Chapitre 4

Des compléments sur la vérification avec UML

Ce chapitre propose un complément de cours sur la vérification avec UML. Des exercices y étant associés dans le chapitre 3.

1 Introduction

Le langage de modélisation unifié UML [RJB99a, RJB98, Con99, Mul97] et le processus unifié UP [RJB99b] constituent une étape importante dans le développement du logiciel. En effet, le langage UML a permis de normaliser la jungle des notations utilisées dans les méthodes de développement à objets et le processus unifié est une synthèse de ce qui semble essentiel au développement orienté composants et à grande échelle.

Le langage UML [Con99] répond à un besoin de clarification des notations visuelles des nombreuses méthodes à objets constituant le paysage du développement des années 90. La notation a depuis été normalisée par l'OMG et la plupart des méthodes du marché proposent un langage de modélisation proche d'UML. Les intérêts de cette normalisation sont multiples : portabilité des modèles et des documentations, apprentissage d'un langage unique par les acteurs du développement, normalisation des documents, simplification des outils d'aide au développement, etc. Le langage UML est large-spectre puisqu'il doit permettre la modélisation de toutes sortes d'applications à objets à tous niveaux d'abstraction.

- Toutes les applications peuvent être modélisées avec UML : applications de gestion (`business`), applications temps réel, applications scientifiques, bureautique, etc.
- La notation couvre ainsi la description des besoins (description de l'environnement et des caractéristiques fonctionnnelles et non fonctionnelles du système), la structuration du système (composants à différents niveaux d'abstraction et de structuration), l'évolution du système (comportement dans le temps), son implantation (en termes de programmes à objets ou pas). Le langage est prévu pour un processus sans coutures (*seamless*) c'est-à-dire que la même notation est utilisée quelque soit le niveau d'abstraction (de la description quasi informelle à la description formelle des programmes). On peut ainsi tracer les différentes décisions de conception car les modèles sont progressivement enrichis.
- La notation se veut un modèle commun à tous les langages de modélisation ou de programmation à objets.
- Un modèle de la notation a été proposé, qui permet la réflexion sur le langage et l'implantation des outils associés (éditeurs, analyseurs, traducteurs).

Le processus de développement unifié pose les bases du développement d'applications complexes : processus itératif et incrémental, basé sur les cas d'utilisation et centré architecture.

Il est décomposé en itérations sur quatre phases (préparation[1], élaboration, construction et transition) ; chaque itération comprend un travail d'analyse des besoins, d'analyse, de conception, de réalisation et de test. Chaque travail donne lieu à un modèle UML et correspond à un niveau d'abstraction donné. Le développement est un suite d'itérations permettant de comprendre le besoin, construire une architecture d'application qui est ensuite développée. La progression dans le développement tient compte des risques du développement et de la réutilisation.

Au premier abord, UML semble bien remplir son rôle puisqu'il a été accepté comme norme par l'Object Management Group (OMG). Actuellement, on dispose de bon nombre d'outils d'édition, qui effectuent certaines vérifications et des traductions de squelettes de classes dans les langages de programmation. Cependant, la validation des modèles et la vérification de propriétés des modèles sont essentiellement à la charge des utilisateurs de la méthode ou des fournisseurs d'outils. Les étapes d'acceptation des produits et des modèles dans les tâches d'analyse et de conception risquent d'être le théâtre de discussions longues et coûteuses. Sachant qu'elles ont lieu à chaque itération du processus, la validation constitue une étape clé dans le processus unifié. Sans outil d'aide à la vérification, la tâche de validation risque d'être réalisée superficiellement par les acteurs du développement et les problèmes risquent d'être reportés au niveau de la programmation pour laquelle des outils et des métriques existent. En conséquence, on risque de perdre une partie des bénéfices du processus unifié et de retomber dans les travers du modèle classique de la cascade : détection tardive des erreurs et coûts élevés de maintenance.

Dans ce chapitre, nous étudions les différents problèmes liés à la vérification dans le langage UML et le processus unifié et des pistes de solutions. Dans la section 2, nous rappelons les éléments essentiels du langage et du processus unifié, qui nous serviront de référence dans la suite du document. Dans la section 3, nous posons la problématique de la validation et la vérification avec UML. Nous déterminons les besoins, les attentes et les problèmes rencontrés. Pour appréhender sa complexité, la notation est découpée en domaines, les domaines en modèles et diagrammes. Nous illustrons ces problèmes sur quelques exemples en détaillant les domaines externes et physiques. Un exposé plus important est consacré au domaine logique dans la section 4. Bien que leur apport pédagogique soit indéniable, nous avons pris le parti de ne pas illustrer chaque règle par un diagramme UML. Un catalogue de 500 pages ne suffirait pas. Dans ces deux sections, nombre de règles de vérification sont proposées et discutées. Pour certaines, nous mettons en évidence les ambiguïtés du métamodèle en préconisant telle ou telle interprétation. Cette préconisation reste discutable, l'intérêt du travail étant de mettre en évidence ces points de discussion. La section 5 résume les difficultés de la vérification et explore deux pistes pour résoudre ces problèmes : l'utilisation du métamodèle et la définition d'un modèle formel. Nous concluons sur les perspectives dans ce domaine.

2 Développement avec UML et UP

Nous ne nous étendrons pas sur la notation UML et le processus unifié. Le lecteur consultera avec profit le tome 2 [AV01b] et les ouvrages de référence sur le langage [RJB99a, RJB98, Con99, Mul97] et le processus unifié UP [RJB99b]. Nous résumons les éléments essentiels, auxquels nous faisons référence dans la suite du chapitre. Après avoir présenté la notation (section 2.1) et le processus (section 2.2) nous mettons en évidence la corrélation entre processus et notation dans la section 2.3. En effet, l'utilisation des notations varie en fonction du processus de développement utilisé. Cela influence fortement la vérification, dont fait l'objet ce chapitre.

[1] *Inception*

2.1 Notation UML

La notation UML inclut un grand nombre de concepts autour de l'objet (objets, classes, opérations, attributs, relations, envois de message, etc.), mais aussi de l'analyse des besoins (acteurs, cas d'utilisation), de la conception du logiciel (composants, modules, processus) ou de l'implantation (nœuds, liaisons, déploiement). La raison est que cette notation est conçue pour décrire des modèles couvrant l'ensemble du cycle de développement. De plus, UML autorise l'enrichissement ou la personnalisation de la notation au moyen des stéréotypes (des mots-clés), qui qualifient les concepts. Par exemple, une classe abstraite est une classe affinée par le stéréotype `«abstract»`.

La variété d'utilisation des concepts est induite par le nombre de combinaisons des concepts au sein d'un modèle. UML propose huit types de combinaisons cohérentes, appelées **diagrammes**.

- Les diagrammes de classes représentent les classes et les relations statiques entre ces classes. Les concepts principaux à ce niveau sont : classe, attribut, opération, visibilité, interface, association, agrégation, héritage, dépendance...
- Les diagrammes d'objets décrivent des objets et des liens. Les objets peuvent être actifs et définir leur flot de contrôle. Sur ces liens (réels ou virtuels) circulent des messages. Les envois de messages sont synchrones ou asynchrones, avec ou sans résultats. Les diagrammes d'objets se retrouvent sous deux formes dans UML :
  - Les diagrammes de séquence, qui donnent une vision temporelle des interactions en objets en mettant l'accent sur l'ordonnancement des échanges entre objets.
  - Les diagrammes de collaboration, qui donnent une vision spatiale des interactions en mettant l'accent sur les liaisons entre objets.
- Les diagrammes de cas d'utilisation (UC - *Use Case*) décrivent les acteurs et l'utilisation du système.
- Les diagrammes états-transitions modélisent le comportement des objets au cours du temps.
- Les diagrammes d'activités décrivent le flot de contrôle interne aux opérations. A grande échelle, ils représentent aussi les échanges entre objets.
- Les diagrammes de composants mettent en évidence les composants d'implémentation et leurs relations.
- Les diagrammes de déploiement définissent la structure matérielle et la distribution des objets et des composants.

La notation propose des éléments généraux pour enrichir ou structurer les diagrammes : stéréotypes, paquetages, notes, contraintes.

Les diagrammes décrivent des aspects complémentaires mais non disjoints du système. Ainsi, les concepts peuvent apparaître dans différents diagrammes avec parfois des noms différents (par exemple, un paquetage se nomme catégorie dans un diagramme d'analyse et sous-système dans un diagramme de conception). De plus certains concepts sont perçus à des niveaux d'abstraction différents. Par exemple, les opérations des objets sont décrites plus finement à la conception.

La première source de complexité est donc la combinaison des concepts dans les diagrammes. UML gère en partie ce problème en proposant une "syntaxe", c'est le **métamodèle**. En ce sens, UML est qualifié de langage.

La seconde source de complexité est la combinaison (l'utilisation) des diagrammes dans la spécification du système. A ce niveau, il n'y a pas de règle précise. Par exemple, les diagrammes d'activités peuvent être utilisés pour décrire des états d'objets, des opérations (vision objet) ou une fonction du système (vision DFD).

Les auteurs principaux de la notation proposent une architecture du système organisée en **vues**. Chaque vue est une perspective sur le système, qui dépend du niveau d'abstraction

ou de préoccupation, mais le lien entre les vues reste relativement flou. Les diagrammes représentent par exemple des coupes (des vues) sur le système modélisé. Par exemple, les diagrammes de classes, de composants ou de déploiement mettent en évidence une structure du système. Les diagrammes états-transitions mettent en évidence l'évolution dynamique des objets. Les diagrammes de séquence, de collaboration ou d'activité mettent en évidence les aspects fonctionnels et la coordination dans le système. D'autres découpages sont possibles. Ainsi, une architecture habituelle est un découpage en vue des cas d'utilisation, vue logique (ou de conception), vue des processus, vue d'implantation et vue de déploiement.

La confusion possible entre notation et modèle de la notation est induite en partie par le fait que les règles syntaxique de représentation et de combinaison de concepts de la notation sont définies dans la notation elle-même (métamodèle) d'UML. Le vocabulaire qu'on utilise pour décrire la notation se confond avec le vocabulaire de la notation elle-même. Les manuels de référence [Con99, RJB99a] sont décrits en s'appuyant sur son métamodèle. La structure du métamodèle donne la structure d'explication du langage : on parle d'**élément de modélisation**, de relation, de diagramme. Par exemple, une note (un commentaire) est un élément de modélisation au même titre qu'une classe. On touche donc plus à l'aspect syntaxique de la notation qu'aux concepts.

Dans [AV01b], nous avons regroupé les diagrammes, qui nous semblaient proches, en quatre familles de modèles : les modèles d'approche (cas d'utilisation et scénarios), les modèles de structure (diagrammes d'objets, de collaboration et de classes), les modèles de la dynamique (diagrammes de séquence, états-transitions et d'activités) et les modèles d'implantation (diagrammes de composants et de déploiement). On peut aussi reclasser ces familles en trois niveaux d'abstraction : les modèles de l'utilisateur (approche), les modèles de conception (structure et dynamique) et les modèles d'implantation. Cela nous donne une forme cohérente d'utilisation des diagrammes qui facilite la vérification. Nous y reviendrons dans la section 3.

2.2 Processus unifié (UP)

Le processus unifié n'est pas à proprement parler innovant. Il s'agit d'un processus itératif (modèle en spirale) qui reprend les étapes habituelles du développement (des besoins aux tests), les principes du prototypage et du développement incrémental (une structure -une architecture- qu'on complète progressivement), et enfin la gestion de projet et le pilotage (évaluation, planification, organisation, suivi et approbation). Tous ces points sur la conduite de projets informatiques sont détaillés dans le chapitre 1 de l'ouvrage de MOREJON et RAMES [MR93] ou dans [CGM97]. L'originalité d'UP repose sur l'intégration harmonieuse de ces différentes approches au moyen des cas d'utilisation et sur son adaptation aux techniques récentes du développement (objet, architectures réparties, Web, etc.), la réutilisation et les modèles (patrons *pattern* ou architectures types *frameworks*). Détaillons le processus unifié.

Dans le développement d'un système informatique, le processus décrit l'enchaînement des travaux qui conduisent à des résultats. On appelle produit le résultat d'un travail, d'une étape dans le développement. Le produit peut être un système informatique mais aussi un ensemble de documents quelconques, des spécifications, des logiciels, des manuels, etc.

Le processus unifié UP [RJB99b] se répète à travers une série de cycles formant la vie du système. Chaque cycle conduit à une nouvelle version, c'est-à-dire un produit utilisable par le client. Cette version comprend la description des besoins (cas d'utilisation, besoins non fonctionnels et tests), l'architecture du système et les modèles produits au cours du développement, le logiciel, la documentation, etc.

Un cycle se décompose en quatre phases : la préparation (*inception*), l'élaboration, la construction et la transition. Chaque phase peut être perçue comme un projet avec une planification des travaux, des ressources (humains, techniques, temporelles, financières), des cert-

fications, etc. Nous ne nous étendrons pas sur cet aspect gestion de projet. La préparation est la phase de lancement du projet. Durant cette phase, on propose une vision du système (besoins attendus, performances, architectures envisagées, etc.), on définit les risques et les critères d'évaluation, on établit un plan de travail prévisionnel des phases suivants. L'élaboration met au point et valide l'architecture du système, c'est-à-dire qu'à l'issue de cette étape, le cadre de développement est bien défini et la faisabilité est assurée. Théoriquement, il ne reste qu'à compléter (remplir) l'architecture et tester l'ensemble, c'est l'objectif de la phase de construction. Le résultat est un produit exploitable qu'il reste à tester en contexte réel. La phase de transition est la mise en situation réelle du produit, tests complets et finalisation de la version du produit pour le cycle courant. La transition établit le bilan du cycle, capitalise les efforts de développement et prépare les versions ultérieures (les cycles suivants) : éléments à ajouter, à améliorer, etc.

Les phases sont elle-même divisées en itérations et produisent des versions du produit. Chaque itération couvre les activités traditionnelles (*workflow*) d'analyse des besoins, d'analyse, de conception, de réalisation et de test. En fait, les parts respectives de ces activités évoluent selon la phase considérée, les activités en amont sont plus poussées dans les itérations des premières phases, les activités en aval sont plus approfondies dans les itérations des phases ultérieures. Chaque itération donne lieu à un produit résultat. Plus le projet est grand, plus il y a d'itérations dans une phase.

A ces activités correspondent des modèles (cas d'utilisation, analyse, conception, déploiement, implantation et test) décrits avec la notation unifiée, une vue architecturale de ces modèles et des descriptions complémentaires. Nous retrouvons alors la notation UML, qui sert à décrire ces modèles. Nous traçons maintenant les grandes lignes de la corrélation entre le processus et la notation unifiée.

2.3 Du processus à la notation unifié

Nous donnons un aperçu de l'utilisation de la notation UML dans le processus unifié au travers des activités des itérations.

Modélisation du besoin

La capture des besoins établit les besoins fonctionnels ou non du système à modéliser et la compréhension de son contexte.

Les besoins fonctionnels sont les services attendus par le demandeur. Les besoins non fonctionnels regroupent des contraintes (systèmes existants, plate-formes, standards, distribution, etc.) et des critères du résultat attendu (performances, coûts, qualité, etc.). Les besoins sont ensuite consignés sous forme d'un modèle des cas d'utilisation, de glossaires, de descriptions des acteurs, des cas d'utilisation, des interfaces utilisateur et des besoins spéciaux. Essentiellement textuels, les produits de l'activité d'analyse des besoins utilisent les diagrammes de cas d'utilisation, illustrés par des scénarios (diagrammes de séquence réduits au système et aux acteurs externes). Plus rarement on utilise des diagrammes états-transitions ou des diagrammes d'activité pour décrire les traitements de cas d'utilisation. Il s'agit là à notre avis d'une entorse, parfois pratique mais sur laquelle il est difficile d'établir des vérifications formelles.

Le contexte du système est une compréhension de son activité réelle. Elle se traduit sous deux formes : modélisation du domaine (*Domain Model*) et modélisation du métier (*Business Model*).

- Le modèle du domaine est la description technique du domaine d'activité : les termes techniques et le vocabulaire employé, les concepts et leurs relations. Il s'agit essentiellement d'un diagramme de classe qui permet de mieux comprendre de quoi il s'agit.

