Conformance Testing from UML Specifications Experience Report[1]

Lydie du Bousquet[†]   Hugues Martin[‡]   Jean-Marc Jézéquel[§]

[†] LRS-IMAG, BP 72, 38402 St Martin d'Hères cedex, France
Lydie.du-Bousquet@imag.fr
[‡] Gemplus Research Lab, BP 100, 13881 Gémenos, France
Hugues.Martin@gemplus.com
[§] IRISA, Campus de Beaulieu, 35042, Rennes cedex, France
Jean-Marc.Jezequel@irisa.fr

Abstract: UMLAUT is a framework for building tools dedicated to the manipulation of models described using the Unified Modeling Language (UML). TGV is a tool for the generation of conformance test suites for protocols. Both tools are connected so that it is possible to specify an application in UML and derive automatically some test cases. In this article, the integration of those tools in an industrial process is evaluated through a case study. This case study, proposed by Gemplus, is a Java Card applet: a classical electronic purse.

1   Introduction

The automatic generation of test for conformance testing has been studied for several years. Academic conformance testing tools are now being transformed into industrial products. This paper reports on an attempt of such a tranformation. UMLAUT and TGV are two academic tools developed at the IRISA research center. UMLAUT is a framework for building tools dedicated to the manipulation of models described using the Unified Modeling Language (UML) [Ho99]. TGV is a tool for the generation of conformance test suites for protocols [JM99]. Both tools have been connected so that it is possible to generate test data from a UML specification [Le01,JLP98].

Between 1999 and 2000, Gemplus and IRISA have collaborated in order to check the ability of UMLAUT/TGV to scale up to industrial applications (Lhusy project). Gemplus proposed an industrial case study: a Java card application (an electronic purse). The work done at IRISA was first to specify the electronic purse application in UML and second to generate test cases and to execute them. In the meantime, Gemplus produced some test data manually. So it was possible to compare manual test generation and automatic test generation.

In this article, we describe the tools and the experimentation. Section two presents the case study used to validate our testing process. Section three is devoted to the presentation of the different tools. Sections four and five describe the UML specification

---

and the result of the test generation experiment. Finally, section six is devoted to the conclusions and current works.

2   The case study: an electronic purse

In this section, we give an informal specification of the purse applet we studied. The complete informal specification, several formal specifications and the Java source code of this application can be found on the Gemplus web site at the URL: http://www.gemplus.com/smart/r_d/publications/case-study/.

The electronic purse offers the following functionalities to the card holder: authentication, debit or credit his purse. The purse balance can be consulted, the currency in which this balance is expressed can be changed. The purse contains the list of supported currencies. It records the transactions done in a "log file".

Moreover, the purse is planned to share information with several fidelity applets (loyalties). A loyalty applet enables to capitalize fidelity points, depending on the user transactions recorded in the log file. A loyalty implements a shareable interface to allow the purse to communicate with it. The loyalty applets can subscribe a purse service to be informed when the purse could loose some transactions (when the log file is full). The purse contains the list of loyalties with which it can communicate.

In order to secure the purse, some cryptographic mechanisms are embedded with the card and with the purse applet. These mechanisms use keys and random numbers. Several attributes and methods in the purse are devoted to the security and the cryptographic mechanism management. For instance, the administrator authentication is managed by the purse (contrary to the user authentication which is from the responsibility of the Java Card).

Figure 1 details an example of security mechanism introduced in order to prevent some malicious uses of the purse. To credit the purse, the user has to perform two actions. The first one is an initialization (done with the *appInitCredit* method). It aims at configuring a secure communication between the purse and the card reader. This operation generates a cryptographic key. The second operation is the "acknowledgment" of the credit operation (*appCredit* method). It verifies the correctness of the secure communication and performs the credit operation. For security reasons, the *appInitCredit* method succeeds only if the user is authenticated. Moreover, a key computed during the execution of the *appInitCredit* method has a limited lifetime. It is invalidated if any another method than *appCredit* is executed just after the execution of *appInitCredit*. It is also invalidated after the execution of *appCredit*. This mechanism is also applied for the user authentication,the debit and the change currency operations.

