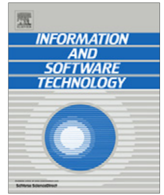
Information and Software Technology 55 (2013) 301–319

Contents lists available at SciVerse ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Automated generation of test oracles using a model-driven approach

Beatriz Pérez Lamancha [a,*], Macario Polo [b], Danilo Caivano [c], Mario Piattini [b], Giuseppe Visaggio [c]

[a] Software Testing Centre, Republic University, Montevideo, Uruguay
[b] Alarcos Research Group, Castilla-La Mancha University, Ciudad Real, Spain
[c] Dipartamento di Informatica, Universitá degli Studi, Bari, Italy

ARTICLE INFO

Article history:
Received 15 November 2011
Received in revised form 16 August 2012
Accepted 21 August 2012
Available online 19 September 2012

Keywords:
Software testing
Automated test oracle
Model-driven testing
UML state machine
Model to text transformation

ABSTRACT

Context: Software development time has been reduced with new development tools and paradigms, testing must accompany these changes. In order to release software products in a timely manner as well as to minimise the impact of possible errors introduced during maintenance interventions, testing automation has become a central goal. Whilst research has produced significant results in test case generation and tools for test case (re)-execution, one of the most important open problems in testing is the automation of oracle generation. The oracle decides whether the program under test has or has not behaved correctly and then issues a pass/fail verdict. In most cases, writing the oracle is a time-consuming activity that, moreover, is manual in most cases.

Objective: This article automates two important steps in the test oracle: obtention of expected output and its comparison with the actual output, using a model-driven approach.

Method: The oracle automation problem is resolved using a model-driven framework, based on OMG standards: UML is used as metamodel and QVT and MOF2Text as transformation languages. The automated testing framework takes the models that describe the system as input, using UML notation and derives from them the test model and then the test code, following a model-driven approach. Test oracle procedures are obtained from a UML state machine.

Results: A complete executable test case at functional test level is obtained, composed of a test procedure with parametrized input test data and expected result automation.

Conclusion: The oracle automation is obtained using a model-driven approach, test cases are obtained automatically from UML models. The model-driven testing framework was applied to an industrial application and has been useful to testing automation for the main functionalities in the system.

© 2012 Elsevier B.V. All rights reserved.

1. Introduction

New paradigms in software development reduce the time needed to obtain the software product, automating the generation of code and reusing artefacts from other products. Software testing must accompany these changes, reducing testing time both when the system is developed and when the product is in the maintenance phase.

In Model-Driven Engineering (MDE), models are treated as first-class entities that are used to generate code automatically. In the maintenance phase, when a change request is required, the models are modified and the code is automatically re-generated. Since manual testing is time-consuming and error-prone, the idea of MDE can be applied to develop automatic tests from models: thus, when a modification is made in the models, not only the code is automatically re-generated, but also the test cases, which must be ready to be re-executed.

Model-Based Testing (MBT) provides techniques for the automatic generation of test cases using models extracted from software artefacts [1]. We are interested in applying the MDE [2] paradigm to testing. MDE considers models for software development, maintenance and evolution through model transformation [3]. We use the term Model-Driven Testing (MDT) to refer to model-based testing that follows the MDE paradigm, i.e., the test cases are automatically generated from software artefacts through model transformations.

In previous works [4–6], we defined an automated model-driven testing framework by means of two types of transformations:

1. Model-to-model transformation (M2M) to generate test models from design models. This transformation takes UML 2.0 models [7] as input and produces UML Testing Profile (UML-TP) models [8,4].

* Corresponding author.
E-mail addresses: bperez@fing.edu.uy (B.P. Lamancha), macario.polo@uclm.es (M. Polo), caivano@di.uniba.it (D. Caivano), mario.piattini@uclm.es (M. Piattini), visaggio@di.uniba.it (G. Visaggio).

0950-5849/$ - see front matter © 2012 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.infsof.2012.08.009

2. Model-to-text transformation (M2T) to generate test code from test models. This transformation takes UML-TP models as input and produces xUnit code through an MOF-to-text transformation.

As result, a "test procedure" is obtained. A test procedure contains instructions for the set-up, execution and evaluation of results for a given test case [2]. A test case contains a set of inputs, execution conditions and expected results which are developed for testing a particular system's objective (such as exercising a given path or verifying the compliance of a specific requirement) [2]. The actual result obtained by the test case is compared to the expected result through an "oracle". An oracle is any (human or mechanical) agent, which decides whether a program behaved correctly in a given test, and accordingly produces a verdict of "pass" or "fail". Behind this single idea, the oracle automation is a quite interesting line of research: automatically determining the expected output of a program for any possible input.

So, this article presents a contribution to the automated generation of test oracles. This implies to add two elements to the test case: one that takes the input test data and automatically calculates the expected result; and other that automatically obtains the verdict comparing the actual and the expected result. Whilst the actual result is obtained when the test case is executed in the system under test, the main difficulty stems in obtaining the expected result (mainly due the infinite set of possible input data for each functionality to test). The inclusion of the oracle in the test case is essential for finding errors in the System Under Test (SUT), which is the main goal of testing [9]. Although the generation of test data and sequences of instructions for setting up and executing test cases has a relatively high automation degree, no generic mechanism exists to describe the oracle [10], and its description is therefore difficult and expensive [11]. Thus, it is almost always implemented by hand. For Baresi and Young [12] and for Bertolino [10], an ideal oracle should verify the fulfilment of all system characteristics, but avoid "over specification". Otherwise, writing a generic oracle could be as costly as manually creating an oracle for each test case.

Our proposal uses UML state machines to describe test oracles. UML notation is a widely known and applied standard in development processes. Our previously defined MDT framework [4–6] is augmented with automated test oracles. Since this is a standardised framework based on UML notation, the use of UML state machines makes it possible to use the same notation for development and testing. UML state machines are very useful to model the behaviour of the most complex and critical components in object-oriented software [13], which, in practice, is an essential advantage for developers, and to improve the communication between developers and testers. These complex components are also the most important for testing, so design models are reused in test models to describe test oracles.

As case study, we use the Monica Mobile system, an application for monitoring load sensors in trucks. Sensors periodically send data to a mobile device that provides different kinds of visual and audible information both to the truck driver (by means of bluetooth communication) and to a central computer (by 3G mobile technology). This behaviour can be easily modelled with a UML state machine and used by designers, developers and testers.

Our approach takes advantage of the design model for generating test cases, thus preventing, as much as possible, the use of different artefacts by testers and developers. Therefore, this article uses UML state machines specified by developers that are later annotated by testers with testing information using a UML Profile. Testers complete the description of the SUT, making possible to automate the test oracle generation. Thus, where in previous works we generated executable test cases in the form of execution scenarios loaded with test data, now those test cases are enriched with oracle information proceeding from the associated state machines.

Since this article extends a previously defined approach for model-driven testing with the inclusion of automated test oracles, it starts with a brief description of previous works. Then, Section 3 describes the oracle automation problem and explains why it is important in the testing process. Section 4 presents how the MDT framework is augmented with automated test oracles and uses a Library system as its running example. Section 5 describes the automated test oracle approach and how it was implemented using a UML state machine. The whole approach was applied to the Monica Mobile system, which, is described in Section 6. Finally, related works are discussed and conclusions and future work are outlined.

2. Model-driven testing framework

Although there are several approaches for model-based testing [14,15], the adoption of model-based testing by the industry remains low and signs of the anticipated research breakthrough are weak [10]. Model-driven testing (MDT) refers to a model-based testing that follows an MDE paradigm, i.e., the test cases are automatically generated using models extracted from software artefacts through model transformations.

Using a model-driven approach, the automation is developed through model transformations. This involves a source metamodel, a target metamodel, and a set of transformation rules that describe how the elements from the source metamodel are mapped into elements of the target metamodel.

We have defined and implemented an automated framework for model-driven testing (see Fig. 1) [5]. The main characteristics of the framework are:

- Standardised: this is based on Object Management Group (OMG) standards. The standards used are UML [7] and UML Testing Profile [8] as metamodels, and the Query/View/Transformation (QVT) [16] and MOF2Text [17] as standardised transformation languages.
- Functional testing level: the framework generates the test cases at the functional testing level using model-driven testing techniques. Test case scenarios are automatically generated from design models and evolve with the product until test code generation. System behaviour is represented in a design layer using UML sequence diagrams.

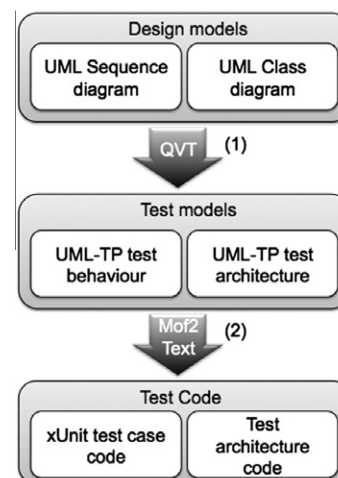


Fig. 1. Model-driven testing approach.

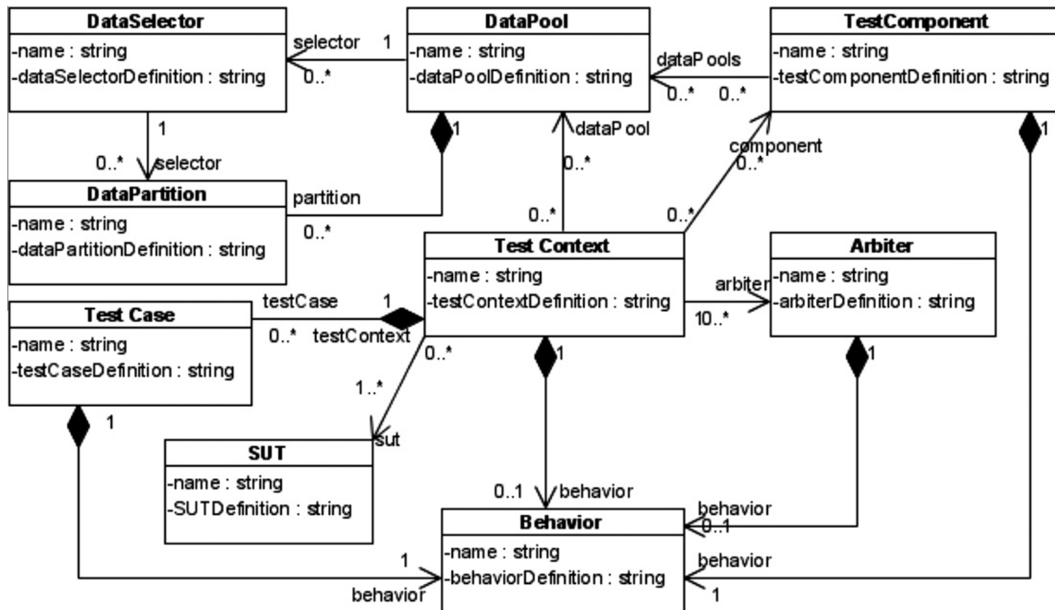


Fig. 2. UML-TP metamodel.

- Framework implementation using existing tools: no tools have been developed to support the framework; existing market tools that conform to the standards can be used. The requisite is that the modelling tool can be integrated with the tools that produce the transformations.

The following subsections explain the metamodels and the transformations used for the framework implementation.

2.1. Metamodels

Three metamodels are used in the framework: UML 2.0, UML Testing Profile and xUnit. The objective of UML 2.0 [7] is to provide system architects, software engineers, and software developers with tools for the analysis, design, and implementation of software-based systems as well as for modelling business and similar processes [7].