– Le modèle du métier s'attache à décrire des processus métier. On l'utilise plutôt lorsqu'on travaille sur une organisation, que nous qualifions sommairement d'informatique de gestion. Il s'agit d'une modélisation du système réel, et non du système (informatisé) futur. Ce modèle permet de mieux appréhender ce qui se passe dans le système. Les notations UML utilisées sont essentiellement celles des hauts niveaux d'abstraction, que nous avions classé en modèles de l'utilisateur et modèles de conception dans la section 2.1).

Analyse

L'analyse est un raffinement et une structuration des besoins en termes de structure du système (interne au système). Cette structure est une structure logique en terme d'objets (et de classes), qui se situe à un niveau d'abstraction supérieur à l'implantation, aucune hypothèse n'étant faite en ce sens. Le modèle d'analyse peut être perçu comme une répartition (logique) du besoin sur les éléments (essentiellement des objets) du système. Chaque objet ayant des responsabilités vis-à-vis de ces besoins. Il s'agit de "réaliser" les cas d'utilisation. On passe du langage du client à celui du développeur.

Le modèle d'analyse comprend essentiellement des diagrammes de classes et des diagrammes de collaboration. Les diagrammes de classes décrivent la structure logique du système (une abstraction de ses objets logiques). Les diagrammes de collaboration décrivent comment les cas d'utilisation sont réalisés par le système. Selon les principes de Jacobson, on sépare nettement les objets de l'interface, ceux du contrôle et de la coordination des activités et ceux de la gestion des informations (entités). L'organisation de ces diagrammes via des paquetages d'analyse et des paquetages de services (et leur dépendances) fournit une première architecture logique du système (*Application-specific layer*, *Application-general layer*).

L'analyse est complétée par la description de besoins spécifiques concernant la persistance, la distribution et la concurrence, la sécurité, la tolérance aux fautes, la gestion de transactions, etc.

Conception

La conception est la détermination du système informatique qui permet de répondre aux besoins mis en évidence dans l'analyse des besoins et structurés dans l'analyse. La conception fait le lien entre l'architecture logique et l'architecture logicielle. Dans une architecture en couche, la conception est grossièrement le lien entre les couches hautes issues de l'analyse et les couches basses provenant de l'environnement d'implantation (*Middleware layer*, *System-software layer*). C'est à ce niveau que sont mises en évidence les décisions générales d'implantation (architecture logicielle -client-serveur, distribué, n-tiers-, environnements de développement et d'interface, communication, persistance, systèmes d'explotation cibles, applications existantes, etc.). La conception définit un contexte précis et fiable pour l'implantation (*blueprint for the implementation model*).

Lors de la conception sont produits principalement un modèle de conception et un modèle de déploiement. La vue architecturale et des descriptions annexes (contraintes...) complètent le produit. Le modèle de conception décrit la réalisation physique des cas d'utilisation par des diagrammes de classes et d'interaction. Il est structuré en sous-systèmes (paquetages de conception) munis d'interfaces. Les sous-systèmes de service, comme les paquetages de services dans l'analyse, regroupent des classes fournissant un service commun et cohérent. Les classes et leurs relations sont décrites selon le vocabulaire et la sémantique de l'environnement cible (avec des stéréotypes pour préciser la notation). Les méthodes, qui réalisent les opérations, sont décrites plus finement, éventuellement par des diagrammes d'activité. Le comportement des objets actifs est précisé par les diagrammes états-transitions. Les interactions sont définies par des séquences ou des collaborations. Le modèle de déploiement utilise les diagrammes de

même nom. On peut affecter des sous-systèmes aux nœuds. On n'utilise pas de diagrammes de composants à ce niveau.

Implantation

L'implantation est la réalisation du modèle de conception et de déploiement à travers les éléments d'implantion (composants, fichiers sources, exécutables, etc.). Il s'agit donc essentiellement de passer des diagrammes de classes et collaborations aux diagrammes de composants puis de procéder à l'implantation réelle et aux tests.

Un modèle d'implantation et un modèle de déploiement sont fournis et enrichis de descriptions telles que la vue architecturale de la conception, le plan d'intégration, etc. Le modèle d'implantation utilise la notation des diagrammes de composants. Il est structuré en sous-systèmes d'implantation munis d'interfaces et reliés par des relations de dépendance.

La traçabilité est assurée par une relation de dépendance entre les éléments du modèle de conception et ceux du modèle d'implantation. Par exemple, une classe de conception est réalisée dans un fichier Java.

Test

Le test consiste à vérifier le résultat de l'implantation. L'activité de test consiste en une planification et une définition des objectifs de test, la réalisation puis l'analyse des tests. Il en résulte une activité de correction de l'implantation (et des modèles associés) et un retour sur les tests (avec régression).

Les produits du test sont spécifiques à cette activité : modèle des tests, planification et évaluation des tests, traitement des erreurs. Le modèle de test est un ensemble de cas de test, de procédures de test et de composants de test. Les cas de test correspondent aux cas d'utilisation et à leur réalisation. Les procédures de test décrivent le test d'un ou plusieurs cas de tests. Les composants de test sont des composants logiciels qui automatisent les procédures de test.

L'ensemble de la notation est utilisé au sens où on se réfère aux modèles des activités précédentes pour réaliser les tests. Mais il n'y pas a priori de diagramme UML dédié à cette activité, même si on peut tracer des diagrammes de cas d'utilisation, des diagrammes de séquence ou organiser les programmes en composants.

2.4 Synthèse

A travers le processus unifié, on se rend compte que certains principes comme le développement sans couture ne sont pas "purs". Il nous semble sage en effet, pour éviter des confusions induites par la complexité de la notation, de ne pas utiliser tous les diagrammes à tous les niveaux et de favoriser telle notation pour tel type d'activité. Ainsi, on a une vision plus claire des niveaux d'abstractions, chers à Merise. Dans UML, on distingue aussi trois niveaux : le niveau externe (ou utilisateur, n'existe pas dans Merise), le niveau logique (conceptuel et logique dans Merise) et le niveau physique (plus développé que dans Merise).

Le niveau externe est celui de l'utilisateur (cas d'utilisation, et scénarios provenant essentiellement de la méthode Objectory [JCJO92]). Le niveau logique est celui de la description abstraite du système (il correspond historiquement aux notations issues de l'analyse à objets, et fortement inspirées de la méthode OMT [RBP+96]). Le niveau physique est celui de la description concrète du système (il correspond historiquement aux notations issues de la conception à objets, et fortement inspirées de la méthode de Booch [Boo92]). On retrouve les trois piliers de la notation UML.

En plus de la meilleure visibilité des activités à réaliser en fonction des notations, il est évident que la vérification des produits (des spécifications) est facilitée s'il y a moins de

diagrammes à confronter. Il n'en reste pas moins que l'articulatation des modèles doit mettre en évidence la continuité du processus et la source des décisions (traçabilité des concepts et des décisions).

3 Vérification de modèles UML

La certification des produits (des résultats) du développement du logiciel est une question essentielle dans la bonne marche et dans l'acceptation du produit résultant. Elle peut se mesurer en termes de qualités des spécifications et du logiciel [AV01a] (fiabilité, robustesse, validité, extensibilité, etc.), de rentabilité (coûts et retour d'investissements), etc. En faisant abstraction des critères économiques, liés à la gestion de projet, l'acceptation du résultat se traduit souvent en deux questions essentielles : le produit est-il valide ? le produit est-il correct ?

La validité est une préoccupation des utilisateurs du système. Le système est valide s'il répond à l'attente des utilisateurs et aux contraintes de l'environnement [GMSB96]. Dans le processus unifié, cette préoccupation est prise en compte dès le début, car les cas d'utilisation forment une description du système adéquate pour les revues et inspections des utilisateurs. De plus, le prototypage rapide permet un retour "réel" et rapide des futurs utilisateurs. Les inspections peuvent aussi porter sur les modèles du métiers, les modèles du domaine et les modèles d'analyse car la notation UML utilisée reste limitée et générale. Nous ne reviendrons pas sur ce point.

La vérification consiste à s'assurer que les modèles successifs satisfont la spécification globale, que le développement est correct par rapport à la spécification de départ [GMSB96]. Outre les revues et les inspections, la vérification inclut des tests et des preuves. Les inspections sont prévues à chaque activité de chaque itération des phases. C'est un travail collectif de discussion. Le test est une activité à part entière du processus unifié. Il fait appel aux techniques de test et de génération de test sur lesquelles nous ne nous étendrons pas. Nous nous intéressons dans la suite à la vérification de propriétés des modèles et à leur preuve. Nous nous appuyons sur des généralités sur le développement du logiciel décrites dans le chapitre 1 de [AV01a].

3.1 Objet de la vérification

Nous distinguons trois catégories de propriétés sur lesquelles peut porter la vérification : les propriétés du système, les propriétés de la spécification (des modèles) et les propriétés du processus.

Propriété du système

Les propriétés du système sont relatives aux trois aspects d'un système :
- Aspect structurel : pas de redondance des informations, volumes finis, modularité, cohérence, etc.
- Aspect fonctionnel : terminaison des calculs, temps fini, unicité des résultats pour certaines entrées, cohérence, complétude, etc.
- Aspect dynamique : pas de blocage, pas de famine, équité, sûreté, etc.

Dans une première approche, nous estimons que ces propriétés peuvent être vérifiées indépendamment dans les diagrammes. Par exemple, dans un diagramme de classe ou un diagramme d'objet, on vérifie des propriétés structurelles. Dans les diagrammes états-transitions ou les diagrammes de séquence, on vérifie des propriétés dynamiques. Dans les diagrammes d'activités, on vérifie des propriétés fonctionnelles.

Propriété des modèles

Nous pouvons ranger dans cette catégorie les propriétés relatives à la qualité des spécifications (cohérence, non redondance, complétude, conformité, etc.)[2]. En considérant chaque modèle comme un ensemble de descriptions complémentaires, on a affaire à des spécifications hétérogènes (intégration de méthodes, systèmes multi-vues). La propriété essentielle à vérifier est la cohérence et la complétude des descriptions entre elles.

Dans UML/UP, il s'agit de vérifier la cohérence et la complétude des descriptions à travers les différents diagrammes. Par exemple, un envoi de message dans un diagramme de séquence peut/doit correspondre à un échange dans un diagramme de collaboration, une opération dans un diagramme de classe, une transition dans un diagramme d'activité, un sous-diagramme d'activité, une opération d'un composant, etc. Cela correspondont à la vérification croisée des données et des traitements dans Merise.

Propriété du processus

Nous pouvons ranger dans cette catégorie les propriétés relatives à la qualité du processus (sûreté, terminaison, rigueur, traçabilité, etc.)[3]. Il s'agit, en gros, de vérifier qu'à travers les niveaux d'abstraction les spécifications sont toujours les mêmes. C'est la propriété de cohérence du raffinement des spécifications. Plus globalement, l'implantation respecte la spécification.

Dans UML/UP, on rattache ces propriétés aux modèles des différentes activités de l'analyse des besoins au test. La propriété essentielle à notre niveau est la traçabilité des éléments dans le cycle de développement et celle des décisions associées aux transformations. La traçabilité permet un contrôle de cohérence et de complétude.

3.2 Principes de mise en œuvre de la vérification

La vérification est la mise en cohérence des modèles et des propriétés attendues. Nous dégageons trois étapes : propriétés, règles, contrôle. Le premier travail consiste à établir les propriétés attendues pour chaque modèle ou chaque groupe de modèle, selon la catégorie de propriétés étudiées. Le second travail est la définition pour chaque propriété de règles (ou des contraintes) de vérification. Si les règles sont respectées alors la propriété est vérifiée. La dernière étape est le contrôle, la vérification effective, des règles sur les spécifications et les modèles.

Compte-tenu de l'étendue du domaine modélisable avec UML, du nombre de diagrammes et de modèles produits lors du processus UP, la vérification est un travail extrêmement difficile. Actuellement, il n'existe pas de document de référence établissant les bases d'une mise en œuvre de la vérification dans le processus et la notation unifiés. C'est sans doute l'une des tâches gigantesque mais essentielle à laquelle les concepteurs d'UML devront un jour s'atteler.

Détermination des propriétés et des règles

Nous n'avons guère trouvé dans la littérature, pourtant abondante sur UML, un exposé détaillé des règles à vérifier. En général, des pistes sont données dans les ouvrages, des indications sous forme de questions à se poser. Une autre source d'information se trouve dans les manuels de documentation d'UML : l'explication des éléments et de leurs relations, les exemples, les combinaisons raisonnables et celles qui sont interdites, le métamodèle qui définit des combinaisons acceptables d'éléments de la notation, etc. Une troisième source d'information est l'utilisation de règles *ad-hoc* dans les outils supportant la notation UML.

[2] Voir chapitre 1 de [AV01a].
[3] Voir chapitre 1 de [AV01a].

Expression des règles de vérification

La forme la plus simple d'expression est le langage naturel. On peut voir l'outil de vérification comme un manuel de procédure avec un lexique des propriétés, une liste structurée des règles permettant de vérifier la propriété et une démarche pour tester les spécifications. Cette forme de mise en œuvre présente deux inconvénients majeurs. Premièrement, l'expression informelle n'est pas exempte de défauts : ambiguïtés, sous-entendus, redondance, incohérences, etc. Elle risque donc d'introduire des erreurs. Deuxièmement, la vérification effective ne peut être que manuelle, en comités de vérification. Elle sera donc lourde et coûteuse, ce qui est un frein à son utilisation.

Pour être exploitable concrètement, la vérification doit, au moins en partie, être automatisée. Pour cela, il faut définir formellement les règles et trouver un algorithme qui permette de les vérifier. Il faut aussi trouver des moyens de structuration, car le nombre de propriétés et de règles est important.

Un premier pas a été franchi en ce sens par les concepteurs d'UML. Le méta-modèle d'UML, c'est-à-dire le modèle de sa notation, permet de définir formellement une partie des règles à vérifier : celle qui concerne l'organisation et l'agencement des éléments de la notation dans les diagrammes. Bien qu'il ne soit pas toujours accessible, et qu'il soit utile de commencer l'apprentissage d'UML par le guide de la notation, nous pensons qu'il est un outil de documentation essentiel. L'expérience nous l'a enseigné. En effet, seule la lecture du métamodèle nous a permis d'appréhender certains concepts et certaines règles de modélisation, car les exemples des autres ouvrages étaient ambigus et parce que les auteurs de ces derniers avaient des interprétations erronées ou partielles.

Le métamodèle fournit un extraordinaire moyen de structuration des règles, puis qu'on peut spécialiser des règles, déléguer via des associations, etc. Le modèle de vérification est un modèle objet à part entière. L'utilisation de la réflexion et du langage OCL (*Object Constraint Language*) [WK98] renforce le côté formel des expressions. Par exemple, l'expression `not self.allParents->includes(self)` dans le contexte d'un élément généralisable signifie que le graphe d'héritage est non circulaire.

On dit que le méta-modèle établit la syntaxe formelle du langage UML. En fait, ce n'est pas complètement vrai pour deux raisons. Premièrement, la sémantique du métamodèle est définie (presque) formellement à partir d'une notation définie informellement ! Ainsi, il n'a jamais été vérifié, à notre connaissance, que les nombreuses contraintes OCL du méta-modèle, qui semblent logique au premier abord, ne sont pas contradictoires. Elles sont certainement incomplètes. Deuxièmement, il s'agit de règles générales de combinaison. Or le sens des éléments d'un diagramme peut varier selon le contexte d'utilisation. Ainsi, un élément généralisable (contexte de l'exemple précédent) peut être une classe, une association, un stéréotype, un événement, une exception, un cas d'utilisation, une collaboration. Ce sont autant de concepts sémantiquement différents pour lesquels la notion de spécialisation diffère. Bon nombre de règles dépendent de la sémantique même des éléments. Par exemple, une association peut être spécialisée au même titre qu'une classe, mais la sémantique en termes de liens n'est pas claire : si les instances de l'association spécialisée sont aussi des instances de l'association générale (comme pour les classes) alors les règles de multiplicité sont certainement à redéfinir. Qu'en est-il de la redéfinition des caractéristiques suivantes : multiplicité, navigabilité, rôles, contraintes d'association, etc. Quoiqu'il en soit, le méta-modèle est un moyen pratique d'implanter des règles de vérification et donc de les automatiser dans les outils supportant UML. Un dernier reproche est fait au méta-modèle, il ne permet pour l'instant d'exprimer que des règles portant sur les relations statiques entre éléments [BHH+97].

