Un cadre pour la vérification de modèles UML

P. André, G. Ardourel, G. Sunyé

Laboratoire d'Informatique de Nantes-Atlantique
2, rue de la Houssinière
B.P. 92208
44322 NANTES CEDEX 3
Prenom.Nom@univ-nantes.fr

— UML, modèles, architectures de logiciels, gestion et vérification de propriétés —

RAPPORT DE RECHERCHE

N° 04.12

Janvier 2005

P. André, G. Ardourel, G. Sunyé

Un cadre pour la vérification de modèles UML

18 p.

Les rapports de recherche du Laboratoire d'Informatique de Nantes-Atlantique sont disponibles aux formats PostScript® et PDF® à l'URL :
http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html

Research reports from the Laboratoire d'Informatique de Nantes-Atlantique are available in PostScript® and PDF® formats at the URL:
http://www.sciences.univ-nantes.fr/lina/Vie/RR/rapports.html

Un cadre pour la vérification de modèles UML

P. André, G. Ardourel, G. Sunyé

Résumé

Ce rapport de recherche pose les bases d'un cadre formel pour la vérification sur des modèles UML de propriétés telles que la correction, la cohérence, la sûreté de fonctionnement... Actuellement, cette vérification est essentiellement syntaxique et repose sur des outils UML disparates. Dans ce papier, nous proposons un cadre pour la vérification de modèles UML, basé sur une structuration des modèles, une combinaison de techniques de vérification, qui s'intègre au développement itératif et incrémental. Une expérimentation de cette approche est actuellement réalisée sur un outil disponible en source libre

Catégories et descripteurs de sujets : D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Termes généraux : UML, Vérification de modèles

Mots-clés additionnels et phrases : Composants, Services, Spécification formelle, Conformité d'interaction

1 Introduction

La modélisation à objets avec UML [13, 7] est maintenant une pratique courante dans le développement de logiciels. En pratique, la notation est utilisée essentiellement pour l'expression des besoins, l'analyse/conception du système et la documentation des applications. De manière générale, on constate une sous-utilisation de la notation, soit parce que produire des modèles est fastidieux et que peu d'outils permettent leur exploitation, soit par manque d'expérience sur les capacités de la notation, soit parce ce que cette dernière n'est pas adaptée à ce qu'on souhaite représenter. Il en résulte une faible qualité des produits (modèles, spécifications) du développement. Ces produits sont peu lisibles, précis, homogènes, communicables, cohérents, complets. Plusieurs facteurs peuvent expliquer cette situation :

1. Les modèles dépendent du processus et non de la notation, autrement dit chacun définit quelle combinaison d'éléments de notation constitue un modèle. Actuellement, il n'existe pas un processus unifié normalisé.
2. La notation est conséquente et complexe. La plupart des acteurs du développement n'utilisent qu'une partie de la notation (principalement les éléments relatifs aux cas d'utilisation, aux classes, parfois aux séquences et aux automates). A l'inverse, malgré la large couverture des concepts d'UML, il est nécessaire de personnaliser la notation soit par des stéréotypes, soit par de nouveaux profils d'UML.
3. Il n'existe pas à l'heure actuelle de sémantique communément agréée, seules quelques règles de combinaison sont fournies par le métamodèle. On constate diverses interprétations d'UML, liées à l'expérience en modélisation à objets, en programmation et aux domaines d'application (systèmes d'information, temps réel, simulation...).

Pour améliorer la qualité les modèles produits, il faut les contrôler. En faisant abstraction des critères liés à la gestion de projets (coûts, rentabilité), le contrôle des modèles UML se focalise sur deux points clés : le modèle est-il valide, le modèle est-il correct ? Le système est valide s'il répond au besoin de l'utilisateur et s'il est conforme avec l'environnement cible. La validité s'obtient par des révisions, des inspections, du prototypage. La vérification consiste à certifier la correction et la cohérence des modèles du développement. Aux techniques utilisées dans la validation s'ajoutent les tests et les preuves.

Dans ce papier, on s'intéresse au problème de la vérification de propriétés des modèles UML produits au cours du développement. L'ensemble des diagrammes et de la notation est donc pris en compte, de même que son usage dans la pratique du développement. Cela signifie par exemple qu'il est tenu compte :

- des différents **niveaux d'abstraction** relatifs aux activités de développement : analyse des besoins, analyse, conception, programmation... Avec un développement "sans coutures", la même notation est utilisée pour différents niveaux d'abstraction.
- des différentes **versions**. Dans un développement *itératif* et *incrémental* (*e.g.* RUP [14], Catalysis [4]). Les versions d'itération n+1 des modèles sont plus complètes que celles du niveau n.
- des **aspects** du système, différents mais non-disjoints du système (spécifications hétérogènes). Par exemple, un automate complète la description d'une classe par des aspects dynamiques, une collaboration doit être conforme à la description des classes (et de leurs automates) des objets de la collaboration.

Les propriétés à vérifier se classent en trois catégories distinctes :

- Les **propriétés du système** dépendent de l'application cible. Parmi les propriétés générales se trouvent la validité, la sûreté, la vivacité, la fiabilité... Les propriétés du système se classent parfois selon trois points de vue : *statique* ou structurel, *dynamique* ou comportemental et *fonctionnel* ou opérationnel.
- Les **propriétés du modèle** sont relatives aux descriptions multi-vues. Ce sont principalement la cohérence et la complétude des vues (ici des diagrammes UML) du modèle.
- Les **propriétés du processus** se focalisent sur l'évolution du système au cours du développement. Ce sont principalement la cohérence et la complétude des versions de modèles dans une itération et entre itérations. Le problème est relativement complexe, et peu de solutions existent hormis la traçabilité et quelques règles d'évolution.