The **UML 2.0 Testing Profile** (UML-TP) [8] extends UML 2.0 with specific concepts for testing, grouping them into test architecture, test data, test behaviour and test time. Fig. 2 shows an excerpt from the UML-TP metamodel. The test architecture in UML-TP is the set of concepts to specify the structural aspects of a test situation. It includes the Test Context, which contains the test cases (as operations) and whose composite structure defines the test configuration. The test behaviour specifies the actions and evaluations necessary to evaluate the test objective, which describes what should be tested. The TestCase specifies one case to test the system, including what to test it with, the required input, result and initial conditions. It is a complete technical specification of how a set of TestComponents interacts with a System Under Test (SUT) to realise a TestObjective and return a Verdict value [8].

xUnit is a family of frameworks, which enable the automated testing of different elements (units) of software. These frameworks are based on a design by Kent Beck, originally implemented for Smalltalk as SUnit [18]. Gamma and Beck ported SUnit to Java, thus creating JUnit.[1] From there, the framework was also ported to other languages, such as NUnit[2] for.NET, PUnit for Python,[3] etc.

2.2. Transformations

The model-driven approach is defined in two layers for testing: test model and test code.

- *Model-to-model transformation (M2M)*: The input of this transformation is the description of the SUT by means of UML sequence diagrams to represent scenarios and UML Class diagrams to represent the static structure of the system. For each scenario represented as a sequence diagram, a QVT transformation (arrow 1 in Fig. 1) produces a test model composed of, on the one hand, a class diagram representing the test architecture and on the other hand, a sequence diagram representing the test behaviour whose involved objects correspond to instances of the class diagram. These models conform to the UML Testing Profile (UML-TP).
- *Model-to-Text transformation (M2T)*: This transformation takes the testing models as input and produces executable test code to test the system as output. According to the model-independence principles of MDE, a test model can be translated into executable test code for different development languages and environments. In our case (Fig. 3), this is carried out by a transformation written with the MOFScript tool,[4] which implements the OMG's MOF model-to-text transformation [17]. Each transformation defined with MOFScript language is composed of a *texttransformation* element, which is the main element that transforms a model into text. It is composed of rules (similar to a function). Each rule performs a sequence of operations or calls to other rules in order to analyse the input models and generates the desired text. A rule has a context type, which is a type of input metamodel and can have a return element, which can be reused in other rules and input parameters to perform the operations defined in the rule. A *texttransformation* element can also have an entry point rule. This is a special type of rule called *main*. This is the first rule to be executed when the transformation is executed and is responsible for executing the rest of the transformation rules.

[1] http://www.junit.org/.
[2] http://www.nunit.org/.
[3] http://punit.smf.me.uk/.
[4] http://www.eclipse.org/gmt/mofscript/.

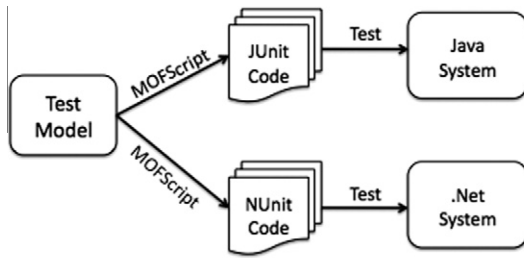


Fig. 3. Test model to test code transformation.

3. Oracle automation problem

Automating the oracle means that the testing system must be capable of including a mechanism in the test cases to give a coherent pass/fail verdict by itself, which must depend on the input test data and on the expected result. Then, given a function to test and the input test data, the expected result is calculated automatically. Fig. 4 shows the generic procedure to automate the oracle. The *oracle problem* is the process to obtain the expected result for the test cases and its automation is still an open issue in software testing research. Using an automated oracle to support testing activities can reduce cost of the testing process and its related maintenance.

A test oracle is defined to contain two parts: the *oracle information* that is used as the expected output; and an *oracle procedure* that compares the oracle information with the actual output [20]. Since manual and human oracles are costly and unreliable, automated test oracles are required to ensure testing quality while reducing testing costs. Due to challenges in providing complete test oracles, it can be expensive and sometimes impossible to provide a complete and reliable oracle.

Although much research has been done to provide test oracles automatically (see surveys [12,19]), none of them completely automates all test oracle activities in all circumstances [19]. It is unlikely that there will ever be an ideal system for creating test oracles. Instead, there are a variety of approaches that make different compromises to produce test oracles that, though not ideal, balance the trade-offs to provide useful capabilities [12].

The current state of MDT frameworks (defined in Section 2) does not consider test oracles, since is a standardised framework based on UML Models, this paper uses UML State Machine to describe the test oracle. Clearly, this solution only resolves the oracle problem in those cases where the system behaviour can be modelled as a state machine, but as was argued previously, a complete solution for all systems is impossible at this time.

UML state machines can be used to express the behaviour of part of a system, modelled as a graph composed of states and transitions that are triggered by dispatching a series of (event) occurrences [7]. The main elements in a UML state machine are [7]:

- State: models a situation during which some invariant condition holds. The invariant may represent a static situation such as an object waiting for some external event to occur. However, it can also model dynamic conditions such as the process of performing some behaviour.

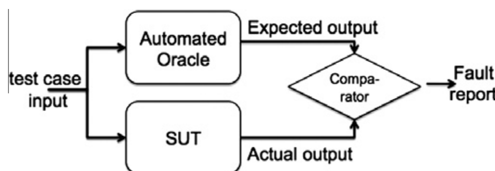


Fig. 4. Automated oracle procedure (taken from [19]).

- Transition: this is a directed relationship between a source state and a target state, which takes the state machine from one state to another, representing the complete response of the state machine to an occurrence of an event of a particular type. The transition has three main elements: an *event* that specifies the triggers that may fire the transition, a *guard* (a constraint that is evaluated when an event occurrence is dispatched by the state machine; if the guard is true at that time, the transition may be enabled; otherwise, it is disabled) and an *effect* (specifies an optional behaviour to be performed when the transition fires).

In our approach, the oracle automation problem is resolved using a model-based approach, where UML state machines are augmented with specific information for testing using a UML Profile for oracles. Later, these models are transformed using a model-driven approach to obtain the oracle class that returns the expected result. The following section explains how the MDT framework is enhanced with automated oracles and Section 5 explains the implementation of the oracle automation proposal.

4. MDT framework enhanced with test oracles

In order to automatically include the oracle in the test cases, we extended the model-driven testing framework presented in Section 2 using UML state machines to represent the oracle information. The objective is the automated generation of test oracles from models describing the system functionalities through model transformations.

Since the framework now includes UML state machines as an essential element of test generation, the models dealt with in Fig. 1 have been extended with the UML state machine metamodel (Fig. 5). In the first step (arrow 2), the tester applies the oracle profile to the state machine with specific information for testing.

The MOF2Text transformation (arrow 3) is completely automated: it takes a UML state machine stereotyped with and transforms it into a class with the information to calculate the expected result (i.e. the test oracle).

The following subsection explains how the model-driven testing framework works to test a library system:

4.1. Running example: library system

A library system is used as the running example. This system provides functionalities to find, reserve, borrow and renew items

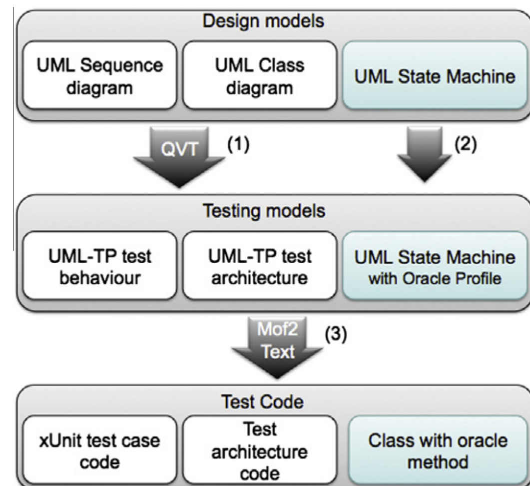


Fig. 5. Model-driven testing with oracle automation.

in a library. The items can be books, CDs, magazines or movies. Fig. 6 shows an excerpt from the Library UML Class diagram. The borrower has a status that can be: normal, delayed (the borrower returns the book late), suspended (the borrower returns the book after the suspension days) or deleted.

Fig. 7 shows the *Return Book* scenario, where the customer decides to return a previously borrowed book. The customer provides the identification and the book copy, the system calculates the days that the book was borrowed. To do that, the system first calculates the borrower's status and returns the status to the actor. If the actor confirms the return, the system calculates the amount to pay (if applicable) and returns this amount to the customer. The borrower only needs to pay if s/he returns the book after the allowed days.

4.2. Test model generation

This section outlines the QVT transformation (labelled 1 in Fig. 2). Taking the *return book* sequence diagram, to generate the test behaviour from a functional testing point of view, each message between the actor and the SUT must be tested [4]. For this, the following steps in the test case behaviour are generated with the QVT transformation:

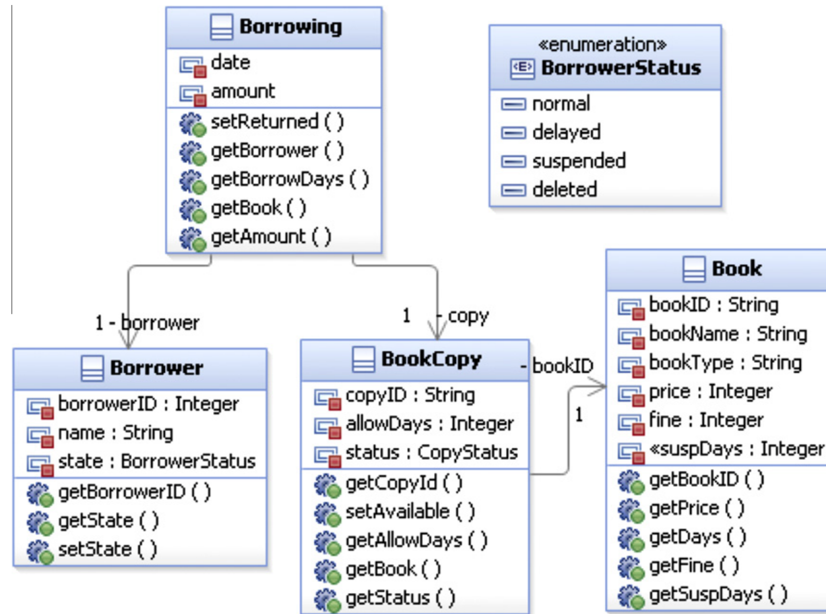


Fig. 6. Excerpt of library system class diagram.

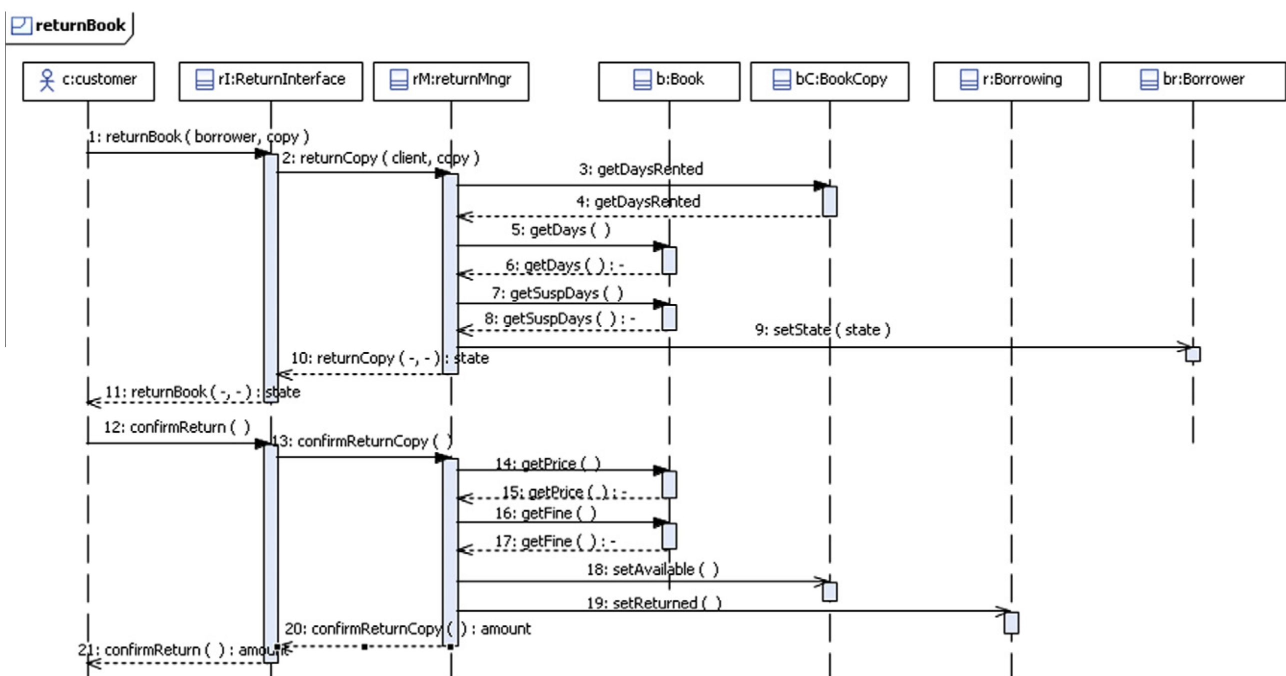


Fig. 7. Return book scenario.