La forme la plus finie d'expression des règles de vérification est celle basée sur une sémantique formelle d'UML. Si la notation est clairement définie, alors il est plus facile d'exprimer rigoureusement les règles et surtout d'automatiser leur vérification. C'est un travail difficile

et beaucoup de travaux y sont consacrés actuellement, nous y reviendrons dans la section 6.

Domaines de vérification

Etablir une vérification des neuf types de diagrammes sur différents modèles correspondant à différentes activités pour les trois catégories de propriétés que nous avons proposées ci-avant ne peut être une tâche atomique, car la complexité est ingérable.

Nous estimons que les diagrammes ont en quelques sortes des relations modulaires entre eux : on peut les grouper selon des critères de cohérence forte et de couplage faible. Nous proposons ainsi un premier découpage qui permet de réduire la complexité de la vérification. Le découpage proposé est basé sur les niveaux d'abstraction de la section 2.4. Chaque domaine correspond à un niveau d'abstraction :

- Le domaine externe regroupe les cas d'utilisation et les scénarios. Sa vérification a lieu dans l'activité d'analyse des besoins.
- Le domaine logique rassemble les autres diagrammes (classes, séquences, collaborations, états-transitions, activités). Sa vérification a lieu dans les activités d'analyse et de conception.
- Le domaine physique regroupe les composants et le déploiement. Sa vérification a lieu dans les activités d'implantation et de test.

La relation entre ces domaines est de type correspondance ou traçabilité.

3.3 Mise en œuvre de la vérification

Les principes de la section précédente nous permettent de poser les bases d'une mise en œuvre de la vérification. On ne s'intéresse ici qu'aux aspects du systèmes qui sont modélisables avec la notation, les besoins non-fonctionnels sont vérifiés dans les tests adaptés.

Vérification hiérarchique

La notation est découpée hiérarchiquement en domaines distincts, formant des sous-ensembles cohérents de notation. La vérification porte donc sur chaque domaine et sur les relations entre domaines.

Vérification inter-domaine En terme de vérification, elle correspond aux propriétés du processus de la section 3.1. On vise principalement la complétude et la cohérence entre les domaines via la traçabilité. Chaque élément d'un domaine devra trouver des éléments correspondants dans un autre domaine.
Notons que plusieurs modèles d'objectif différent peuvent utiliser le même domaine, c'est par exemple le cas des modèles d'analyse et de conception. La relation entre les modèles, dans ce cas, est aussi une relation de correspondance ou de raffinement. Leur vérification s'incrit donc dans le cadre de la vérification inter-domaine.

Vérification intra-domaine Il s'agit du cœur de la vérification. Les domaines sont composés de différents diagrammes qui doivent être cohérents et complets entre eux. En terme de vérification, cela correspond aux propriétés des modèles de la section 3.1. Nous détaillons dans la suite de la section la vérification pour chaque domaine.

Vérification diagramme Chaque diagramme décrit le système selon un aspect donné. La notation individuelle des diagrammes est en général restreinte et cohérente, elle s'appuie sur des théories plus ou moins formelles mais en général bien établies. En terme de vérification, les propriétés à vérifier sont celles du système de la section 3.1.

Nous détaillons maintenant ces différents aspects de la vérification.

Vérification inter-domaine ou inter-modèles

La vérification doit établir les correspondances entre éléments de modélisation de niveau (abstraction ou modèle) différent. Tout élément du niveau supérieur a pour cible au moins un élément du niveau inférieur. Tout élément du niveau inférieur a pour origine au moins un élément du niveau supérieur. Par exemple, un scénario (niveau externe) est réalisé par une collaboration (niveau logique), une classe d'analyse (niveau logique) est bien implantée dans le modèle de conception (niveau logique) et dans le modèle d'implantation (niveau logique). Elle doit aussi vérifier la cohérence des éléments liés. Par exemple, une opération ne peut être tracée par une classe. On vérifie donc la cohérence et la complétude hiérarchiques des modèles.

La relation entre éléments de domaines ou de modèles différents est notée en UML sous forme d'une spécialisation «trace» de la relation de dépendance. Même si une notation existe, elle est virtuelle en ce sens qu'elle n'a pas une sémantique précise et des calculs associés. Il s'agit d'une relation qui sert à la vérification du processus et non à la description du système. C'est pourquoi elle apparaît plutôt dans les manuels d'UML et d'UP que dans la réalité du développement.

La vérification est principalement manuelle, sous forme de revues de spécifications. On peut imaginer des solutions automatisées en implantant la relation dans les diagrammes UML. Cette solution présente deux risques :

- Confusion. Représenter explicitement la relation de traçabilité nuit à la lisibilité des modèles : le lecteur risque de confondre cet outil de documentation et d'aide à la vérification avec la modélisation intrinsèque du système.
- Explosion combinatoire. Sa représentation impliquerait des diagrammes énormes et délicats à structurer. C'est une relation hiérarchique non arborescente qui se combine mal avec les outils de structuration. Par exemple, un élément d'un paquetage d'analyse peut correspondre à plusieurs éléments d'un sous-système de conception et inversement un élément d'un sous-système peut correspondre à des éléments de plusieurs paquetages. La dépendance entre paquetage n'est plus stricte.

De plus les outils actuels n'incluent pas la notion de niveau d'abstraction.

D'autres solutions d'assistance sont envisageables :

- Représentation implicite. Les éléments de modélisation peuvent être enrichis de manière à prendre en compte la traçabilité : soit dans les champs de textes de la documentation soit dans des champs explicites. Dans le premier cas, on offre un support à la vérification manuelle, dans le second cas, on peut implanter des outils de test. La gestion de projet devra inclure cet effort supplémentaire de la modélisation.
- Bases de données. La traçabilité peut aussi donner lieu à une tâche à part entière dans les activités du développement. Pour ne pas confondre modèle et processus, on peut stocker les éléments dans des lexiques ou des bases de données, créer des relations entre éléments et définir des algorithmes de comparaison.

Niveau externe

Le niveau externe inclut les diagrammes de cas d'utilisation et les scénarios rattachés à ces diagrammes. La cohérence entre ces deux vues du niveau externe est basée simplement sur une association entre cas d'utilisation (UC) et scénarios : un scénario est rattaché à un seul cas d'utilisation. La syntaxe et le nombre de concepts sont réduits.

Certains contrôles syntaxiques sont implantables dans l'outil de modélisation :

- Unicité des noms, y compris dans les catégories (paquetages structurant les cas d'utilisation).
- La relation entre UC et acteur est une association.

- Les cas d'utilisation reliés à un acteur sont définis.
- La relation entre UC est une association, une dépendance ou une spécialisation.
- Les acteurs et les UC reliés à un cas d'utilisation sont définis.
- Les acteurs apparaissant dans un scénario d'un UC sont reliés à cet UC.

La représentation étant essentiellement textuelle, la vérification sémantique (cohérence, complétude) doit faire l'objet de revues et d'inspection. D'autant que certains points de la notation sont imprécis :

- La spécialisation de cas d'utilisation désigne-t-elle des fonctionnalités étendues ou réduites, des liens supplémentaires ou restreints ?
- Comment s'articulent associations, dépendances et spécialisations ?
- Quelles sont les conséquences des relations d'un cas d'utilisation avec les scénarios de ce cas : partage, enrichissement, invocation externe, etc ?
- Les stéréotypes d'extension et d'inclusion ne sont pas réellement intégrés dans la notation et leur sémantique est informelle.

Dans tous ces cas, la vérification est manuelle et doit s'appuyer sur une définition précise et claire, à défaut d'être formelle, des règles de modélisation et d'interprétation des concepts. Une présentation particulière des cas d'utilisation facilite leur vérification (chapitre 7, [AV01b]).

Niveau physique

Nous traitons le niveau physique avant le niveau logique car ce dernier est de loin le plus complexe. De plus le niveau physique présente des similitudes avec le niveau externe : la syntaxe et le nombre de concepts sont réduits, la sémantique n'est pas intrinsèque à la notation et pousse la vérification en dehors du domaine (le langage naturel pour le niveau externe, le langage de programmation pour le niveau physique).

Le niveau physique est basé sur les diagrammes de composants et de déploiement. La cohérence entre ces deux vues du niveau externe est basée simplement sur une association entre éléments et nœuds : un élément est rattaché à des nœuds. Comme pour le niveau externe, on peut distinguer la syntaxe de la sémantique.

Certains contrôles syntaxiques sont implantables dans l'outil de modélisation :

- Unicité des noms dans le système et les sous-systèmes.
- L'unique relation entre éléments est la dépendance. La relation entre nœuds est l'association.
- Les éléments reliés à un nœud sont définis. Inversement, tout élément doit être lié directement ou indirectement à un nœud.
- Les interfaces peuvent être associées aux composants et servir de point d'encrage aux dépendances entre composants.

La notation des diagrammes de composants et de déploiement est relativement pauvre. Il s'agit avant tout d'une définition structurelle du système logiciel (composants) et matériel (déploiement). La sémantique associée est tout aussi pauvre, ce qui est en complète contradiction avec la réalité puisque le logiciel et le matériel constituent des domaines extrêment complexes et variés. Notons que l'ajout de stéréotypes permet d'enrichir la notation. Par exemple, on distingue différentes sortes de composants (programmes, sous-programmes, tâches, composants génériques, processus, etc.). Les stéréotypes permettent aussi de référencer l'environnement système sous-jacent : composants (classe java, paquetage java, fichier source, exécutable, processus, interfaces, bibliothèque, répertoires, sous-systèmes, système existants...), dépendances (compilation, liaison dynamique...), liaisons (internet, middleware...), nœuds (PC, mainframe, dispositifs périphériques...). Mais, il n'y a là rien de rigoureux qui nous permette d'établir les bases d'une vérification.

En fait, la sémantique réelle est repoussée à l'implantation réelle du système, et à ce niveau là, il existe des descriptions formelles, des règles à respecter et des outils de manipu-

lation des spécifications (des programmes). En ce sens, nous prenons les modèles du niveau physique comme une représentation simplifiée de l'implantation système qui met en évidence son architecture, sans se soucier des détails. La vérification est celle du système implanté.

Niveau logique

Le niveau logique est véritablement celui sur lequel le travail de vérification a de l'importance car les erreurs non détectées à ce niveau risquent d'avoir des répercussions importantes pour la suite du développement.

Le travail consiste à définir un modèle objet cohérent sous ses différents aspects et qui permette de poser les propriétés attendues du système modélisé. On considère un seul modèle appelé modèle logique. La vérification est celle des propriétés du système (section 3.1) et celle des propriétés des modèles (section 3.1). Nous lui consacrons la section suivante.

4 Vérification du modèle logique

4.1 Introduction

Dans cette section, nous abordons la notion de description multi-vue cohérente. La vérification qui nous intéresse dans cette partie est statique c'est-à-dire avant l'implantation du système. La vérification statique prend en compte les combinaisons licites d'éléments de modélisation et des propriétés de la spécification (cohérence, complétude). Elle est basée sur la syntaxe, le typage des éléments, ou la cohérence des contraintes (cardinalités, notations de contraintes, expressions OCL). D'un point de vue langage et compilation, il s'agit de la syntaxe et de la sémantique statique. La syntaxe est la disposition correcte des éléments de modélisation, la sémantique statique est une restriction des combinaisons possibles, selon certaines propriétés des spécifications (cohérence, complétude, etc.). On ne tient pas compte d'un "contenu" réel ou d'une exécution (sémantique dynamique).

En UML, la description du langage s'appuie sur le métamodèle. On peut dire que la structure du métamodèle définit la syntaxe. Par exemple, un "classificateur" ou *classifier* (classe, UC, nœud) a des propriétés (opérations ou attributs) et peut faire l'objet de connexions avec des associations. Une opération ne peut être en relation directe avec une association. La vérification syntaxique ne tient pas compte a priori des commentaires, mais dans le cas des contraintes exprimées avec OCL, on peut vouloir inclure la vérification (syntaxique) des expressions OCL.

Du fait de l'héritage, de nombreuses combinaisons sont acceptées par le métamodèle, par exemple une classe peut être associée à un cas d'utilisation. Or, cette combinaison acceptable "syntaxiquement" ne correspond à rien en terme de langage UML. Elle doit être interdite. Des contraintes OCL ont donc été ajoutées dans le métamodèle, qui réduisent le nombre de combinaison licites. On peut qualifier ces contraintes de sémantique statique. Nous distinguons trois niveaux de règles statiques :

1. Règles d'agencement : par exemple, une interface ne contient pas d'autres propriétés que des opérations, la relation d'héritage est anti-réflexive.
2. Contraintes : typage, cardinalités, contraintes entre associations, contraintes OCL.
3. Cas particuliers liés à une connaissance plus profonde de la sémantique des concepts : par exemple, il n'y a pas d'héritage multiple pour des sous-classes d'un discriminant, un objet passif ne peut envoyer deux messages simultanément... Dans cette catégorie rentrent aussi les règles associées aux stéréotypes.

Comme nous l'avons vu précédemment, le modèle logique est constitué d'un ensemble de diagrammes représentant des aspects complémentaires et non disjoints du système à modéliser. Nous avons donc naturellement deux niveaux de vérification : le niveau diagramme et le niveau

modèle. Nous étudions la vérification individuelle des diagrammes dans la section 4.2 et la vérification intégrée des diagrammes dans la section 4.3. De manière approximative, on peut dire que l'effort de vérification des diagrammes est plus proche de la syntaxe tandis que celui de la vérification des modèles est orienté vers la sémantique.

Les règles statiques sont très nombreuses et nous ne pouvons toutes les citer ici. Nous donnons quelques exemples significatifs de règles statiques et de "trous" pour ces règles dans le métamodèle actuel. Nous les avons classées selon les niveaux proposés ci-avant. Nous nous sommes basés sur la référence actuelle d'UML [Con99] en séparant lorsque cela est nécessaire les règles qui viennent du métamodèle (*UML Semantics* !) et celles issues du guide de la notation (*UML Notation Guide*) qui contient des explications informelles et des exemples. Nous évitons autant que possible les autres sources car ce sont souvent des interprétations ou des usages pratiques. Nous précisons aussi, lorsque nous l'avons trouvée, la page dans le manuel de référence de la notation UML [Con99]. Enfin, pour des raisons d'espace, nous n'avons pas illustré les règles par des schémas UML. Un exemple simple de vérification est proposé dans l'exercice 6.12 du chapitre 9 de [AV01b]. Un autre exemple est donné dans un article de Gogolla [Gog98]. Le lecteur trouvera en complément dans [SG98a, SG98b, SG01] un catalogue de différents problèmes de modélisation avec UML 1.3. Périn [Pér00] propose des solutions aux problèmes de cohérence entre vues.

4.2 Niveau diagramme

Un diagramme est un cadre de description homogène et comportant un nombre réduit de concepts. Pour chaque diagramme du niveau logique, nous précisons le sens de la vérification sans donner la liste exhaustive des règles à vérifier.

Les règles de vérification associées à la syntaxe (générale) des descriptions UML sont définies par la structure du métamodèle. Elles ne présentent pas beaucoup d'intérêt dans ce chapitre car elles sont vérifiées par la plupart des outils d'UML qui respectent la norme. C'est pourquoi nous en faisons abstraction dans ce qui suit.

Règles générales

Dans cette section, nous illustrons quelques règles statiques générales. Par convention du méta-langage, le concept "*classificateur*" est une généralisation des éléments du langage relatifs aux types (classe, type de données, interface, nœud, composant, cas d'utilisation, paquetages, etc.).

Les règles suivantes sont de niveau 1.

1. La première vérification consiste à savoir pour un diagramme donné quels sont les éléments de modélisation UML autorisés. Assez curieusement, ce n'est pas la préoccupation du métalangage. Ce dernier définit un modèle "à plat" des concepts (éléments de modélisation), des relations entre concepts et des règles de bon usage (informelles ou en OCL). Les seuls éléments pour lesquels il y a un lien évident élément-diagramme sont les collaborations, les machines à états et les graphes d'activités. Pour les autres, la réponse est dans le guide de la notation (informel + exemples).
2. Chaque élément de modélisation a un nom unique dans son espace de nommage (un classificateur) (p 2-39). Le métamodèle ne précise pas, mais on peut le deviner, que le nommage des paquetages est hiérarchique.
3. Chaque élément de modélisation a une visibilité dans son espace de nommage. Si pour les éléments statiques (classe, attribut, opération, association) la visibilité a un sens dans leur concept englobant (classe, paquetage) on a du mal à voir ce que cela signifie pour un message, une collaboration, un cas d'utilisation, un état...