Vérifier l'ensemble de ces propriétés est un travail ardu à cause du nombre de concepts de la notation, des interprétations variées (absence de sémantique précise) et de l'incomplétude inhérente au style de développement (incréments). Quelle que soit la cible de la vérification, le processus de vérification se déroule en trois étapes : construire les modèles, exprimer les propriétés, vérifier (automatiquement) les propriétés sur les modèles.

Nous proposons un cadre pour la vérification qui prend en compte la complexité de la notation en la découpant en domaines et niveaux d'abstraction, ce qui facilite l'établissement des règles et leur vérification. Pour appréhender la complexité de la vérification, nous basons le cadre de la vérification sur les quatre principes suivants :

1. **Réduire l'espace combinatoire de la notation**. En limitant l'usage de la notation à certains domaines et en structurant les modèles, la vérification se divise en sous-problèmes plus restreints. Un effet de bord est une meilleure lisibilité et cohérence des modèles produits lors du développement.
2. **Mixer les approches de vérification**. Les approches étant complémentaires, il semble judicieux de trouver un cadre qui permette de les utiliser conjointement.
3. **Gérer l'incomplétude**. Le développement itératif et incrémental étant par nature incomplet[1], il faut adapter le niveau d'exigence attendu de la vérification au processus de développement et pas seulement aux modèles ou aux éléments de notation.
4. **Conserver l'indépendance aux AGL et s'adapter aux évolutions des standards**. Les modèles pouvant être décrits dans des versions différentes de la norme (sans compatibilité ascendante) il faut s'y adapter. Inversement, il est risqué de lier l'outil de vérification à quelque outil AGL du fait de la pérénité et de l'évolutivité de ces outils.

Le papier est organisé selon ces quatre principes. Dans la section 2, nous étudions la réduction de l'espace combinatoire par une structuration des modèles, la source de la vérification. Puis, nous fixons le cadre général de la vérification, en mixant les approches à base de règles et les approches formelles dans la section 3. Le problème de l'incomplétude est traité par la hiérarchisation des propriétés et des vérifications dans la section 4. Enfin, l'indépendance avec les AGL et l'adéquation à l'évolutivité des normes se fait par une implantation dans l'outil Bosco (section 5).

2 Structuration des modèles

Dans le contexte de la vérification, on considère qu'un modèle est une combinaison structurée d'éléments de modélisation (les briques de base d'UML). Un modèle est le résultat d'une activité du processus de développement. Sa structuration est donc implicitement liée au processus ou aux outils qui le mettent en œuvre. Puisque le processus n'est pas standard, vérifier un modèle revient à le vérifier dans chaque outil de chaque méthode. Pour éviter ce problème, il semble intéressant d'associer une structuration[2] standard à la notation et y faire référence dans le processus. Dans la notation UML, un modèle est un paquetage (cf. *UML Metamodel* [7], p. 2-184). Nous proposons de promouvoir la notion de diagramme, considérablement utilisée dans les ouvrages décrivant la notation, dans le métamodèle. Le diagramme limite l'espace des combinaisons d'éléments de modélisation, ce qui n'était que partiellement vérifié par les règles (*well-formedness rules*) informelles ou OCL, du métamodèle, appelées abusivement sémantique d'UML. Ainsi, nous proposons de structurer la notation en trois niveaux : **domaine**, **diagramme**, **élément de modélisation**.

Un *domaine* représente un niveau d'abstraction dans le processus de développement. Inversement, à chaque activité du processus de développement correspond un domaine : les modèles du processus sont donc basés sur les domaines. On réduit ainsi l'usage de la notation en fonction de l'activité, ce qui facilitera évidemment la vérification.

- Le domaine externe, relatif à l'analyse des besoins, se focalise sur la notation à haut niveau d'abstraction (cas d'utilisation, scénarios[3]). Nous préconisons de ne pas utiliser de diagrammes d'activités avec couloirs à ce niveau, car ils induisent un découpage données/traitements peu compatible avec une modélisation à objets.
- Le domaine logique, relatif aux activités d'analyse et de conception, se focalise sur une structuration logique du système, indépendante de l'implantation. On y retrouve les éléments clés d'une modélisation à objets : classes, objets, séquences, collaborations, automates, activités.
- Le domaine physique, relatif à l'implantation et aux tests, se focalise sur la notation à bas niveau (composants, déploiement, code).

[1] Selon Jacobson [14], seule l'implantation finale donne la sémantique des modèles produits.
[2] Il s'agit de structuration horizontale, de type composite, et non verticale (héritage).
[3] Les scénarios sont des diagrammes de séquences restreints aux interactions entre le système et les acteurs de son contexte.

Chaque domaine regroupe un ensemble de diagrammes. Ceux-ci sont classés en diagrammes d'instances et diagrammes de types, car les vérifications associées sont différentes. Chaque diagramme regroupe des éléments de modélisation. La notion de diagramme, qui représente une unité de cohérence fondamentale dans la notation puisque c'est elle qui permet de spécifier les propriétés du système, est donc conservée et devra être « promue »dans le métamodèle. En pratique, cette promotion se fait par une annotation ou un stéréotype pour les paquetages concernés. Cette annotation s'obtient naïvement par une inférence sur un échantillon des éléments du paquetage. La structuration proposée est résumée dans la figure 1.