- Obtaining the test data: In the UML-TP, the test data required to execute the test case is stored in the DataPool. The TestComponent asks for the test data using the DataSelector operation in the DataPool.
- Executing the test case on the SUT: the TestComponent simulates the actor and stimulates the SUT. The TestComponent calls the SUT functionality to be tested: i.e., TestComponent calls the message to test in the SUT.
- Obtaining the test case verdict: the TestComponent is responsible for checking whether the value returned for the SUT is correct, and uses the Validation Action for that.

Fig. 8 shows the automatically generated test case for the Return Book scenario. To understand the model, take the first message from the actor to the system in Fig. 7. The message is: *returnBook (borrower, copy): state*. Following the steps described above:

- Obtaining the test data: to test this message, we need the input data*(client and copy)* and the expected result *(expected_state)*, which are stored in the DataPool. They are recovered by a call to the *DataSelector ds_returnBook (client, copy, expected_state)* operation.
- Executing the test case on the SUT: The TestComponent calls the operation *returnBook (borrower, copy)* in the SUT, which returns the actual result (*state* in this case).
- Obtaining the test case verdict: once the operation is executed in the SUT, the next step is to compare whether the expected result (*expected_state*) and the result actually obtained from the SUT (*state*) are equal. This is represented with the sentence {*expected_state==state*} in the validation action.

The same steps would be automatically generated for other messages from the actor to the SUT, such as the further call to *confirmReturn (): amount*.

4.3. Test code generation

This section outlines the transformation from test model to test code (labelled 2 in Fig. 2). The MOFScript transformation takes the test model as input and, for each UML Interaction (i.e. sequence diagram) stereotyped as ≪TestCase≫, transforms it into a xUnit test method. To illustrate this transformation we use Java and its corresponding test language JUnit.

Listing 1 shows the JUnit test code generated for return book (Fig. 8), the class described is a TestContext; in the UML-TP this class is responsible for invoking the test cases (in this case, the test method). Basically, the transformation creates the header of the method (lines 1–7) and calls the DataPool to obtain the test data (line 9). We are interested in testing each test procedure with more than one data value. For this, the transformation creates a loop to iterate with the test data within the test scenario (lines 10–23). Inside the *for* loop, the *dataPool* first returns a vector with four elements: *clientds_returnBook* and *copyds_returnBook* (lines 11 and 12) are the test input data. The expected result is also stored in the dataPool, which is retrieved as *amountds_returnBook*and *stateds_returnBook* (lines 13 and 14).

Next, the *returnBook* method of the SUT is called (line 15) with the test data retrieved from the *DataPool*. The result returned by the SUT is compared to the expected result, as defined in the *validation action* (line 18). The same occurs for the *confirmReturn* method (line 20) and its validation action (line 22). More information about the semantics of the transformations from test models to*xUnit*code and about how MOFScript transformations were developed can be consulted in [6].

4.4. Test oracle motivation

It is important to note at this point that in Listing 1, the expected result was manually calculated by the tester and stored in dataPool. Following the UML-TP, each test case procedure (i.e. a se-

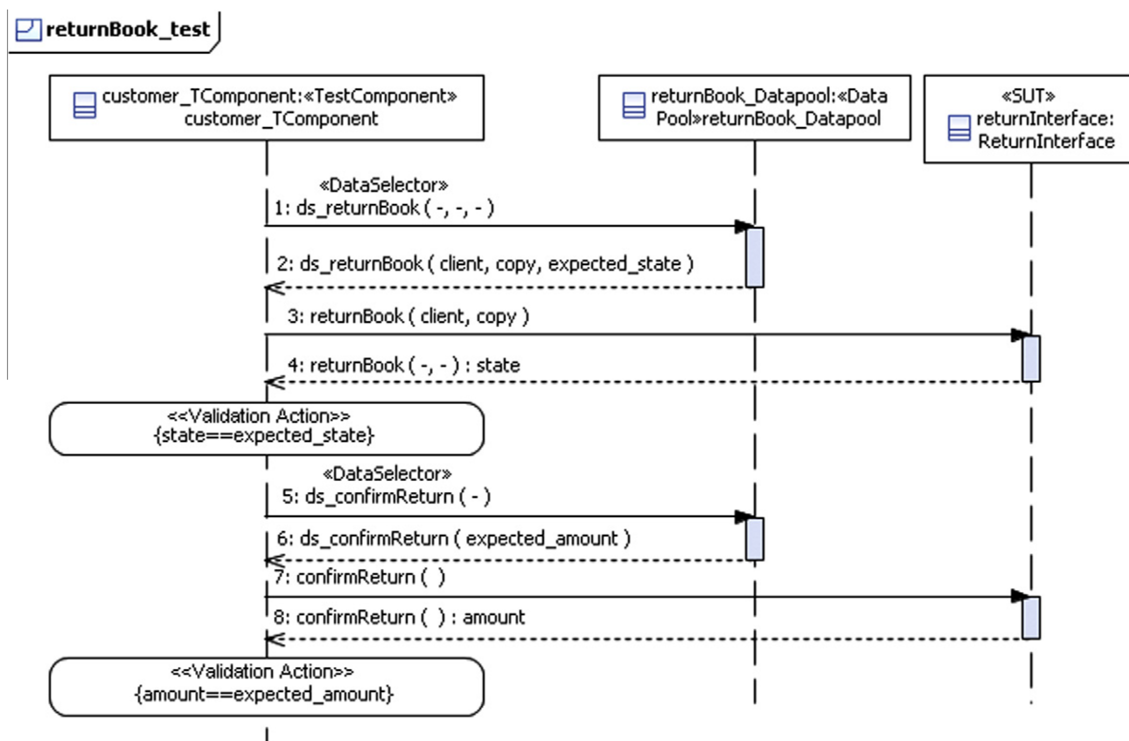


Fig. 8. Automatically generated test case for return book scenario.

```
1.  import java.util.Vector;
2.  import junit.framework.TestCase;
3.  import Library.ReturnInterface;
4.  public class ReturnBook_TestContext extends TestCase {
5.  //Atributes
6.  private ReturnInterface returnInterface;
7.  public void testReturnBook_test(){
8.      //Ask the data from DataPool
9.      Vector<ValueSet> v = returnBook_Datapool.getreturnBook_test();
10.     for(ValueSet vs:v){
11.         String clientds_returnBook = (String) vs.getValue("clientds_returnBook");
12.         String copyds_returnBook = (String) vs.getValue("copyds_returnBook");
13.         String stateds_returnBook = (String) vs.getValue("stateds_returnBook");
14.         Integer amountds_confirmReturn = (Integer) vs.getValue("amountds_confirmReturn");
15.         //Call returnBook in SUT
16.         String state = returnInterface.returnBook( clientds_returnBook, copyds_returnBook);
17.         //Validation Action for returnBook
18.         assertTrue(stateds_returnBook.equals(state));
19.         //Call confirmReturn in SUT
20.         Integer amount = returnInterface.confirmReturn();
21.         //Validation Action for confirmReturn
22.         assertTrue(amountds_confirmReturn.equals(amount));
23.     }
24. } //End testReturnBook_test
25. } // End of class returnBook_TestContext
```

Listing 1. JUnit automatically generated test code.

quence diagram) which corresponds to an execution scenario, has one or more associated tuples in the DataPool in order to test it. Besides the input test data, these tuples also have the expected value.

Taking *return book* as the scenario to test, the test inputs are the borrower and the copy. The result is the borrower state and the amount to pay. If the oracle is not automated, the expected result must be manually calculated by the tester and stored in the DataPool.

Fig. 9 shows an example of the test data stored in the DataPool. In *TestCase 1*, *bperez* borrows book copy *XR2011A*. When she returns the book and the *return book* scenario is executed, the total days borrowed is 6. Up to now, the tester must manually calculate the *expected_state* (*delayed*) and *the expected_amount* (10), and then store them in the datapool (last two rows in Fig. 9). When the test case is executed, the expected result is compared with the result returned by the system.

Calculating the expected result for each test case by hand is a tedious and error-prone task. To deal with a large number of cases, a UML state machine can be provided with the functionality to obtain the expected result. Note that the state of a UML classifier can be described as a function of its attributes and that, thus, the sequences of messages represented in an interaction diagram modify

		Test Case 1	Test Case 2
Borrower	borrowerID	bperez	mpolo
	state	normal	delayed
Book Copy	copyID	XR2011A	SP2010B
	allowDays	5	3
	bookID	XR2011	SP2010
Book	bookID	XR2011	SP2010
	price	10	12
	fine	0,15	0,1
Borrowing	borrowerID	bperez	mpolo
	copyID	XR2011A	SP2010B
	BorrowDays	6	2
Oracle	expected_amount	10	0
	expected_state	delayed	normal

Fig. 9. Example of input test data stored in DataPool.

the states of the objects involved in the scenario, according to the corresponding state machine.

Fig. 10 shows the UML state machine for the borrower status (normal, delayed and suspended) when a book is returned. Transitions have three variables in their guards: *daysBorrowed* (number of days that the borrower has the book), *allowedDays* (number of days that the book can be borrowed) and *suspDays* (sanction days when the book is not returned on time).

Also, it uses three variables in the effects of the transitions: *amount* (represents the total amount the borrower must pay if the book is returned late); *price* (amount to pay for these books); and *fine*(fine per day to apply to this book). The meaning of each transition is the following:

- t1: When the borrower status is *normal* and the days borrowed exceeds the allowed days, the borrower status changes to *delayed* and the amount to pay is the price.
- t2: When the borrower status is *normal* and the days borrowed are fewer or equal to the allowed days, the borrower status remains *normal* and the amount to pay is zero.
- t3: When the borrower status is *delayed* and the days borrowed are fewer or equal to the allowed days, the borrower status changes to *normal* and the amount to pay is zero.
- t4: When the borrower status is *delayed* and the days borrowed exceeds the allowed days, the borrower status remains *delayed* and the amount to pay is $price + fine * daysBorrowed$.
- t5: When the borrower status is *delayed* and the days borrowed exceeds the suspension days, the borrower status changes to *suspended* and the amount to pay is $price + fine * daysBorrowed$. When a borrower is in a suspended state, s/he cannot borrow a book.

This state machine is in the design layer (see Fig. 5) and is also used to automate the test case oracle, producing a method to be used in test cases. The following section explains how the UML state machine is used to obtain the expected result in an automated way instead of using a manual procedure, which is the main contribution of this work.

