TOBIAS : un environnement pour la création d'objectifs de tests à partir de schémas de tests

Pierre Bontron, Olivier Maury, Lydie du Bousquet,
Yves Ledru, Catherine Oriat, Marie-Laure Potet
Laboratoire LSR IMAG
BP 72
38402 Saint Martin d'Hères Cedex, France

Tel (+33) 4 76 82 72 32, Fax (+33) 4 72 82 72 87, Contact : lydie.du-bousquet@imag.fr

Résumé :

Le projet RNTL COTE aborde le problème du test de conformité des composants. Il a pour objectif de décharger l'ingénieur de test des nombreux détails routiniers d'une activité de test (calcul des préambules des tests, implantation dans une technologie cible, etc.). L'idée est de lui permettre de concentrer son activité sur les aspects créatifs de la génération de test (conception de l'oracle, identification des hypothèses de test, choix des propriétés à tester, …). Pour y parvenir, plusieurs niveaux d'abstraction sont proposés : test exécutable pour une cible technologique, test abstrait, objectif de test. TOBIAS poursuit cet effort en proposant des schémas qui correspondent à une famille d'objectifs de test, et en fournissant les outils qui permettent de les exploiter.

Abstract :

The RNTL COTE Project addresses the problem of testing conformity of components. It aims at discharging the engineer from clerical testing activities (calculation of tests preambles, taking into account target technology, etc.). The goal is to focus his activity on the creative aspects of test generation (design of the oracle, identification of test assumptions, choice of the properties to test, ...). For that purpose, several levels of abstraction are proposed: executable test for a given technology, abstract test, test purpose. TOBIAS continues this effort by proposing diagrams which correspond to a family of test purposes, and by providing the tools which ease this exploitation.

1 Introduction

Le projet COTE[1] (COmponent TEsting) est destiné à fournir des méthodes, techniques et outils pour tester, vérifier, et certifier les composants logiciels, à la fois aux réalisateurs de composants et aux utilisateurs de composants. Ce projet s'intéresse essentiellement au test de conformité de composants à partir de spécifications UML. L'élément central de ce projet est un langage d'expression de tests abstraits appelé TeLa. Il permet de décrire des tests abstraits indépendamment de la technologie sous-jacente (CORBA, EJB, .Net, …).

Dans ce projet, la génération de tests est axée autour de UMLAUT/TGV [4,5]. Cet outil prend en entrée une spécification comportementale de l'application à tester et un *objectif de test* qui décrit le but du test sous la forme de transitions qui sont les points de passage obligés du test dans la spécification. Il produit en retour un cas de test abstrait.

Dans le cadre de l'utilisation industrielle d'un tel outil, l'écriture d'un ensemble satisfaisant d'objectifs de test reste un point difficile. D'une part, il est nécessaire de « couvrir » l'ensemble des fonctionnalités que l'on souhaite tester. D'autre part, l'écriture d'un grand nombre d'objectifs de test est une tâche répétitive et fastidieuse. Le travail que nous présentons dans cet article s'inscrit dans cette problématique.

L'idée de base de TOBIAS (Test OBjective desIgn ASsistant) est d'exploiter les similarités entre les objectifs de tests, pour les générer à partir de schémas de plus haut niveau.

Dans la partie 2, nous présentons un exemple de distributeur bancaire. Cet exemple nous servira à illustrer notre discours tout au long des parties 3, 4 et 5. Dans la partie 3, nous montrons comment sont exprimés les tests de manière abstraite. Dans la partie 4, nous présentons le niveau d'abstraction utilisé dans TOBIAS. Enfin, la partie 5 décrit le processus qui permet la génération d'objectifs de tests à partir de schémas de haut niveau.

2 Un exemple : un distributeur bancaire

Une banque possède plusieurs distributeurs permettant à ses clients de consulter leur compte à distance et de retirer de l'argent. Chaque client ne possède qu'un seul compte accessible à travers les distributeurs. Pour accéder à leur compte, les clients utilisent une carte de retrait. La *figure 1* représente le diagramme de classes d'un tel système.

