Validation dynamique de modèles UML avec points de variation sémantique

Frank Chauvel, Jean-Marc Jézéquel, et Didier Vojtisek

Abstract: Le Model-Driven Engineering, ou ingénierie des modèles, exploite les principes de la métamodélisation pour permettre à une entreprise de capitaliser son processus de raffinement depuis l'analyse jusqu'à la génération de code, en passant par l'application de design patterns, le tissage d'aspects extra-fonctionnels (persistance, sécurité, fiabilité, qualité de service, etc.). L'Unified Modeling Language (UML) est aujourd'hui incontournable lorsqu'il s'agit de modéliser des systèmes complexes en suivant une démarche orientée objets. Mais comme le méta-modèle UML permet des points de variation sémantique (i.e. différentes interprétations d'un jeu de concepts sont possibles), le processus de raffinement n'est plus linéaire mais arborescent, ce qui pose des problèmes ardus de validation de tels modèles, en particulier sur le plan comportemental. Nous avons donc construit un simulateur UML paramétré par un ensemble extensible de sémantiques ayant pour objectif de construire le système de transitions étiquetées correspondant à un modèle UML dans chacune de ses interprétations sémantiques possibles. Ce simulateur peut être couplé à toute une gamme d'outils exploitant ce formalisme, offrant ainsi la possibilité de vérifier par simulation certaines propriétés des modèles UML, ou encore de synthétiser des tests de conformité.

1 Introduction

Depuis sa standardisation par l'Object Management Group (OMG) en 1997, l'Unified Modeling Language (UML [9]) s'impose progressivement comme un standard de fait pour la modélisation à objets de systèmes, qu'ils soient logiciels, matériels ou organisationnels. UML est suffisamment complet pour pouvoir remplacer ou compléter les notations à objets l'ayant précédée, et sa souplesse permet de l'utiliser pour modéliser toutes les facettes d'un système, depuis la phase d'identification des besoins jusqu'au test et au déploiement final du système opérationnel.

Le Model-Driven Engineering, ou ingénierie des modèles, exploite les principes de la métamodélisation pour permettre à une entreprise de capitaliser son processus de raffinement depuis l'analyse jusqu'à la génération de code, en passant par l'application de design patterns, le tissage d'aspects extra-fonctionnels (persistance, sécurité, fiabilité, qualité de service, etc.).

Mais comme le méta-modèle UML permet des points de variation sémantique (i.e. différentes interprétations d'un jeu de concepts sont possibles), le processus de raffinement n'est plus linéaire mais arborescent, ce qui pose des problèmes ardus de validation de tels modèles, en particulier sur le plan comportemental. Tel est le cas notamment des systèmes répartis, dans lesquels la concurrence et l'asynchronisme rendent la compréhension du système particulièrement ardue.

Nous proposons ici d'adapter à UML ces techniques issues de la recherche sur les méthodes

formelles afin que les développements s'appuyant sur cette notation puissent en bénéficier. Il ne s'agit pas pour nous de traduire une spécification UML dans un autre langage formel afin d'utiliser directement les outils qui lui sont dédiés. En effet, bien que possible dans une certaine mesure, une telle conversion se heurte parfois aux spécificités propres au formalisme cible. Du fait de la richesse d'UML, cette conversion ne pourrait probablement pas se faire sans contorsion de la sémantique que nous souhaitons donner à UML. Par conséquent, nous donnons à UML une sémantique opérationnelle qui lui est propre, au travers du formalisme des *systèmes de transitions étiquetées* (Labelled Transition System ou LTS).

Nous avons donc construit un simulateur UML paramétré par un ensemble extensible de sémantiques ayant pour objectif de construire le système de transitions étiquetées correspondant à un modèle UML dans chacune de ses interpretations sémantiques possibles. Ce simulateur peut être couplé à toute une gamme d'outils exploitant ce formalisme, offrant ainsi la possibilité de vérifier par simulation certaines propriétés des modèles UML, ou encore de synthétiser des tests de conformité.