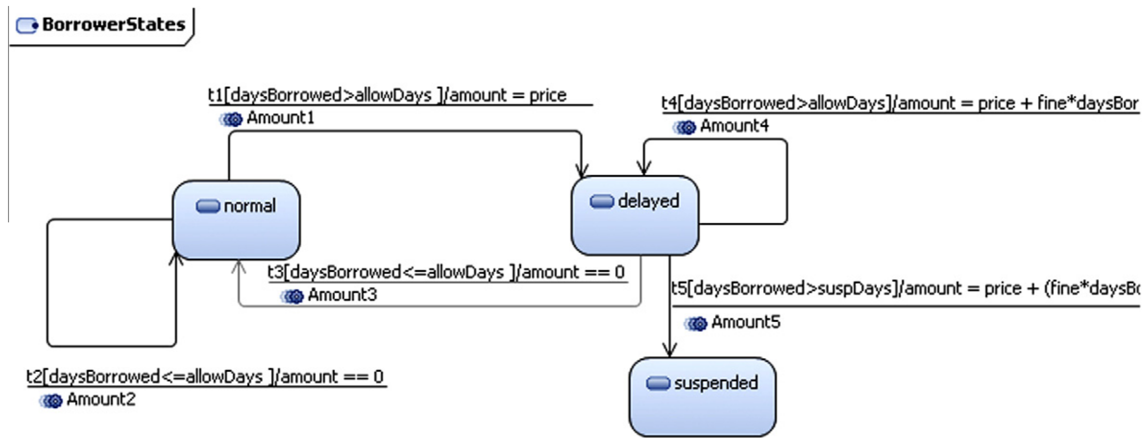


Fig. 10. UML state machine for the borrower states.

5. Automated test oracle implementation

To better understand our proposal for automated test oracles, the high-level view of the automated oracle process presented in Fig. 4 has been extended. Fig. 11 describes the way in which the test case verdict is obtained using the metamodels defined in the model-driven testing framework (see Fig. 5).

To automate the generation of oracles, the UML state machine must be augmented with specific information for testing. For this reason, a UML Profile for Oracles is defined to be applied to UML state machines.

Fig. 12 specifies the steps to obtain the oracle method from a UML state machine. The addition of the profile elements is a manual task performed by the tester. Once the UML state machine with the Oracle profile is obtained, the state machine is automatically transformed using MOFScript. This transformation returns a class (in the language used to implement the SUT) with two methods: one that returns the expected state and another that returns the expected result.

This approach has been standardised and made UML compliant so that existing tooling can also be used for oracle automation. The same UML editor used to describe UML sequence diagrams and UML class diagrams is used to describe UML state machines. Also, UML tools provide functionalities to define and apply UML Profiles. Specifically, examples in this work are depicted using *IBM Rational Software Architect*.[5]

The following subsections explain the steps in Fig. 12 in detail. First the UML Profile defined for test oracles is explained, then, the MOFScript transformations to obtain the oracle class are described. Finally, the automatically generated code for test oracles is presented.

5.1. UML profile for oracle determination

First of all, it is necessary to know when a UML state machine represents an oracle and when it represents the system behaviour. We define a profile that complements the UML-TP to automate the oracle.

Fig. 13 shows the stereotypes defined in the profile. The semantics for the stereotypes in the profile are explained below using the UML state machine for the Library example in Figs. 10 and 14 show the resulting UML state machine.

[5] http://www.ibm.com/developerworks/rational/products/rsa/.

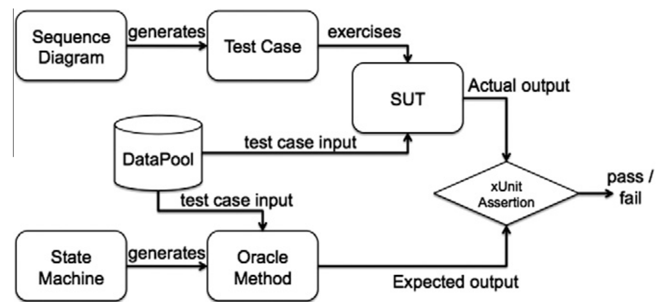


Fig. 11. Automated test oracle proposal.

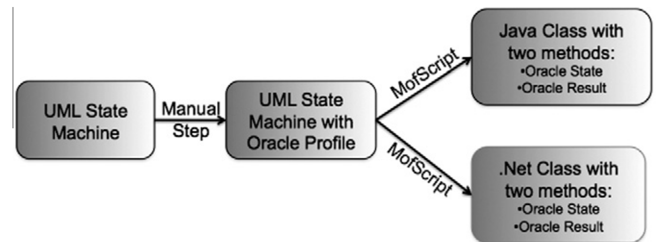


Fig. 12. Oracle automation proposal.

- TestOracle: a UML state machine that defines an oracle has the stereotype ≪TestOracle≫. Fig. 14 is stereotyped ≪test oracle≫.
- Preconditions and post-conditions: depending on the transition, states can be considered as pre- or post-states. For example, Fig. 10 has three states: normal, delayed, suspended. Normal and delayed are pre-states, because the borrower must be in one of these states to borrow a book. Likewise, states can also be post-states for some transitions: in the example, the three states can be post-states. Depending on the role, states are stereotyped as ≪precondition≫ or ≪postcondition≫.
- TestPrecondition: As noted in Section 3, a UML state machine uses variables in the guard and effect clauses. We need to know how to obtain these variables from the dataPool to generate the oracle. This information is missing in the UML state machine and is necessary to obtain the test oracle. TestPrecondition is defined in a UML Comment stereotyped as ≪TestPrecondition≫, which is attached to the UML State Machine. The UML Comment stereotyped as ≪TestPrecondition≫ may have different tags to obtain the data for the oracle (see Figs. 14 and 15). The text inside

Stereotype	Applied to	Semantic
<<testOracle>>	State Machine	Indicates that the UML state machine represents an oracle for testing
<<precondition>>	State	Indicates that the state can be a initial state for testing
<<postcondition>>	State	Indicates that the state can be a final state for testing
<<TestPrecondition>>	Comment	includes information about how the variables used in the state machine are configured

Fig. 13. UML profile defined for oracle automation.

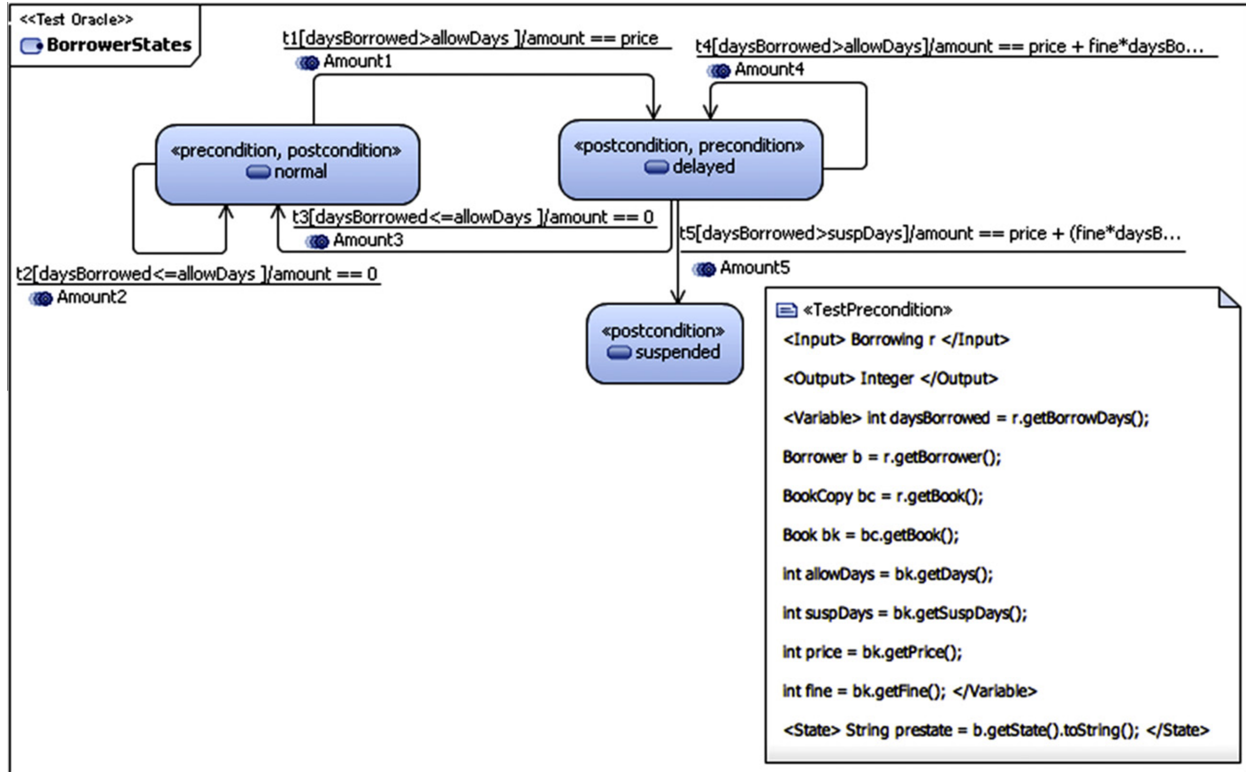


Fig. 14. UML state machine using the UML profile defined for oracle automation.

the tags is written using the same programming language that uses the system under test, in the case of Library example, the text is written using *Java* language. The possible tags are:

```
<<Test Precondition>>

<Input> Borrowing r </Input>
<Output> Integer </Output>
<Variable>
   int daysBorrowed = r.getBorrowDays();
   Borrower b = r.getBorrower();
   BookCopy bc = r.getBook();
   Book bk = bc.getBook();
   int allowDays = bk.getDays();
   int suspDays = bk.getSuspDays();
   int price = bk.getPrice();
   int fine = bk.getFine();
</Variable>
<State>
   String prestate = b.getState().toString();
</State>
```

Fig. 15. UML comment attached to BorrowerStates (see Fig. 14).

– Input: The information between ⟨Input⟩ and ⟨/Input⟩ are input parameters for the oracle method. For the example, a *Borrower* is needed as input.
– Output: The information between ⟨Output⟩ and ⟨/Output⟩ is the type that returns the oracle method. For the example, this state machine returns an *Integer* value.
– Variable: The information between ⟨Variable⟩ and ⟨/Variable⟩ are variables that are used in the state machine. For the example, it must be known how to obtain *daysBorrowed, allowDays, suspDays*, and other variables used in Fig. 10.
– State: The information between ⟨State⟩ and ⟨/State⟩ is used to obtain the initial state. For the example, the information about the current state for the *Borrower b* is obtained using the function *b.getState ().toString ()*.

5.2. Transformation to obtain oracle methods

The implementation of the oracle method is performed through a model-to-text transformation (M2T). As input, it takes a UML state machine stereotyped as ≪TestOracle≫ and returns a class in the same language as the SUT with two oracle methods:

1. A method to return the result: this method calculates the pre-state and, depending on its value, evaluates the guard and returns the effect corresponding to this guard.
2. A method to return the post-state: this method returns a string representing the post-state.

MOFScript Transformations: Listing 2 shows an excerpt from the *Model2Text* transformation using MOFScript language. The main rule is applied to UML state machines stereotyped ≪TestCondition≫ (lines 1–4). This main rule calls the mapClass rule (lines 5–9) that creates the text file, the header of the oracle class and calls the rules *addOracleState ()* and *addOracleResult ()*.

As depicted in Fig. 12, the result of executing the MOFScript transformation is a Class with two methods. Taking the UML state machine for the Library System in Fig. 14 as example, the Java method presented in Listing 4 is obtained from the *addOracleState ()* rule.

The *addOracleState ()* rule (lines 10–36 in Listing 2) generates the method that returns the expected post-state. First, the transformation loads the information stored in the UML Comment stereotyped ≪TestPrecondition≫ (lines 13–16); as a result, lines 1–11 in Listing 4 are generated. The transformation takes each stereotyped state as a precondition (lines 17 and 18 in Listing 2) and, for each guard, generates the corresponding text code. Listing 4 shows the generated code (lines 12–29).

Listing 3 shows the *addOracleResult ()* rule. This rule is similar to *addOracleState ()* but generates the method that returns the expected result, depicted as an effect in the state machine. For each state stereotyped as ≪Precondition≫, the corresponding effect is generated (lines 14 to 24) as a text code result for the method. Listing 5 shows the java method generated to obtain the oracle result.

5.3. Actual versus expected result comparison

Listing 1 showed the JUnit code without the oracle for the UML sequence diagram in Fig. 8. The MOFScript transformation from the test model to xUnit code also needs to be changed. Now, the transformation has taken into account that there is a UML state machine stereotyped ≪TestOracle≫ related to the UML sequence diagram.

