Model-Based Testing from UML Models

Eddy Bernard[1], Fabrice Bouquet[2], Amandine Charbonnier[3], Bruno Legeard[3], Fabien Peureux[3], Mark Utting[4], Eric Torreborre[3]

(1) LEIRIOS GmbH
Elisabethstrasse 91
80797 MUENCHEN
Germany
eddy.bernard@leirios.com
www.leirios.com

(2) University of Besançon
LIFC
16 route de Gray
25030 Besançon
France
bouquet@lifc.univ-fcomte.fr

(3) LEIRIOS Technologies
18 rue Alain Savary
25000 Besançon
France
{charbonnier, legeard, peureux, torreborre}@leirios.com

(4) University of Waikato
Department of Computer Science
Private Bag 3105
Hamilton – New-Zealand
marku@cs.waikato.ac.nz

Abstract: This paper presents an original model-based testing approach that takes a UML behavioral view of the system under test and automatically generates test cases and executable test scripts on the basis of model coverage criteria. This approach is embedded in the LTG model-based testing tool and is currently deployed in domains such as electronic transactions, embedded control software and information systems.

1 Introduction

UML 2.0 [RJB05] contains a large set of diagrams and notations, defined in a flexible and open-ended way using a meta-model and with some freedom allowed for different interpretations of the semantics of the diagrams by different UML tools. So for practical model-based testing it is necessary to select a subset of UML and clarify the semantics of the chosen subset so that model-based testing tools can interpret the UML models. Each model-based testing solution takes a different approach to this, by supporting various kinds of diagrams and defining a safe subset of those diagrams that can be used in models. It is necessary to define both the data part of the model (class and instance diagrams are typically used for this) and the dynamic behavioral aspects of the model.

In this paper, we describe an original model-based testing solution that we have developed, embedded into the LEIRIOS Test Generator (LTG) tool, and deployed in numerous industrial validation projects in domains such as smartcards, electronic transaction, automotive control software and banking or administrative information systems. It takes as input a UML model of the expected behavior of the system under test (SUT), uses model coverage criteria to automatically generate test cases, and uses adaptors to translate the generated test cases into executable test scripts.

2 Model-Based Testing process with LTG

The test generation process can be split into five main steps:

1st step - Model: The validation engineer first constructs the model from a textual specification. The goal of this step is to translate the description of the features of the tested system into a precise UML model of the expected behavior.

2nd step - Animate & Validate: The newly created model can then be checked by the validation engineer by means of LTG animation facilities. This enables the behavior of the modeled system to be checked to make sure it reacts to invoked operations and specified arguments as it was meant to. This can be seen as a "model debugging" step.

3rd step - Generate: Once the model is reasonably trustworthy, the validation engineer starts the test generation. In most cases, this step will be iterative; each iteration enhancing or refining results obtained in the previous iteration. Details of generation of different parts of test cases are described in the next sections. The goal of this step is to generate abstract test sequences. Note that generated test cases are called *abstract* because they are displayed in an implementation independent way.

4th step - Export: The translation of the generated cases into the target test script language is performed in an *adaptation* step. This step involves user-specific adapters designed once for the project by the validation engineer to define script patterns and mappings. They are used to translate abstract names from the model into concrete names in the target language and to translate the test cases into executable scripts on the test bench.

5th step - Execute: Generated test scripts can be run with user's test execution environment. At this stage, test failures may be caused either by modeling mistakes, adapter errors or by actual bugs in the tested system.

3 Test generation from UML models

3.1 Modeling with UML to generate tests

This section discusses UML modeling from the perspective of automated generation of black box tests from UML models. This perspective means that we want to use UML to model the *behavior* of systems, rather than their physical structure (e.g., deployment diagrams) or the design of their implementation (e.g., component diagrams, collaboration diagrams, or internal structure diagrams).

The LTG approach uses a subset of the UML 2.0 language with the following diagrams and expression language:

- **Class diagrams** – These represent the data model of the SUT behavior, and exhibit the points of control and observation of the SUT as operations within classes;
- **Instance diagrams** – This kind of diagram is used to set up the initial state of the system for test generation, by defining objects, initial values of their attributes and the associations between objects.
- **State Machine diagrams** – A UML state machine is associated with the main SUT class and formalizes the expected behavior of that class using transitions between states to respond to user events;
- **The Object Constraint Language (OCL)** – We use OCL [WK03] expressions within the class diagram to formalize the expected behavior of operations of a class using preconditions and postconditions. OCL is also used within the state machine to formalize the transitions between states – the guards and the effects of transitions are expressed as OCL predicates.