Le reste de cet article est organisé de la manière suivante : la section 2 présente le problème des variations sémantiques d'UML, en particulier au travers de la notion de statechart. Nous décrivons ensuite en section 3 comment la modélisation de ses points de variation sémantiques nous donne l'ossature d'un programme de transformation de modèle UML permettant de construire une famille de simulateurs UML s'appuyant sur le formalisme des systèmes de transitions. Un tel simulateur peut être couplé à des outils ouverts capables d'exploiter ces systèmes de transitions. La section 4 présente quelques techniques formelles que ce couplage nous a permis d'appliquer, comme la validation par simulation et la génération de tests.

2 Points de variation sémantique en UML

La norme UML n'a pas associé de sémantique complète à cette notation, laissant le soin aux utilisateurs de définir les éléments les plus spécialisés. Il existe donc un certain nombre de points de variation sémantique dont l'interprétation influe sur le comportement d'un modèle UML comme par exemple, la politique de sélection des évènements dans la file associée à un automate.
Le cas de l'héritage entre classes illustre parfaitement la notion de point de variation sémantique. La plupart des langages objets offre la possibilité de définir une relation d'*héritage* entre 2 classes. La sémantique associée à cette relation n'est pourtant pas la même dans tous les langages : certains langages utilisent un héritage contra variant, d'autres un héritage co-variant, d'autres encore, un héritage non variant. Comme UML intervient dans le cycle de développement en amont de la phase de codage, les concepteurs d'UML ont donc choisi de laisser aux utilisateurs le choix de la sémantique associée à la relation d'héritage : il s'agit donc bien d'un point de variation sémantique.

2.1 Quelques variations sémantiques des statecharts dans UML 2.0

Les statecharts tels qu'ils sont décrits dans la norme UML se résument à des automates avec comme extensions principales les hiérarchies d'états et la concurrence.

Déroulement du temps
La progression du temps (et celle de l'automate) peut être envisagée de deux façons opposées. Le temps peut être synchrone, c'est-à-dire équivalent à une horloge qui fait progresser

l'automate à chaque déclic, ou bien asynchrone, lorsque l'automate réagit instantanément aux évènements qui surviennent. Dans le cas synchrone, lorsque plusieurs évènements sont survenus à une date, aucune relation de précédence ne peut être établie entre eux. Ces évènements sont alors considérés comme simultanés, ce qui peut nécessiter d'établir une politique de sélection.

Sélection des évènements
Lorsque les évènements parviennent à un automate, ils sont stockés dans une structure de données dont la gestion n'est pas imposée dans les spécifications d'UML. Dans le cas général, cette structure est une file (FIFO) et la sélection d'un élément s'y fait par ordre d'arrivée. On peut toutefois envisager d'autres solutions telles que la pile (LIFO) pour sélectionner les éléments les plus récents, ou tout autre système de priorité. Lorsque le temps est synchrone et que plusieurs évènements sont présents dans la file, il n'existe pas, à priori, de relation d'ordre entre eux.

Durée des évènements
Les évènements discrets sont consommés instantanément par l'automate et ne peuvent être à l'origine que d'une et une seule transition. Par opposition, les évènements continus peuvent déclencher plusieurs transitions pendant leur durée de vie.

Priorités entre les transitions
L'utilisation d'états imbriqués amène généralement des conflits entre transitions. En effet une transition sortant d'un état peut entrer en collision avec celles sortant des sous états de son état source. Pour résoudre ce genre de conflit, UML propose de donner la priorité aux transitions les plus internes.

Interruptions préemptives
Sous l'hypothèse du temps synchrone plusieurs évènements peuvent survenir "en même temps" et provoquer un conflit entre plusieurs transitions. Lorsque le conflit met en jeu des états imbriqués, il peut être nécessaire de favoriser les *interruptions*, c'est-à-dire les transitions de plus faible profondeur dans la hiérarchie des états. Les transitions sont alors dites "préemptives".