Listing 6 shows the new code generated: now, only the test input needs to be recovered from the datapool (lines 9 and 10). Due to the fact that the state machine in Fig. 14 is associated with the *return book* functionality, the transformation takes the data from the UML comment attached to the state machine. This input is *Borrowing b*, so this value is also returned by the dataPool as *bds_oracleInput* (see line 11). Now, the expected state and the amount are requested from the oracle methods. Line 16 requests the expected state and line 17 requests the expected result (amount), passing as parameter *bds_oracleInput* in both cases.

Each operation is called in the SUT obtaining the actual result (lines 20 and 26). Finally, the comparison between the expected

```
1.  uml.StateMachine::main(){
2.     if(self.hasStereotype("testCondition")){
3.       self.mapClass()}
4.  }

5.  uml.StateMachine::mapClass() {
6.       ...
7.     self.addOracleState()
8.     self.addOracleResult()
9.  }


10. uml.StateMachine::addOracleState(){
11.
12.    self.ownedElement->forEach(r:uml.Region){
13.      r.ownedComment->forEach(cm:uml.Comment|cm.hasStereotype("TestPrecondition")){
14.      //Load Input and Output parameters, Variables and States specified in the Comment
15.         ...
16.      }
17.    r.ownedElement ->forEach(s:uml.State){
18.      if(s.hasStereotype("precondition")){
19.        <%\n if (prestate.equals("%> s.name <%")){ %>
20.        r.ownedElement -> forEach (t:uml.Transition){
21.              if (t.source == s) {
22.                t.ownedRule -> forEach (g:uml.Constraint){
23.                  g.ownedElement -> forEach (o:uml.OpaqueExpression){
24.                    String body = o.body
25.                    <%\n if ( %> body.substring(1, body.size()-1) <%) { %>
26.                     <% poststate = "%> t.target.name <%"; %>
27.                  }
28.                }
29.              }
30.        }
31.     }
32.    }
33.    <%\n return poststate;%>
34.    <%\n  } %>
35.   }
36. } //Oracle State
```

Listing 2. MOFScript transformation.

```
1.  uml.StateMachine::addOracleResult(){
2.
3.     self.ownedElement->forEach(r:uml.Region){
4.       r.ownedComment->forEach(cm:uml.Comment|cm.hasStereotype("TestPrecondition")){
5.          //Load Input and Output parameters, Variables and States specified in the Comment
6.          ...
7.       }
8.       r.ownedElement ->forEach(s:uml.State){
9.       if(s.hasStereotype("precondition")){
10.        <%\n if (prestate.equals("%> s.name <%")){ %>
11.        r.ownedElement -> forEach (t:uml.Transition){
12.             if (t.source == s) {
13.               t.ownedRule -> forEach (g:uml.Constraint){
14.                 g.ownedElement -> forEach (o:uml.OpaqueExpression){
15.                   String body = o.body
16.                   <%\n if ( %> body.substring(1, body.size()-1) <%) { %>
17.                 }
18.               }
19.               String effect = t.effect.body
20.               <%\n result = %> effect.substring(1, effect.size()-1)  <%; %>
21.               <%\n } %>
22.             }
23.           }
24.       }
25.     }
26.    }
27.    <%\n return result = null;%>
28.    <%\n   } %>
29.  }
30. } //OracleResult
```

Listing 3. Rule addOracleResult.

```
1.   public String BorrowerStates_oracle_state ( Borrowing r ) {
2.     String poststate = "";
3.     int daysBorrowed = r.getBorrowDays();
4.     Borrower b = r.getBorrower();
5.     BookCopy bc = r.getBook();
6.     Book bk = bc.getBook();
7.     int allowDays = bk.getDays();
8.     int suspDays = bk.getSuspDays();
9.     int price = bk.getPrice();
10.    int fine = bk.getFine();
11.    String prestate = b.getState().toString();
12.    if (prestate.equals("normal")){
13.       if ( daysBorrowed>allowDays ) {
14.           poststate = "delayed";
15.       }
16.       if ( daysBorrowed<=allowDays ) {
17.           poststate = "normal";
18.       }
19.    }
20.    if (prestate.equals("delayed")){
21.       if ( daysBorrowed<=allowDays ) {
22.            poststate = "normal";
23.       }
24.       if ( daysBorrowed>suspDays) {
25.            poststate = "suspended";
26.       }
27.       if ( daysBorrowed>allowDays) {
28.            poststate = "delayed";
29.       }
30.    }
31.    return poststate;
32. }
```

Listing 4. Oracle state automatically generated.

and actual results is done using the *Assertion* sentence in JUnit (lines 23 and 29), as depicted in Fig. 11. The tester could use the functionalities that bring the xUnit framework to manage defects and re-execute the test cases. If the defects is due to an error in the oracle procedure, the tester or the designer only need change the UML state machine, execute again the model transformation that obtains the test oracle procedure and re-execute the test cases again using the xUnit framework.

6. Case study: Monica Mobile system

Empirical studies are necessary in real environments if we are to deepen the knowledge and validity of the application of Software Engineering practices [21]. A *Case study* explores a phenomenon within its real context, especially when the boundaries between phenomenon and context are not clearly evident [22]. According to Runeson and Höst [21], the steps to follow in a case study process are: (i) Case Study design and planning, (ii) Preparation of data collection (definition of procedures and protocols for data collection), (iii) Collecting evidence, (iv) Analysis of collected data, and (v) Reporting.

This section explains how each step was carried on for the Monica Mobile system.

Case study design: The objective of the case study is to evaluate the application of a model-driven testing framework to automate the generation of complete test cases, what includes the corresponding oracles, in order to improve the testing process in model-driven environments. The case study analyses Monica Mobile, a system for monitoring and controlling the conditions of the load transported in a truck developed. This system has been developed by the Ser & Practices company. It was selected because: (1) during its development, UML diagrams and tools were used; (2) Ser & Practices is a spin-off of Bari University (Italy) and, thus, it is really interested in the integration of the best research results in their software projects. The data collection is obtained using tool instrumentation and the units of analysis were selected to fit the specific objectives of the case study. The main sources of data were collected interviews and analysis of code and of technical documentation.

Case study protocol: *Action-research* is the manner in which to meet the required conditions, to learn from our own experiences and make them accessible to others [23]. It is a qualitative research method that brings theory and practice, and researchers and practitioners together to solve a problem [24], and that we have

```
1.  public  Integer BorrowerStates_oracle_result ( Borrowing r ) {
2.   Integer  result = null;
3.   int daysBorrowed = r.getBorrowDays();
4.   Borrower b = r.getBorrower();
5.   BookCopy bc = r.getBook();
6.   Book bk = bc.getBook();
7.   int allowDays = bk.getDays();
8.   int suspDays = bk.getSuspDays();
9.   int price = bk.getPrice();
10.  int fine = bk.getFine();
11.  String prestate = b.getState().toString();
12.  if (prestate.equals("normal")){
13.      if ( daysBorrowed>allowDays ) {
14.            result = price;
15.       }
16.      if ( daysBorrowed<=allowDays ) {
17.                 result = 0;
18.      }
19.  }
20.  if (prestate.equals("delayed")){
21.      if ( daysBorrowed<=allowDays ) {
22.            result = 0;
23.      }
24.      if ( daysBorrowed>suspDays) {
25.            result = price + (fine*daysBorrowed)*3;
26.      }
27.      if ( daysBorrowed>allowDays) {
28.            result = price + fine*daysBorrowed;
29.      }
30.  }
31.  return result;
32. }
```

Listing 5. Oracle result automatically generated.

```
1.  public void testReturnBook_test(){
2.
3.   //Ask the data from DataPool
4.   Vector<ValueSet> v = returnBook_Datapool.getreturnBook_test();
5.   BorrowerStates oracle = null;
6.   for(ValueSet vs:v){
7.
8.     //Get the data for test case input
9.     String clientds_returnBook = (String) vs.getValue("clientds_returnBook");
10.    String copyds_returnBook = (String) vs.getValue("copyds_returnBook");
11.
12.    //Get the data for oracle input
13.    Borrowing bds_oracleInput = (Borrowing) vs.getValue("bds_oracleInput");
14.
15.    //Get the expected result from the oracle
16.    String stateds_returnBook = oracle.BorrowerStates_oracle_state(bds_oracleInput);
17.    Integer amountds_confirmReturn = oracle.BorrowerStates_oracle_result(bds_oracleInput);
18.
19.    //Call returnBook in SUT
20.    String state = returnInterface.returnBook( clientds_returnBook, copyds_returnBook);
21.
22.    //Validation Action for returnBook
23.    assertTrue(stateds_returnBook.equals(state));
24.
25.    //Call confirmReturn in SUT
26.    Integer amount = returnInterface.confirmReturn();
27.
28.    //Validation Action for confirmReturn
29.    assertTrue(amountds_confirmReturn.equals(amount));
30.  }
31. }
```

Listing 6. JUnit automatically generated test code with oracle.

	Iteration 1	Iteration 2
Researcher	Alarcos Group SerLab Group	Alarcos Group SerLab Group
Researched Object	Monica Mobile	Monica Mobile
Reference group	SER&Practices developers	SER&Practices developers
Stakeholder	SER&Practices	SER&Practices
Subject	Methodology for model-driven testing	Methodology for model-driven testing with oracle automation
Place	Bari, Italy	Bari, Italy
Year	2010	2011

Fig. 16. Iterations in the case study.

applied in several research projects since more than 10 years ago (we documented one of our first experiences in [25]).

The roles of Action-Research are: Researcher (person or group of people who actively carry out the research process), Researched object (the problem to solve), Critical reference group (group on which research is performed in as much as it has a problem that needs to be solved), and the Stakeholder (anyone that can benefit from the research but does not directly participate in it) [26]. For this research, two main iterations were conducted (Fig. 16). The first iteration applies the model-driven testing framework explained in Section 4. The second iteration adds the oracle automation proposal. The reference group was composed by the development team of Monica Mobile and the direction board of Ser & Practices.

Collecting evidence: Data is collected mainly through interviews, analysis of design models and study of the application code. The test models and test cases were automatically obtained using model transformations. The Ser & Practices company gave feedback on the results.

Monica Mobile is a sensor system. Along the truck there are several nodes, and each node has up to five sensors to measure the temperature, pressure, humidity and concentrations of CH4 and CH6 gases. Via bluetooth, the nodes periodically send the sensor data to a smart phone in front of the driver (Fig. 17).

The icons on the screen change their colour (green, yellow, red) depending on the information received: when the measurements are within their limits, the icons are green; if an alert is received from a specific sensor, the corresponding icon turns to red; if a message of no alert is received after an alert, then it goes to yellow. When a normal measurement is received, it goes back from yellow to green. The data are also sent via GPRS to a remote central machine, although this functionality is beyond the scope of this paper.

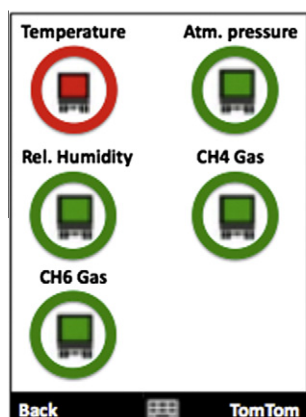


Fig. 17. Sensor interface in Monica Mobile.

The functionality in charge of processing messages is Process Packet. It is called each time a package is sent from the sensor system. The package consists of a byte string of variable length and has information about measurements in each sensor, as well as one byte (the 19th) that identifies the package type. There are four types:

- Announce (code 0×01): this is the first package sent by the sensor system when it is turned on. It contains information about the types of the sensors, the number of nodes and the numbers of sensors in each node for a truck.
- DataSens (code 0×03): this type of package is sent periodically and contains information about the measurements for each sensor and node. The measurements in this package are within the tolerable limits.
- AlertSens (code 0×05): this type of package is sent when one or more sensors are in alert. It contains information about the node, the sensor and the anomalous measure.
- NoAlertSens (code 0×07): this type of package is sent when one or more sensors that were in alert are now in normal situation.

