https://orcid.org/0000-0001-6176-4475
Ángeles S. Places https://orcid.org/0000-0001-8539-304X
Ignacio García Rodríguez de Guzmán https://orcid.org/0000-0002-0038-0942

REFERENCES

1. Grindal M, Offutt J, Andler SF. Combination testing strategies: a survey. *Softw Test Verif Rel*. 2005;15:167-199.
2. Polo M, Reales P. Mutation testing cost reduction techniques: a survey. *IEEE Softw*. May 2010;27(3):80-86. https://doi.org/10.1109/MS.2010.79

3. Beizer B. *Software testing techniques*. London; Boston: International Thomson Computer Press; 1990.
4. Baresi L, Young M. Test oracles, technical report CIS-TR-01-02. University of Oregon, Dept. of Computer and Information Science. 2001.
5. Bertolino A. Software testing research: achievements, challenges, dreams. In *Future of Software Engineering, 2007. FOSE'07*. 2007;85-103. https://doi.org/10.1109/FOSE.2007.25.
6. Barr ET, Harman M, McMinn P, Shahbaz M, Yoo S. The oracle problem in software testing: a survey. *IEEE Trans Software Eng*. May 2015;41(5):507-525. https://doi.org/10.1109/TSE.2014.2372785
7. Pezzè M, Zhang C. Chapter one — Automated test oracles: a survey. In: Memon A, ed. *Advances in Computers*. Vol.95. USA: Elsevier; 2014:1-48.
8. Grieskamp W, Hierons RM, Pretschner A. 10421 summary— model-based testing in practice. Dagstuhl Seminar Proceedings. 2011.
9. Anand S, Burke EK, Chen TY, et al. An orchestrated survey of methodologies for automated software test case generation. *J Syst Software*. Aug. 2013; 86(8):1978-2001. https://doi.org/10.1016/j.jss.2013.02.061
10. Lebeau F, Legeard B, Peureux F, Vernotte A. Model-based vulnerability testing for web applications. In 2013 IEEE sixth international conference on software testing, Verification and Validation Workshops. 2013;445-452. https://doi.org/10.1109/ICSTW.2013.58.
11. Ball T, Hoffman D, Ruskey F, Webber R, White L. State generation and automated class testing. *Softw Test Verif Rel*. 2000;10(3):149-170.
12. Offutt J, Liu S, Abdurazik A, Ammann P. Generating test data from state-based specifications. *Softw Test Verif Rel*. 2003;13:25-53.
13. Hong HS, Lee I, Sokolsky O, Cha SD. Automatic test generation from Statecharts using model checking. *In Proceedings of FATES'01, Workshop on Formal Approaches to Testing of Software, volume NS-01-4 of BRICS Notes Series.* 2001;15-30.
14. Burton McDermid S. Automatic generation of tests from statechart specifications. In *Proc. of Formal Approaches to Testing of Software (FATES'01*, Aalborg, Germany
15. Utting M. *Practical Model-Based Testing: A Tools Approach*. USA: Morgan-Kaufmann; 2010.
16. Weißleder S. *Test Models and Coverage Criteria for Automatic Model-Based Test Generation with UML State Machines*. Berlin: Humboldt-Universität zu Berlin; 2010.
17. Hierons RM, Merayo MG. Mutation testing from probabilistic finite state machines. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, 2007. TAICPART-Mutation 2007. 2007;141-150. https://doi.org/10.1109/TAIC.PART.2007.20.
18. Salas PAP, Krishnan P, Ross KJ. Model-based security vulnerability testing. In *2007 Australian Software Engineering Conference (ASWEC'07).* 2007;284-296. https://doi.org/10.1109/ASWEC.2007.31.
19. Belli F. Finite state testing and analysis of graphical user interfaces. In *Proceedings 12th International Symposium on Software Reliability Engineering.* 2001;34-43. https://doi.org/10.1109/ISSRE.2001.989456.
20. Jahangirova G. *Oracle assessment, improvement and placement*. London: University College London; 2019.
21. Kilincceker O, Silistre A, Challenger M, Belli F. Random test generation from regular expressions for graphical user interface (GUI) testing. In 2019 IEEE 19th international conference on software quality, Reliability and Security Companion (QRS-C). 2019;170-176. https://doi.org/10.1109/QRS-C.2019.00044.
22. Tuglular T, Muftuoglu A, Belli F, linschulte M. Model-based contract testing of graphical user interfaces. *IEICE Trans Inf Syst*. 2015;98(7):1297-1305. https://doi.org/10.1587/transinf.2014EDP7364
23. Kirani S, Tsai WT. Specification and verification of object-oriented programs. 1994.
24. Polo M, Tendero S, Piattini M. Integrating techniques and tools for testing automation: research articles. *Softw Test Verif Reliab*. Mar. 2007;17(1):3-39. https://doi.org/10.1002/stvr.v17:1
25. OMG. Unified modeliing language. UML 2.5.1, formal. Dec. 2017.
26. Kilinççeker O, Turk E, Challenger M, Belli F. Regular expression based test sequence generation for HDL program validation. 2018 *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C).* 2018; doi: https://doi.org/10.1109/QRS-C.2018.00103.
27. Belli F, Dreyer J. Program segmentation for controlling test coverage. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*. 1997;72-83. https://doi.org/10.1109/ISSRE.1997.630849.
28. Belli F, Grosspietsch K. Specification of fault-tolerant system issues by predicate/transition nets and regular expressions-approach and case study. *IEEE Trans Software Eng*. Jun. 1991;17(6):513-526. https://doi.org/10.1109/32.87278
29. Liu P, Ai J, Xu Z. A study for extended regular expression-based testing. In 2017 *IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, Los Alamitos, CA, USA. 2017;821-826. https://doi.org/10.1109/ICIS.2017.7960106.
30. Briand L, Labiche Y. A UML-based approach to system testing. *Softw Syst Model*. Sep. 2002;1(1):10-42. https://doi.org/10.1007/s10270-002-0004-8
31. Tretmans J, Brinksma E. TorX: automated model based testing - Côte de Resyste. 2003.
32. de Vries RG. Towards formal test purposes. In *Formal Approaches to Testing of Software 2001 (FATES'01).* 2001;61-76.
33. Cohen D, Society IC, Dalal SR, Fredman ML, Patton GC. The AETG system: an approach to testing based on combinatorial design. *IEEE Trans Software Eng*. 1997;23:437-444.
34. Myers GJ. *The Art of Software Testing*. 2nd ed. New Jersey, U.S.A.: Wiley; 2004.
35. Polo M. Regular expressions expansion engine. [Online]. Available:. https://bitbucket.org/macariopolo/expresionesregulares
36. Usaola MP, Romero FR, Aranda RR, Rodríguez IG. Test case generation with regular expressions and combinatorial techniques. In 2017 IEEE international conference on software testing, Verification and Validation Workshops (ICSTW). 2017;189-198. https://doi.org/10.1109/ICSTW.2017.38.
37. Fowler M.The value object pattern, martinfowler.com. [Online]. Available: https://martinfowler.com/bliki/ValueObject.html. Accessed: 07-Oct-2019.

How to cite this article: Polo M, Pedreira O, S. Places Á, García Rodríguez de Guzmán I. Automated generation of *oracled* test cases with regular expressions and combinatorial techniques. *J Softw Evol Proc*. 2020;e2273. https://doi.org/10.1002/smr.2273