Non déterminisme
Un automate peut également contenir des cas de non déterminisme "dur" où deux transitions partant du même état sont déclenchées par le même évènement. Sélectionner une transition de manière aléatoire permet de simuler un comportement équitable, mais il peut être également intéressant dans certains cas de fixer arbitrairement la transition à tirer.

2.2 Modélisation des choix sémantiques

Pour permettre de spécifier les différents choix sémantiques relatifs aux statecharts, il est nécessaire de disposer d'un méta modèle permettant d'exprimer les différents choix possibles. Ce méta modèle est présenté par la **Erreur ! Source du renvoi introuvable.**.
Le principal atout de cette solution est qu'elle permet une grande souplesse dans la définition de la sémantique associée aux statecharts. Les portions de code associées à la progression de l'automate ne sont donc pas figées et peuvent être facilement modifiées dans le modèle adéquat.

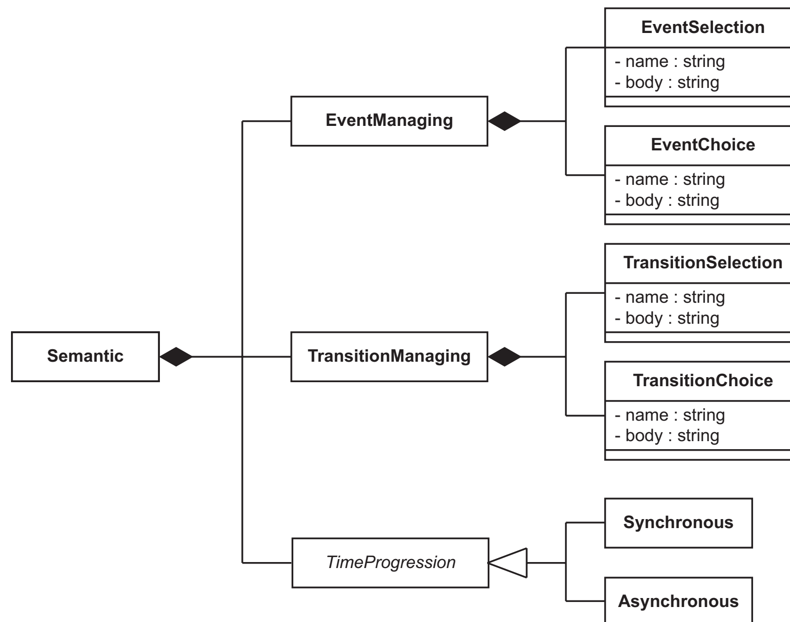

Figure 1. Métamodèle des points de variation sémantique

Un profil spécifique peut permettre de décorer les statecharts en précisant les choix associés à un automate. On pourra par exemple définir pour l'élément « StateMachine » du méta-modèle une tagged value nommée EventSelection. Cette valeur fera ainsi le lien entre le modèle de l'automate et le modèle de la sémantique.

3 Modélisation de la sémantique d'UML

L'approche que nous proposons consiste à transformer la spécification UML initiale en une version équivalente ne faisant usage que de constructions relativement simples de UML (telles les classes et les opérations), auxquelles il est plus aisé de donner une sémantique précise. Nous nous concentrons ici sur les transformations concernant les machines d'états.