The UML models that are used for test generation must have a precise and unambiguous meaning, so that the behavior of those models can be understood and manipulated by the test generator. This precise meaning also makes it possible to use the model as an *oracle,* either by predicting the expected output of the SUT, or checking the actual output of the SUT against the model. OCL expressions associated with class operations and state machines provide the expected level of formalization necessary for model-based testing modeling. LTG performs symbolic execution of the model by taking an operational interpretation of OCL postconditions and using specialized constraint logic programming technology to compute the next state after invoking an operation of the model. This symbolic execution capability enables the generation of test sequences and the computation of expected SUT outputs for oracle purposes.

3.2 Test selection criteria

Test selection criteria are supported by the LTG to control the choice of tests from all the possible tests that can be derived from the behavior UML model. It is the validation engineer's knowledge of the SUT that is the key factor in generating a good suite of tests and achieving the test objectives of the validation project. Test selection criteria are the means of communicating that validation expertise of tests to a model-based testing tool. LTG supports several different kinds of criteria, to give precise control over the generated tests:

- **Transition-based** coverage criteria, like All-States, All-Transitions and All-Transition-Pairs;
- **Decision-based** coverage criteria, like Decision coverage, Condition/Decision coverage, Modified Condition/Decision coverage and Multiple Condition coverage;
- **Data-oriented** coverage criteria, like boundary value coverage [KLPU04] and one-Value/All-Values coverage.

It is important to notice that all these test selection criteria are related to models and define how well the generated test suite covers the model. None of them rely on the source code of the SUT or on coverage measurements made on the SUT (see [ZHM97] for a survey on code test coverage criteria). Notice also that the level of coverage of the model based on test selection criteria does not directly imply the level of code coverage by the generated test suite. Model coverage and code coverage are different issues.

4 An example of model and generated tests

In this section, we describe an example of model-based testing with UML in the domain of electronic ticketing. This example concerns the validation of the Paris metro electronic ticket Navigo with the ImagineR contract. This contract allows the user to take public transportation within 2 travel zones for a period of one year. During week-ends, public holidays and school holidays, the ImagineR contract allows travel within all 8 possible zones (see www.ratp.fr for more details on this contract). We present now a simplified version of the UML model designed for test generation, the test cases generated with LTG and a discussion on generating executable test scripts.

4.1 ImagineR UML model

The UML model used for test generation is composed with a class diagram (main classes in figure 1 and enumeration classes in figure 2), an instance diagram to set up the initial state (figure 3) and a state machine (figure 4) to formalize the expected SUT behavior. These diagrams have been developed with Borland Together® Designer 2006 for Eclipse, and then imported directly into the LTG model-based testing tool.

The 'validator' class represents the SUT; the other classes just store data. Two kinds of operations are defined on the 'validator' class:

- **Events** – The events define user actions on the SUT that are used in the associated state machine to trigger transitions between states (see figure 4);
- **Operations** – The operations are internal actions that modify the system state during the computation of user actions. They are called from the action part of transitions in the state machine (see figure 4) and their meaning is defined by OCL postconditions in the class diagram.

The four enumeration classes in figure 2 define abstract data values: states (card valid or defective), messages, dates and zones.

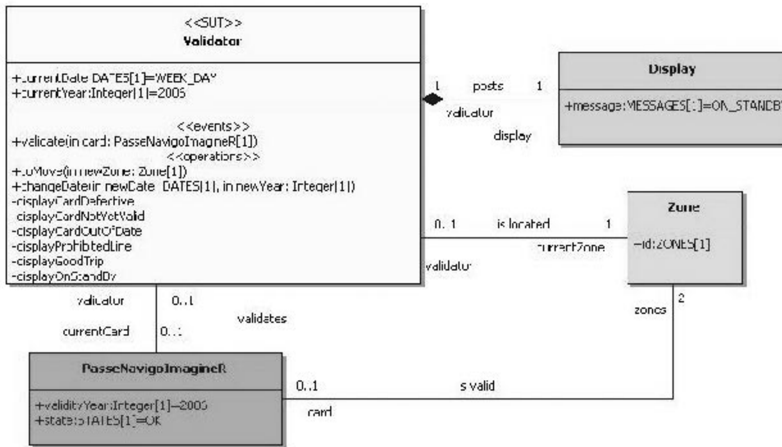

Figure 1 – ImagineR class diagram (main classes)