In its full version, the purse interface contains nine operations intended to the card holder and forty operations intended to the applet manager. Its compiled size, to be embedded in a Java Card, exceeds 23Ko. Its code size exceeds 7000 lines of Java code. It uses the Java Card API and the mechanisms of shareable interfaces to communicate with other embedded Java Card applets.
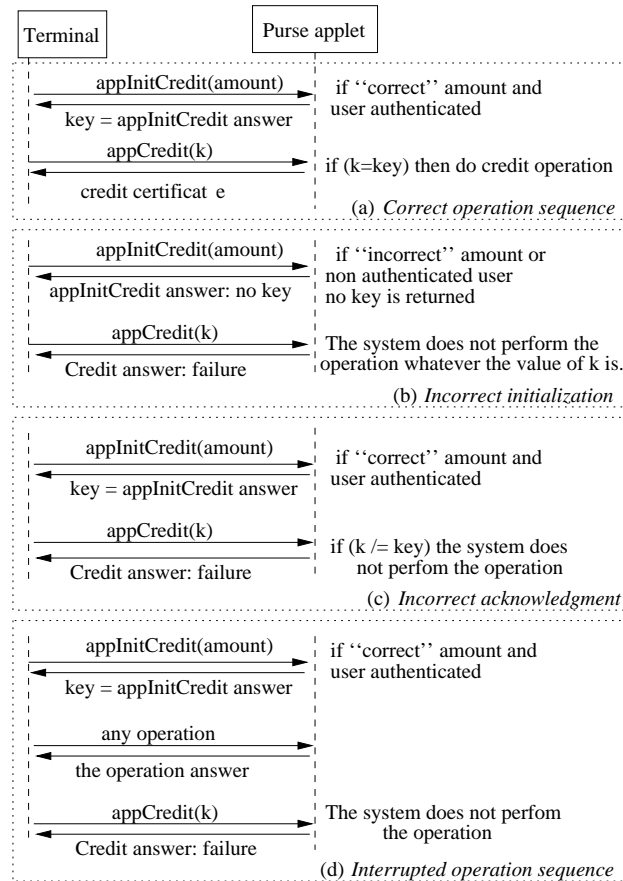
Fig. 1: The credit secure messaging behavior

3 Tools for specification and test generation

3.1 TGV: a conformance testing tool

Conformance testing

Conformance testing is a black box functional testing [Is91]. It is one of the most rigorous testing techniques [Br88,Tr92]. The usual theoretical approach is to consider a formal specification of the System Under Test (SUT) intended behavior. A conformance relation defines the correctness criterion of the implementation with respect to the formal specification.

The conformance relation formally defines the verdict associated the test case execution. Three verdicts are generally distinguished. Informally, "fail" means rejection by the test case, "pass" means that the goal of the test experiment (described by a so called test

purpose) is reached, and "inconclusive" means that the implementation correctly behaves but, due to a lack of control on the SUT, does not allow to reach the expected goal.

TGV

TGV (Test Generation with Verification technology) is a conformance test generator [JM99]. It has been developed as a collaboration between Verimag Grenoble and IRISA. It uses the libraries of the Caesar-Aldebaran Distribution Package developed by Verimag Grenoble and VASY team from Inria Rhône Alpes [Ga98].

TGV takes as inputs the SUT specification and a test purpose. The specification can be expressed as a Input-Output Labeled Transition System (IOLTS). A IOLTS consists of states with transitions between them, where transitions are labeled with actions or events. An example of a coffee machine specification is given figure 2. This specification indicates that the coffee machine should first "receive" one or two coins, and an order (tea or coffee, with or without sugar), and then "emit" the right product(s).