3.1 Réification du comportement d'un automate

Le pattern *State* [3] propose une solution élégante pour réifier le comportement associé à un statechart. Le principe est d'isoler les comportements liés à un état particulier dans un objet. Les différents états de l'automate y sont représentés par une hiérarchie de classes ce qui permet notamment de gérer les états imbriqués.
Le pattern *Command* [3] offre une solution élégante pour représenter les évènements de l'automate et les organiser en hiérarchie. Il s'agit ici de représenter chaque évènement par une classe.
Les différentes solutions ne permettent cependant pas de représenter la notion de file de messages relative à la progression de l'automate. Le temps n'entre pas en jeu et les implantations traitent instantanément les occurrences des évènements.
Le pattern *Active Object* [7] permet de modéliser ceci. Il permet également de définir une facade unique pour l'automate. L'application du pattern *Active Object* présentée par la figure 2 n'est pas parfaitement fidèle. A l'origine, l'objectif de ce pattern est de séparer l'appel des méthodes d'un objet de leur exécution. L'objet qui joue le rôle du proxy doit donc construire un objet évènement pour chaque appel de méthode. Ici, ce pattern nous permet de représenter correctement les communications asynchrones avec un objet. L'objet proxy ne dispose donc

que d'une seule méthode dédiée à la gestion de ce type de communication, les autres appels de méthode (les appels synchrones) sont transmis au *servant* directement, sans être placés dans la file de messages.

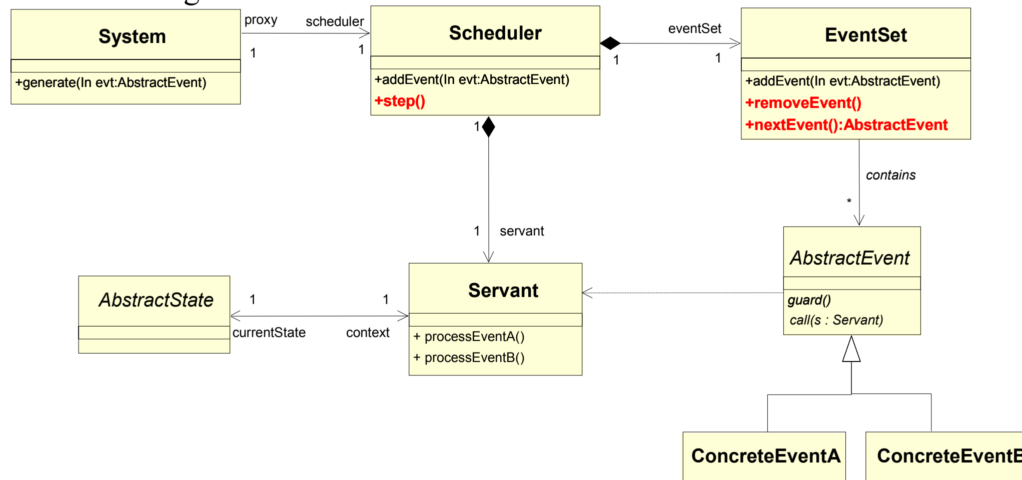

Figure 2. Pattern *Active Object*

3.2 Tissage des aspects sémantiques

L'intérêt du pattern *Active Object* est faire de apparaître la plupart des points de variation sémantique explicitement dans le modèle objet réifiant le comportement de l'automate. On voit ici que les notions de transformations de modèle et de tissage d'aspects [12] sont très proches. En effet, dans le cas des variations sémantiques, la transformation associée remplit le corps des méthodes spécifiques au pattern *Active Object*.
La question de la sémantique temporelle est représentée par la procédure step(). Dans le cas synchrone, le déclenchement de cette procédure est pris en charge par un tiers qui gère l'ensemble des objets actifs du système et les fait progresser simultanément. Dans le cas asynchrone, la procédure step() exécute instantanément le code associé à la transition déclenchée par l'évènement.

```
procedure step()
begin
     eventSet := eventPool.select();
     anEvent := eventSet.choice();
     transitionSet := getFirableTransition(event).select();
     aTransition := transitionSet.choice();
     aTransition.fire();
end.
```

La gestion des évènements est également facilitée par l'utilisation de ce pattern. En effet la sélection des évènements est prise en charge par une méthode. Faire varier la sélection des évènements revient donc à modifier le code de cette opération. Le cas des évènements différés est pris en charge par la méthode removeEvent() qui retire un évènement de l'ensemble. Elle pourrait également permettre de gérer les cas des évènements discrets et continus, même si ces cas ne concernent pas UML directement.
Le cas des transitions et du non déterminisme est plus difficile. En effet, sous l'hypothèse de l'utilisation du pattern *State*, le non déterminisme se traduit par plusieurs méthodes ayant la même signature (même nom, mêmes paramètres) au sein de la même classe; ce qui est évidemment impossible dans le monde UML. Ceci est dû au pattern *State* qui associe à chaque transition une méthode dans la classe représentant l'état source de la transition.