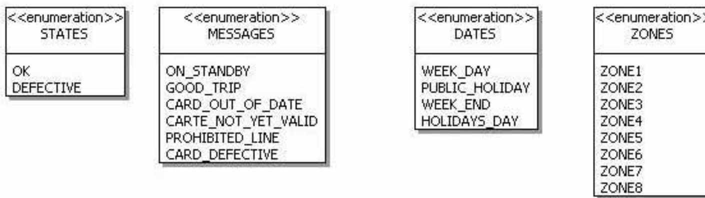

Figure 2 – ImagineR class diagram (enumeration classes)

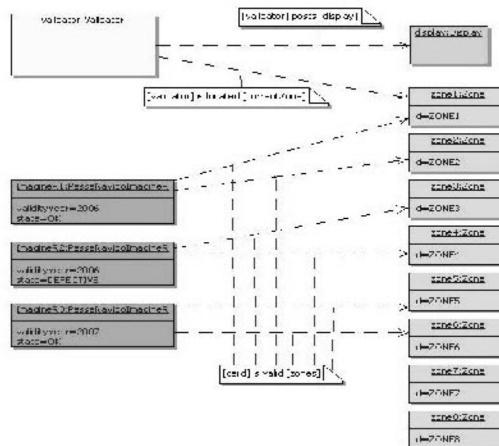

Figure 3 – ImagineR enumeration diagram (initial state)

The object diagram (see figure 3) gives the initial state of the system for test generation purposes. The various objects in the diagram give the test data that are managed during the test sequences. The initial value of attributes can be given at the instance level or within the class diagram, but all object attributes must be initialized.

Figure 4 presents the state machine associated with the SUT class. This state machine gives the dynamic behavior of the system. The labels of the transitions are formalized in OCL with the following structure: *Event[Guard]/Action*. *Event* is a user event defined in the SUT class, *Guard* is an OCL predicate, and *Action* is an operation defined in the SUT class or an OCL postcondition. Each part is optional.

The internal action sets up new values for attributes: for example the 'changeDate(in newDate: DATES, in newYear: Integer)' action is defined by the following postcondition: ***self***.*currentDate* = *newDate* ***and*** ***self***.*currentYear* = *newYear* and 'toMove(in newZone: Zone)' is defined by the following OCL postcondition: *self.currentZone = newZone*.