4. Une propriété (*feature*) est un attribut ou une opération. Une méthode est une opération avec un corps. Les propriétés sont "encapsulées" (relation de composition) dans les classificateurs (p. 2-14 et 2-32). L'usage de la composition colle assez mal avec la cardinatité `0..1` dans ce cas. Cela signifie qu'on peut trouver des propriétés indépendantes ou reliées à autre chose.
5. Une contrainte est associée à un élément de modélisation, le contexte de la contrainte est le nom de l'élément de modélisation associé et son espace de nommage. Notons qu'une contrainte peut être associée à une contrainte (quel est le sens d'une contrainte de contrainte ?) et à elle-même !
6. Un élément généralisable, support de la relation de généralisation, ne peut être une généralisation de lui-même.
7. Les stéréotypes sont des éléments généralisables (supports de la relation de généralisation) associés à des éléments de modélisation et en particulier à des contraintes. Chose curieuse, dans le métamodèle ([Con99], p. 2-67), chaque contrainte est rattachée à un et un seul stéréotype. En y regardant de plus près, on constate qu'une contrainte est associée à au moins un élément de modélisation "et" exclusivement (xor) à un stéréotype. Il y a visiblement une erreur d'interprétation de la part des concepteurs. Nous interprétons la règle comme suit : "*une contrainte est associée soit à un stéréotype soit à des éléments de modélisation*". C'est donc une contrainte de totalité (au moins un des deux) et d'exclusion (pas les deux). Cela se traduit par des cardinalités minimales de 0 dans les deux cas et une contrainte mixte `xor`.
8. Chaque élément de modélisation est défini obligatoirement par un nom dans le métamodèle. En pratique, certains noms sont facultatifs (objets, associations, contraintes, etc.). Le (langage du) métamodèle ne permet pas la définition de propriétés, cela ne correspond donc pas à la réalité. La propriété *name* est aussi surchargée, elle n'a pas le même sens pour un objet (son identité ?), une opération (son profil), un lien, un message, etc.

Les exemples prédécents mettent en évidence le fait que association et spécialisation se marient esthétiquement bien dans le métamodèle mais induisent nombre d'erreurs si on ne contrôle (contraint) pas l'usage des éléments hérités. C'est à notre avis la faille essentielle dans le métamodèle d'UML.

Les règles générales de niveau 2 s'appuient sur le typage. Le typage permet de vérifier le profil des opérations, le type des attributs et le type des expressions OCL.

Il n'y a pas de règle de niveau 3, puis qu'il n'y a pas de sémantique "générale". On peut placer dans cette catégorie la vérification ou l'évaluation d'expressions OCL.

Diagramme de classes

Nous étudions dans cette section la notation relative aux diagrammes de classe. La notation du diagramme de classes est la plus complète (complexe) de la notation UML. Elle est en quelque sorte une généralisation des modèles objets des langages de programmation classiques (séquentiels) tels que C++, Smalltalk, Eiffel ou Java, abstraction faite des concepts de la concurrence. Pourtant, à la base, il s'agit d'un simple réseau sémantique avec des concepts (classes) et des liens (relations).

Discutons de quelques exemples de règles de niveau 1.

1. Un diagramme de classe comprend les éléments de structure statique : classes, types, relations et leurs variations. Il ne comprend pas d'éléments du comportement temporels ([Con99] *UML Notation Guide*, p. 3-33). Rien ne permet de le vérifier formellement dans le métamodèle.
2. Chaque élément de modélisation a un nom unique dans son espace de nommage (règle de la section 4.2). Une propriété d'instance ne peut de ce fait avoir le même nom qu'une

propriété de classe. Deux opérations se distinguent par leur nom (p. 2-45). La surcharge avec un profil différent est donc interdite, ce qui n'est pas un usage courant en objet.

3. Les propriétés sont définies dans les classificateurs. Dans le diagramme de classes, on doit restreindre ces classificateurs aux classes, aux associations ou aux qualificatifs d'association. Une interface ne contient que des opérations.
4. Les opérations sont définies dans les classes ou classes-association (spécialisation "multiple" de classe et association) mais pas dans les nœuds ou les cas d'utilisation.
5. Le type des paramètres d'une opération est inclus dans l'espace de travail du classificateur associé (p. 2-45). L'usage des types primitifs et des types paramétriques entre-t-il dans le cadre de cette règle ?
6. Si une classe est concrète (non abstraite) toutes ses opérations doivent avoir une implantation en terme de méthode (p. 2-46). Cela colle difficilement avec la réalité des modèles UML rencontrés.
7. Les relations entre classes peuvent être de type généralisation, association ou dépendance. Les relations de spécialisation, d'agrégation ou de composition sont anti-réflexives et anti-symétriques. Les graphes de ces relations sont orientés et sans circuit.
8. Une association est une relation composée de deux terminaisons vers des classificateurs. Par héritage, une classe peut donc être associée à un composant. Il faut définir des contraintes dans les sous-classes pour qu'une classe soit associée uniquement à des classes dans le modèle des classes. Cet exemple illustre parfaitement le problème des règles générales et les abus de la relation de spécialisation dans le métalangage.
9. Selon le métamodèle, toute association a un nom (unique comme nous l'avons vu). En pratique, les noms d'associations ou de rôles sont facultatifs. Si deux associations (sémantiquement) différentes existent entre deux classes alors d'une part elles ont obligatoirement un nom d'association ou de rôle et d'autre part tous ces noms doivent être différents.
10. Une association est navigable dans au moins un sens.
11. Deux attributs de qualification d'une association doivent avoir des noms différents. Il n'y a pas d'interférence avec les propriétés des classificateurs.

Abordons maintenant les vérifications de niveau 2. Certaines règles viennent des modèles Entité-Association. La liste est loin d'être exhaustive.

1. Le type des propriétés et des paramètres peut-il être :
   - contraint à certaines sous-classes de *Classifier* ? (exemples : attribut de type UC dans une classe ?)
   - restreint par une expression OCL ? (exemples : généricité contrainte des classes ?)
   - une expression de type ? un type existant ?

   Le métamodèle ne fournit pas de réponse à ces questions.
2. Toute propriété (resp. association) dérivée doit être accompagnée d'une expression OCL valide indiquant le mode de calcul de la dérivation. Une association dérivée doit être compatible, d'un point de vue cardinalités et contraintes, avec les associations dont elle dépend.
3. La composition est exclusive. Dans la relation de composition, un objet est composant d'au plus un composé (cardinalité maximale de 1) (p. 2-45, *AssociationEnd*) et cet objet ne peut être lié directement par une autre relation de composition sauf en présence de cardinalités minimales de 0 à chaque fois et d'une contrainte d'exclusion (`xor` ou `nand`) entre les deux associations de compositions. Notons que la composition indirecte est implicite par transitivité de la relation de composition.

4. Si deux sous-classes sont exclusives (discriminant ou contrainte `disjoint`) alors un héritage multiple de ces classes est interdit.
5. Certaines vérifications portent sur la cohérence entre contraintes posées sur les associations (cardinalités, notations de contraintes, expressions OCL).
   - Les cardinalités des associations doivent être cohérentes.
     (a) Une association avec une cardinalité (unique) 0 sur une extrémité n'a pas de sens.
     (b) Une association binaire avec la cardinalité 1 sur chaque extrémité signifie que les deux objets sont intimement liés. On doit vérifier qu'il s'agit bien de deux objets différents. Sauf cas particulier, une telle association doit être une composition.
   - Les contraintes entre associations doivent être compatibles avec les cardinalités des associations.
     (a) La contrainte standard d'exclusion `xor` n'est pas compatible avec la cardinalité minimale de 1. La règle est identique pour la contrainte de totalité disjonctive `or` et la contrainte d'unicité (présence dans une seule association).
     (b) La contrainte de totalité `and` est redondante avec la cardinalité minimale de 1 sur chaque extrémité d'association concernée.
     (c) La contrainte d'égalité `=` implique des cardinalités identiques sur chaque extrémité d'association concernée.
   - Les contraintes entre associations doivent être cohérentes entre elles.
     (a) Les contraintes sur associations ne sont acceptables que pour des ensembles comparables. Par exemple, l'inclusion (`subset`) d'association n'est possible que si les extrémités sont égales.
     (b) Les contraintes portant sur une association doivent être cohérentes. Par exemple, l'inclusion (`subset`) d'association est incompatible avec les contraintes `xor` et `nand` et partiellement redondante avec les contraintes `or` et `and`.
     (c) La vérification entre associations deux-à-deux n'est pas suffisante. Il faut vérifier les contraintes globalement ([ARRV00b], p. 10).
6. La vérification des expressions OCL doit prendre en compte les règles de visibilité des éléments.

Abordons maintenant les vérifications de niveau 3. Ces vérifications se traduisent plus souvent par des questions ou des conseils que par de véritables règles.

1. Quel est le sens de lecture d'une association n-aire (n>2) ? Autant en Merise la réponse est simple, autant en UML ou OMT les interprétations varient.
2. Le type d'un attribut est un classificateur selon l'association `type` (p. 2-14). Cela soulève les problèmes suivants :
   - Ce type doit-il être uniquement un type primitif (et donc respecter par là les règles de normalisation des modèles entité-association ou relationnels) ou un type général (primitif ou classe) ?
   - Quel sens cela a-t-il de typer un attribut par un cas d'utilisation, un nœud, une interface ?
   - Le type d'un attribut est obligatoire. Comment modéliser un attribut optionnel ?

   Le type d'un attribut est fixé par une contrainte dans le métamodèle (p. 2-56) : il est une classe, un type de données ou une interface. Deux solutions sont possibles pour modéliser un attribut optionnel : créer une association avec une cardinalité `0..1` ou utiliser des valeurs "fantômes" (ex. un pointeur `nil`).
3. La classe `Parameter` (p. 2-14, 2-41) pose également quelques problèmes :
   - Le type d'un paramètre est un classificateur selon l'association `type` (p. 2-14), ce type peut-il être un cas d'utilisation, un nœud, une interface ou même une classe ? Dans chaque cas, quel est le sens associé ? Le métamodèle ne fixe pas de solutions.

- Les règles sont-elles les mêmes pour un paramètre classificateur et un paramètre d'opération ? La nature `kind` du paramètre définit son rôle (`in`, `out`, `inout`, `result`). Cela a un sens pour les opérations mais pas pour les classificateurs. Cela devrait être une propriété de l'association entre `Parameter` et `Operation`.
- Le type est obligatoire, que fait-on si le paramètre est aussi paramètre d'une opération ? Le métamodèle de la page 2-14 est erroné : les cardinalités doivent être `0..1` dans chaque cas avec une contrainte d'exclusion `xor` (voir page 254).

4. Le corps d'une méthode est une *ProcedureExpression* (p. 2-14) c'est-à-dire une expression définie par un nom de langage et une chaîne de caractères (p. 2-75). Aucune vérification n'est donc possible à ce niveau sur les méthodes *i.e* les opérations et objets invoqués.

5. De nombreuses questions sont relatives à l'utilisation de la relation de spécialisation :
   - Quelle est la sémantique de l'agrégation par rapport à celle de l'association ou à celle de la composition ? Ce problème est discuté dans [HSB99]. A notre niveau, nous considérons que l'agrégation est une association asymétrique non réflexive et que la composition est une agrégation exclusive.
   - Les cardinalités incluent-elles les sous-classes ?
   - Les contraintes entre associations incluent-elles les sous-classes ?
   - Peut-on définir des contraintes entre associations de classes et de sous-classes ?
   - Peut-on redéfinir les relations ? Lesquelles ? Selon quelles règles ?
   - Peut-on redéfinir les propriétés ? Lesquelles ? Selon quelles règles ?

   Des exemples de tels problèmes sont étudiés dans [ARRV00b, ARRV00a, AV01b].

6. D'autres questions sont relatives à l'usage de stéréotypes. Le manuel de référence propose plusieurs stéréotypes pour certaines notations. Le métamodèle indique des contraintes sur certains de ces stéréotypes "standards" tels que la contrainte `xor`, les contraintes d'héritage, les classes abstraites, l'agrégation, etc.

   Le traitement des stéréotypes n'est pas homogène et standard : certains stéréotypes comme les classes abstraites ou l'agrégation/composition sont traités à part entière par des attributs (`isAbstract`, `aggregation`, `isQuery`) dans le métamodèle, d'autres se déduisent -sans que la notation le précise- (par exemple une méthode abstraite est une opération sans méthode), d'autres sont uniquement des commentaires (voir les descriptions des éléments de la notation p. 2-18 à 2-43 [Con99]).

   Peut-on trouver un traitement homogène des stéréotypes dans lequel on puisse définir la notation et la sémantique (contraintes OCL par exemple) de stéréotypes ? Il faut dans ce cas se pencher sur le problème de la représentation de la relation entre un élément et son stéréotype. Actuellement, il s'agit d'une association : on peut lier chaque élément à (au plus) un stéréotype (p. 2-67). La spécialisation nous semble plus adaptée car on hérite de l'ensemble de la sémantique de l'élément "spécialisé".

7. La relation de dépendance n'a pas de sémantique précise. Elle désigne le fait qu'un élément a besoin (utilise) d'un autre élément pour sa définition. Elle peut correspondre à l'importation modulaire (entre paquetages ou entre classes), à la relation de clientèle, etc. Elle peut être considérée dans certaines interprétations comme redondante avec la relation d'association. Elle peut être stéréotypée en instanciation (dépendance entre l'objet et sa classe), raffinement (*«derive»*, *«trace»*, *«realize»*, *«refine»*), inclusion (*«include»*, *«extend»*), etc. Toutes ces interprétations nécessitent une sémantique et induisent des contrôles adaptés.

Diagramme d'interaction

Nous étudions dans cette section la notation relative aux diagrammes de collaboration et diagrammes de séquence. Nous avons choisi de ne pas les traiter séparément car ces deux

diagrammes partagent la même sémantique. Dans les deux cas, il s'agit de collaboration entre objets, c'est pourquoi ils partagent le même modèle "sémantique" dans le métamodèle, le concept de *Collaboration*. Les diagrammes diffèrent par l'aspect mis en évidence : le temps pour les diagrammes de séquence et la structure pour les diagrammes de collaboration.

Le fait d'avoir deux vues différentes sur les mêmes concepts permet de factoriser la notation mais rend difficile la séparation des deux. En effet, à notre connaissance, rien dans le métamodèle, c'est-à-dire le manuel de référence ([Con99] *UML Semantics*), ne permet de distinguer les deux utilisations (et donc les deux diagrammes). Seules les règles syntaxiques générales sont fournies. La différence se trouve explicitée informellement dans le manuel de la notation ([Con99] *UML Notation Guide*) mais avec un vocabulaire qui entretient la confusion entre collaboration et diagramme de collaboration.

Compte-tenu de cette remarque, nous organisons cette section en trois parties : la collaboration, le diagramme de collaboration et le diagramme de séquence. Les règles générales sont concentrées dans la première partie, les deux autres parties précisent des restrictions ou des interprétations particulières.

a) La collaboration Contrairement au diagramme de classe, la collaboration dispose dans le métamodèle d'une unité de structuration, l'élément de modélisation -le concept- *Collaboration*. Ainsi, un diagramme de collaboration ou de séquence est typiquement une "instance" de ce concept. Une collaboration fait référence exclusivement soit à un classificateur soit à une opération, dont il est une réalisation, et qui représente en quelque sorte son contexte.

Structurellement, une collaboration inclut des éléments de modélisation, des interactions, des rôles d'association et de classificateurs (p. 2-104). Détaillons ces éléments :

- Les rôles signifient ici des usages particuliers des concepts associés. Il ne faut pas les confondre avec la notion de rôle dans une association. Pour ce faire, nous utiliserons parfois les termes association type et classificateur type. Ce sont des exemple types des concepts mais pas directement des instances. Il faut bien comprendre la nuance apportée par le métamodèle à ce niveau. La plupart des ouvrages sur UML présentent les diagrammes de collaboration comme des diagrammes d'instance, avec des instances et des liens. Mais le métamodèle ne fait référence, pour une collaboration, qu'à des associations type et classificateurs type. S'agit-t-il d'un abus de notation des ouvrages sur UML ? Non, car les instances et les liens sont liés à des classificateurs et des associations et donc par spécialisation à des classificateurs type et des associations type. Une autre réponse est dans le guide de la notation ([Con99] *UML Notation Guide*, p. 3-97 et 3-109) : *"A collaboration diagram can be given in two different forms : either at specification level (ClassifierRoles, AssociationRoles, and Message) or at instance level (Objects, Links, and Stimuli)."*; dans le premier cas, la collaboration est un patron (*pattern*). En résumé, dans un diagramme d'interaction, on trouve trois niveaux d'abstractions :

  1. instances (objets, liens),
  2. rôles (classificateurs type, associations type),
  3. types (classificateurs, associations, relations).