Pour disposer d’un mécanisme de sélection des transitions, il est nécessaire de réifier également le concept de transition. Ce qui amène vers une réification complète du méta-modèle des statecharts, qui permettrait une transition directe entre les modèles et le code.

4 UMLAUT et la boite à outils CADP

Le module de tissage décrit ci-dessus est intégré dans UMLAUT NG [6]. UMLAUT est un outil dédié à la manipulation de modèles dont les modèles UML. Il a été conçu dès le départ comme un outil ouvert, capable de lire des spécifications sauvegardées dans le format standardisé XMI et donc facilement interfaçable avec les ateliers de génie logiciel existants supportant la notation UML.
UMLAUT est un outil modulaire et extensible. Il dispose d'une bibliothèque de transformations pouvant être appliquées à des modèles UML. Enfin, UMLAUT donne accès aux techniques formelles grâce à son module de simulation, capable de compiler une spécification UML sous la forme d'un système de transitions étiquetées construit à la volée. Une spécification UML compilée de la sorte est ainsi directement exploitable par les outils de la boite à outils CADP [*2*], qui sont tous basés sur la même interface vers un système de transitions qu'implémente UMLAUT. CADP offre ainsi toute un gamme d’outils parmi lesquels, on peut citer un simulateur interactif permettant d'explorer le comportement du système et un *model checker* Nous ne décrirons dans cet article qu’un exemple de chaîne d’outils basée sur la synthèse de test avec TGV [5].

4.1 Le simulateur de spécification UML

Cette section présente le module de simulation de UMLAUT, dont le but est de traduire une spécification UML en un système de transitions, en s'appuyant sur les transformations présentées précédemment. Partant d'un état global de la spécification, le simulateur propose l'ensemble des transitions tirables depuis cet état.

Les transitions sont étiquetées par les actions *atomiques* exécutées par le système. Une action est toujours exécutée directement ou indirectement par un flot de contrôle précis (appelé "thread"). Si une action appartient à une opération d'un objet passif, elle ne peut être exécutée que si un objet actif lui transmet le contrôle par un appel d'opération ("nested flow of control" en terminologie UML).

Il est important de noter que le contrôle n'est pas forcément initié au sein du système. L'environnement peut également le stimuler, les acteurs étant des objets actifs pouvant envoyer des requêtes à des objets (actifs ou passifs) contenus dans le système. Un système purement réactif n'aura donc aucun comportement spontané en absence de stimulus provenant de l'environnement. Il faut donc également simuler un environnement capable d'offrir tous les stimuli possibles au système. On ferme donc le système en faisant des acteurs des *processus chaos*, c'est-à-dire en les munissant d'automates "marguerites" capables de produire tous les stimuli attendus par le système et capables d'accepter toutes les réponses renvoyées par celui-ci. Cela implique notamment de parcourir tout le domaine de valeur des types utilisés en paramètre des messages. Afin d'éviter des divergences inutiles, on supposera l'environnement *raisonnable*, c'est-à-dire qu'un message ne sera envoyé au système que si celui-ci est capable d'y répondre immédiatement. La version actuelle de UMLAUT n'automatise pas la fermeture du système, qui reste pour le moment à la charge de l'utilisateur. Cette contrainte devrait cependant être levée rapidement.