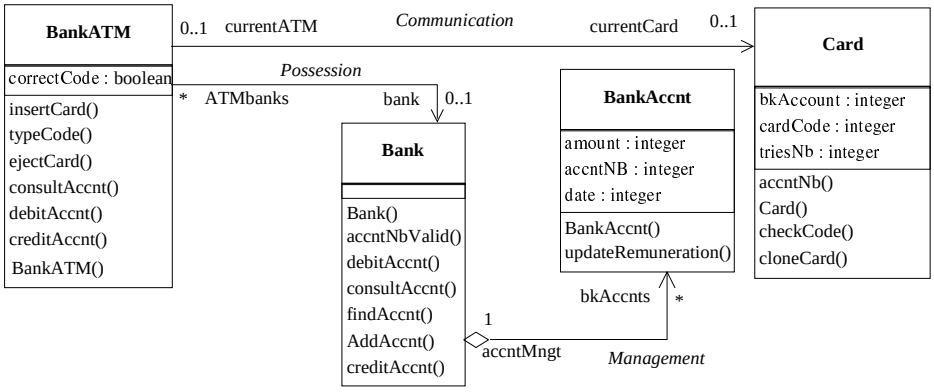

figure 1 : Diagramme de classes du distributeur bancaire

La classe Card modélise les cartes bancaires. Une carte contient le numéro du compte client, le code personnel du client et le nombre de saisies erronées du code lors de l'utilisation de la carte. La classe Bank modélise les banques. Elle est en relation avec zéro ou plusieurs distributeurs et gère zéro ou plusieurs comptes. La classe BankAccnt modélise les comptes bancaires. Chaque compte est composé d'un solde, d'un numéro et de la date du dernier retrait. La classe BankATM modélise les distributeurs. Chaque distributeur est composé d'une variable indiquant si une personne est connectée et offre les méthodes permettant à l'utilisateur de s'authentifier, de se déconnecter, de consulter son compte, de rajouter ou de retirer de l'argent.

[1] COTE est un projet RNTL réunissant Softeam, l'IRISA, le LSR-IMAG, Gemplus et France Télécom. http://www.irisa.fr/cote/

3 Des cas de tests concrets à la spécification

Quelle que soit la manière dont on souhaite tester une application (test fonctionnel ou de conformité, test aléatoire, test structurel …), il faut exécuter les tests sur une implantation de l'application : programme exécutable, prototype physique … Ce sont ces tests exécutables que nous appelons tests concrets. Dans le cas qui nous intéresse, les tests concrets doivent permettre de mettre en évidence les erreurs fonctionnelles des composants testés. C'est après l'exécution des tests que nous pourrons décider de l'arrêt du test et ce, selon certains critères. Nous présentons dans cette partie différents niveaux d'abstraction qui sont utilisés dans COTE.

3-1 La notion de test concret

Un test concret est un test qui sera effectivement exécuté. Dans le cadre du test de composant, ces tests doivent tenir compte de la technologie du composant testé : CORBA, EJB, .NET.
Sur le plan de la structure d'un test, il nous faut identifier et définir divers éléments :

- le composant sous test (CUT) qui est le composant pour lequel le test est défini,
- le système sous test (SUT) qui contient les divers composants pouvant être nécessaires pour tester le CUT,
- le testeur qui stimule le SUT et initialise, déclenche, coordonne les tests. De plus, il fournit le verdict global.

Voici un exemple de test exécutable décrit en java. Le but de ce test est de vérifier que le solde du compte d'un utilisateur est positif.

```
public class Testeur {
   public BankATM ATM1 ;
   public Testeur(){}
   public Verdict sequence1(){
      if(ATM1.insertCard(card01) && ATM1.typeCode(3920)
         && ATM1.consultAccnt()>=0 && ATM1.ejectCard())
           return PASS ;
      else return FAIL ;}
   public static void main(){
       ATM1 = new BankATM() ;
       Testeur1 = new Testeur();
       Testeur1.sequence1();}}
```

figure 2 : Un test exécutable codé en JAVA

Si, lors de l'exécution du test, la carte ou le code est refusé, si le solde du compte est négatif ou si la carte n'est pas éjectée, le testeur renverra le verdict FAIL qui signifie qu'une erreur s'est produite ; si le test n'a pas détecté d'erreur, le verdict PASS est renvoyé.