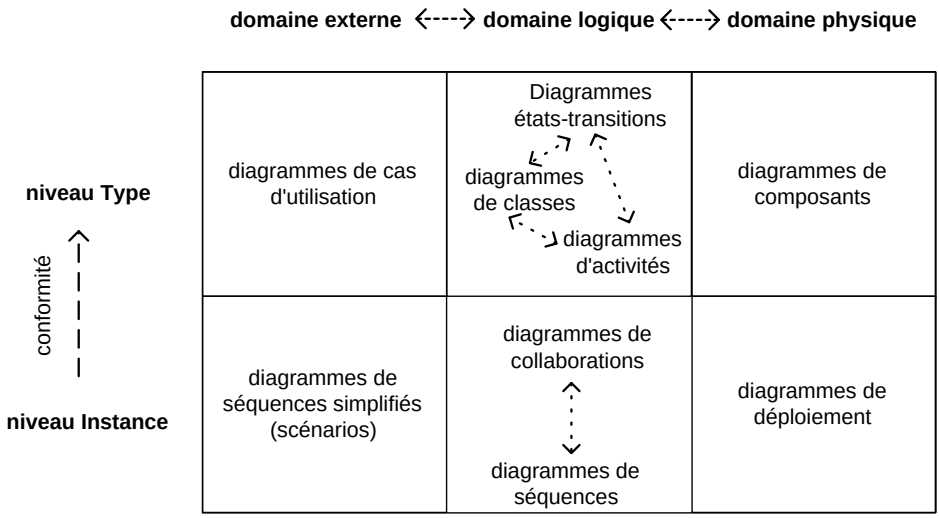

Figure 1 : *Structuration des domaines*

Chaque diagramme (hormis les scénarios) apparaît dans une seule cellule, il s'agit dont d'une partition de la notation. Une telle structure limite l'expressivité mais améliore l'apprentissage de la notation et donc son usage. Les relations sont multiples : des instances aux types (conformité), entre diagrammes (cohérence, complétude...), entre domaines (raffinement, traçabilité...). Ces relations supportent le processus de vérification.

3 Un cadre pour la vérification

La vérification de modèle UML se fait en général selon l'une des deux approches suivantes : les méthodes formelles et les systèmes à base de règles. On trouvera dans [9, 8] des exemples relatifs à ces approches pour la vérification de cohérence.

3.1 Comparaison des approches de vérification

Dans l'approche par méthodes formelles, l'objectif est de définir un modèle formel sur lequel on peut ensuite exprimer et vérifier des propriétés. Ce modèle peut être une théorie mathématique (logiques, algèbres, λ- ou π-calculs, systèmes de réécriture, automates, réseaux de Petri...). C'est une réponse au manque de sémantique précise d'UML [12, 15]. Dans les années 90, la tendance était à la définition d'un langage formel à objets *e.g.* Object-Z, OOZE, Z++, VDM++, Troll, Maude, etc [2]. La combinaison avec UML est un domaine de recherche actif depuis 1997, avec la création du groupe *pUML* [5], dont l'objectif affiché est de fournir un manuel de référence formel pour la notation, c'est-à-dire d'associer à chaque concept d'UML une correspondance dans un domaine sémantique formel. Les travaux varient selon le niveau d'abstraction (modèle ou métamodèle), les aspects pris en compte (statiques, dynamiques, fonctionnels), la couverture de la notation et les outils de preuve. Une des difficultés majeures est que les approches formelles occupent souvent des niches, c'est-à-dire qu'elles sont adaptées à certains des trois aspects mais pas à tous. Ainsi, on distingue les approches à dominante dynamique (systèmes de

transitions, algèbres de processus) des approches orientées modèles (Z, B) ou des approches à dominante fonctionnelle (spécifications algébriques). Notons de plus que le processus de développement formel, qui est en principe linéaire, s'accorde assez mal avec un processus itératif et incrémental. Ainsi, travailler avec des modèles volontairement incomplets pose forcément des problèmes pour les méthodes formelles. Enfin, il existe des sémantiques opérationnelles dans les versions exécutable d'UML [10, 11], qui sont utiles pour le prototypage et la validation mais ne s'intègrent pas vraiment avec l'idée de vérifier des modèles UML, puisqu'on veut le faire avant codage.

Dans l'approche des systèmes à base de règles, une règle décrit un moyen de vérifier une (partie d'une) propriété. La description peut être "formelle" (OCL, algorithme, code Java...) ou informelle. C'est l'approche choisie dans les AGL pour implanter des algorithmes de vérification, qui travaillent sur une représentation interne (non connue) des modèles UML. Ces algorithmes sont souvent confidentiels mais des exemples de systèmes à base de règles sont donnés dans [16, 9, 8]. La principale difficulté est la gestion d'un ensemble conséquent de règles, sa cohérence et son degré de complétude. L'avantage de cette approche est qu'elle se marie bien avec un développement itératif et incrémental, bien que nous n'ayons pas observé que ce problème soit traité en pratique.

La table 1 compare ces deux approches. Le *modèle* est la source de la vérification. On ne distingue pas modèle UML et métamodèle ici. La partie *propriétés* indique quelles sont les propriétés vérifiées et comment elle sont décrites. La partie *vérification* indique comment les propriétés sont vérifiées sur les modèles. On constate que les deux approches sont complémentaires. Pour pouvoir les combiner, il est nécessaire de disposer d'un cadre permettant de considérer les vérifications indépendamment de leur approche.