Un état global (encore appelé *configuration*) est donc constitué de :

- l'état de chacun des objets du système
- la topologie du réseau formé par les liens entre objets
- le "locus" de chaque flot de contrôle (et donc la pile d'exécution associée)

Lorsqu'une transition est tirée, un nouvel état global est extrait (par copie profonde de l'ensemble du réseau d'objets). Le simulateur offre aussi une fonction de comparaison d'états globaux permettant la détection de cycles dans le graphe d'accessibilité. Cette fonction de comparaison étant définie sur la base des fonctions de comparaison des objets constituants le système (états locaux), il est possible en redéfinissant ces fonctions de comparaison locales de réaliser des abstractions sur le graphe d'accessibilité. L'utilisation d'abstractions pertinentes peut ainsi conduire à une réduction significative de l'espace d'état tout en conservant certaines des propriétés du système.

4.2 Limitations actuelles et développements futurs du simulateur

Le simulateur inclus dans UMLAUT est encore en développement, et souffre donc de quelques limitations.

La première limitation concerne la spécification des actions exécutées par les objets. UML n'offre pas à l'heure actuelle de définition officielle pour les actions. Un groupe s'est formé qui travaille à la définition d'un langage d'actions et à sa formalisation, ce qui permettra à terme de combler cette lacune. Néanmoins, il est possible de pallier ce problème de plusieurs façons :

- en anticipant et en ajoutant ainsi à UML dès maintenant un ensemble limité d'actions très simples que l'on retrouvera forcément dans le futur langage d'actions, comme l'appel d'opérations et l'affectation d'expressions OCL
- en permettant l'utilisation de fragments de code écrits dans un langage de programmation existant (alternative proposée par certains ateliers de génie logiciel).

La seconde possibilité est souvent séduisante, mais les actions ainsi spécifiées restent bien souvent hors de portée des analyses que peut conduire un outil UML ; il est par exemple difficile de s'assurer de l'atomicité de telles actions, ou d'éviter certains effets de bord indésirables.

Enfin, la manière dont les états globaux sont stockés (par copie profonde) n'est pas optimale. En effet, les actions exécutées lorsque l'on tire des transitions ont un impact souvent très localisé (à un objet seulement). De plus, il est courant que la structure du réseau évolue peu (ou même pas du tout), alors que les états locaux des objets ainsi que les files de messages évoluent constamment. Il est donc probable qu'une factorisation des informations communes à plusieurs états apporte des gains importants en mémoire lors de la simulation.

Pour plus de détails, le lecteur pourra se référer à la thèse d'Alain Le Guennec [4]

4.3 Application à la génération de tests

Après avoir vérifié certaines propriétés de la spécification, il est important de pouvoir s'assurer qu'une implantation finale du système sera effectivement conforme à sa spécification. L'outil TGV [5] peut être utilisé dans ce but. En effet, cet outil permet de générer des cas de tests à partir d'une spécification et d'un *objectif de test*. L'objectif de test guide la génération du cas de test pour produire des cas de tests ciblés, en restreignant l'exploration du graphe d'accessibilité aux seules transitions que l'on souhaite autoriser. La théorie sous-jacente à TGV se fonde sur les systèmes de transitions étiquetées pour lesquels entrées et sorties sont distinguées. Les modèles et algorithmes de TGV sont présentés en détail dans [5] et nous ne nous étendrons pas davantage sur cet aspect dans cet article.

Pour pouvoir générer des tests, TGV a donc besoin d'objectifs. La spécification UML du système fournit déjà des objectifs de tests intéressants sous une forme abstraite : les cas d'utilisations. Un cas d'utilisation peut ainsi être décrit plus précisément par une collaboration et un diagramme de séquence représentant les interactions entre l'environnement et le système qui concernent directement la réalisation de ce cas d'utilisation. UMLAUT convertit ensuite ces interactions en objectifs de test pour TGV c'est-à-dire des patrons de test IOLTS.