3-2 La notion de test abstrait

Comme il a été dit précédemment, un test concret est lié à une technologie. Ce lien peut parfois s'avérer très contraignant : si nous voulons tester des composants ayant des fonctionnalités identiques mais fonctionnant sur des technologies différentes, il faudra définir un ensemble de tests concrets pour chaque technologie. Ce type de contraintes peut être un frein à l'élaboration de meilleures campagnes de tests. Il semble donc intéressant de rajouter un niveau d'abstraction.
Dans le cadre du test fonctionnel, l'ingénieur doit s'appuyer sur la spécification du composant, si elle existe, pour synthétiser des tests. Cette spécification peut être exprimée à l'aide de différents langages allant de la langue naturelle à des langages de spécification tels qu'UML.
L'idée est de permettre à l'utilisateur de décrire ses tests au même niveau d'abstraction que la spécification. Ainsi, si la spécification ne tient pas compte de la technologie cible, l'utilisateur pourra définir des tests abstraits faisant abstraction des contraintes technologiques. Dans le cadre du projet COTE, nous exprimons ces tests abstraits en UML. Ceux-ci seront par la suite compilés vers la technologie choisie par l'utilisateur. La *figure 3* représente le test abstrait correspondant au test concret donné *figure 2*.

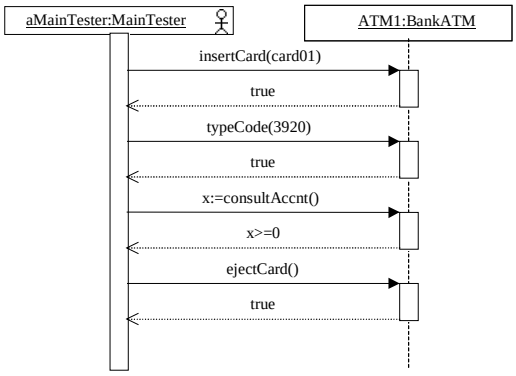

figure 3 : Un test abstrait

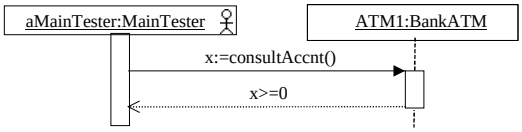

figure 4 : Un objectif de test

3-3 La notion d'objectif de test

Au moment de définir un test, l'ingénieur cherche à tester une certaine fonctionnalité qui apparaît dans la spécification à travers un ensemble de transitions. Un test abstrait représente un chemin complet dans la spécification comportementale du système. Un objectif de test a pour but de simplifier la conception d'un test en ne précisant que les transitions que l'on souhaite tester. On représente un objectif de test comme un ensemble ordonné de transitions de la spécification [3]. La *figure 4* décrit un objectif de test qui vise à s'assurer que le solde du compte est positif. L'outil UMLAUT/TGV part d'un tel objectif de test et d'une spécification comportementale de l'application et génère automatiquement un cas de test comme celui de la *figure 3*.

4 TOBIAS

Nous avons vu, dans la *section 3*, comment le fait de monter le niveau d'abstraction permettait de factoriser l'effort de test, et donc le travail de l'ingénieur de test. Si ce niveau d'abstraction peut sembler satisfaisant, nous montrons ici pourquoi il peut être utile de définir et d'utiliser un niveau encore plus élevé.

Certaines applications sont particulièrement critiques en termes de comportements. Il faut donc les tester de la façon la plus complète possible. Or, comme le montre H. Martin dans sa thèse [2], il faut pour cela décrire un grand nombre de tests.

L'outil UMLAUT/TGV permet de générer automatiquement un test abstrait à partir d'un objectif de test. Cet outil a néanmoins certaines limitations. D'une part, il est nécessaire que les paramètres des méthodes, dans les objectifs de tests, soient instanciés pour que l'outil puisse générer un test abstrait. D'autre part, l'outil ne génère qu'un seul test abstrait par exécution à partir d'un objectif de test.

Ainsi, l'ingénieur de test écrit autant d'objectifs de tests qu'il y aura de tests. C'est pourquoi nous proposons ici de décrire des schémas de tests qui sont des abstractions des objectifs de tests. Chaque schéma de tests permet de générer plusieurs objectifs de tests. Les schémas de tests, et l'outil TOBIAS qui leur est associé, permettent donc la génération de nombreux tests abstraits à partir d'une description synthétique.

4-1 Abstraction sur les valeurs des paramètres

La *figure 5* montre des objectifs de tests similaires aux valeurs des paramètres près. L'écriture de ces objectifs de tests peut s'avérer longue et fastidieuse, d'autant plus lorsque certaines méthodes peuvent comporter de nombreux paramètres. C'est pourquoi il nous semble intéressant de faire une abstraction sur les valeurs. L'utilisateur pourra définir à un niveau plus élevé, dans TOBIAS, les valeurs effectivement utilisées. Lorsque celui-ci souhaitera faire des tests avec de nouvelles valeurs, il n'aura qu'à les rajouter dans TOBIAS et n'aura pas besoin d'écrire de nouveaux objectifs de tests.