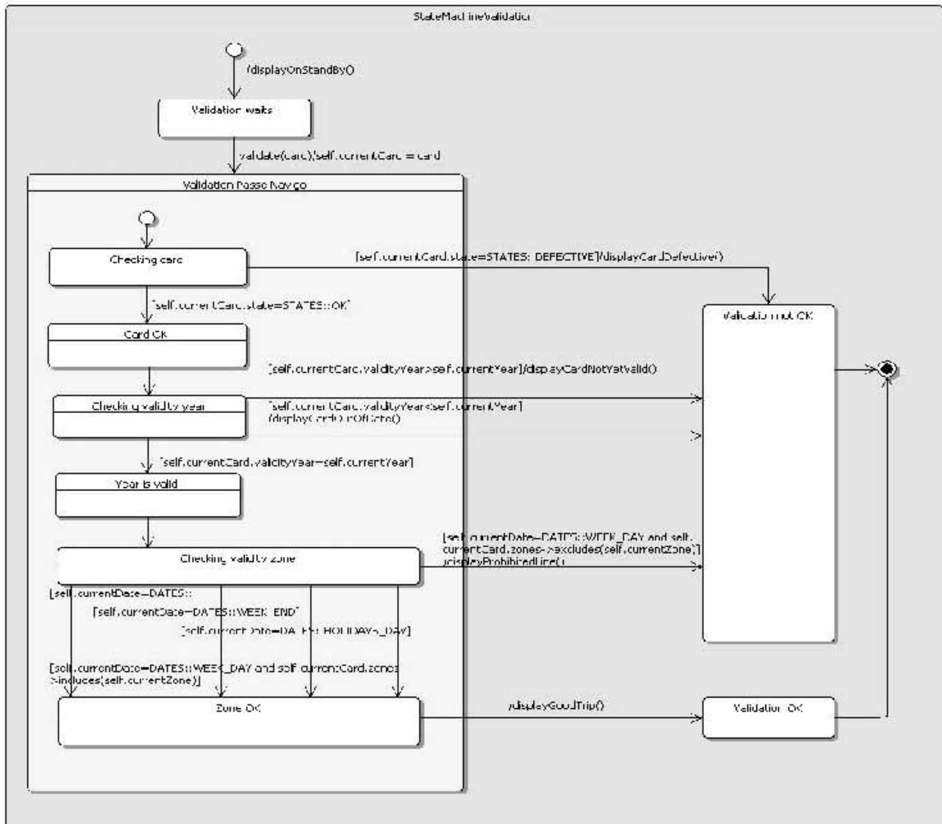

Figure 4 – ImagineR state machine

4.2 Generating test cases and executable test scripts

To generate the test cases, we decide to cover all the transitions of the state machine as well as all paths in the OCL postconditions of the internal actions. The data coverage criterion All-Values is set up for the input parameters of the 'changeDate' operation in order to explore the relevant combinations of dates and years of contracts. When these test selection criteria are applied to the ImagineR UML model, the LTG tool produces 20 test cases.

	Test case definition	**Expected results**
TC1	changeDate(newDate=HOLIDAYS_DAY, newYear=2006) ; validate (card=ImagineR1)	Message = GOOD_TRIP
TC2	toMove(newZone=ZONE3) ; validate (card=ImagineR1)	Message = PROHIBITED_LINE

Table 1 – Extract of generated test cases

Table 1 presents two of the generated test cases: TC1 is a case where the electronic ticket validation is accepted and TC2 is a case where the ticket validation is not accepted. In TC1, 'changeDate' is a preamble operation which sets the context and 'validate' is the tested operation.

This electronic ticket validation example is a simplification of a real validation project at PARKEON (see www.parkeon.com) where all contracts of the electronic ticket have been validated using model-based testing with UML. In this application, the test execution environment is a proprietary tool of PARKEON, developed in the scripting language Tcl. Therefore, executable scripts were generated using a LTG/Tcl adapter and a mapping table that links abstract operation signatures from the model with concrete interfaces provided by the test execution environment to set up the card and invoke the validation terminal.

5 Conclusions

The unique features of the LTG approach come from the conjunction of the following:

- The UML model defines the expected behavior of the SUT; it is defined using class and object diagrams for the static description and state machines with OCL expressions for the dynamic part;
- The test selection criteria are based on a large variety of model coverage criteria including transition-based criteria, decision based criteria and data-oriented criteria; These model coverage criteria make it possible for the validation engineer to precisely tune the automated test generation computation;

- An adaptation process of the generated test cases into executable test scripts is based on defining mappings between abstract and concrete names and templates written in the target test execution language.

In conclusion, the success of model-based testing in industrial settings is related to several major issues:

- **The adequacy of the modeling notation to easily develop test-oriented models**: UML models offer various paradigms (OCL preconditions and postconditions, state machines, sequence charts…) that make it possible to model various aspects of the expected behavior of the SUT;
- **The accuracy of automatically generated test cases with respect to the test objectives**: the test selection criteria must provide the validation engineer flexible control over the test generation.
- **The capacity to automate the execution of the generated tests**, in regards to the number of tests and automation of the full test process.

References

[KLPU04] Kosmatov, N.; Legeard, B.; Peureux, F.; and Utting, M.: Boundary Coverage Criteria for Test Generation from Formal Models. In Proc. of the 15th Int. Symp. on Software Reliability Engineering (ISSRE'04), Saint-Malo, France, pages 139--150, November 2004. IEEE Computer Society Press.

[RJB05] Rumbaugh, J.; Jacobson, I.; Booch, G.: The Unified Modeling Language Reference Manual, Second Edition. Addison-Wesley, 2005.

[UPL06] Utting, M.; Pretschner, A.; Legeard, B.: A Taxonomy of Model-Based Testing, Technical report 04/2006, Dept. of Computer Science, The University of Waikato, April 2006. www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf

[WK03] Warmer J.; Kleppe A.: The Object Constraint Language, Second Edition. Addison-Wesley, 2003.

[ZHM97] Zhu, H.; Hall, P.A.V.; May, J.H.R.: Software Unit Test Coverage and Adequacy. ACM Computing Surveys, 29(4):366-427, 1997.