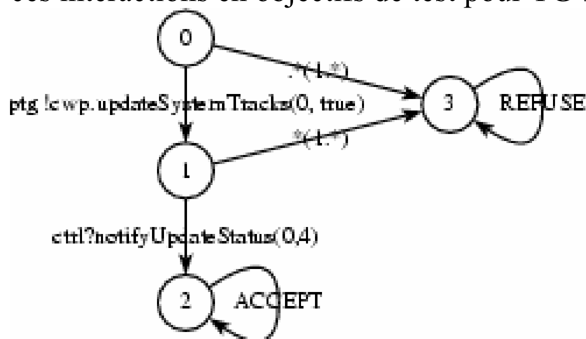

Figure 3. exemple d'objectif de test IOLTS

Pour obtenir un test, il faut commencer par identifier chaque intervenant (rôle) dans l'interaction parmi les objets du système déployé. Les points d'interactions entre objets acteurs et objets internes au système deviennent les points de contrôle et d'observation du point de vue du testeur. Seuls les messages transitant par ces points sont observables, les messages internes au système n'étant pas accessibles pour le testeur.

L'interaction dans laquelle on connaît l'identité de tous les objets jouant un rôle constitue alors la trame de base de l'objectif de test. Cet objectif de test est représenté sous la forme d'une séquence d'évènements. Comme ce n'est qu'une projection partielle du comportement, d'autres événements qui ne concernent pas directement ce cas d'utilisation peuvent éventuellement survenir pendant sa réalisation. Une transition implicite étiquetée avec "*" permet d'indiquer que tous les événements non-mentionnés sont autorisés.

Les transitions menant dans l'état `REFUSE` de l'objectif de test permettent d'élaguer l'exploration du graphe lors de la construction des cas de test.

Enfin UMLAUT traduit les cas de test IOLTS en cas de test UML sous la forme de diagrammes de séquence HMSC (High level Message Sequence Chart) qui seront plus facilement compris par le testeur qui vérifiera l'implantation finale.

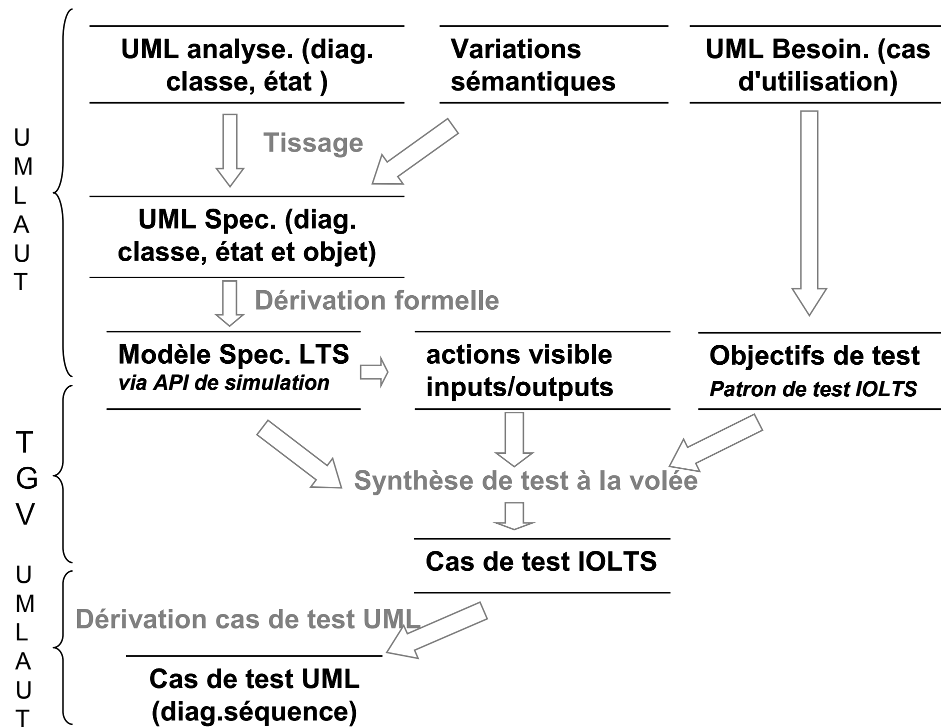

Figure 4. Gestion de la variation sémantique et génération de cas de test avec UMLAUT NG