- Les éléments de modélisation (agrégation *constrainingElement* p. 2-104) sont des contraintes *"The model element that add extra constraints, like Generalization and Constraint"* ([Con99] *UML Semantics*, p. 2-106).
- Une interaction est un ensemble de messages (composition, p. 2-104). Chaque message est associé à une association type (*communicationConnection*), deux classificateurs type (*sender*, *receiver*) et une action. On peut préciser pour chaque message les liens de causalité (*activator* : le messages qui est à l'origine du message courant), et de précédence (*predecessor* : les messages qui doivent être exécutés avant le message courant).

– Une action est une communication, une invocation d'opération, un résultat, une création, un envoi de signal, une terminaison, une destruction, etc. (p. 2-85). Elle génère le stimulus associé au message. C'est l'action qui fixe le mode de la synchronisation de la communication.

La cardinalité minimale entre la collaboration et les associations ou classificateurs type définit, approximativement, la différence entre une collaboration du diagramme de collaboration et une collaboration du diagramme de séquence.

Discutons de quelques exemples de règles de niveau 1.

1. Comme pour les classes -règle 1 du niveau 1-, la première règle consiste à vérifier que les éléments de modélisation d'un diagramme d'interaction sont pertinents (un cas d'utilisation ne le serait pas). Au premier abord et contrairement au diagramme de classe qui n'a pas de représentation dans le métamodèle, on peut s'appuyer sur l'élément de modélisation *Collaboration* du métamodèle (p. 2-104) pour déterminer les éléments autorisés. En fait c'est un peu plus subtil : nous avons vu qu'une collaboration peut contenir des éléments de modélisation, et a priori n'importe lequel. Le manuel indique (informellement) que ce sont des contraintes. Tout cela reste à préciser et vérifier.
2. Une collaboration est la représentation d'un classificateur ou d'une opération. Le métamodèle ne précise pas quels classificateurs sont autorisés ni le sens de la collaboration pour une opération.
3. Une collaboration est perçue soit au niveau spécification soit au niveau instance (cf discussion en préambule). Il y a confusion possible entre les niveaux car chaque niveau est facultatif (objet anonyme, rôle implicite, pas de classe spécifiée) dans un diagramme d'interaction mais il doit y en avoir au moins un pour que l'objet ait un sens. Par ailleurs, dans le métamodèle, tous les noms (donc tous les niveaux) sont obligatoires. Comment faire ?
4. La relation entre un classificateur type ou une association type et leur concept de base est double : spécialisation et association. Il n'y a pas redondance entre les deux relations :
   – La spécialisation met en évidence que les règles de structuration d'une collaboration sont les mêmes que celle des classes et relations : un diagramme d'interaction (en fait seuls les diagrammes de collaboration l'autorisent -mais cela n'est indiqué qu'informellement-) est un diagramme de classe avec des messages en plus. Cette relation n'existe pas pour les instances, c'est ce qui fait la différence entre un rôle et une instance.
   – L'association met en évidence le fait que le rôle est défini par un type. C'est pourquoi, dans un diagramme d'interaction par exemple, les objets ont trois noms : un nom d'instance, un nom de rôle et un nom de classe (la classe de rattachement de l'instance).

   A ce niveau les points suivants nous semblent à préciser pour la vérification :
   – Dans la pratique, le nom du classificateur type (le nom de rôle) est facultatif ; quel est le rôle par défaut d'un objet ?
   – Doit-il y avoir cohérence entre les éléments hérités et les éléments associés ? Par exemple, les propriétés et relations d'un classificateur type sont-elles exactement celles du classificateur correspondant : mêmes noms d'attributs, d'opérations, d'associations, de rôles d'association, de cardinalités, etc ? C'est un problème à la fois crucial et complexe qui rejoint les préoccupations de la section 4.3.
   – Dans la pratique toujours, lorsqu'on représente un objet (ou son classificateur type), on représente très rarement les compartiments attributs et opérations. Mais la notation l'autorise ([Con99] *UML Notation Guide*), p. 3-119). L'interdire en partie (par exemple, on peut vouloir donner des valeurs d'attributs pour une instance) ou complètement permettrait de simplifier les contrôles de cohérence.

5. Chaque élément a un nom unique dans son espace de nommage. Quel est le nom d'un objet anonyme, d'un rôle ? Deux objets différents ont des représentations différentes. Si une même action est envoyée par le même objet au même objet, les messages supports sont-t-ils identiques ou pas ? En pratique, dans le diagramme de séquence, on traduira cela par deux flèches différentes et dans le diagramme de collaboration par un numéro d'ordre différent.
6. Le résultat d'un appel d'opération a une double représentation (flèche à trait discontinu et notation " :="). S'il y a redondance, peut-il y avoir incohérence ? Y-a-t-il un modèle normalisé (à nombre minimal de concepts) pour faciliter la vérification ?

Les vérifications de niveau 2 sont essentiellement spécifiques aux diagrammes de collaboration, de séquence et surtout à la cohérence avec les autres diagrammes du niveau logique.

1. La première interrogation soulevée à ce niveau concerne la dépendance entre une collaboration et un classificateur ou (exclusivement) une opération (p. 2-104). Peut-on avoir des collaborations en dehors de ces cas ? Si non, il faut placer une contrainte de totalité.
2. La cible d'une action d'un message (*target*, p. 2-85) doit être compatible avec le receveur du message (un classificateur type). Nous n'avons pas trouvé une telle règle.
3. Concernant les vérifications de type liées au nommage des éléments, deux objets peuvent-ils avoir le même nom pour une classe différente ? Cela contredirait la règle de nommage des éléments dans leur espace de nommage. Il faut le vérifier.
4. Une action de retour doit être précédée d'une invocation d'opération.
5. A l'intérieur d'un diagramme d'interaction, on doit vérifier la cohérence entre les éléments et leurs rôles (typage). La cohérence entre rôles et classificateurs est assurée dans la vérification de cohérence globale (section 4.3).

Discutons de quelques exemples de règles de niveau 3.

1. Une collaboration est un élément de modélisation avec une représentation visuelle (ellipse en trait discontinu faisant référence à la notion de cas d'utilisation). Mais il n'existe pas de diagrammes dont la collaboration serait un constituant. Dans les ouvrages, cette représentation apparaît pour souligner qu'un cas d'utilisation est réalisé par des collaborations (traçabilité).
2. Une collaboration est un élément généralisable. Mais nous n'avons pas trouvé de règles de spécialisation d'une collaboration. Ce problème est lié à celui de la règle précédente.
3. La notation ne met pas assez en évidence les différences entre instance/rôle/type, même si des règles de nommage existent pour les objets (p. 3-119 et 3.120). Nous estimons que cela induit des erreurs d'interprétation dans les diagrammes d'interaction. Nous y reviendrons notamment pour le diagramme de collaboration. La suppression du niveau rôle nous semble s'imposer.
4. La diffusion de messages n'existe pas, même si on parle d'envoi multiplexé à un multi-objet. Dans le métamodèle, un message a un émetteur unique et un récepteur unique. Pour envoyer un message à plusieurs objets, on passe par une collection. La collection devra alors être un objet spécifique dont la classe définit une opération qui prend en compte la diffusion vers les éléments de la collection par un envoi de message itératif. En aucun cas, il y a envoi simultané (en parallèle) des messages.
5. Même si le métamodèle permet l'usage des stéréotypes dans une collaboration, en pratique les diagrammes d'interaction ne définissent pas de stéréotypes propres. Les stéréotypes sont ceux des diagrammes de classe. Les variations de messages sont définis par spécialisation de la classe `Action` et non par stéréotype. Ainsi certaines vérifications sont fixées dans le métamodèle (p. 2-95 à 2-99) :

- Une action de création n'a pas de cible mais est reliée à une classe. A qui est alors envoyé le message ?
- Une action de suppression n'a ni cible ni arguments.
- Une action de terminaison n'a pas d'arguments.
- Un appel d'opération a des paramètres effectifs conformes à son opération -cohérence instance/classe de la section 4.3).
- Un envoi de signal est toujours asynchrone.

6. Si un objet envoie des messages asynchrones, alors il doit être actif. Si un objet reçoit des messages asynchrones, il n'est pas forcément actif.
7. Tout envoi de résultat est précédé d'un appel d'opération. Notons que l'objet appelant n'est pas nécessairement le receveur du résultat, un autre objet, la continuation, peut recevoir le résultat.
8. On ne connaît pas la sémantique de la réception de messages (non stocké, ordonnancé dans une file, gérée avec priorité ou pas, etc.).

b) Le diagramme de collaboration Comme le diagramme de classes, le diagramme de collaboration est un diagramme structurel avec des concepts (objets) et des relations (liens). Les éléments d'un diagramme de collaboration font référence aux éléments des diagrammes de classes (voir section 4.3).

Revenons sur quelques règles de niveau 1.

1. La règle 1 du niveau 1 de la partie collaboration peut être reprise et affinée au niveau du diagramme de collaboration.
2. Une collaboration est la représentation d'un classificateur ou d'une opération. Le métamodèle ne précise pas les représentations utilisables pour le diagramme de collaboration.
3. Par définition du métamodèle, un diagramme de collaboration est un diagramme de classe (voir les remarques de la règle 4 du niveau 1 de la partie collaboration). De ce fait, TOUTES les règles des diagrammes de classes sont applicables ici ! Cela répond alors en partie aux interrogations de la règle 1. Nous estimons qu'il s'agit d'un abus d'utilisation de la relation de spécialisation pour les classificateurs type et les associations type.
4. Quelques éléments de la notation UML présentés dans le guide de la notation ([Con99] *UML Notation Guide*) ne figurent pas dans le métamodèle (*UML Semantics*) : multi-objets, numérotation hiérarchique des messages dans une collaboration, flots de contrôle multiple, gardes, envoi itératif. Soit nous ne les avons pas vus, soit il y a là une lacune. La diffusion de messages est réalisée par un envoi à un multi-objet (une collection) comme nous l'avons vue dans les règles de niveau 3 de la collaboration.
5. Le diagramme de collaboration est utilisé à deux niveaux : le niveau instance et le niveau spécification (rôle) ([Con99] *UML Notation Guide*, p. 3-101, p. 3-109 et exemple p. 3-113). Les règles d'utilisation et de modélisation nous apparaissent différentes au travers de l'exemple de la page 3-113, mais elles ne sont pas définies formellement et contredisent en partie le métamodèle. Voici ce que nous avons pu déduire de nos observations. Nous appelons objets les boîtes et liaisons les traits entre les boîtes.
   - Niveau Instance.
     (a) Le nom de l'objet est souligné (il a au moins un nom d'instance ou un nom de classe).
     (b) Les liaisons entre objets sont des liens : il n'y a pas de cardinalités mais on peut trouver des rôles d'associations, des attributs d'associations qualifiées, des notations d'agrégation et de composition.
     (c) Le métamodèle autorise la relation de dépendance (quel sens ?) mais pas la spécialisation (c'est une relation entre classificateurs).

– Niveau Spécification.
  (a) Le nom de l'objet n'est pas souligné (il a au moins un nom de classe et optionnellement un nom de rôle mais pas de nom d'instance). Les objets sont des classificateurs type (des rôles).
  (b) Les liaisons entre objets sont des associations type (des rôles) et donc des associations, avec les notations et règles des diagrammes de classes.
  (c) Les relations de dépendance ou d'héritage sont possibles.

  Dans les deux cas, la notation des messages est identique et il n'y a pas de concepts structurants *i.e.* paquetage. Il faut pouvoir distinguer les deux utilisations et vérifier les règles associées dans chaque cas. Comment ? Le métamodèle ne résoud pas ce problème.

Dans le diagramme de collaboration, les règles de niveau 2 sont implicitement celles de niveau 2 des diagrammes de classe de la section 4.2 (par spécialisation des éléments de modélisation). Nous discutons de quelques points particuliers.

1. Il y a correspondance implicite entre le niveau instance et le niveau collaboration ([Con99] *UML Notation Guide*, p. 3-101) mais il faut vérifier la cohérence des noms et des types. Par exemple, un rôle sur un lien doit-il être conforme à un nom de rôle sur une association type ?
2. Les vérifications de type liées aux éléments non déterminés (règle 4 du niveau 1) sont applicables ici.
3. La vérification des annotations et des contraintes suit les règles générales de la notation.
4. Lors d'un envoi synchronisé (association *predecessor* du métamodèle), il faut vérifier que les messages de la liste des prédécesseurs existent et sont conformes.
5. Certains envois de messages impliquent un retour de valeur, le résultat est stocké dans une variable qui est parfois utilisée comme paramètre d'un autre envoi de message. On doit vérifier les règles de typage des arguments, des variables et des résultats. En particulier, le résultat d'une action (opération) est un argument spécial de l'envoi de message (dans le métamodèle). Ces vérifications de cohérence dans la collaboration s'ajoutent aux vérifications de cohérence entre appel d'opération (collaboration) et spécification d'opération (classes) de la section 4.3.
6. Vérification des contraintes de précédence dans les envois synchronisés.

Discutons de quelques règles de niveau 3.

1. Revenons sur la règle 3 du niveau 3 des collaborations (p. 260). Un trait entre deux objets doit-il être perçu comme un lien (avec un nom spécifique) ou une association type (un rôle) ? Une relation d'héritage entre objets (p. 2-117) a-t-elle un sens ? Les règles de vérification sont différentes mais on ne sait pas quelle est la bonne interprétation. Nous pensons que l'introduction des trois niveaux, facilitée par la définition orientée objet du métamodèle et l'utilisation abusive de la relation de spécialisation, présente plus de risques d'erreurs de modélisation que d'apport dans la notation.
2. La diffusion de message n'existe pas (voir les règles du niveau 3 de la collaboration). Cela est mis en évidence dans les exemples de diagrammes de collaboration par un envoi simple à un objet multiple (collection) qui se charge de faire suivre les messages. Il n'y a pas de parallélisme "pur".

c) Le diagramme de séquence Le diagramme de collaboration met en évidence l'ordre des interactions et les contraintes temporelles. La structure des "objets" de l'interaction est implicite.

Revenons sur quelques règles de niveau 1.

1. La règle 1 du niveau 1 de la partie collaboration peut être reprise et affinée au niveau du diagramme de séquence. Un diagramme de séquence est une collaboration sans représentation des relations structurelles : toutes les relations sont implicites. Aucune règle du métamodèle ne fixe cela.
2. Une collaboration est la représentation d'un classificateur ou d'une opération. Le métamodèle ne précise pas pour le diagramme de séquence quelles sont les représentations utilisables.
3. Quelques éléments de la notation UML présentés dans le guide de la notation ([Con99] *UML Notation Guide*) ne figurent pas dans le métamodèle (*UML Semantics*) : activité au sein d'un objet, flots de contrôle multiple, gardes, envoi multiplexé à des collections, envoi itératif, envoi avec durée de transmission non négligeable (flèche inclinée). Soit nous ne les avons pas vus, soit il y a là une lacune.
4. L'ordre des flèches entre deux objets implique un ordonnancement des messages, mais il n'est pas précisé quand cela relève de la règle de précédence (*predecessor* d'un message), la règle de causalité, le parallélisme possible ou obligatoire, le transfert de contrôle.
5. Les envois ou réceptions synchrones sont liés à la notion de prédécesseur d'un message mais nous n'avons vu le lien explicite avec le métamodèle ni la même notation que dans le diagramme de collaboration.

Revenons sur quelques règles de niveau 2.