?i: specification input
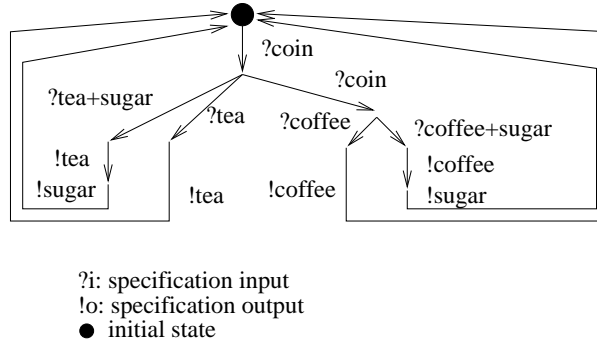!o: specification output
● initial state
Fig. 2: A coffee machine specification, expressed as an IOLTS

A test purpose (TP) is used to select a part of the specification, for which a test case will be generated. A "good" test purpose should be simple (typically, much simpler than the specification) and should select exactly the scenarios that the user has in mind. Two test purposes are given figure 3. The test purpose TP1 specifies a test case in which the coffee machine would "emit" coffee and then sugar. The end of the test is denoted by a transition labeled "accept" in the test purpose. So in TP1, the test will be completed after the "emission" of the sugar. The test purpose TP2 means that one wants to select a test case in which the coffee machine receives an input whose name starts with "co" (i.e. coffee or coin), and then "emit" sugar.

Given a specification and a test purpose, TGV generates a test case. If TP1 is used, TGV will produce TC1. If TP2 is chosen, TGV will generate non-deterministically one of the two test cases TC1 or TC2.

As one can notice, the test case actions have not the same orientation as the specification ones. Indeed, the specification expresses the implementation under test viewpoint. On the contrary, a test case specifies what the environment (here the coffee machine users) should do and observe during the test.TC1 means that after the insertion of two coins and a "coffee+sugar" command, the environment should obtain some coffee and then some sugar.
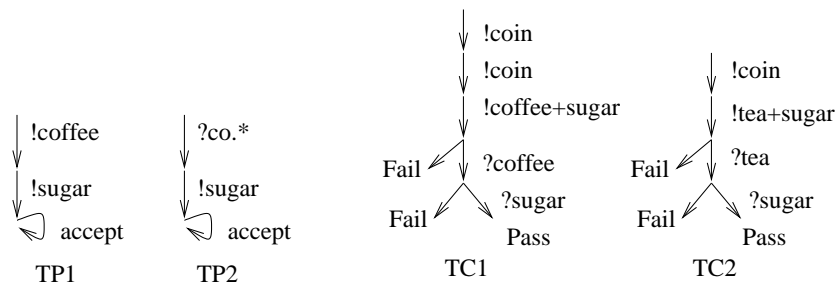
Fig. 3: Some examples of test purposes and test cases

TGV relies on efficient algorithms which are based on adaptations of on-the-fly model-checking algorithms [JM99]. "On-the-fly" means that the test case generation is done in a lazy way. During the computation, the specification state space is not completely stored, so that state explosion problem is limited.

3.2 UMLAUT: a UML transformation framework

UMLAUT (Unified Modeling Language All pUrposes Transformer) is a tool dedicated to the manipulation of UML models. It is a general framework for UML model transformation [JLP98, Ho99, Le01]. UMLAUT is developed within the Triskell Project and is distributed as freeware (http://www.irisa.fr/UMLAUT/).

The VALOOD module of UMLAUT is dedicated to the transformation of a UML model into an accessibility graph.The accessibility graph of a model describes the evolution of a system in terms of states and transitions labeled by events (operation calls, timer expirations, message exchanges).

To build the accessibility graph, VALOOD needs a UML model which describes the behavior of the System Under Test and the behavior of its environment. The environment behavior is denoted by one or several actor(s) in the class diagram. The UML model should be composed of a class diagram a deployment diagram, and the statecharts of all the objects present in the deployment diagram. The latter is used to describe the initial state of the SUT.

For instance, let us consider a toy example of purse (see figure 4). The SUT is specified by one class (a TOY-PURSE) and its environment is a terminal, i.e. the card-reader (see fig. 4(a)). In this example, it is only possible to credit the purse with a positive amount, as it is specified in the statechart given fig. 4(c). The deployment diagram specifies that the test configuration is composed of only one purse and one terminal (figure 4(d)). Deployment diagrams do not necessary refer only to the initial state of the system. They are *snapshots* of the system at a given moment, susceptible of evolution as new objects or connections are dynamically created. If a new active object is created, then its state machine starts executing, which adds a new flow of control in the system.