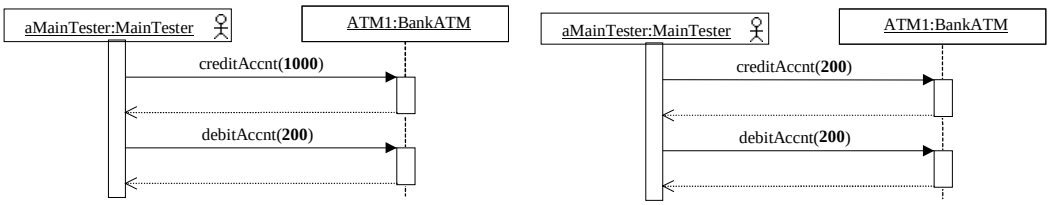

figure 5 : Deux objectifs de tests qui diffèrent sur les valeurs des paramètres

Un schéma de tests permettant de générer les objectifs de tests de la *figure 5* est :
ATM1.creditAccnt(*) ; ATM1.debitAccnt(200) *- ici le caractère '***** ' représente une abstraction sur les valeurs -*

4-2 Les groupes de méthodes

Les groupes de TOBIAS offrent à l'utilisateur un moyen de regrouper sous une même étiquette un ensemble de méthodes. La stratégie de regroupement dépend essentiellement des choix de l'ingénieur de test : il peut choisir de regrouper des méthodes de même signature, des méthodes appartenant à la même classe, des méthodes ayant les mêmes fonctionnalités ou selon tout autre critère lui semblant utile.
Nous pouvons par exemple définir un groupe ***opérations :*** *{creditAccnt(), debitAccnt(), consultAccnt()}*.

4-3 Abstractions sur les instances

Dans la *figure 6,* nous faisons appel aux mêmes méthodes mais sur différentes instances. Faire abstraction des instances de classe permettra là encore de factoriser l'effort de test. Par exemple, si l'utilisateur souhaite prendre en compte de nouvelles configurations d'instances sous test, il n'aura pas à écrire de nouveaux objectifs de tests.

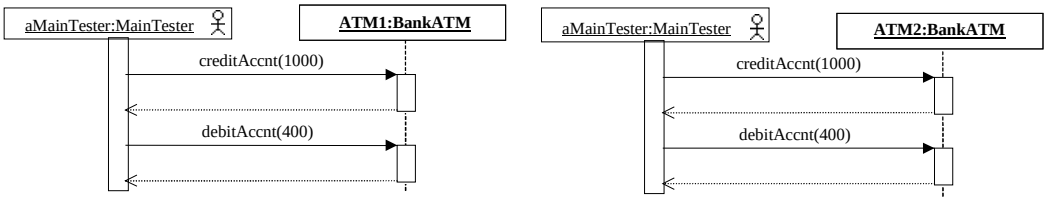

figure 6 : Deux objectifs de tests qui diffèrent sur les instances

Un schéma de tests permettant de générer les objectifs de tests de la *figure 6* est :
****.creditAccnt(1000);*. debitAccnt(400)*** *- ici le caractère '***** ' représente une abstraction sur les instances -*

4-4 Abstractions sur les boucles

Lors d'un test, il peut être nécessaire d'envoyer un message ou une séquence de messages plusieurs fois d'affilé. L'abstraction porte sur le nombre de répétitions. Par exemple, il peut être intéressant de faire un test pour un ou deux débits successifs. Chaque cas de figure va obliger l'utilisateur à écrire un nouvel objectif de test et ceci sans tenir compte des contraintes liées aux données.
La *figure 7* montre un cas où l'on souhaite tester le fonctionnement du distributeur lorsque l'on fait un ou deux débits successifs.