	Approches formelles	**Systèmes à base de règles**
Modèle		
description	un modèle formel	un modèle
notation couverte	partiel	total
Propriétés		
notation	expressions formelles (e.g prédicats)	expressions semi-formelles
utilisateur	expressions formelles	peu extensible
Prop. du système	oui, selon la sémantique cible	non
Prop. du modèle	oui	oui
Prop. du processus	raffi nage	traçabilité, évolution, transformation
Vérification		
processus	automatique pour les propriétés du modèle individuel pour les propriétés du système	sélection/activation -
vérification	preuve formelle	algorithme (ou revue si informel)
Utilisateur		
bases	UML, spécifi cation formelles	UML
rôle	défi nitions et preuve de certaines propriétés	sélection des règles
expérience	théorie de la preuve	maîtrise de la base de règles
Avantages	sémantique formelle (non-ambigu, consistant...) (raffi nage vers le code)	adaptable (selon l'étape de développement) (selon la sémantique de l'application)
	automatisation	seulement UML
	bien fondé	programmation et tests
Inconvénients	lours, expérience	dépend de l'implantation
	notation UML partielle	non fondé (cohérent, complet ?)
	deux modèles (source + formel)	sélection implique expérience

TABLE I– Synthèse comparative

3.2 Définition d'une vérification

A chaque propriété peuvent correspondre plusieurs vérifications. Par exemple, la propriété de cohérence d'un modèle du domaine logique suppose notamment les vérifications suivantes :

– conformité des messages envoyés à un objet dans un diagramme de séquence avec ceux que le type de l'objet supporte et ceux que l'objet émetteur est autorisé à envoyer,

– conformité de la séquence de messages reçus avec le diagramme d'états de la classe de l'objet receveur.
– cohérence entre les actions d'un diagramme d'états avec sa classe.

Une vérification est définie par un nom, un but (commentaire informel), un espace de travail. Elle peut être réalisée à l'aide de **règles** (algorithmes, prédicats, code) ou de **preuves** dans un modèle formel. Une bonne pratique est de définir, si possible, les règles par des expressions OCL (*Object Constraint Language*) pour conserver un cadre homogène.

L'*espace de travail* correspond à la partie du métamodèle qui est utilisée ou concernée par la vérification. Dans le cadre d'UML, ce métamodèle est celui qui décrit UML, appelé niveau M2 par l'OMG. L'espace de travail est constitué de :

– l'ensemble U des éléments du métamodèle utilisés, c'est-à-dire ceux qui sont nécessaires à la réalisation de la vérification, que l'on partitionne en U_v, ensemble des éléments à vérifier, et U_n ensemble des éléments qui ne sont pas à vérifier
– l'ensemble C des éléments du métamodèle concernés mais non utilisés, dont les instances devront être considérées invalides si l'un des éléments qui les compose est jugé invalide.

U est une composante dépendante de la vérification, alors que sa partition en U_v et U_n, ainsi que le choix de C sont dépendants de l'intégration au processus de vérification. Par exemple, pour la vérification de conformité de diagrammes de séquence avec un diagramme de classe, les éléments de U sont les diagrammes de classe et de séquence[4], les classes, interfaces, méthodes, objets et messages ; les éléments de U_v sont les messages, et ceux de C les diagrammes de séquence[4].

Les traitements associés à une vérification sont :

– collecte des éléments de M1 instances de U et conversion éventuelle,
– exécution de la règle ou de la preuve associée à la vérification,
– interprétation du résultat et annotation des éléments du modèle d'origine,
– répercussion du résultat sur les éléments instances de C.

Collecte

La réalisation d'une vérification d'un modèle M1[5] nécessite de fournir les éléments $U1$ de ce modèle qui sont instances de U. $U1$ est partitionné en $U1_v$ et $U1_n$ en suivant la partition de U en U_v et U_n. Ces instances sont ensuite éventuellement converties dans le format nécessaire à l'exécution de la vérification.

Interprétation

Le résultat d'une vérification peut être vu comme une partition des éléments de $U1_v$ en deux ensembles $U1_{vt}$ et $U1_{vf}$ qui contiennent respectivement les éléments pour lesquels la propriété est vérifiée et ceux pour lesquels elle ne l'est pas. Les éléments de $U1_{vf}$ sont **annotés** comme invalides *pour cette vérification* dans le modèle M1'. Dans le cas où la vérification est réalisée par un outil dont la réponse est peu détaillée (ex : un booléen), le soin que l'on aura pris d'identifier les éléments vérifiés et concernés (U_v et C) permet de répercuter le résultat de manière plus fine.

Répercussion

Les éléments de M1 qui sont instances de C et qui contiennent des éléments invalidés pour la vérification sont également annotés. Le décision de considérer un élément invalide *pour toute vérification* est dépendante du processus et sera abordée plus loin.

[4]Comme indiqué précédemment, les paquetages contenant des diagrammes sont supposés annotés et nous les appellerons diagrammes par abus de langage.

[5]Plus précisément, il s'agit de la sélection d'une partie d'un modèle de niveau M1 selon la classifi cation de l'OMG. Le modèle d'analyse dans le processus unifi é est un exemple de modèle de niveau M1.