L'approche du test utilisée par TGV ne traite pas les données de manière symbolique (les valeurs effectives apparaissent au sein des labels des transitions). D'autres travaux complémentaires comme [8], basés sur des approches symboliques, abordent le problème de la génération de données pertinentes vis-à-vis d'un critère de couverture des comportements de la spécification. Néanmoins le simulateur n'étant pas lié à un outil en particulier, il est tout à fait possible de le connecter à d'autres outils utilisant d'autres techniques.

Pour plus de détails sur les techniques de génération de test abordées ci-dessus, le lecteur pourra se référer aux travaux décrits dans la thèse de Simon Pickin [10].

6 Conclusion et perspectives

Grâce aux techniques issues du MDE et plus particulièrement à celle de tissage d'aspects, il nous a été possible de réifier le comportement des modèles UML en systèmes de transitions étiquetées. Nous avons ainsi montré que la variabilité dans un modèle UML n'est pas un frein à l'utilisation des méthodes formelles dans un contexte de validation et de vérification de ce modèle.

Il serait maintenant intéressant de compléter la chaîne par l'intégration d'outils fournissant d'autres techniques de validation issues des méthodes formelles comme par exemple la génération de test symboliques, ou le modelchecking.

De même, on peut améliorer les phases amonts en offrant une aide à la conception des objectifs de test, ou mieux encore, en automatisant partiellement cette tâche.

References

[1] D. D'Souza and A. Wills. *Objects, Components and Frameworks With UML: The Catalysis Approach*. Addison-Wesley, 1998.
[2] H. Garavel. Open/caesar: An open software architecture for verification, simulation and testing. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384. Springer-Verlag, Lecture Notes in Computer Science, 1998.
[3] E. Gamma, R. Helm, R. Johnson et J. Vlissides. Design Patterns: Elements of reusable Object Oriented Software. *Addison-Wesley Longman Publishing Co., Inc. 1995*
[4] A. Le Guennec. Génie logiciel et méthodes formelles avec UML : Spécification, validation, génération de tests. PhD thesis, Ecole doctorale MATISSE, Université de Rennes 1.
[5] T. Jéron and P. Morel. Test generation derived from model-checking. In Nicolas Halbwachs and Doron Peled, editors, *CAV'99, Trento, Italy*, pages 108-122. Springer, LNCS 1633, July 1999.
[6] JM. Jézéquel, WM Ho, A. Le Guennec, and F. Pennaneac'h. UMLAUT: an extendible UML transformation framework. In Robert J. Hall and Ernst Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999. Additional information on http://modelware.inria.fr/mtl
[7] R. G. Lavender and D. C. Schmidt, Active Object: an Object Behavioral Pattern for Concurrent Programming. Volume2 of Pattern Languages of Program Design *Addison-Wesley Longman Publishing Co., Inc. 1996*
[8] J. Offutt and A. Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 416-429. Springer, 1999.
[9] Object Management Group (OMG): *Unified Modelling Language Specification, version 1.4*. OMG, Needham, MA, USA. Sep. 2001.
[10] S. Pickin. *Test des composants logiciels pour les télécommunications*. PhD thesis, Université de Rennes 1, 2003.
[11] G. Övergaard. A formal approach to collaborations in the unified modeling language. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723 of *LNCS*, pages 99-115. Springer, 1999.
[12] W.M. Ho, J.-M. Jézéquel, F. Pennaneac'h, and N. Plouzeau. A toolkit for weaving aspect oriented UML designs. In *Proceedings of 1st ACM International Conference on Aspect Oriented Software Development, AOSD 2002*, Enschede, The Netherlands, April 2002.