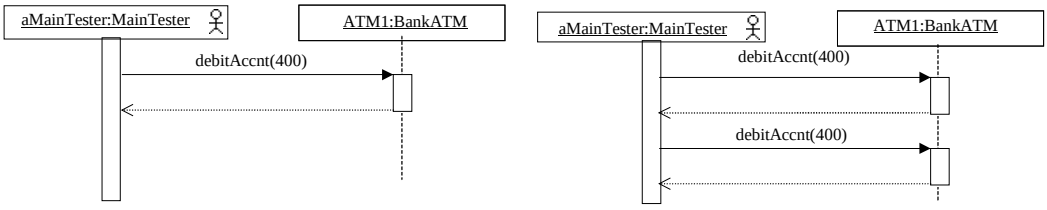

figure 7 : Deux objectifs de tests variant sur le nombre d'appel à une méthode

Un schéma de tests permettant de générer les objectifs de tests de la *figure 7* est : ***ATM1.debitAccnt(400)***$^{\wedge 1..2}$

5 Les principes généraux de TOBIAS

TOBIAS est un outil de description de schémas de tests pour aider à la génération d'objectifs de tests. Cette aide à la génération s'effectue en plusieurs étapes. La construction de schémas de tests est une première étape, la génération d'objectifs de tests à partir de schémas de tests en constitue une deuxième.
Nous allons présenter dans cette partie la création des schémas de tests et l'itération sur ces schémas de tests pour générer des objectifs de tests. Nous reprenons l'exemple de la *partie 2* pour illustrer nos propos.

5-1 Description des schémas de tests

La création de schémas de tests dans TOBIAS s'appuie sur différents diagrammes UML qui offrent différentes vues du système sous test. Dans la version finale, il est prévu que TOBIAS prenne en entrée trois types de diagrammes différents : un diagramme de classes, un diagramme de déploiement et des diagrammes d'états-transitions.

Le diagramme de classes permet de répertorier toutes les classes du système, ainsi que les différentes méthodes qui leurs sont associées avec leur paramètres. A partir de ces informations, on peut créer les groupes tels qu'on les a vus dans la *partie 4-2*. Le diagramme de déploiement présente les différentes instances qui composent le système étudié. Par la suite, TOBIAS offrira la possibilité de définir d'autres diagrammes de déploiement en accord avec le diagramme de classes. Quand aux diagrammes d'états–transitions, ils permettront à TOBIAS de gérer des stratégies de sélection de valeurs pour les paramètres des méthodes des classes dans des extensions futures.
La description des schémas de tests nécessite d'avoir extrait les instances des classes sous test dans le système (via les diagrammes de classes et de déploiement) ainsi que les méthodes qui leurs sont associées. On définit alors différents groupes avec les méthodes pour exprimer les schémas de tests.
Les schémas de tests sont exprimés dans un langage proche des expressions régulières avec comme éléments les groupes et les méthodes. Dans le cadre de l'exemple de l'ATM, nous pouvons définir le groupe ***opération*** dans lequel nous regrouperons les méthodes *creditAccnt(), debitAccnt()* et *consultAccnt()* de la classe BankATM. Nous pouvons alors définir le schéma de tests suivant : ***opération***$^{\wedge 1..2}$
Ce schéma de tests signifie qu'on fait appel une ou deux fois à une méthode appartenant au groupe ***opération***. Nous verrons que ce schéma de tests permet de générer les objectifs de tests de la *figure 7* mais aussi 154 autres objectifs de tests.

5-2 Des schémas de tests aux objectifs de tests

Après avoir décrit ses schémas de tests dans l'environnement TOBIAS, l'utilisateur peut générer des objectifs de tests de façon automatique. Cette génération se fait en rendant plus concrets les schémas de tests suivant plusieurs dimensions : les valeurs, les groupes, les instances et le nombre d'appels à un groupe ou une méthode.
En combinant ces différentes dimensions, on obtient l'ensemble des objectifs de tests pouvant être générés à partir du schéma de tests pris en compte. Reprenons le schéma de tests défini *section 5-1* et observons de quelle manière TOBIAS calcule les objectifs de tests correspondants.
Le système correspondant est formé de la façon suivante : il existe deux instances de la classeBankATM, ce sont les instances *ATM1* et *ATM2*. L'ensemble de valeurs défini pour *creditAccnt()* est {200, 1000}, celui de *debitAccnt()* est {100, 200, 400}.
Le principe du passage des schémas de tests aux objectifs de tests consiste à « déplier » le schéma de tests selon les abstractions définies *section 4*. En commençant par les boucles, on obtient des objectifs de tests avec un **ou** deux appels au groupe ***opération***. Ensuite on déplie suivant les groupes, on leur substitue leurs méthodes, pour ***opération*** on obtient *{creditAccnt(), debitAccnt(), consultAccnt()}*. Chaque méthode est associée à autant d'instances qu'il est possible, *creditAccnt()* sera ainsi associée aux instances *ATM1* et *ATM2*. Enfin, pour chaque méthode associée à une instance, on instancie l'ensemble de ses paramètres avec les ensembles de valeurs définis lors de la description des schémas de tests, pour *ATM1.creditAccnt()*, on a deux appels possibles à cette méthode de possible : *ATM1.creditAccnt(200)* et *ATM1.creditAccnt(1000)*. Pour connaître le nombre d'objectifs de tests pouvant être généré à partir d'un schéma de tests, il faut d'abord connaître le nombre d'objectifs de tests pouvant être généré pour chaque groupe composant ce schéma de tests, ici : ***Opération :*** (2x2 + 3x2 + 2)
Cette formule se lit comme étant 2 valeurs de paramètres possibles pour *creditAccnt()* pour 2 instances possibles, ou 3 valeurs de paramètre possibles pour *debitAccnt()* pour 2 instances possibles ou *consultAccnt()* pour 2 instances possibles. Le groupe ***opération*** permet ainsi de générer 12 objectifs de tests. En dépliant l'abstraction sur les boucles qui donnait un appel ou deux au groupe ***opération***, on obtient :