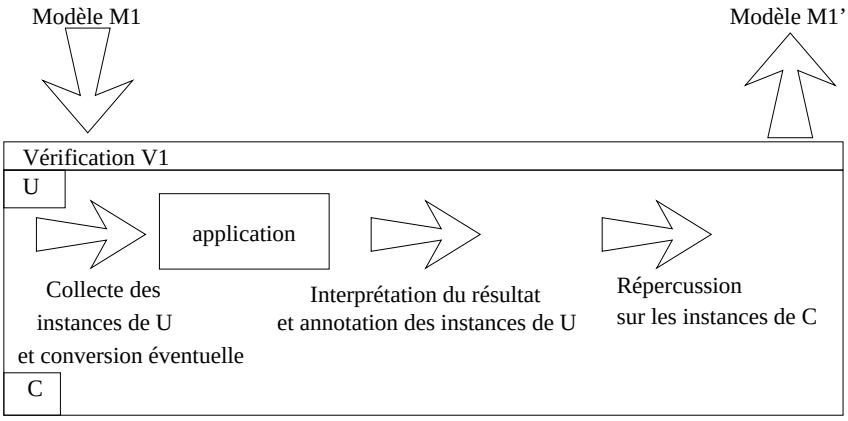

Figure 2 : *Application d'une vérification sur un modèle*

3.3 Processus et composition de vérifications

Dans une vérification implantée par un système à base de règles, l'utilisateur choisit les règles (listes, choix multiple) et lance la vérification [16]. Le moteur prend ensuite les règles et les vérifie *indépendamment* sur le modèle en entrée. Cela implique une connaissance approfondie de la part de l'utilisateur des règles à vérifier et des enchaînements raisonnables. On souhaite fournir une aide à la fois au concepteur de vérification et aux utilisateurs, via la notion de processus de vérification.

Un processus de vérification consiste en l'application successive de fonctions opérant sur des modèles. Ces fonctions peuvent être des **vérifications** ou des **filtrages**. Le filtrage le plus simple consiste à éliminer tout élément qui a été annoté comme invalide pour une vérification donnée. L'ordre d'enchaînement des vérifications change le résultat si des filtrages sont introduits. En effet, une vérification $v1$ peut invalider un élément en entrée d'une vérification $v2$ si $U(v2) \cap (U_v(v1) \cup C(v1)) \neq \emptyset$. Lorsque des vérifications sont en dépendances croisées, choisir laquelle doit influer sur l'autre est une décision importante du processus.

La figure 3 illustre l'application d'un processus de vérification simple.

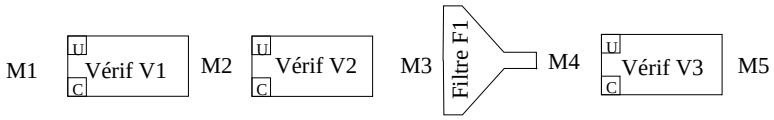

Figure 3 : *Application d'un processus de vérifications à un modèle M1*

Pour systématiser la gestion des vérifications et la création de processus de vérifications, nous introduisons un opérateur de composition qui permet de définir une vérification à partir d'une application successive de fonctions :

comp(nom, but, liste de fonctions)→ verifi cation(U, C).

La vérification obtenue possède un ensemble U qui est l'union des U des vérifications contenues dans la liste. Un ensemble C peut y être ajouté, mais est vide par défaut. Les traitements associés à une vérification issue de composition sont les suivants :

- collecte : aucun filtrage nécessaire
- exécution : application des fonctions contenues
- interprétation du résultat : la valeur de l'annotation pour la vérification composite est obtenue par un prédicat portant sur la valeur des annotations des vérifications contenues
- répercussion du résultat sur les élements instances de C

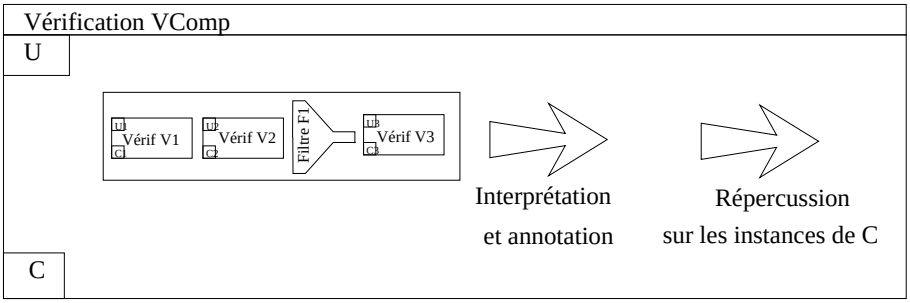

Figure 4 : *Composition de vérifications*

A ce stade, le processus obtenu par la composition des vérifications est un processus « général ». Il peut s'appliquer à n'importe quel modèle M1 décrit selon le métamodèle M2. Cependant, le processus peut nécessiter un paramétrage par des sélections du modèle à vérifier. Par exemple, on peut vouloir indiquer que la vérification de cohérence entre diagramme d'activités et diagramme d'état transition ne doit s'appliquer que sur deux sous-ensembles de M1 comportant chacun un diagramme d'états et un ensemble de diagrammes d'activités.

4 Hiérarchisation de la vérification

L'ensemble des vérifications doit être organisé afin de construire des processus de vérifications aisément.