1. Les vérifications de type liées aux éléments non déterminés (règle 3 du niveau 1) sont vérifiables ici.
2. La vérification des annotations et des contraintes suit les règles générales de la notation. Le diagramme de séquence ajoute des contraintes temporelles, des notations de contraintes (variables de nommage de messages) et des primitives spécifiques aux envois de message (*receiveTime*, *sendTime*, p. 3-99).
3. La vérification des annotations et des contraintes suit les règles générales de la notation.
4. Lors d'un envoi synchronisé (association *predecessor* du métamodèle), il faut vérifier que les messages de la liste des prédécesseurs existent et sont conformes.
5. La vérification des messages à retour de valeur (règle 5 du niveau 2 du diagramme de collaboration) s'applique aux séquences.

Discutons de quelques exemples de règles de niveau 3.

1. La diffusion de message n'existe pas (voir les règles du niveau 3 de la collaboration). Lorsque deux messages partent d'un même instant dans un objet, cela symbolise une synchronisation et non un envoi en parallèle à différents objets. Par ailleurs, dans les exemples de diagrammes de séquence, la diffusion se fait par un envoi de message simple à un objet multiple (collection) qui se charge de faire suivre les messages.
2. Dans [AV01b], nous utilisons les commentaires pour représenter des structures de contrôle (alternatives, répétitives). Cela permet de grouper des diagrammes similaires en une seule représentation. On peut imaginer inclure de telles facilités pour enrichir la notation. Il faut alors fixer les règles de vérification d'une telle algorithmique.

Diagramme états-transitions

Nous étudions dans cette section la notation relative aux diagrammes états-transitions (`Statechart Diagram`). Bien que les diagrammes d'activités partagent une sémantique commune, nous avons choisi de les traiter à part dans la section 4.2, car nous les utilisons dans des contextes clairement différents (cf discussion de la section 4.3), ce qui n'était pas le cas pour les diagrammes de collaboration et de séquence.

Comme dans le cas des collaborations, le diagramme états-transitions dispose dans le métamodèle d'une unité de structuration, l'élément de modélisation -le concept- *StateMachine*. Ainsi, un diagramme états-transitions est une "instance" de ce concept, *i.e.* une machine à états. Par souci de simplification, nous utilisons le terme "machine" pour désigner un diagramme états-transitions, sa machine à états.

Structurellement, dans le métamodèle, une machine contient uniquement des états et des transitions ([Con99] *UML Semantics*, p. 2-130). Elle fait référence à des (états)sous-machines, qui ne dépendent que d'elle. Une machine à états décrit le comportement d'au plus un élément de modélisation ([Con99] *UML Semantics*, p. 2-130). Détaillons ces différents points :

- Un état fait partie d'une seule machine. Il comprend éventuellement une action d'entrée, une action de sortie, une activité, des transitions internes (relation de composition) et des événements retardés (*deferredEvents*). La classe *State* est abstraite. Un état est soit un état final [⊙], soit un état simple, soit un état composite. L'état composite inclut des sous-états et permet la décomposition hiérarchique d'états (*nested states*), les sous-états pouvant être eux-même composites. Ce peut aussi être un (état)sous-machine. Un état composite est composé de quatre types de sous-états (ou sommets, *StateVertex*). Nous les détaillons en précisant leur notation entre [] :
  - un état "normal", au sens ou nous l'avons vu,
  - un pseudo-état : initial [•], historique superficiel [H] ou profond [H*], point de synchronisation ou d'éclatement [||], point de jonction, point de choix [○],
  - un état de synchronisation [⊛],
  - un état souche (*stub*), c'est-à-dire un état dont l'intérieur est masqué et détaillé ailleurs dans un autre état.
- L'action est le même concept que dans les diagrammes d'interaction (section 4.2, page 258).
- L'élément de modélisation (agrégation *context* p. 2-130) est un élément dont la machine décrit le comportement, plus précisément cet élément est un objet ou une interaction ([Con99] *UML Notation Guide*, p. 3-131).
- Une transition relie deux sommets -au sens précisé ci-avant- (*source*, *target*). Elle fait partie d'une machine ou d'un état. Elle comprend optionnellement un événement, une action et une garde.
- Un événement est la spécification du type d'une occurrence observable (p. 2-133). Il peut avoir des paramètres. Un événement est l'occurrence d'un signal, d'un appel d'opération, d'événement temporel (chien de garde) ou d'un changement (démon).

Nous ne discutons pas des différentes représentations graphiques de chaque concept, consulter pour cela la partie 9 du guide de la notation (p. 3-131 et suivantes).

Discutons de quelques exemples de règles de niveau 1. Les concepteurs ont été plus attentifs à la description de ce diagramme, c'est pourquoi la plupart des règles suivantes sont issues des contraintes du métamodèle.

1. Comme pour les autres diagrammes, la première règle consiste à vérifier que les éléments de modélisation d'une machine sont pertinents. On s'appuie sur l'élément *StateMachine* du métamodèle (p. 2-130) pour déterminer les éléments autorisés. L'ensemble apparaît cadenassé et cohérent hormis le contexte de la machine. Une machine est formée d'un seul état, appelons-le l'état global, qui doit être composite. Elle est agrégée dans un classificateur ou une propriété comportementale (opération ou méthode) (p. 2-141). Il nous semble important de limiter l'usage des machines aux classes.
2. Chaque élément a un nom unique dans son espace de nommage. On doit vérifier qu'à un niveau de composition d'état donné, tous les noms d'états sont différents. Quel est le nom d'une transition, d'un état final, d'un état souche, d'un pseudo-état, d'un état de synchronisation ?
3. Autant l'utilisation des transitions est simple -elles relient deux états-, autant celle des

états est complexe. Nous discutons de quelques règles de combinaisons, dont certaines sont explicitées dans les contraintes du métamodèle (p. 2-139 à 2-143).

(a) L'état global n'est pas un sous-état et n'a pas de transitions sortantes (p. 2-141).

(b) Un état simple n'a pas de sous-états (p. 2-135).

(c) Un état composite contient des sous-états dont il fixe les règles et des transitions (p. 2-139).
- Un état composite a au plus un sous-état initial, un sous-état historique superficiel, un sous-état historique profond.
- Les sous-états sont inclus dans un seul état composite.
- Si un état composite est concurrent alors il est composé uniquement d'états composites (au moins 2). Ces états forment des régions (p. 2-133) dénotant des flots d'exécution parallèles. Ils ont une notation particulière : un trait pointillé. Noter que les transitions entre états composites (les régions) d'un état composite concurrent se font via des sous-états de synchronisation ([Con99] *UML Notation Guide*, p. 3-149). Mais ceci n'est pas matérialisé dans le métamodèle.

(d) Un (sous-)état de synchronisation est utilisé avec les pseudo-états, point de jonction ou d'éclatement de transitions, pour synchroniser des transitions entre régions d'un état composite (p. 2-138, 2-141, 3-149). Il n'est pas un état habituel (actif/inactif) mais contient une limite (*bound*) positive (un nombre) ou infinie (noté [⊛]) qui fixe la différence maximale entre les mises à feu de transitions entrantes et sortantes. Toutes les transitions entrantes viennent d'une même région et toutes les transitions sortantes vont dans une même région. Mais ceci n'est pas formalisé dans le métamodèle.

(e) Les pseudo-états font partie des états composites.
- Un sous-état initial n'a pas de transitions entrantes et une seule transition sortante (p. 2-140). On ne peut donc imaginer un aiguillage par des gardes (voir discussion règle 2 du niveau 3).
- Un sous-état historique a au plus une transition sortante (p. 2-140).
- Un point de synchronisation a au moins deux transitions entrantes et exactement une transition sortante (p. 2-140).
- Un point d'éclatement a exactement une transition entrante et au moins deux transitions sortantes (p. 2-140).
- Un point de jonction a au moins une transition entrante et une transition sortante (p. 2-140).
- Un point de choix a au moins une transition entrante et une transition sortante (p. 2-140).

(f) Un état final n'a pas de transitions sortantes (p. 2-133, 2-140), mais il peut avoir des transitions internes et des actions.

4. Les éléments d'une transition sont tous facultatifs hormis les deux états. Les contraintes sur les transitions sont fonction des états qui les composent (p. 2-141 et 2-142) :

(a) Une transition associée à un point de synchronisation ou d'éclatement est appelée transition concurrente, elle n'a ni garde ni événement déclencheur.

(b) Une transition sortant d'un point de synchronisation a pour cible un état et les transitions sortant d'un point de synchronisation ont pour origine des états de régions différentes.

(c) Une transition entrant dans un point d'éclatement a pour origine un état et les transitions sortant d'un point d'éclatement ont pour cible des états de région différentes.

(d) Les transitions sortant d'un pseudo-état n'ont pas d'événement déclencheur.

(e) Si une transition sortant d'un état initial a un événement déclencheur, c'est soit un événement de création soit, dans le cas d'une machine associée à une opération, une invocation d'opération. Dans les autres cas, il n'y a pas d'événement déclencheur.

5. Les sous-états d'un état sous-machine sont des états souches. Les états sous-machine ne sont jamais concurrents (p. 2-141).

Les vérifications de niveau 2 sont liées aux gardes.

1. L'expression OCL associée à une garde doit être typée correctement.
2. L'expression OCL associée à une garde ne doit pas être toujours fausse.
3. Une action de retour doit être précédée d'une invocation d'opération.

Discutons de quelques exemples de règles de niveau 3.

1. Une transition est un composant d'une machine ou interne à un état. Il semble utile d'ajouter une contrainte de totalité et d'exclusion (`xor`) dans le méta-modèle.
2. Lorsqu'on veut présenter hiérarchiquement et modulairement une machine, on cache l'intérieur d'un état composite. Sa représentation externe est une souche d'état (*StubState*). Sa représentation interne est l'intérieur d'un état composite. L'interface entre les deux nous semble manquer : dans les exemples, l'état composite est soit une souche soit une boîte blanche dans laquelle les transitions externes "rentrent" dans les sous-états. Nous avons proposé une solution dans un ouvrage précédent ([AV01b], p. 233) mais elle ne respecte pas les contraintes posées sur les sous-états initiaux et les états-finaux.
3. Une transition sans événement déclencheur issue d'un état simple est déclenchée par la fin d'activité de l'état source. S'il n'y a pas d'activité, alors on doit se poser la question de la pertinence de l'état (son passage est instantané).
4. La sémantique des événements retardés est relativement ambiguë. Si un événement, qui n'est pas déclencheur d'un état (pas de transitions associées), intervient alors il est perdu, sauf s'il est référencé dans la liste des événements retardés de l'état en question. Il sera éventuellement déclencheable dans l'état suivant. Nous n'avons pas vu d'illustration dans le guide de la notation.
5. La sémantique des états composites et des transitions associées est relativement complexe. Elle est indiquée informellement ([Con99] *UML Semantics*, p. 2-145 et 2-146). Nous avons relevé les points suivants, qui nous semblent importants :
   (a) Invariant d'activité (récursif).
   - Un état séquentiel est actif si un et un seul de ses sous-états est actif.
   - Un état concurrent est actif si tous ses sous-états (ses régions) sont actifs.

   (b) Arrivée dans un état.
   - Dans un état séquentiel, on entre soit dans l'état par défaut (initial), soit explicitement dans un sous-état (il n'est pas précisé de quel type, mais on imagine que ce ne peut être qu'un état simple ou un état composite - à fixer), soit dans un état historique (le sous-état le plus récemment actif ou l'état par défaut s'il n'y en a pas). Le raisonnement est appliqué récursivement si le sous-état est composite. Lorsqu'on entre dans un état on exécute l'action d'entrée (*entry :*).
   - Dans un état concurrent, on entre soit dans l'état par défaut, soit explicitement dans un sous-état. L'état par défaut est l'union des états par défaut de chaque région. Si on rentre explicitement dans des sous-états des régions, on utilise un point de synchronisation ; l'état entrant des régions non concernées par la synchronisation est leur état par défaut.
     On en déduit que l'historique est interdit dans un état concurrent (!). On suppose que les actions d'entrée sont exécutées dans l'ordre et qu'un état concurrent peut avoir une action d'entrée.

(c) Sortie d'un état.
- Lorsqu'on sort d'un état séquentiel, on exécute l'action de sortie, et ce récursivement et dans l'ordre, si plusieurs niveaux sont à traverser.
- Lorsqu'on sort d'un état concurrent, on sort de chaque région puis on exécute l'action de sortie de chaque région. On suppose qu'il y a aussi une action de sortie pour l'état concurrent.

(d) Evénement interne. La sémantique des événements internes aux états composites n'est pas clairement définie : est-ce un événement (externe) d'un sous-état, est-ce un événement interne global "hérité" dans tous les sous-états ?

(e) Evénement différé. Il est différé dans chacun des sous-états. Quel sens cela a-t-il pour des états finaux, des pseudo-états ou des états de synchronisation ?

(f) Activité d'un état. Que représente l'activité d'un état composite ?

6. Un état sous-machine est l'invocation d'une machine définie ailleurs mais dans le même contexte. A première vue, cette spécialisation de l'état composite ressemble à un état-souche. Peut-être est-ce utilisé lorsqu'un motif d'état apparaît plusieurs fois dans une machine. Cela nous évoque le problème de la duplication d'état ; un état quelconque peut-il être "référencé" plusieurs fois ? La machine à états n'a pas de représentation visuelle spécifique, elle ne peut donc pas être vue comme un élément de modélisation dans un diagramme quelconque et il n'y a pas de relations explicites entre machines à états.
7. Même si le métamodèle permet l'usage des stéréotypes dans une collaboration, en pratique les machines ne définissent pas de stéréotypes propres. Les stéréotypes sont ceux des diagrammes de classe. Les variations d'événements sont définis par spécialisation de la classe `Event` et non par stéréotype. Ainsi nombre de vérifications du niveau 1 sont fixées dans le métamodèle en fonction du type d'état ou d'événement.
8. Dans OMT, il existe une notation spécifique pour les envois de messages ou d'événements vers d'autres objets. Le destinataire d'un message est-il la cible contenue dans l'action ? Dans quels cas peut-on symboliser visuellement une action ?

Diagramme d'activités

Comme dans le cas des machines à états, le diagramme états-transitions dispose dans le métamodèle d'une unité de structuration, l'élément de modélisation -le concept- *ActivityGraph*. Ainsi, un diagramme d'activités est une "instance" de ce concept. Par extension, tout diagramme d'activité peut se traduire par un diagramme états-transitions.

Structurellement, dans le métamodèle, un graphe d'activité est une machine à états ([Con99] *UML Semantics*, p. 2-160). Plusieurs particularités sont à souligner :

- Dans le patron des machines à états, on redéfinit, spécifiquement pour le diagramme d'activités, les éléments suivants :
  - Un état action (`ActionState`), qu'on appelle abusivement une **activité**, est un état simple. Un état d'appel est une activité dont l'action d'entrée est un appel d'opération.
  - Un état de sous-activité est un état de sous-machine. Pour nous, il joue le rôle d'activité souche, c'est-à-dire une activité dont le contenu n'est pas visible au niveau où elle apparaît.
- La machine à états a été largement simplifiée. Pour prendre en compte une répartition des activités sur plusieurs unités, on utilise des partitions. Une partition est un élément de modélisation représentant une unité de réalisation des activités. Ce n'est pas un objet mais plutôt une unité organisationnelle (telle qu'on peut la trouver dans une modélisation métier). Graphiquement, les partitions sont représentées par des couloirs (*swimlanes*) et les activités sont réparties sur ces couloirs. Une partition figure dans un seul graphe d'activité.

- Les actions opèrent par des objets et sur des objets. Ces objets sont de trois types :
  - Les objets responsables des actions. En principe c'est l'objet auquel est rattaché le graphe d'activité sauf si des partitions sont introduites. Dans ce cas, les couloirs peuvent désigner des objets ou des groupes d'objet. Nous le voyons, la notation devient vague. Si les couloirs sont des objets, alors autant utiliser un diagramme de séquence (et éviter une redondance). Si les couloirs sont des objets, alors la vérification de cohérence avec les autres diagrammes devient difficile. Nous préconisons cette utilisation uniquement pour l'analyse des besoins, sans y associer de vérifications strictes. Nous conseillons donc, au niveau logique, d'utiliser un diagramme d'activité uniquement pour expliciter une opération.
  - Les objets en entrée ou en sortie d'actions. Un état flot d'objet (*ObjectFlowState*) est un état simple qui a des paramètres et un classificateur pour type. Il définit un flot d'objet sur une transition (notée avec un trait discontinu), par exemple ([Con99] *UML Notation Guide*, p. 3-158).
  - Un classificateur-état (*Classifier-in-state*) caractérise les instances du classificateur qui sont dans des états donnés, cela ressemble à la notation de *class as state* d'OMT. Dans un diagramme d'activité on utilise des objets état pour symboliser l'évolution d'un objet dans un graphe d'activité. Un objet état est un objet symbolisé avec la valeur de son état (`[state]`). Un tel objet peut apparaître sur un flot d'objet. Cette notation est aussi utilisable dans les diagrammes d'interaction (p. 3-157) ou les diagrammes de classes pour indiquer par exemple que certaines caractéristiques (association, propriétés) ne sont disponibles que dans certains états.