(a) Class diagram

(b) Terminal Statechart

(c) Toy-Purse Statechart

(d) Deployment diagram

?i specification input
?o specification output
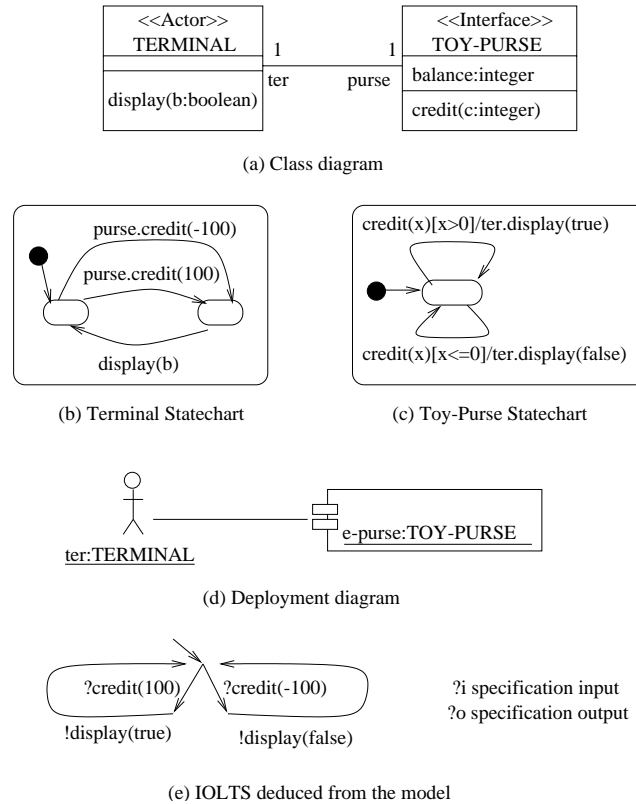
(e) IOLTS deduced from the model

Fig. 4: Illustration of the UML model characteristics needed by UMLAUT/TGV

The accessibility graph denotes the behavior of the full system. Intuitively, it is given by the statechart composition of all objects (actor included). The accessibility graph is not empty if the statechart of the actors call some methods of the SUT. For such calls, all the parameters of the method should be instantiated with concrete values. The result of this is to multiply the number of transition in the actor statechart(s).

For the toy-purse example, we choose to describe a statechart for the terminal in which the credit operation has two possible parameter instantiations (fig. 4(c)).

One should note that the accessibility graph is usually an infinite graph. Therefore it is not built exhaustively by VALOOD,but explored progressively, as needed by TGV during the test case generation. Starting from the initial state, VALOOD computes the set of all fireable transitions outgoing this state. The IOLTS is built by labeling the transitions with atomic actions executed by the system. The accessibility graph of the toy-purse is given figure 4(e) as an IOLTS.

Along the lines of Aspect Oriented Programming [Ki97], the approach chosen in UMLAUT consists in weaving the various dynamic aspects of the initial model into a simple subset of UML, made of classes and operations only. Most transformations deals

with UML state machines, probably one of the most complex construction available in UML.

A state machine normally comprises a set of states, an event queue, and a thread that dispatches events taken from the queue. We have decided to make the event queue and the thread explicit in the transformed model (which contains classes dedicated to these concepts). The states of the state machines are also reduced to the simpler concept of class: each state can indeed be seen as a specific subtype of the class to which the state machine is attached (the state machine's context). This subtype has the same interface as the context, but operations are redefined according to the transitions outgoing the corresponding state (inner transitions override outer ones in UML, which fits perfectly in this scheme). Figure 5 illustrates the transformation, which is similar in principle to the State Design Pattern [Ga94].