Fig. 18 describes an excerpt from the Process Packet functionality. Due to space limitations, not all the types of packages are shown, only the DataSens type. In this case, the data for each sensor are updated and the interface is refreshed by a call to the RefreshSensorForm method.

The behaviour of the responses to sensor measurements is easily modelled as a state machine: as noted above, they can be green, yellow or red, depending on the sequence of the packages received. Fig. 19 shows the corresponding UML state machine.

Guards in each transition in Fig. 19 take into account the package type that was sent. The effect compares the image that is shown in the interface (Fig. 17) with the expected one. Fig. 19 also shows the TestPrecondition where the Input, Output, Variables and State for the oracle are defined.

Following the steps defined in Section 3 to obtain the oracle method using a model-to-text transformation, the UML state machine for the Sensor oracle was transformed to C# code (which is the code in which Monica Mobile was developed). The MOFScript transformation generates the SensorOracle class with two methods. Listing 7 shows one of them, the Sensor_oracle_result method, which returns whether the image from each sensor is the expected one. Also, the MOFScript transformation generates the Sensor_oracle_state method (not shown here), which returns the resulting state for the sensor after the package is received.

In general, the terminology to produce the oracle method header is: state machine name + "_oracle" + "_XXX", where XXX can be state or result. Then, for the *Sensor* state machine (Fig. 19), the methods obtained are: *Sensor_oracle_state* and *Sensor_oracle_result.*

Now, the oracle methods are ready to be used. The Model-Driven Framework will be applied with the automated test oracles to test Monica Mobile (see Fig. 5). The first step is to execute the QVT transformation to the sequence diagram in Fig. 18 to obtain the test case procedure (see Fig. 20). ProcessPacket functionality does not return a result. Then, the comparison of the expected result with the obtained one (which is automatically generated with the ≪Validation Action≫) is not possible. In this case, the tester needs to add a UML Comment stereotyped ≪Test Postcondition≫ to add information about the oracle. Fig. 20 shows the UML comment attached to the ProcessPacket functionality. It iterates for all sensors in each node; then, it calls the oracle methods in the

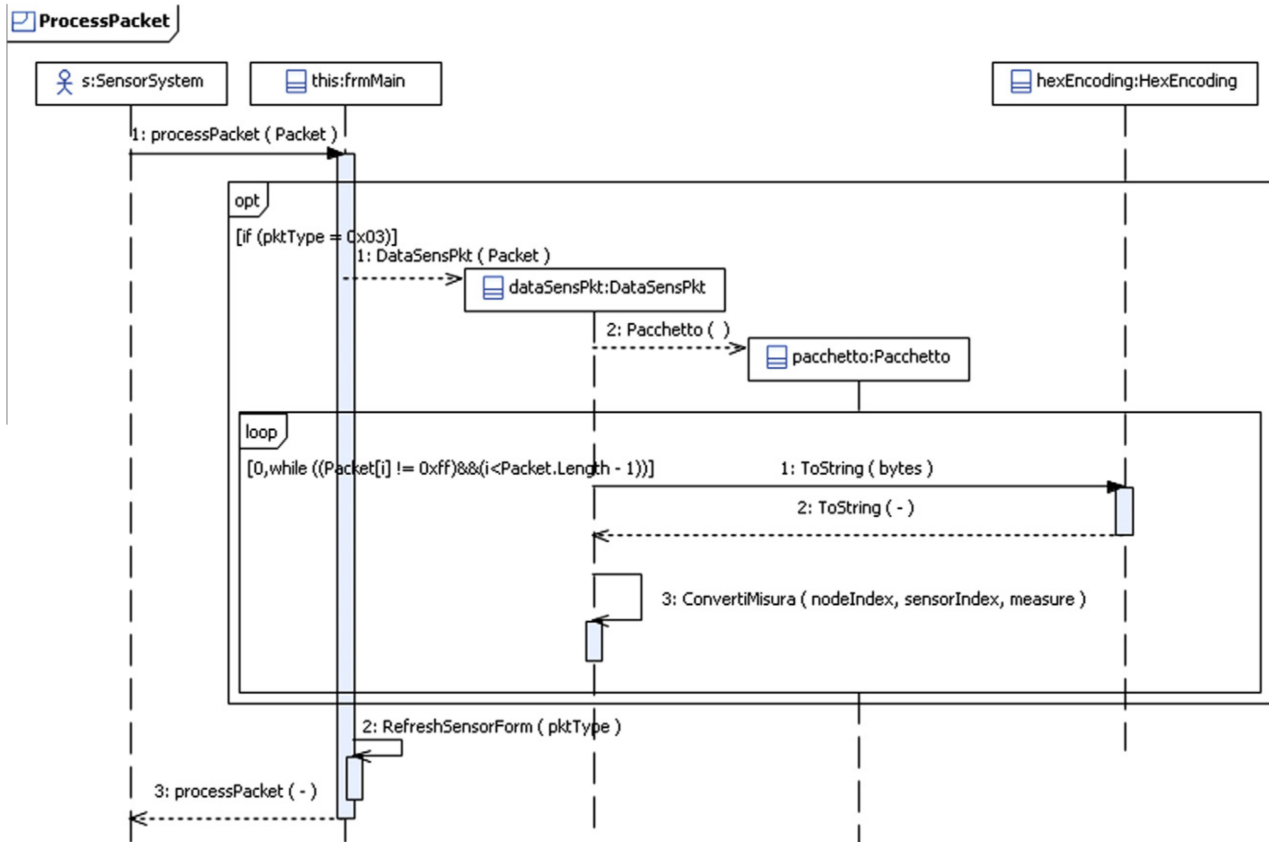


Fig. 18. Excerpt of process packet functionality.

SensorOracle class and uses the Assert function of NUnit to check whether the test passes or fails. This comment is copied to the ProcessPacket_test in the QVT transformation as a UML Comment. When the MOFScript transformation is later executed, the information in the UML Comment is pasted at the end of the NUnit test case.

To obtain the NUnit code to test ProcessPacket, the MOFScript transformation that has the test procedure in Fig. 20 as input is executed.

Listing 8 shows the code of the automatically generated test. The meaning of the test case in Fig. 8 is the following:

- Obtain test data from dataPool: Lines 1–6. The dataPool returns an array with all the packets stored for this test case (test inputs).
- Execute the functionality in SUT: The processPacket is executed in line 8, with the test data *Packet* as parameter.
- Obtaining the test case verdict: In this case, the verdict is calculated using the information in the UML Comment and appears in lines 9–20.

Analysis of collected data: The number of test cases fully automated, the number of test cases obtained with the expected result automatically generated and the number of defects obtained were studied, combined with qualitative analysis.

ProcessPacket is one of the main functionalities in Monica Mobile and must be automated, since manual testing is quite time-consuming. New versions are planned and regression testing for them must ensure the product quality in a reasonable time: for example, new sensors will be incorporated to measure the radioactivity of the truck load. The functionality process packet handles four types of packages: *announce*, *dataSens*, *alert* and *no alert*. Each type of message corresponds to one scenario for the functionality. The same oracle method used to test the *dataSens* packet was used to test the other types of packets. For the types of packet announce, *alert* and *no alert*, we automatically generated the test case procedure following the same steps in the model-driven framework that we used for the *dataSens* packet. Once the test procedure and the test oracle method were obtained, these test cases could be executed with several different input data.

Table 1 shows, for each type of packet, the number of different test inputs used to test it. Announce, the first message that arrives to the system, was tested with five different packets as input. One test case procedure was generated for this type of packet. Finally, for each test input, the test procedure was executed, calling the oracle methods to calculate the expected result. For each test input, a complete test case was executed, composed of the test input, test procedure, expected result and the validation of the expected versus the actual result obtained from the SUT. The same occurs with the other types of packets.

Table 2 summarises the results obtained by automating the functionality Process Packet in Monica Mobile system. Two oracle methods were automatically generated from the UML state machine; four scenarios were tested, one for each type of packet. For each scenario, represented by a sequence diagram, first, the test model was automatically generated and then, the code for the test case procedure with the oracle method was obtained. The four test case procedures were executed in the system with 85 different input data. This number was obtained by adding the test cases executed for each type of packet (see Table 1). Finally, the number of defects found in the system as result of executing the test cases were 14.

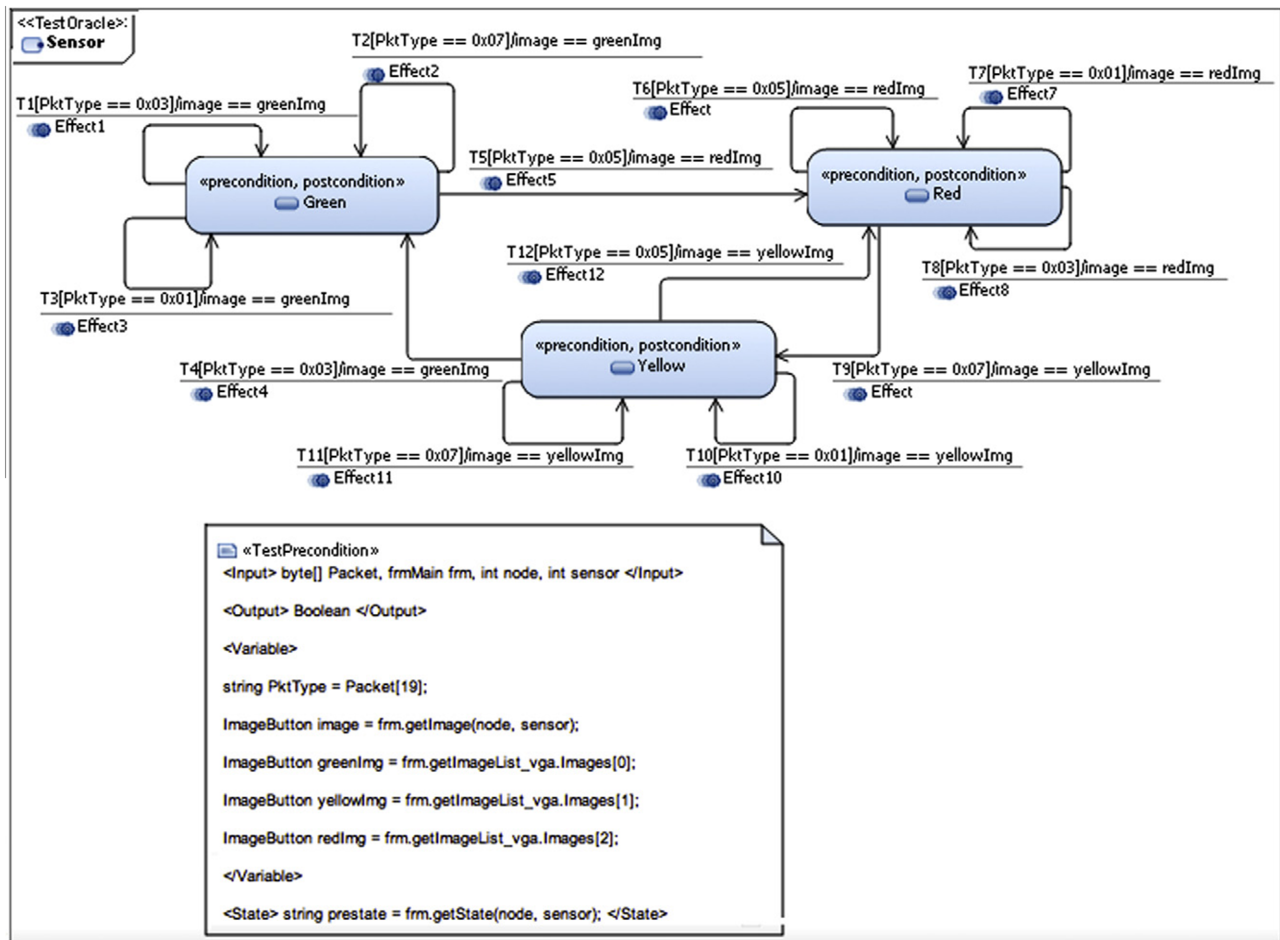


Fig. 19. UML state machine for a sensor.