Les règles de vérification sont celles des machines à états. Certaines règles sont restreintes, d'autre modifiées, d'autres règles sont ajoutées.

Prenons quelques exemples de règles de niveau 1.

1. Comme pour les autres diagrammes, la première règle consiste à vérifier que les éléments de modélisation d'une machine sont pertinents. On s'appuie sur l'élément *ActivityGraph* du métamodèle (p. 2-160) pour déterminer les éléments autorisés. Les règles suivantes sont relatives au contenu :
   (a) Les types d'états autorisés sont : activité (état-action), sous-activité, état final, pseudo-états (initial, point de synchronisation ou d'éclatement, point de jonction, point de choix -ces deux derniers sont notés [⋄]-). Les autres devraient être interdits, mais ils ne le sont pas dans le méta-modèle.
   (b) Les partitions qui sont spécifiques aux graphes d'activités (relation de composition et cardinalité 1, p. 2-160). Une partition n'est pas une région (état composite concurrent). Notons qu'une région est décrite par des machines. Une règle devrait imposer des restrictions sur ces machines.
   (c) Par extension d'une machine, un graphe d'activité est formé d'un seul état, qui doit être composite. Compte-tenu de la règle précédente, on considère qu'il s'agit d'une sous-activité, mais cela reste à formaliser car cela s'intègre mal avec les régions.
   (d) Toutes les transitions sur les états autorisés sont a priori autorisées.
   (e) Le contexte d'un graphe d'activité est un paquetage, un classificateur ou une propriété comportementale (opération) (p. 2-164). Nous pensons que la seule utilisation rigoureuse concerne le dernier cas.
2. Une activité (**ActionState**) représente l'exécution d'une action atomique, définie par son action d'entrée. La fin de l'action entraîne automatiquement la sortie de l'état. L'activité est donc un état sans activité interne, sans événement interne et sans action de sortie (p. 2-161). Les règles suivantes sont relatives à l'activité :
   (a) L'action d'entrée est obligatoire (p. 2-164). Nous avons détecté une ambiguïté à ce sujet dans le manuel de référence ([Con99] *UML Semantics*). La règle précise

que la collection des actions d'entrée est positive strictement (p. 2-164). Le texte indique qu'une activité peut exécuter plusieurs actions (p. 2-161) mais dans un état, et selon ce que nous avons vu dans la section états-transitions, il y a au plus une action d'entrée (p. 2-130). L'exécution dynamique et concurrente d'une activité nous semble sujette à caution dans ce cas (p. 2-161).

(b) L'activité n'a pas d'activités ou d'événements internes, ni d'actions de sortie (p. 2-164).

(c) Les transitions issues d'une activité n'ont pas d'événement déclencheur (p. 2-164).

(d) Un état d'appel (**CallState**) est une activité avec une seule action d'appel comme entrée.

3. Chaque élément a un nom unique dans son espace de nommage. On doit vérifier qu'à un niveau de composition d'état donné tous les noms d'états ou d'activités sont différents. Les mêmes questions se posent que dans le contexte d'une machine et d'un graphe d'activité (règle 2, niveau 1, machine).

4. Revenons sur quelques règles de combinaisons d'états de la règle 3 du niveau 1 de la machine.

   (a) L'état global d'une machine n'est pas un sous-état et n'a pas de transitions sortantes ; qu'en est-il pour un graphe d'activité ? voir la règle 1-c.

   (b) Un état simple n'a pas de sous-états ; par extension une activité n'a pas de sous-activités.

   (c) Le seul état composite autorisé est la sous-activité. Les règles relatives à la concurrence états composites sont donc inutiles mais les autres règles 3 du niveau 1 de la machine sont vérifiables pour les formes d'états autorisées.

5. Les règles de transitions sont celles de la règle 4 du niveau 1 de la machine avec les particularités suivantes :
   - Les transitions issues d'une activité n'ont pas d'événement déclencheur (règle 2-b ci-avant).
   - Les transitions associées aux points de synchronisation ou d'éclatement ont pour source ou cible un état ou un pseudo-état quelconque (p. 2-166). Il n'y a pas de restriction sur les régions. Signalons que tous les pseudo-états ne sont pas autorisés (règle 1-a). Nous préférons la règle suivante.
   - Les états associés aux points de synchronisation ou d'éclatement sont des activités.

6. La machine d'une sous-activité est un graphe d'activité (p. 2-166).

7. Les états d'une sous-activité sont des activités.

8. Un état flot d'objet a un type et une direction compatibles avec son classificateur ou son classificateur-état (p. 2-165). Il est compatible avec le profil de l'action associée (argument ou résultat) (p. 2-165).

Les vérifications de niveau 2 sont celles des machines.

Discutons de quelques exemples de règles de niveau 3, qui s'ajoutent à celles de la machine à états.

1. Comment s'articulent activité d'un état et graphe d'activité ? Contrairement à OMT, l'activité d'un état est une action et à toute action on peut associer une opération, décrite par un graphe d'activité et donc interruptible. En résumé, et contrairement à OMT, les transitions et les états sont "interruptibles" par des événements.

2. Les éléments de décision (*decision/merge*) sont définis dans le guide de la notation ([Con99] *UML Notation Guide*, p. 3-154 et 3-155) mais pas dans la référence ([Con99] *UML Semantics*). Nous les assimilons aux pseudo-états point de synchronisation et point d'éclatement avec une notation différente. C'est une interprétation discutable.

3. De même, émissions et réceptions explicites d'événements (*signal sending and receipt*) figurent dans le guide de la notation ([Con99] *UML Notation Guide*, p. 3-159 et 3-161) mais pas dans la référence ([Con99] *UML Semantics*). La réception est une notation alternative à l'événement déclencheur d'une transition. L'émission est une notation alternative à l'envoi de message dans une action.
4. L'invocation dynamique des actions d'une activité nous semble contradictoire avec la définition des graphes d'activités. La présence de flots parallèles peut être explicitée avec les points de synchronisation et d'éclatement.
5. La relation entre partition et graphe d'activités nous semble à préciser. Visuellement on comprend que les activités sont rangées par couloir mais le méta-modèle ne reflète pas directement et clairement cette lecture. Qu'en-est-il pour les sous-graphes d'activité ?
6. Le lien entre événements retardés et réception d'événements est illustré dans le guide de la notation (p. 3-161).
7. La sémantique des sous-activités se déduit de celle des états composites séquentiels. Le lien entre sous-activité et les autres activités est-il du même type que celui entre état-composite et état ?
8. La sémantique et l'usage des états flot d'objet doivent être précisés : pourquoi ces objets et pas d'autres arguments des envois de messages ? Quels effets particuliers ont-ils sur les activités ?

4.3 Niveau modèle

Après avoir examiné individuellement les diagrammes du niveau logique, nous nous attelons à vérifier la cohérence globale du modèle logique. Le modèle est un ensemble hétérogène de diagrammes. Nous avons étudié les principes d'une telle vérification dans le chapitre 10 de [AV01a] et illustré ces principes sur des exemples concrets dans le chapitre 10 de [AV02]. Le but était de croiser "manuellement" les concepts et de vérifier leur cohérence. Dans le modèle logique, l'unité de cohésion est la notion d'objet. Elle s'appuie sur la triade instance/type/communication. Dans cette section, nous présentons d'abord la démarche de vérification, puis nous l'appliquons à la vérification du modèle logique.

Démarche

Dans certaines méthodes, le modèle logique est la composition de plusieurs points de vue disjoints du système. Par exemple, dans OMT [RBP+96], le niveau logique est la donnée de trois modèles : le modèle objet, le modèle dynamique et le modèle fonctionnel. L'intégration des points de vue (plus généralement l'intégration de méthode) est un problème difficile, surtout lorsqu'on veut formaliser chaque vue[4]. La difficulté est de trouver "la colle" entre les différents morceaux, chaque morceau correspondant à une approche ou une "théorie", en générale cohérente, qui est très différente des autres. Les auteurs de la méthode OMT, par exemple, n'ont pas résolu ce problème.

La base de la vérification de cohérence entre les diagrammes est la traçabilité de l'envoi de message : un objet reçoit un événement (séquence, états-transitions) qui déclenche éventuellement des actions ou une activité (états-transitions, opérations) ; ces opérations peuvent engendrer de nouveaux événements (états-transitions, séquence). Cette apparente simplicité masque la complexité induite par diverses relations (agrégation, héritage notamment).

Nous choisissons de ne pas attaquer de front le problème de l'intégration en considérant que tous les diagrammes sont de même niveau et à rapprocher. Dans UML, selon la lecture

[4] Voir la conférence IFM ou le projet SHES
http ://www.sciences.univ-nantes.fr/info/recherche/mgl/FRANCAIS/ThemesetProjets/SHES/index.html

que nous faisons d'une utilisation rigoureuse de la méthode, les diagrammes ne sont pas de même niveau, on peut dire qu'ils sont hiérarchisés. Un peu comme pour les langages de programmation, nous séparons types et valeurs. Nous distinguons clairement deux groupes de diagrammes : les diagrammes de types et les diagrammes d'instances.

Diagrammes de types Les diagrammes de types définissent le type de tous les concepts d'un modèle à objet : classes, opérations, envois de message, changements d'états... Les diagrammes sont hiérarchiquement structurés de la façon suivante. Au niveau supérieur, on trouve le diagramme de classe. Ensuite, dans chaque classe on trouve des opérations qui peuvent être explicitées par des diagrammes d'activité et un diagramme états-transitions qui définit les enchaînements d'opérations autorisés, et les différents états que peut prendre une instance de la classe. Chaque niveau définit un contexte, c'est sensiblement la même notion de contexte qu'on utilise pour exprimer les contraintes avec OCL.

Diagrammes d'instances Les diagrammes d'instances portent sur le comportement des instances (collaborations et séquences). Ils constituent la colle entre les différents aspects des diagrammes de types.

Dans cette lecture, nous avons restreint l'usage des diagrammes états-transitions et des diagrammes d'activités. En principe, selon le métamodèle (p. 2-160 de [Con99]), un diagramme d'activités est une spécialisation d'une machine à états (ce qui est un abus de (méta)langage car il n'y a pas d'états dans un diagramme d'activités) et une machine à états est le comportement d'un élément de modélisation (qui définit son contexte). De ce fait, on peut associer un diagramme états/transitions à quasiment n'importe quel concept de modélisation (attribut, association, classe, opération événement, paquetage...). Il s'agit à nouveau d'un abus de la relation de généralisation/spécialisation dans le (méta)langage. L'usage commun (règles informelles issues de la pratique) veut qu'on associe les diagrammes états-transitions (les machines à états) aux classes et les diagrammes d'activités aux opérations appelées activités se trouvant à l'intérieur des états ou globalement au système (fonctions générales avec ou pas des couloirs pour les objets concernés). Nous restreignons l'usage des machines à états aux classes et celui des activités aux opérations. De ce fait, la distinction entre action (opération atomique) et activité (interruptible) du diagramme états-transitions est mise en adéquation avec le diagramme d'activité associé à l'opération : ce n'est plus la machine à états qui fixe l'atomicité d'une opération mais l'opération elle-même. Nous avons ajouté un niveau de contrôle supplémentaire : une opération non atomique d'une classe ne peut être associée à une transition de la machine à états de cette classe.

Cette structuration permet d'organiser le travail de vérification du modèle en trois parties (plus petites) :

- vérification croisée des diagrammes d'instance,
- vérification croisée des diagrammes de types,
- vérification croisée des diagrammes de types et d'instance.

Les propriétés à vérifier sont la correction, la cohérence et la complétude. Plus les modèles sont proches sémantiquement, plus la vérification est simple car la comparaison s'établit sur des concepts similaires. Dans le cas contraire, les concepts sont orthogonaux et, plus prosaïquement, les éléments de modélisation se trouvent dans des paquetages (et des pages) différents du métamodèle.

Avant de passer à la vérification croisée, il nous faut éclaircir un point fondamental à l'interface des différentes vues du modèle logique : la sémantique du comportement (ou de la communication). Le métamodèle nous semble ambigu à ce sujet, ce qui pose évidemment des problème dans la vérification. Une autre raison est que cela nous permet d'y ranger les règles

de vérification qui concernent les classes et les actions (elles sont communes à la vérification type-type et type-instance).

Comportement commun

Le comportement commun est un support de la liaison entre modèles. Il est défini par le paquetage *Common Behavior* dans le manuel de référence ([Con99] *UML Semantics*, p. 2-83 à 2-102). Nous avons vu dans les sections consacrées aux diagrammes individuels les concepts de stimulus, d'événement, d'action, d'opération, et l'envoi ou l'invocation de ces concepts. Essayons d'éclaircir la sémantique du comportement d'un système.

Collaboration Dans une collaboration, un message est échangé entre deux objets (ou leur rôle), l'émetteur et le receveur. Le stimulus désigne la communication au niveau des instances (en quelque sorte une instance de message). Chaque message contient une action ; *"The Action which causes a Stimulus to be sent according to the Message."* ([Con99] *UML Semantics*, p. 2-107). Une action est donc associée à des stimuli.

Action La définition de l'action est fondamentale car elle se retrouve dans les collaborations et les machines. *"An Action is a specification of an executable statement that forms an abstraction of a computational procedure that results in a change in the state of the model, and can be realized by sending a message to an object or modifying a link or a value of an attribute."* ([Con99] *UML Semantics*, p. 2-86). Une action est définie par un objet cible, un indicateur de synchronisation (sync/async) et un corps (script et itération) et éventuellement des arguments et une séquence d'actions.

- Le corps est une chaîne de caractères. Aucune vérification ne peut y être associée. Si la description de l'action s'arrête là, on ne peut étudier l'effet sur d'autres objets.
- La cible est une expression dont le résultat désigne un objet ou un ensemble d'objets. Y-a-t'il confusion entre la cible et l'émetteur ou le receveur dans l'action associée à un envoi de message ? Non. Nous interprétons l'envoi de message comme suit : l'émetteur envoie un message, le receveur est activé et exécute une action qui génère des modifications ou d'autres messages vers d'autres cibles.
- Plusieurs formes d'action caractérisent le comportement (p. 2-85) :
  - instanciation *CreateAction*, associée à une classe,
  - appel d'opération *CallAction*, associé à une opération,
  - retour de résultat *ReturnAction*,
  - envoi de signal *SendAction*, associé à un signal ou une exception,
  - terminaison *TerminateAction*,
  - destruction *DestroyAction*,
  - action "abstraite" *UninterpretedAction*, concrétisée dans le langage de programmation.

Machine à états Dans une machine, une transition, qu'elle soit interne ou externe à un état, est déclenchée par un événement. Une transition a éventuellement pour effet une action. Des actions sont aussi associées à des états (entrée, sortie ou activité). Revenons sur la notion d'événement : *"An event is a specification of a type of observable occurrence. The occurrence that generates an event instance is assumed to take place at an instant in time with no duration. "* ([Con99] *UML Semantics*, p. 2-133). Plusieurs formes d'événements peuvent déclencher une transition (p. 2-131) :

- appel d'opération *CallEvent*, associé à une opération,
- envoi de signal *SignalEvent*, associé à un signal ou une exception,
- événement temporel *TimeEvent*, défini par une condition temporelle,
- événement de changement *ChangeEvent*, défini par une condition booléenne.

Cohérence La cohérence entre collaborations et machines à états (ou activités) repose sur la sémantique du triplet action-message-événement. Nous comparons ces concepts deux à deux selon la sémantique du métamodèle :

1. *Message* → *Action* : un envoi de message déclenche une action.
2. *Message* ← *Action* : une action peut générer des envois de message.
3. *Evénement* → *Action* : un événement, via une transition, peut déclencher une action.
4. *Evénement* ← *Action* : pas de lien explicite.
5. *Message* → *Evénement* : pas de lien explicite.
6. *Message* ← *Evénement* : pas de lien explicite.