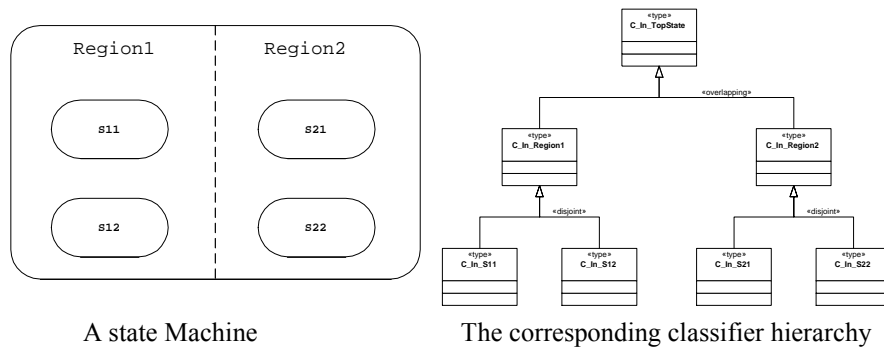A state Machine   The corresponding classifier hierarchy

Fig. 5: Normalization of UML state machines

Of course, the state of an object can change, which maps to dynamic typing after the transformation is done (but this is not a problem in principle since UML already supports dynamic and multiple classification).

Finally, the initial deployment is also transformed so as to include the objects representing communication queues and event dispatchers.

4   A UML model for the purse

Modeling and validation phases are strongly related. When one wants to derive test cases automatically, one has to cope with two opposite objectives. The formal specification should be detailed enough to allow the generation of significant tests. But the specification should also be abstract enough to make the model readable. For instance, the cryptographic mechanisms were not supposed to be tested (they are proved). However, those cryptographic mechanisms are used by the user secure messaging, which is managed by the purse. If one wants to test the last situation described figure 1, one has to model the validity of the key values.

In the following, we describe the UML model we designed in order to validate the functional behavior of the purse.
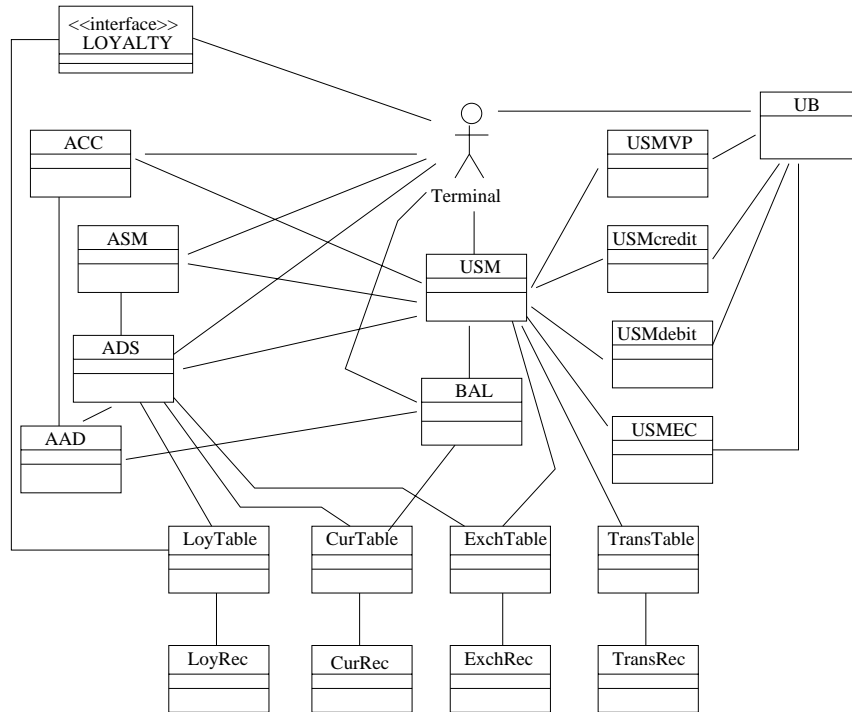
Fig. 6: Abstract view of the purse model class diagram

The class diagram

As we said before, to use UMLAUT/TGV, we need to provide a model which describe both the System Under Test (SUT) and its environment. The SUT is the purse applet and the loyalty applets loaded in the card. The model is oriented to test the purse functionalities and the interaction between the purse and the loyalties (from the purse point of view).

In the final UML model, the SUT is modeled by 20 classes (see figure 6). There are one class to model the loyalty applet interface and the 19 other classes describe the purse.

- Six classes are devoted to the user secure messaging (USM) management USM, USMcredit, USMdebitm, USMEC, USMVP and UB.

- Four others are dedicated to the administrator rights and the cryptographic attributes management (ACC, ASM, ADS, AAD).