Without using the automated oracle procedure, the testers need to calculate the expected result for each data input and to store it in the dataPool. This is a manual process that takes time and is error prone. Moreover, in this case study, the developers of Monica Mobile have no documented test cases, so being very difficult the comparison with our approach.

Table 3 shows the time involved in executing the model transformations to obtain the test cases. The transformation that obtains the test model is a model-to-model transformation implemented using mediniQVT language and takes 146 ms. The transformation to obtain the test case procedure is a model-to-text transformation implemented using MOFScript and takes 120 ms. The oracle procedure is obtained using a model-to-text transformation that takes a UML state machine and returns the oracle method and takes 93 ms. In summary, transformations take less than one second.

If the UML state machine used to obtain the oracle procedure has an error or if the UML models used to obtain the test cases changes, the transformation could be re-run and the test model, test code and oracle procedure is obtained again.

In summary, with the addition of the oracle information to the test case scenario, we obtained an executable test case that can be run using the NUnit framework each time that a new version of Monica Mobile is released. This regression testing ensures that the functionality does not decrease in quality when changes to the application are made.

Obviously, this approach is not the "magical method", valid for all systems that, for Bertolino [10], is the "ideal oracle". But it constitutes an important advance for automating the test oracle generation in the emergent model-driven paradigm. In particular, it is specially applicable to systems or functionalities whose behaviour is suitable to be described with UML state machines. As well as the source code of a program may contain errors, the model itself can also be erroneous: but, since test models are automatically obtained from design models and test code is automatically obtained from test models through model transformation, the methodology keeps the traceability between design models, test models and test code, preserving the correspondence between abstraction levels. This traceability also provides maintainability, since when a change occurs and the design models are modified, transformations are re-executed and test models and test code are automatically obtained again.

7. Related works

Related works can be divided into three categories: oracle automation, use of state machines for testing and model-driven testing.

Oracle automation: One of the most important tasks in software testing is the definition of the oracle. In the last decade (2001 and 2009), two surveys analysing the oracle problem in depth were published [12,19], highlighting in both cases the challenges inherent in automation.The main approaches in oracle automation are summarised below:

Richardson [27] proposes several levels of test oracles, one for checking the range of the values of variables, and the other using specification-based testing. Memon et al. [28] developed an automated GUI test oracle, using formal models that represent the

```
1.   public Boolean Sensor_oracle_result(byte[] Packet, frmMain frm, int node, int
    sensor){
2.              byte PktType = Packet[19];
3.              ImageButton image = frm.getImage(node, sensor);
4.              ImageButton greenImg = frm.getImageList_vga.Images[0];
5.              ImageButton yellowImg = frm.getImageList_vga.Images[1];
6.              ImageButton redImg = frm.getImageList_vga.Images[2];
7.              string prestate = frm.getState(node, sensor);
8.              if (prestate.Equals("Green")){
9.                  if (PktType == 0x01){
10.                     result = (image == greenImg);}
11.                 if (PktType == 0x03){
12.                     result = (image == greenImg);}
13.                 if (PktType == 0x07){
14.                     result = (image == greenImg);}
15.                 if (PktType == 0x05){
16.                     result = (image == redImg);}
17.             }
18.             if (prestate.Equals("Yellow")){
19.                 if (PktType == 0x05){
20.                     result = (image == yellowImg);}
21.                 if (PktType == 0x01){
22.                     result = (image == yellowImg));}
23.                 if (PktType == 0x07){
24.                     result = (image == yellowImg);}
25.                 if (PktType == 0x03){
26.                     result = (image == greenImg);}
27.             }
28.             if (prestate.Equals("Red")){
29.                 if (PktType == 0x03){
30.                     result = (image == redImg);}
31.                 if (PktType == 0x01){
32.                     result = (image == redImg);}
33.                 if (PktType == 0x05){
34.                     result = (image == redImg);}
35.                 if (PktType == 0x07){
36.                     result = (image == yellowImg);}
37.             }
38.             return result;
39.  }
```

Listing 7. C# code generated for the SensorOracle.

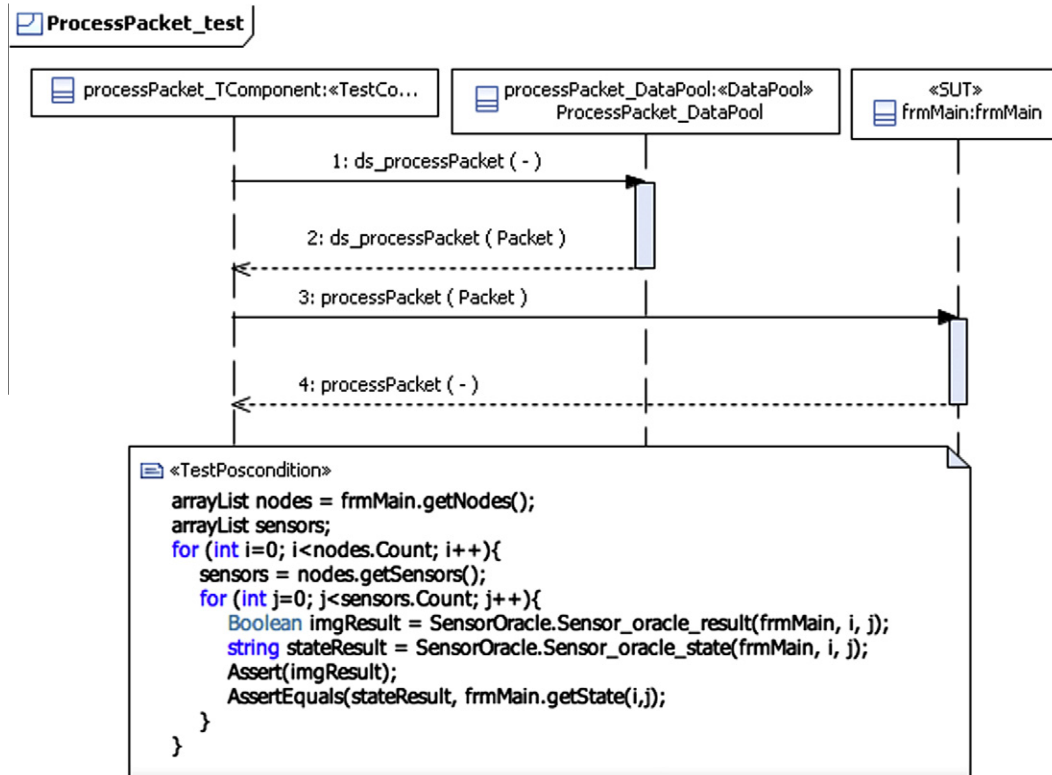


Fig. 20. Test behaviour for process packet functionality.

```
1.  public void ProcessPacket_test() {
2.  //Call dataSelector in DataPool
3.  List<ValueSet> v = processPacket_dataPool.ds_ProcessPacket();
4.  for (int i = 0; i < v.Count; i++){
5.      ValueSet vs = v[i];
6.      byte[] Packet = (byte[])vs.getValue("Packet");
7.      //Call SUT
8.      frmMain.processPacket(ref Packet);
9.      //UML Comment
10.     arrayList nodes = frmMain.getNodes();
11.     arrayList sensors;
12.     for (int i=0; i<nodes.Count; i++){
13.        sensors = nodes.getSensors();
14.        for (int j=0; j<sensors.Count; j++){
15.          Boolean imgResult = SensorOracle.Sensor_oracle_result(Packet, frmMain, i, j);
16.          string stateResult = SensorOracle.Sensor_oracle_state(Packet, frmMain, i, j);
17.          Assert(imgResult);
18.          AssertEquals(stateResult, frmMain.getState(i,j);
19.        }
20.     }
21.  }
22. }
```

Listing 8. NUnit code to test ProcessPacket functionality.

Table 1
Types of process packet with expected results generated for Monica Mobile.

Type of packet	Test input	Test case procedures	Test cases obtained
Announce	5	1	5
DataSens	40	1	40
Alert	20	1	20
No alert	20	1	20

Table 2
Test elements generated for Monica Mobile.

Test oracle methods	2
Scenarios tested	4
Test case procedures	4
Test cases with expected results	85
Defect founds	14

Table 3
Time involved in executing model transformations.

Model-to-model transformation to obtain test model	146 ms
Model-to-text transformation to obtain test case procedure code	120 ms
Model-to-text transformation to obtain test oracle procedure code	93 ms

GUI. The expected state is derived from the formal model and the test case actions. Xie and Memon [20] defined a technique to declaratively specify differents types of automated GUI test oracles and used it in a controlled experiment.

The approach found in the work by Manolache and Kourie [29] for oracle automation is based on various implementations of a program that implements the same functionalities (N-versions diverse systems) and are used later as test oracles. Last et al. [30] uses an input–output analysis of execution data automated by the IFN (Info-Fuzzy Network) methodology of data mining. This approach uses previous versions of the SUT to obtain the expected outputs. Vanmali et al. [31] use an artificial neural network as an automated oracle that is trained on a set of test cases applied to the original version of the system. The network training is based on the "black-box" approach, since only inputs and outputs of the system are presented to the algorithm, which is later used as an artificial oracle.

Shahamiri et al. [32] also use artificial neural networks as an automated oracle to test decision-making structures (using nested if-then-else structures). The network is modelled using a training dataset generated based on software specifications and domain expert knowledge.

In summary, most of the existing approaches that automate test oracles uses previous versions of the SUT and are thus only applicable in regression testing. Other proposals require formal models that represent the SUT. In our work, a UML metamodel is used to describe SUT behaviour, UML state machines are used to oracle representation, augmented using an UML Profile. This approach can be used for testing during development and also in regression testing. Furthermore, UML is a widely accepted standardised notation that allows designers and testers to share the same notation.

Later works have made proposals to solve this problem using artificial neural networks [33,34], metamorphism [35,36], formal approaches [37], etc.

State Machines applied to testing: Using finite state machines for test data generation is not new. Finite state machines, extended state machines and, recently, UML state machines are used for testing. The work by Lee and Yannakakis [38] still is a good reference to have a good overview of this area.

In general the literature uses state machines for testing for two proposes: (i) to generate test cases (ii) to generate the data input and, in some cases, data output as well. These proposals are summarised below:

- State machines to generate test cases: These approaches are based on deciding how the transitions can be tested and several coverage criteria were defined. Chow [39] defines the W-method for finite state machines, which generates a transition tree using full tree paths as testing coverage. This was later used for state machines by Binder [40] who called it a round-trip path. Offut and Abdurazik [41] and Offut et al. [42] define criteria for the transition coverage level for UML state machines (transition coverage, full predicate coverage, transition-pair coverage and complete sequence). Kim et al. [43] transformed UML state machines into an Extended Finite State Machine to use conventional control flow analyses, obtaining test cases from paths. Kim et al. [44] generates test sequences from state charts for concurrent systems in Java. Mouchawrab et al. [45] use round-trip paths testing applied to UML state machines

for class clusters as a test strategy and compare it with structural testing through a controlled experiment.
- State machines to generate test data: These works are related to obtaining test data for test cases. The test data input includes the initial state and the values for the variables used in the guards and effects in the transitions. One of the first methods to derive test data using finite state machines was presented by Chow [39] and was later applied to a UML state machine by Lliuying and Zhichang [46]. Hong et al. [47] and Offut et al. [42] present general criteria to obtain test input from state-based specifications. Chevalley and Thevenod [48] adapt a probabilistic method, called statistical functional testing, for the generation of test cases from UML state diagrams, using transition coverage as the testing criterion. Briand et al. [49] use operation contracts and transition guards written with the Object Constraint Language (OCL) to derive input and output test data.

Existent approaches for state machines applied to testing focus mainly on test case procedures and test data, but none of them generates procedures to obtain the decision about the test oracle, i.e, whether the expected and actual results are equal as our research does.