4.1 Hiérarchisation des propriétés

D'une manière général, ce travail de classification est ambitieux et il n'existe pas à notre connaissance de travaux de synthèse sur ce sujet, de même que la littérature pléthorique au sujet d'UML se focalise peu sur les aspects vérification. Voici quelques critères de classification :

- par propriété : système / modèle / processus
- par domaine : externe / logique / physique
- par niveau d'abstraction : instance / types
- par aspect : statique / dynamique / fonctionnel
- par niveau d'analyse : syntaxe / typage / sémantique

Pour corréler l'invocation et la déclaration des vérifications, la hiérarchie H_M est basée d'abord sur la structuration des modèles (figure 5). A chaque niveau de modèle (sommet de H_M) correspond un ensemble de propriétés.

Les propriétés sont elles-mêmes structurées en propriétés et vérifications. Les autres critères de classification enrichissent la description des propriétés et leur groupage. Par exemple, l'absence d'interblocage, l'absence de précondition fausses et de contraintes fausses sont des propriétés du système. La conformité instance/type et la cohérence entre diagrammes sont des propriétés de domaine.

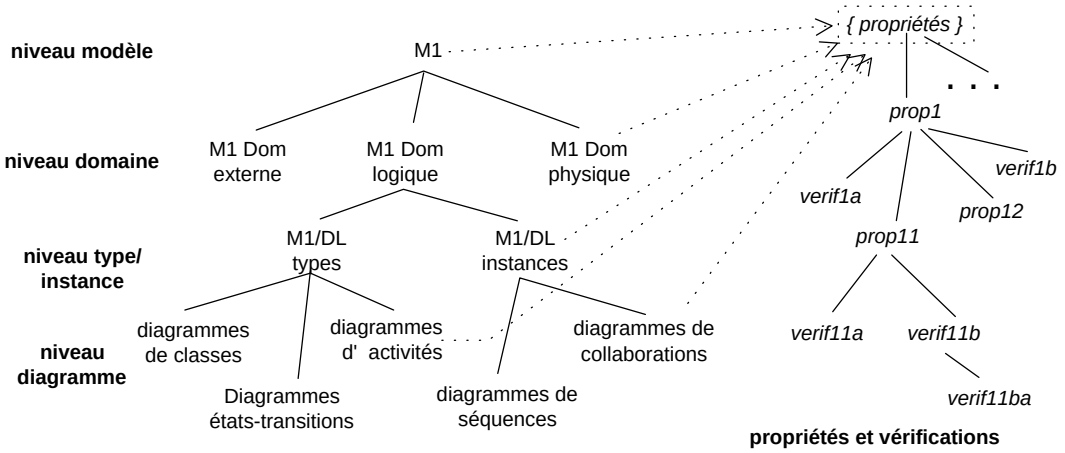

Figure 5 : *Hiérarchisation des modèles, propriétés et vérifications*

4.2 Génération de processus

Lorsque l'on souhaite introduire une nouvelle vérification $v1$ dans l'ensemble de vérifications, il faut la rattacher à une ou plusieurs propriétés P_i et à un sommet S dans H_M. Si P_i n'appartient à aucun sommet de H_M alors elle est ajoutée à S et son processus de vérification par défaut est $v1$. Si P_i appartient déja à S, alors son nouveau processus est obtenu par la composition de l'existant avec $v1$. Les propriétés associées à chaque sommet S de H_M sont constituées d'un nom et d'un processus de vérification. Ce processus peut être défini par le concepteur, ou construit automatiquement. Le processus par défaut est composé de l'appel de toutes les vérifications portant sur la même propriété ou sur une sous-propriété, puis d'un filtrage, pour chacun des fils de S. Cet appel est suivi des vérifications associées à cette même propriété pour le sommet S.

Pour obtenir une vérification à plusieurs niveaux de précision comme on en trouve dans les systèmes à bases de règles et sous la forme de niveaux de preuve dans les systèmes formels, il suffit de créer pour chaque propriété concernée un ensemble de sous-propriétés qui serviront à partitionner en niveaux les vérifications qui leur seront rattachées.

4.3 Hiérarchisation basée sur les domaines

Pour déterminer les vérifications, on se base essentiellement sur la norme (*UML Semantics Notation Guide*[7]. La définition des vérifications nous a permis de mettre en évidence des failles [3], comme d'autre auteurs ont pu le faire [15].

- La vérification *inter-domaines* correspond aux propriétés du processus de développement. Actuellement, seule la cohérence et la complétude entre les domaines est prise en compte par la traçabilité entre éléments de domaines de niveau différents. Plus formellement, on peut imaginer une relation de raffinage. Noter que plusieurs modèles peuvent être produits sur le même domaine.
- La vérification *intra-domaine* correspond aux propriétés des modèles. Comme le montre la figure 1, la vérification se décompose en conformité entre diagrammes d'instances et de types, et en vérification hétérogène (notamment de cohérence [9, 8]) entre diagramme de même niveau d'abstraction.
- La vérification *intra-diagramme* (ou entre diagrammes de même type) correspond aux propriétés des systèmes. Chaque diagramme couvre un aspect du système (statique / dynamique / fonctionnel). La vérification est optimale par une utilisation conjointe des méthodes formelles et des règles syntaxiques.

Prenons l'exemple du domaine logique, qui est au cœur de la vérification de modèles. Les erreurs non détectées à ce niveau ont une influence sur la suite du développement. Des exemples de vérification au niveau logiques sont donnés dans [3] et [6]. On les classe en trois couches : syntaxe (les combinaisons licites *e.g.* une interface ne comprend que des opérations, l'héritage est anti-symétrique), sémantique statique(*e.g.* typage, cardinalités, contraintes OCL), sémantique (usage et exceptions *e.g.* pas d'héritage multiple d'un discriminant, pas d'envoi simultané pour des objets passifs, contraintes de stéréotypes). Une partie de ces vérifications, essentiellement la syntaxe, est décrite par des règles OCL dans le métamodèle [7]. La vérification du niveau logique comprend la vérification individuelle des diagrammes, la vérification entre diagrammes de même niveau (type, instance) et la conformité entre diagrammes de niveau différents.