- Eight classes implement the four tables of the purse (TransRec and TransTable to record transactions done by the user, LoyRec and LoyTable to record loyalties with which the purse can communicate, CurRec and CurTable to record available currencies, ExchRec and ExchTable to record currency changes).

- The last class deals with the balance of the purse (BAL).

There are 73 attributes and 99 methods distributed in those 19 classes.

We choose to model the environment as one automaton, which corresponds to the terminal. This actor class has no attribute and 5 methods. The methods allow the SUT to display the purse balance (*display* method) and the content of the different tables (for instance *displayTransRec* method to display one record of the transaction table).

The deployment diagram

To test the purse, we choose the following configuration. The purse was loaded in the card with three loyalty applets (Loy1, Loy2 and Loy3). Two of them were recorded in the purse loyalty table (Loy1 and Loy2). Moreover, Loy1 has subscribed to the service offered by the purse (see §2).

The statecharts

We attached a statechart to each class of the purse. There are 50 states and 275 transitions distributed in the statecharts associated to the purse classes.

For instance, figure 7 denotes the skeleton of the statechart associated with the USMcredit class. It is composed of 2 states, 3 pseudo-states and 11 transitions. The USMcredit class is one of the five classes dedicated to User Secure Messaging management. The USMdebit, USMVP and USMEC class have statecharts similar to the USMcredit class one.

The environment statechart has 2 states and 143 transitions, which correspond to the chosen parameter instantiations for the methods of the purse. The instantiations were selected manually, by identifying partition classes [RC85] and limit values.
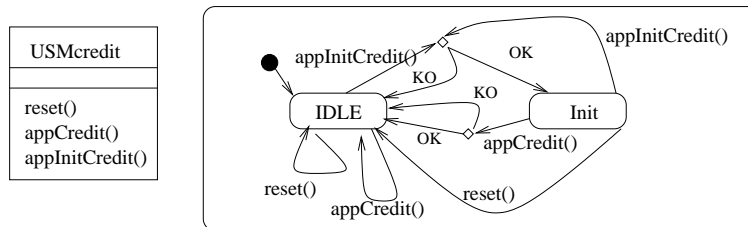

Fig. 7: An abstraction of the USMcredit class and statechart

5   Test generation for the Purse case study

Our strategy for the test generation was the following. We first identified a set of properties we wanted to validate. We derived a set of test purposes to test each of these properties, and then we used TGV to generate a test case for each test purpose.

An example of property is that every functionality should have a correct behavior for every normal use and every possible misused. For instance, to valid the user secure messaging associated to the credit function, we have to take into account the four types of situations described figure 1. To be more precise, a credit can only be done if the user

is authenticated, if the required credit amount is positive, if the new balance does not exceed a maximum value and if the secure messaging is correctly done. Each of these operations can be done with correct or incorrect parameters: correct or incorrect PIN and key for *appVerifyPIN* method call, correct or incorrect key for *appCredit*. Moreover, for *appInitCredit*, the amount can be positive, null, negative, or greater or equals to the balance maximum value. To check the credit function, we found it interesting to generate one test case for each combination of the different parameters values, thus 4*2*4 test purposes.

Thus for each property to validate, we had to produce a lot of similar test purposes, which differ only from the value of the parameters. The process of creating by hand those kind of test purposes is therefore repetitive, and is a curb to a large scale use of TGV. We decided to build a small program called *BuildTP*. It takes as input a description of a test purpose were the method parameters are not instantiated (high-level test purpose) and a set of instantiations for each parameters. BuildTP generates automatically the all the combination of the instantiations.