Model-driven testing: Many proposals for model-based testing exist [15,14], but few of them focus on automated test model generation using model transformations. Dai [50] describes a series of ideas and concepts to derive UML-TP models from UML models, which are the basis for a future model-based testing methodology. Test models can be transformed either directly to test code or to a platform specific test design model (PST). After each transformation step, the test design model can be refined and enriched with specific test properties. However, to the best of our knowledge, this interesting proposal has no practical implementation for any tool. Baker et al. [51] define test models using UML-TP. Transformations are done manually instead of using a transformation language. Naslavsky [52] uses model transformation traceability techniques to create relationships among model-based testing artefacts during the test generation process. They adapt a model-based control flow model, which they use to generate test cases from sequence diagrams. They adapt a test hierarchy model and use it to describe a hierarchy of test support creation and persistence of relationships among these models. Although they use a sequence diagram (as does this proposal) to derive the test cases, they do not use it to describe test case behaviour. Javed [53] generates unit test cases based on sequence diagrams. The sequence diagram is automatically transformed into a unit test case model, using a prototype tool based on the Tefkat transformation tool and MOFScript for model transformation. This work is quite similar to ours, but they do not use the UML-TP. We generate the unit test case in two steps while they use only one. We think that using an intermediate model with UML-TP as PIT is more appropriate for an MDE approach, due that the same test model could be used to test different platforms (i.e. to test differents PSM), later the test code is generated for each specific platform to execute the test cases.

8. Conclusions

In this paper, we proposed an approach allowing the test oracle generation, automating two important steps in the test oracle: the expected output and its comparison with the actual output. We described an automated testing framework to support the test case automation. This framework takes the models that describe the system as input, using UML notation and derives from them the test model and then the test code, following a model-driven approach. As a result, a complete executable test case is obtained, with the following information:

- Parametrized input test data: The tester stores the input test data in the dataPool and the test case automatically retrieves it. The operation in the dataPool returns all the test input for the test case and the test case is executed for each set of inputs.
- Obtaining the expected result in an automated fashion: This part of the test oracle is automatically obtained from the UML state machine. As a result, two methods are obtained to generate the expected result and the expected state.
- Test case procedure: The test case procedure is automatically generated from the sequence diagram that represents the functionality to test and uses the information about the test input and the expected result. The test case procedure focuses on functional testing. Then, the interaction between the system and the actors is tested. As result, a method is obtained in xUnit that calls the test input in the dataPool, calls the operations to test in the SUT with the test input, uses the oracle methods to calculate the expected result and finally, validates whether the expected result is equal to the actual one using the Assert sentence in xUnit.

The approach has been applied to a Library system coded in Java language and an industrial project coded in C# language.

We are currently generating test cases at functional level. In the future, we will extend our framework to obtain test cases at integration and unit level. Current work includes adding functional test cycles to the framework. By using the information in the UML State Machine, we can derive the order in which the functionalities must be called to be tested, taking into account the previous and post-state of the SUT. Future work includes the extension of our approach to handle complex types in UML state machine likes data structures or collections, to obtain the test oracle from it.

Acknowledgments

This research was financed by the projects: DIMITRI (Ministerio de Ciencia e Innovacin, grant TRA2009_0131) and the project PEGASO/MAGO (TIN2009-13718-C02-01) from MICINN and FEDER. Přez has a doctoral Grant from JCCM, Orden de 13-11-2008.

References

[1] S.R. Dalal, A. Jain, N. Karunanithi, J.M. Leaton, C.M. Lott, G.C. Patton, B.M. Horowitz, Model-based testing in practice, in: Proceedings of the International Conference on Software Engineering, 1999, pp. 285–294.
[2] IEEE, IEEE Standard Glossary of Software Engineering Terminology, Tech. rep., IEEE, 1990.
[3] T. Mens, P. Van Corp, A taxonomy of model transformation, Electronic Notes in Theoretical Computer Sciences 152 (2006) 125–142.
[4] B. Pérez Lamancha, P. Reales Mateo, I. García, M. Polo Usaola, M. Piattini, Automated model-based testing using the uml testing profile and qvt, in: International Workshop on Model-Driven Engineering, Verification and Validation, ACM, Denver, Colorado, 2009, pp. 1–10.
[5] B. Pérez Lamancha, M. Polo, M. Piattini, An automated model-driven testing framework for model-driven development and software product lines, in: International Conference on Evaluation of Novel Approaches to Software Engineering, SciTePress, Athens, Greece, 2010, pp. 112–121.
[6] B. Pérez Lamancha, P. Reales Mateo, M. Polo Usaola, D. Caivano, Model-driven testing: transformations from test models to test code, in: International Conference on Evaluation of Novel Approaches to Software Engineering, SciTePress, Beijing, China, 2011.
[7] OMG, Unified Modeling Language, Superstructure Specification, Tech. Rep. formal/2007-11-02, OMG, 2007.
[8] OMG, UML Testing Profile Version 1.0, Tech. Rep. formal/05-07-07, OMG, July 2005 2005.
[9] B. Beizier, Black-Box Testing: Techniques for Functional Testing of Software and Systems, John Wiley & Sons, 1995.
[10] A. Bertolino, Software Testing Research: Achievements, Challenges, Dreams, in: Workshop on the Future of Software Engineering, IEEE Computer Society, 2007, pp. 85–103.
[11] A. Abran, P. Bourque, SWEBOK: Guide to the Software Engineering Body of Knowledge, IEEE Computer Society, 2004.
[12] L. Baresi, M. Young, Test Oracles, Tech. Rep. Technical Report CIS-TR01 -02, Dept. of Computer and Information Science, Univ. of Oregon, 2001.

[13] H. Gomaa, Designing concurrent, distributed, and real-time applications with UML, in: International Conference on Software Engineering, ACM, 2006, pp. 1059–1060.
[14] A.C. Dias Neto, R. Subramanyan, M. Vieira, G.H. Travassos, A survey on model-based testing approaches: a systematic review, in: Workshop on Empirical Assessment of Software Engineering Languages and Technologies, ACM, 2007, pp. 31–36.
[15] M. Prasanna, S. Sivanandam, R. Venkatesan, R. Sundarrajan, A survey on automatic test case generation, Academic Open Internet Journal 15 (2005) 1–5.
[16] OMG, Mof Query/View/Transformation Specification, Tech. rep., OMG, 2007.
[17] OMG, Mof Model to Text Transformation Language, Tech. Rep. formal/2008-01-16, OMG, 2008.
[18] K. Beck, Kent Beck's Guide to Better Smalltalk: A Sorted Collection, vol. 14, Cambridge Univ. Pr., 1999.
[19] S. Shahamiri, W. Kadir, S. Mohd-Hashim, A comparative study on automated software test oracle methods, in: International Conference on Software Engineering Advances, IEEE, 2009, pp. 140–145.
[20] Q. Xie, A. Memon, Designing and comparing automated test oracles for gui-based software applications, ACM Transactions on Software Engineering and Methodology (TOSEM) 16 (1) (2007) 4–24.
[21] P. Runeson, M. Höst, Guidelines for conducting and reporting case study research in software engineering, Empirical Software Engineering 14 (2) (2009) 131–164.
[22] R. Yin, Case study research: design and methods, Applied Social Research Methods Series 5 (2003).
[23] R. McTaggart, Principles for participatory action research, Adult Education Quarterly 41 (3) (1991) 168.
[24] D. Avison, F. Lau, M. Myers, P. Nielsen, Action research, Communications of the ACM 42 (1) (1999) 94–97.
[25] M. Polo, M. Piattini, F. Ruiz, Using a qualitative research method for building a software maintenance methodology, Software: Practice and Experience 32 (13) (2002) 1239–1260.
[26] Y. Wadsworth, What is participatory action research, Action Research International 2 (1998) 1–18.
[27] D. Richardson, Taos: testing with analysis and oracle support, in: International Symposium on Software Testing and Analysis, ACM, 1994, pp. 138–153.
[28] A. Memon, M. Pollack, M. Soffa, Automated test oracles for guis, ACM SIGSOFT Software Engineering Notes 25 (6) (2000) 30–39.
[29] L. Manolache, D. Kourie, Software testing using model programs, Software: Practice and Experience 31 (13) (2001) 1211–1236.
[30] M. Last, M. Friedman, A. Kandel, The data mining approach to automated software testing, in: International Conference on Knowledge Discovery and Data Mining, ACM, 2003, pp. 388–396.
[31] M. Vanmali, M. Last, A. Kandel, Using a neural network in the software testing process, International Journal of Intelligent Systems 17 (1) (2002) 45–62.
[32] S. Shahamiri, W. Kadir, S. Ibrahim, An automated oracle approach to test decision-making structures, International Conference on Computer Science and Information Technology, vol. 5, IEEE, 2010, pp. 30–34.
[33] H. Jin, Y. Wang, N. Chen, Z. Gou, S. Wang, Artificial neural network for automatic test oracles generation, International Conference on Computer Science and Software Engineering, vol. 2, IEEE, 2008, pp. 727–730.
[34] M. Ye, B. Feng, L. Zhu, Automated oracle based on multi-weighted neural networks for gui testing, Information Technology Journal 6 (3) (2007) 370–375.
[35] A. Gotlieb, B. Botella, I. IRISA, F. Rennes, Automated metamorphic testing, in: International Conference on Computer Software and Applications, 2003, pp. 34–40.
[36] J. Mayer, R. Guderlei, An empirical study on the selection of good metamorphic relations, in: Proceedings of Computer Software and Applications Conference, COMPSAC, 2006, pp. 475–484.
[37] T. Xie, Augmenting automatically generated unit-test suites with regression oracle checking, Lecture Notes in Computer Science 4067 (2006) 380.
[38] D. Lee, M. Yannakakis, Principles and methods of testing finite state machines – a survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123.
[39] T. Chow, Testing software design modeled by finite-state machines, IEEE Transactions on Software Engineering 3 (3) (1978) 178–187.
[40] R. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley Professional, 2000.
[41] J. Offutt, A. Abdurazik, Generating Tests From UML Specifications, in: Conference on Unified Modeling Language, Springer, 1999. p. 76.
[42] J. Offutt, S. Liu, A. Abdurazik, P. Ammann, Generating test data from state-based specifications, Software Testing, Verification and Reliability 13 (1) (2003) 8.
[43] Y. Kim, H. Hong, D. Bae, S. Cha, Test cases generation from UML state diagrams, in: Software, IEE Proceedings, vol. 146, IET, 1999, pp. 187–192.
[44] S. Kim, L. Wildman, R. Duke, A UML approach to the generation of test sequences for java-based concurrent systems, in: Software Engineering Conference, 2005. Proceedings 2005 Australian, IEEE, 2005, pp. 100–109.
[45] S. Mouchawrab, L. Briand, Y. Labiche, M. Di Penta, Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments, IEE Transactions on Software Engineering 37 (2011) 161–187.
[46] L. Liuying, Q. Zhichang, Test selection from UML statecharts, in: Conference on Technology of Object-Oriented Languages and Systems, IEEE, 1999, pp. 273–279.
[47] H. Hong, I. Lee, O. Sokolsky, S. Cha, Automatic test generation from statecharts using model checking, in: Workshop on Formal Approaches to Testing of Software, BRICS Notes Series, 2001, pp. 15–30.
[48] P. Chevalley, P. Thévenod-Fosse, Automated generation of statistical test cases from UML state diagrams, in: International Computer Software and Applications, IEEE, 2001, pp. 205–214.
[49] L. Briand, Y. Labiche, J. Cui, Automated support for deriving test requirements from UML statecharts, Software and Systems Modeling 4 (4) (2005) 399–423.
[50] Z. Dai, Model-driven testing with UML 2.0, Canterbury, England, 2004.
[51] P. Baker, Z. Dai, J. Grabowski, I. Schieferdecker, O. Haugen, C. Williams, Model-Driven Testing: Using the UML Testing Profile, Springer, 2007.
[52] L. Naslavsky, H. Ziv, D.J. Richardson, Towards traceability of model-based testing artifacts, in: International Workshop on Advances in Model-Based Testing, ACM, London, United Kingdom, 2007, pp. 105–114.
[53] A. Javed, P. Strooper, G. Watson, Automated generation of test cases using model-driven architecture, in: International Workshop on Automation of Software Test, IEEE Computer Society, 2007.