(opération) $\vee$ ***(opération ; opération)*** : 12 + (12 x 12)

Soit 156 objectifs de tests

6 Conclusion

Le projet COTE vise à créer un environnement pour générer automatiquement des tests exécutables à partir de tests décrits en UML. Dans le cadre de ce travail, nous avons constaté que monter le niveau d'abstraction des tests rendait plus rapide la phase de synthèse de tests quand le passage à un niveau plus concret était automatisé. Ce constat nous a amené à décrire les tests à un niveau d'abstraction très élevé : les schémas de tests. A ce niveau d'abstraction, il est possible de générer de nombreux tests à partir d'un seul schéma de tests. Pour gérer cette automatisation de la génération des tests et l'aide à la description des schémas de tests, un premier prototype de l'environnement TOBIAS sera disponible en décembre 2001.
Nous avons présenté au cours de cet article la principale fonctionnalité de TOBIAS qui est l'aide à la description de schémas de tests et la génération automatique des objectifs de tests à partir de ces schémas. Néanmoins, les différentes abstractions gérées lors de la création des objectifs de tests dans TOBIAS font vite apparaître le problème de l'explosion combinatoire. Un effort d'affinage de ces différentes abstractions devra être effectué pour générer un nombre raisonnable d'objectifs de tests les plus pertinents possibles. Cet effort pourra se faire de plusieurs manières : en posant des contraintes sur les schémas de tests, en choisissant mieux les valeurs des

paramètres grâce notamment à l'intégration de CASTING [1], ou encore en ne sélectionnant qu'un pourcentage des objectifs de tests pouvant être générés à partir d'un schéma de tests.
L'élaboration de TOBIAS nous amène aussi à réfléchir à d'autres problèmes qui pourraient être gérés à l'intérieur de l'outil tels que la définition de critères d'arrêt du test en intégrant, par exemple, des notions de couverture de test à partir des schémas de tests, ou la méthodologie pour mener et structurer une campagne de tests.

Remerciements :

Les auteurs remercient les autres membres du projet COTE : Softeam, l'IRISA, Gemplus et France Télécom.

Références :

[1] L. Van Aertryck, M. Benveniste, and D. Le Métayer. CASTING : A formally based software test generation method ; In The 1*st* Int. Conf. on Formal Engineering Methods, IEEE, ICFEM'97, Japan, Nov. 1997.

[2] H. Martin. Une méthodologie de génération automatique de suites de tests pour applets java-card ; Thèse de doctorat, Université de Lille 1, mars 2001.

[3] Y. Ledru, L. du Bousquet, P. Bontron, O. Maury, C. Oriat, M.-L. Potet. Test purposes : adapting the notion of specification to testing. In Proc. Automated Software Engineering, ASE 2001, USA, Nov. 2001.

[4] T. Jéron and P. Morel. Test generation derived from model-checking ; In N. Halbwachs and D. Peled, editors, CAV'99, Italy, vol. 1633 of LNCS. Springer, July 1999.

[5] W.-M. Ho, JM. Jézéquel, A. LeGuennec, and F. Pennaneac'h. UMLAUT : an extendible UML transformation framework ; In Proc. Automated Software Engineering, ASE'99, USA, Oct. 1999.