We expressed 50 high-level test purposes, which were derived into 2100 test cases. The produced test cases are generally quite simple. Most of them consist of less than ten interactions between the purse and the terminal. Figure 8 gives an example of test purpose and associated test case. Both are given in a textual format[2]. The produced test case correspond to a valid credit operation.
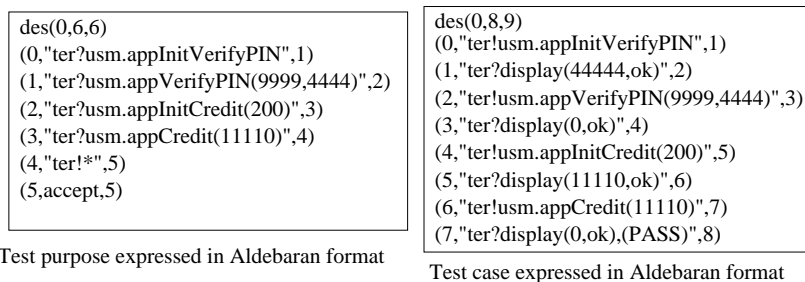
<div style="display:flex; gap:20px;">
<div>

```
des(0,6,6)
(0,"ter?usm.appInitVerifyPIN",1)
(1,"ter?usm.appVerifyPIN(9999,4444)",2)
(2,"ter?usm.appInitCredit(200)",3)
(3,"ter?usm.appCredit(11110)",4)
(4,"ter!*",5)
(5,accept,5)
```

Test purpose expressed in Aldebaran format
</div>
<div>

```
des(0,8,9)
(0,"ter!usm.appInitVerifyPIN",1)
(1,"ter?display(44444,ok)",2)
(2,"ter!usm.appVerifyPIN(9999,4444)",3)
(3,"ter?display(0,ok)",4)
(4,"ter!usm.appInitCredit(200)",5)
(5,"ter?display(11110,ok)",6)
(6,"ter!usm.appCredit(11110)",7)
(7,"ter?display(0,ok),(PASS)",8)
```

Test case expressed in Aldebaran format
</div>
</div>

Fig. 8: Example of a test purpose and its corresponding test case

Executable test cases

The last step of our work consisted in translating abstract test suites previously generated into Java programs. In order to perform it, we have developed a translator: *aut2Java*. Using a dictionary of actions, aut2Java replaces the labels in a test case with concrete method calls.

---

[2] "des()" indicates the initial state, the number of transitions and the number of state. The expressions "ter!" and "ter?" identify the environment emission and reception events in the test case. They respectively indicate the system emission and reception events in the test purpose.The user PIN is 9999. The numbers 4444 and 11110 represent two key values, used for the secure messaging.

Result of the experiment

With UMLAUT/TGV, we found forty-two errors, among which the fact that the user secure messaging was interruptible for the change currency function. It took us eight days to specify the purse applet in UML, eight days to program aut2Java and BuildTP, and less than two days to generate the abstract test cases, to translate them in Java and to execute them.

In parallel with the use of UMLAUT/TGV, a manual test generation process has been applied to the same applet in the Gemplus Lab. There were 126 test cases produced by hand. Those test cases were more complex than those produced with UMLAUT/TGV. It took twenty-two days to specify, to implement and to execute the test cases. There were also forty-two errors found.

The errors found by both processes were not exactly the same. Globally, fifty different errors were found and each process missed eight errors. In the automated process, seven omitted errors involve some attribute values that were not explicitly considered in the model (there were supposed to be constant). The last error omission is due to a property we forgot to test. In the manual process, an erroneous test hypothesis was done (the "equivalence" of the user secure messaging for the four functionalities). Moreover, we forgot to test three limit values.

Process	Errors found	Number of test suites	Time spent
Manual	42	126	22 men-day
Automated	42	2100	18 men-day
Both	50	NA	NA

Fig. 9: Case study results

6   Conclusion and perspectives

The Lhusy project had two objectives. The first one was to evaluate the ability of UMLAUT/TGV to be used in a industrial context. The second one was to compare manual and automated testing processes.

We concluded that the automated test generation process gives better results than the manual one. The number of detected errors were equivalent, but the automated process was shorter. Moreover, it seems to be more rigorous: less omissions have been done (only one property). Finally, a UML specification is provided, and it is possible to use it for documentation.

The evaluation of UMLAUT/TGV is globally positive. The tools were able to manage a large UML model. It was also possible to generate a lot of test cases, quite quickly (with respect to a manual test generation process). However, four lacks were identified.

- The first one is a methodological difficulty. It is very hard to determine the right abstraction level for the UML model. During the experiment, several errors were missed because some variables were abstracted as constant. It is necessary to offer some methodological guides to the user, to help him in the model design.