Chaque diagramme est une vue homogène, avec un nombre limité de concepts. Les vérifications communes à tous les diagrammes sont groupées dans la catégorie des mécanismes communs. Dans la couche 1 d'un diagramme de classes, on trouve par exemple les vérifications suivantes : un diagramme de classe n'inclut pas de collaborations ou de cas d'utilisation (!), unicité des espaces de nommage, une interface contient uniquement des profils d'opérations, une classe non abstraite contient des méthodes. Tout élément dérivé est associé à un calcul, la composition est exclusive, sont des vérifications de la couche 2. La couche 3 traite de la sémantique de la redéfinition d'associations, d'opérations, de la sémantique de la relation de dépendance, des types d'un attribut. Dans un diagramme états-transitions, des exemples de vérifications de la couche 1 sont : un état initial n'a pas de transitions entrantes, un état historique n'a pas de transitions sortantes. Le typage des événements, des gardes correspondent à la couche 2. Dans la couche 3, on pourra vérifier les gardes, préserver l'inclusion des états-hiérarchiques.

Une fois les diagrammes vérifiés individuellement, la correction, la cohérence, et la complétude sont examinés pour les diagrammes de types, les diagrammes d'instance et la conformité comme indiqué dans la figure 1. Au niveau type, on vérifie deux à deux ou globalement les diagrammes de classes, états-transitions, et activités : correction et cohérence des transitions, gardes et actions. Au niveau instance, on vérifie deux à deux les diagrammes de collaborations et de séquences représentant la même collaboration, ou le même ensemble d'objets, on assure la correction et la cohérence des messages.

5 Intégration dans Bosco

Bosco [1] est un projet disponible en source libre[6], dont l'objectif est la mise en oeuvre de référentiels (*repositories*)– ou modèles sous-jacents de représentation – de langages de modélisation, à partir des modèles MOF de ces langages. La principale caractéristique de Bosco est sa flexibilité. En effet, la génération de code est pilotée par un ensemble de *templates*, qui assurent l'indépendance entre le code qui contrôle la génération et le formatage du texte généré. Dit autrement, la génération de code ne se fait pas par une suite d'instructions *print()* confondue au milieu du code, mais grâce à l'interprétation d'un fichier texte. Cette flexibilité permet que certains traitements définis dans 3.2 soient facilement ajoutés au référentiel généré.

Le référentiel généré par Bosco peut être utilisé à travers deux interfaces, la première peut lire et écrire des fichiers XML (compatibles XMI) et la deuxième (compatible JMI) permet qu'un modèle soit directement manipulé par des programmes Java. Ces interfaces permettent que le référentiel soit intégré à d'autres outils de développement. Bien que cette intégration se fasse encore à très bas niveau, notre objectif est de suivre l'exemple des *web services* et que la vérification de modèles soit vue comme un service offert à d'autres outils. En plus de la manipulation de modèles, l'interface Java permet leur introspection. Elle permet, par exemple, la collecte d'éléments de M1 qui sont des instances d'un élément M2. De même, l'annotation des éléments du modèle en fonction du résultat de l'interprétation (partitionnement en instances de U_v et U_n) et la répercussion de celui-ci sont indépendantes du modele sous jacent.

Une autre caractéristique importante de Bosco est la possibilité de fusionner des méta-modèles, qui nous a permis, par exemple, d'intégrer un méta-modèle du langage OCL à ceux de UML et de MOF. Cette fusion nous semble importante lors de l'annotation des éléments M1 dans des langages qui, contrairement à UML, n'offrent pas d'éléments de modélisations tels que les étiquettes.

6 Conclusions et perspectives

Ce papier propose un cadre général pour la vérification de modèles UML dans lequel les vérifications sont associées à une structuration commune de la notation UML, indépendamment des outils et méthodes de développement. La vérification devient alors une activité à part entière de la modélisation avec UML. Le cadre de vérification favorise l'utilisation conjointe des deux approches de vérification, les méthodes formelles et les bases de règles, par la génération modulaire de processus de vérification. Les processus de vérification peuvent être construits automatiquement sur la base d'une hiérarchisation des vérifications. L'implantation dans Bosco permet l'évolution de la vérification en fonction des standards de notation et l'automatisation du processus de vérification. L'utilisation d'un outil qui est indépendant du méta-modèle est particulièrement importante dans le contexte du MDA, où différents langages de modélisation doivent coexister avec le langage unifié. Les perspectives à court terme visent une implantation d'un ensemble représentatif de vérifications dans Bosco, et à étudier plus finement les interactions entre approches formelles et bases de règles. A plus long terme, nous souhaitons intégrer le processus de vérification dans le processus de développement en fournissant des services de vérification aux outils de modélisation.

Références

[1] Pascal ANDRÉ, Gilles ARDOUREL, et Gerson SUNYE. The Bosco Project, A JMI-Compliant Template-based Code Generator. In W. DOSCH et N. DEBNATH, réds., *Proceedings of the 13th International Conference on Intelligent and Adaptive Systems and Software Engineering*, pages 157–162, July 2004. ISBN 1-880843-52-X.