- The second lack concerns the test purpose design. The test purpose design is an important phase in the testing process with TGV. The number of errors that can be discovered depends of the set of test purposes. If a large test purpose subset is forgotten, then some errors can be missed. So it is important to provide some help and/or to facilitate the work of test purpose design.

- As we said in section 3.2, in the UML model needed by UMLAUT/TGV, the statecharts of the actors determine the parameter instantiations of the system under test methods. This is a requirement due to TGV "limitations". The result of this is a multiplication of the transitions in the actor statechart(s). Currently, a more powerful testing tool is developed by IRISA. This tool, named STG is devoted to symbolic conformance testing [Cl01]. If UMLAUT and STG are connected, parameter instantiations would no longer be required.

- The last point is that the test cases produced by TGV are not directly executable, and require a transformation tool such as aut2Java.

At the end of Lhusy project, it was decided to develop an environment to integrate UMLAUT/TGV in an industrial process. Two academic institutions (IRISA and LSR-IMAG) and three industrial partners (Softeam, Gemplus and France-Telecom) are involved in this project named COTE (COmponent TEsting)[3]. The aim of the COTE project is to supply the actors involved in the provision and acquisition of software components with the methods, tools and techniques needed to test, verify and certify these components, and in doing so, to provide the means to achieve the level of quality control required for the development of a mass market in components. The COTE project is supported by the French Government through the RNTL program. It started on November 1st 2000 and will last for a total of two years.

The COTE project relies on UMLAUT/TGV technologies. The COTE environment will be integrated in the Objecteering tool developed by Softeam. In the framework of COTE, a tool is developed by the LSR-IMAG to help the tester to design the test purposes. This tool is named TOBIAS (Test OBjective desIgn ASsistant). Moreover, the translation of test cases derived from the UML model into executable test programs is studied. These test programs will be synthesized for different component architectures (EJB,COM+,CORBA,.NET).

Acknowledgment

Bibliography

[Br88]   E. Brinksma : A theory for derivation of tests. In S. Aggrawal and K. Sabnani, editors, *Protocol Specification, Testing and Verification VIII*, 1988.

[Cl01]   D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva : Automated test and oracle generation for smart-card applications. In *International Conference on Research in Smart Cards (e-Smart'01)*. To appear in LNCS, Springer-Verlag, 2001.

---

[3] http://www.irisa.fr/cote/

[Ga94]   E. Gamma, R. Helm, R. Johnson, and J. Vlissides : Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1995.

[Ga98]   H. Garavel : OPEN/CAESAR: An Open Software Architecture for Verification, Simulation, and Testing}. In *Proceedings of the First Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS 1384, Springer Verlag, 1998.

[Ho99]   W.-M. Ho, J.-M. Jézéquel, A. Le Guennec, and F. Pennaneac'h : UMLAUT: an extendible UML transformation framework}. *In Automated Software Engineering (ASE)*, Florida, USA, October 1999.

[Is91]   ISO : International Standard IS-9646 : Information Technology, Open Systems Interconnection, Conformance Testing Methodology and Framework. Geneve, 1991. Also: CCITT X.290--X.294.

[JLP98]   J.-M. Jézéquel, A. LeGuennec, and F. Pennaneach : Validating distributed software modelled with UML. In *Proc. Int. Workshop UML98*, Mulhouse, France, June 1998.

[JM99]   T. Jéron and P. Morel : Test Generation Derived from Model-checking. In *Computer Aided Verification (CAV)*. LNCS 1633, Springer-Verlag, 1999.

[Ki97]   G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin : Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th European Conference Object-Oriented Programming (ECOOP)*, Jyväskylä, Finland. LNCS 1241, Springer-Verlag, 1997.

[Le01]   A. Le Guennec. Génie Logiciel et Méthodes Formelles avec UML : Spécification, Validation et Génération de tests. PhD thesis, Ecole doctorale MATISSE, Université de Rennes 1, 2001.

[RC85]   D. Richardson and L. Clarke. Partition Analysis : A Method Combining Testing and Verification. *IEEE Transactions on Software Engineering*, december 1985.

[Tr92]   J.Tretmans. A formal approach to conformance testing. PhD thesis, University of Twente, Enschede, The Netherlands, 1992.