[6] Disponible sur http ://bosco.tigris.org

[2] Pascal ANDRÉ et Jean-Claude ROYER. *Ingéniérie Objet, Concepts et techniques, Mourad Oussalah (Ed.)*, Chapitre de livre 8 : Objets et spécifications formelles, pages 271–314. ISBN : 2-7296-60642-4. Interéditions, Mai 1997.

[3] Pascal ANDRÉ et Alain VAILLY. *Exercices corrigés en UML ; Passeport pour une maîtrise de la notation.*, volume 5 of *Collection Technosup*. Editions Ellipses, 2003. ISBN 2-7298-1725-5.

[4] Desmond D'SOUZA et Alan WILLS. *Objects, Components and Frameworks With UML : The Catalysis Approach*. Addison-Wesley, 1998.

[5] Andy S. EVANS, Jean-Michel BRUEL, Robert B. FRANCE, Kevin C. LANO, et Bernhard RUMPE. Making uml precise. In *OOPSLA'98 Workshop on "Formalizing UML. Why and How ?"*, October 1998. Vancouver, Canada.

[6] Martin GOGOLLA. UML for the impatient. Research Report 3/98, Universität Bremen, 1998.

[7] Object Management GROUP. The OMG Unified Modeling Language Specification, version 1.5. Rapport technique, Object Management Group, available at http ://www.omg.org/cgi-bin/doc ?formal/03-03-01, Juin 2003.

[8] Ludwik KUZNIARZ, Zbigniew HUZAR, Gianna REGGIO, Jean-Louis SOURROUILLE, et Miroslaw STARON, réds.. *Proceedings of «UML» 2003 Workshop on Consistency Problems in UML-based Software Development II*, San Francisco, California, USA, October 20 - 24 2003. IEEE and Blekinge Institute Of Technology. ISSN 1103-1581 also Research Report 2003 :06, http ://www.ipd.bth.se/consistencyUML/Consistency_Problems_in_UML_II.pdf.

[9] Ludwik KUZNIARZ, Gianna REGGIO, Jean-Louis SOURROUILLE, et Zbigniew HUZAR, réds.. *Proceedings of «UML» 2002 Workshop on Consistency Problems in UML-based Software Development*, Dresden, Germany, September 30 - October 4 2002. Blekinge Institute Of Technology. ISSN 1103-1581also Research Report 2002 :06, http ://www.ipd.bth.se/uml2002/RR-2002-06.pdf.

[10] Stephen J. MELLOR et Marc J. BALCER. *Executable UML - A Foundation For Model-Driven Architecture*. Addison Wesley, 2002. ISBN 0-201-74804-5.

[11] Chris RAISTRICK, Paul FRANCIS, Ian WILKIE, John WRIGHT, et Colin B. CARTER. *Model Driven Architecture with Executable UML*. Cambridge University Press, 2004. ISBN 0-521-53771-1.

[12] Gianna REGGIO et Roel WIERINGA. Thirty one problems in the semantics of uml 1.3 dynamics. In *Proceedings of the OOPSLA'99 Workshop on "Rigorous Modelling and Analysis of the UML : Challenges and Limitations*, 1999.

[13] James RUMBAUGH, Ivar JACOBSON, et Grady BOOCH. *The Unified Modeling Language*. Object-Oriented Series. Addison-Wesley, 1999. User Guide (ISBN 0-201-57168-4), Reference Manual (ISBN 0-201-30998-X).

[14] James RUMBAUGH, Ivar JACOBSON, et Grady BOOCH. *The Unified Software Development Process*. Object-Oriented Series. Addison-Wesley, 1999. ISBN 0-201-57169-2.

[15] Anthony SIMONS et Ian GRAHAM. 30 things that go wrong in object modelling with uml 1.3. In I Simmonds H KILOV, B RUMPE, réd., *Behavioral Specifications of Businesses and Systems*, chapitre 17, pages 237–257. Kluwer Academic Publishers, 2001.

[16] Ambrosio TOVAL, Jose SÀEZ, et Francisco MAESTRE. Automated Property Verification in UML Models. In Stefan Gruner MICHAEL LEUSCHEL et Stéphane Lo PRESTI, réds., *Proceedings of the 3rd Automated Verification of Critical Systems (AVoCS'03)*. DSSE Technical Report DSSE-TR-2003-2, Avril 2003.

Table des matières

Un cadre pour la vérification de modèles UML

P. André, G. Ardourel, G. Sunyé

Résumé

Ce rapport de recherche pose les bases d'un cadre formel pour la vérification sur des modèles UML de propriétés telles que la correction, la cohérence, la sûreté de fonctionnement... Actuellement, cette vérification est essentiellement syntaxique et repose sur des outils UML disparates. Dans ce papier, nous proposons un cadre pour la vérification de modèles UML, basé sur une structuration des modèles, une combinaison de techniques de vérification, qui s'intègre au développement itératif et incrémental. Une expérimentation de cette approche est actuellement réalisée sur un outil disponible en source libre

Catégories et descripteurs de sujets : D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; D.2.11 [**Software Engineering**]: Software Architectures—*Languages*

Termes généraux : UML, Vérification de modèles

Mots-clés additionnels et phrases : Composants, Services, Spécification formelle, Conformité d'interaction

LINA, Université de Nantes
2, rue de la Houssinière
B.P. 92208 — F-44322 NANTES CEDEX 3