Il semble y avoir un "trou" dans la sémantique du comportement. Il faut pouvoir répondre aux deux questions suivantes :

1. Y-a-t'il des événements sans messages ?
2. Y-a-t'il des messages sans événements ?

Si nous répondons non, alors la cohérence est simple : un envoi de message et un événement sont un même concept. Sinon, il faut définir la corrélation entre les deux. Avant de répondre, quelques remarques :

- Rappelons que la notion d'événement vient des Statecharts et plus généralement du domaine temps réel (des événements arrivent dans le système et sont interceptés par un (ou plusieurs) gestionnaires d'événements, l'émetteur ou le receveur peuvent être inconnus) tandis que l'envoi de message vient de la programmation à objet (l'émetteur et le receveur sont bien déterminés).
- Le métamodèle indique que la création et la destruction d'instance, l'appel d'opération et l'envoi de signal sont des envois de messages (p. 2-107). La cible d'une action de création d'instance est une classe et non une instance. La destruction d'instance peut être vue soit comme un appel d'opération de destruction, soit comme une primitive vers le système de gestion d'objets.
- Si on compare la hiérarchie de spécialisation des actions (p. 2-85) et celle des événements (p. 2-131), on s'aperçoit que, parmi les différentes formes d'actions, seuls deux types d'action (*CallAction* et *SendAction*) correspondent à des événements (*CallEvent* et *SignalEvent*).
- Un événement est le résultat d'une action dans le système ou dans son environnement. Un événement est transmis à une ou plusieurs cibles (p. 2-144). Un événement est reçu quand il est placé dans la file des événements d'un objet ; il est consommé quand l'objet le prend en compte.

En conclusion, si on restreint l'envoi de message à l'appel d'opération et l'envoi de signal, on peut considérer **arbitrairement** qu'un envoi de message est un événement et que d'autres événements peuvent exister (signaux externes, réveils d'horloges, etc.). On en déduit les corrélations suivantes :

1. *Evénement* ← *Action* : une action peut générer des envois de messages et donc des événements.
2. *Message* → *Evénement* : un envoi de message génère un événement chez le receveur.
3. *Message* ← *Evénement* : certains événements sont des envois de message.

De ce fait, nous considérons que, parmi les différentes formes d'actions, seuls l'appel d'opération *CallAction* et l'envoi de signal *SendAction* correspondent à des envois de messages. Ces deux formes d'action correspondent aux deux formes de communication mises en évidence pour les événements (*CallEvent* et *SignalEvent*). Le métamodèle indique que la création et la destruction d'instance sont des envois de messages.

Compte-tenu de ces hypothèses, nous pouvons maintenant détailler la vérification croisée des diagrammes. Nous commençons par la vérification actions-classes dans le paragraphe suivant, puis nous poursuivons par les vérifications entre diagrammes.

Vérification croisée des actions et des classes

1. Dans un envoi de message, le receveur du message est la cible de l'action (`msg.receiver = msg.action.target`). C'est une règle de cohérence fondamentale que nous n'avons pas trouvé dans le métamodèle.
2. Si le type de la cible est calculable statiquement, alors ce type est une classe déclarée. Si la classe a des paramètres, ceux de la classe de la cible doivent être compatibles.
3. Si l'action modifie des attributs alors ceux-ci doivent être définis dans la classe de la cible. Pour cela, il faut que le script soit une expression analysable, une expression OCL par exemple.
4. Si une action est asynchrone, alors la classe de l'objet à l'origine de l'action doit être active.
5. Si l'action invoque une opération alors la classe de la cible (règle 1) doit déclarer ou hériter cette opération. Les paramètres des actions doivent être compatibles (au sens du typage) avec la signature (le profil) des opérations de la classe. Le métamodèle n'indique qu'une vérification du nombre de paramètres (p. 2-95).
6. Si l'action crée une instance, alors la classe associée doit exister et les paramètres de l'instanciation doivent être compatibles avec la méthode d'instanciation de cette classe. La cible n'a pas de sens (p. 2-96). Remarquons que si le langage autorise les métaclasses, la création d'instance est un appel d'opération ordinaire.
7. Si l'action rend un résultat, alors elle doit être corrélée à un appel d'opération et être conforme au profil de cette opération. Le manuel indique que la cible n'est pas précisée (p. 2-93). Cela signifie que les continuations sont interdites et qu'il faut obligatoirement corréler le retour à l'appel (c'est pourquoi nous pouvons considérer que ce n'est pas un envoi de message). Le retour de résultat est-il asynchrone quel que soit le mode d'invocation de son opération ? La notation laisse libre choix.
8. Si l'action envoie un signal, ce dernier devra être capté par la cible. Il est défini par une classe ; les paramètres de l'envoi sont compatibles avec la spécification du signal. Le métamodèle indique une seule vérification, celle du nombre de paramètres (p. 2-99). L'envoi de signal est asynchrone (p. 2-93 et 2-99).
9. Si l'action supprime une instance, définie dans la cible, alors cette dernière devait exister !
10. Une action de terminaison n'a ni argument ni paramètre (p. 2-99).
11. Un stimulus (ce qui est envoyé à une instance) est associé aux actions de création, d'appel d'opération, d'envoi de signal ou de destruction d'opération (p. 2-99).
12. Si une action modifie un attribut, alors elle respecte l'invariant de la classe.

Vérification croisée des diagrammes d'instance

Commençons par un constat. Hormis la présentation et les commentaires, il y a équivalence entre diagrammes de collaboration et diagrammes de séquence. Cela s'explique par le fait que la sémantique support est la même. Ainsi, l'outil *Rational Rose* possède-t-il une fonction de passage de l'un à l'autre, bien que les annotations soient perdues dans la traduction. La vérification croisée se résumerait alors à des traductions d'un formalisme vers l'autre. Cette réponse n'est pas satisfaisante pour plusieurs raisons :

- Si c'était le cas, alors on ne voit pas pourquoi nous utiliserions deux formalismes, au risque d'introduire du travail supplémentaire et des incohérences.
- Les diagrammes présentent des points de vue différents et ce qui fait leur apport n'est pas le formalisme de base de la collaboration mais les annotations (commentaires et contraintes) liées au point de vue mis en évidence.
- En pratique, on trace très rarement pour une même collaboration (abstraite) un diagramme de collaboration ET un diagramme de séquence. On utilise l'un ou l'autre des formalismes selon le point de vue qui prime (structure, temps) et la lisibilité souhaitée.
- Les collaborations sont a priori indépendantes les unes des autres. Si deux objets sont en interaction dans un diagramme de séquence, il n'est nullement obligatoire d'avoir un diagramme de collaboration qui explicite le lien entre ces deux objets. Cette indépendance invalide bon nombre de vérifications croisées entre diagrammes.

En résumé, on ne s'attache pas à vérifier l'exacte correspondance entre un diagramme de collaboration et un diagramme de séquence, ce qui prouverait la cohérence et la complétude entre les deux modèles. Le problème est plus général, on cherche à vérifier un ensemble de diagrammes d'interaction. Le principe général de vérification est de comparer deux à deux tous les diagrammes d'interaction (collaboration vs collaboration, collaboration vs séquence, séquence vs séquence). La vérification porte sur la cohérence mais pas sur la complétude. En effet, les diagrammes d'interaction sont des exemples de collaboration et ils ne représentent pas toutes les collaborations possibles. De même, deux collaborations peuvent "partager" des interactions communes si elles sont des variantes de réalisation d'un cas d'utilisation. L'objectif est donc uniquement de détecter des incohérences.

On suppose travailler avec deux diagrammes d'interaction quelconques. Pour chaque point de vérification envisagé, nous présentons la règle et nous discutons de sa pertinence. Ce qui différencie deux collaborations est plus important que ce qui les unit, c'est pourquoi le nombre de règles est limité.

La base de la cohérence est la vérification que tout objet d'un diagramme est cohérent avec les autres objets dans d'autres diagrammes. Par objet, nous entendons instance ou classificateur type (rôle).

1. Si deux objets ont le même nom d'instance, ils représentent la même instance. Selon le métamodèle, il n'y a pas d'ambiguïté mais les spécifieurs se soucient peu de cette règle en pratique. On ne peut rien affirmer de tel lorsqu'on dispose uniquement d'un nom de classe ou d'un rôle.

2. Si deux objets sont de la même classe, alors les règles suivantes sont à vérifier :
   - Si l'un est actif, l'autre l'est aussi.
   - Si l'un est multiobjet, l'autre l'est aussi.
   - Ils ont les mêmes liaisons (liens, associations type) même si elles ne sont pas toutes nommées. Lorsqu'elles sont représentées, ces liaisons doivent avoir les mêmes caractéristiques selon le niveau de description (instance/spécification) : noms, rôles d'association, agrégation, cardinalités, qualification, etc.
   - Deux messages supportant une même action ont mêmes caractéristiques (arguments, type de synchronisation, etc.) pour ces deux objets. Ont-ils même comportement ? C'est-à-dire les mêmes conséquences en termes d'envois de message ? C'est rarement le cas, cela dépend des paramètres et des conditions locales.

3. Si deux diagrammes représentent la même collaboration, alors ils doivent être équivalents :
   - Mêmes objets/classes et caractéristiques.
   - Mêmes relations explicites/implicites et caractéristiques.
   - Mêmes messages (actions, émetteurs, receveurs, etc.) et dans le même ordre. Chaque message est rattaché à une unique collaboration.

- Seules les annotations peuvent changer.

4. L'ordre des envois de message est représenté linéairement dans le diagramme de séquence et par des numéros dans le diagramme de collaboration. La notation est plus riche dans le diagramme de collaboration car elle est hiérarchique. Autrement dit, les corrélations entre envois de messages sont mises en évidence par une structure imbriquée des numéros.

On pourrait ranger dans cette catégorie la comparaison entre des scénarios de cas d'utilisation et les collaborations qui les réalisent.

Vérification croisée des diagrammes de types

La vérification croisée des diagrammes de types est fondamentalement le collage des différentes vues du système : structurelle, dynamique et fonctionnelle. Nous disposons de trois vues organisées selon la coordination indiquée dans la démarche de la section 4.3 : le diagramme de classes met en évidence la structure, les diagrammes états-transitions décrivent la dynamique individuelle et les diagrammes d'activité décrivent les opérations (fonctions). Pour que la vérification soit complète sur les trois axes, il faudrait avoir le détail des actions, à défaut d'avoir le corps des méthodes sous forme de diagramme d'activités. La vérification peut se faire en comparant les vues deux à deux, c'est ce que nous développons maintenant.

Vérification entre classes et machines

On appelle classe à comportement une classe dont le comportement dynamique est spécifié par une machine. Du point de vue de la machine, cette classe sera la classe de référence de la machine.

Nous proposons une liste de vérifications entre classes et machines. Nous indiquons, lorsque nous les avons trouvées, les références au métamodèle.

1. Une classe à comportement peut avoir plusieurs machines (p. 2-130). Si c'est le cas, comment sont-elles coordonnées ? Nous préconisons une machine au plus par classe pour éviter ce problème.
2. Une classe a un comportement défini optionnellement par une machine. Les sous-classes héritent-elles de cette machine, peuvent-elles la spécialiser, la redéfinir, la modifier ou la contraindre ? Nous n'avons rien trouvé de précis à ce sujet dans UML. Une discussion sur le raffinement de machines est proposée (p. 2-155) avec plusieurs sémantiques différentes (sous-typage, héritage strict, raffinement).
3. Les états de la machine sont définis par des noms, on peut associer chaque état à une combinaison des attributs de la classe de référence. Ce n'est pas une règle, mais cela permet un codage différent des états et une meilleur cohérence.
4. Plusieurs règles sont relatives aux transitions :
   (a) Si l'événement est un envoi de message, alors l'action est une invocation d'opération ou une réception de signal. Dans ce contexte, on ne connaît pas l'émetteur du message, sauf s'il est explicitement ajouté dans les arguments du message. Les contraintes suivantes sont à respecter :
   - Le type de la cible de l'action est la classe de la machine.
   - Lors d'une invocation d'opération, la classe de référence doit déclarer ou hériter cette opération. Les paramètres des actions doivent être compatibles (au sens du typage) avec la signature (le profil) des opérations de la classe. Le métamodèle n'indique qu'une vérification du nombre de paramètres (p. 2-95).
   - Si une action est asynchrone, alors la classe de l'objet à l'origine de l'action doit être active.

- Si, pour un appel d'opération, un résultat est attendu, conformément au profil de l'opération invoquée, on devra trouver une action de retour de résultat soit pour le même objet (appel synchrone) soit pour un objet donné (appel asynchrone).
- Les actions engendrées sont reportées au niveau de la méthode si c'est un appel d'opération ou dans la séquence des actions de l'action considérée. On sépare ainsi action "reçue" et action "envoyée". Il faudra alors vérifier la cohérence à l'intérieur de l'opération, soit par analyse du corps, soit par analyse du graphe d'activité associé à l'opération.

(b) Si l'événement est temporel (y compris la fin d'une activité) ou de type démon, alors l'action est quelconque.

(c) Plusieurs règles sont relatives aux actions "envoyées" :

i. Les règles de la page 274.

ii. Si l'action modifie des attributs, alors ceux-ci doivent être définis dans la classe de référence de la machine.

(d) Pour toute action, les références acceptables sont définies par le contexte de la classe et de ses super-classes : propriétés d'instances ou de classes, liens d'associations, etc.

(e) Les contraintes de la classe doivent être respectées par les actions de la classe.

5. La garde doit être correcte dans le contexte de la classe (code OCL à vérifier s'il existe).
6. La transition vers un état final apparaissant au plus haut niveau d'une machine est induite par une action de destruction.
7. Si un objet n'a pas de machine, alors il ne peut recevoir que des envois de messages et des signaux. Il doit avoir des opérations correspondant à ces envois de messages ou signaux.
8. Si la machine a un état concurrent, alors sa classe de référence définit des objets actifs.
9. Une classe-état référencée dans une machine est une sous-classe directe de la classe de référence de la machine.
10. Une opération partielle, définie seulement sur certains états, doit inclure le domaine de définition dans sa précondition.
11. La précondition d'une opération doit être cohérente avec les gardes des transitions auxquelles elle est associée.
12. La postcondition d'une opération qui rend un résultat doit être cohérente avec le retour du résultat.

Vérification entre classes et activités

Les vérifications entre classes et machines sont aussi applicables entre classes et activités à ceci près que le contexte est celui d'une opération. Précisons quelques règles :

1. Un objet flot d'état doit être référencé dans le contexte de l'opération.
2. Les arguments à l'état initial sont ceux de l'opération associée.
3. Pour toute action ou activité, les références acceptables sont définies par le contexte de l'opération et de sa classe : variables locales, etc.
4. Les contraintes de l'opération et celles de sa classe doivent être respectées par les actions du graphe d'activités.
5. Lorsqu'une opération est redéfinie ou héritée, quels sont les effets sur le graphe d'activités ?

6. Un état flot d'objet a un type et une direction compatible avec son classificateur ou son classificateur-état (p. 2-165). Il est compatible avec le profil de l'action associée (argument ou résultat) (p. 2-165).
7. Un objet état a un classificateur-état référencé dans le diagramme de classes et compatible avec sa spécification.
8. L'état initial de l'activité doit être cohérent avec la précondition de l'opération.
9. La postcondition d'une opération qui rend un résultat doit être cohérente avec le retour du résultat.

Vérification entre deux machines

La répartition des événements dans les machines et la gestion des files d'événements à l'intérieur des machines sont supposées exister par le métamodèle (p. 2-143) ; la sémantique concrète est différée au langage d'implantation. On peut vouloir établir des corrélations entre machines.

1. Les événements du système ne sont pas partagés, mais un événement peut être dirigé vers plusieurs cibles (p. 2-144). Chaque machine ne reçoit que les événements envoyés à l'objet auquel est associée la machine.
2. Tout événement reçu doit avoir été émis. On peut vérifier les liens de causalité :
   (a) Si une machine reçoit un message ou un signal, alors l'émetteur est un objet. On doit vérifier qu'une communication est possible entre l'émetteur et le receveur :
   - Si l'objet émetteur a une machine, on peut vérifier qu'elle émet ce message ou ce signal à un moment donné.
   - Il peut avoir une opération et/ou son graphe d'activité qui contient l'envoi de message.
   - Si les deux objets échangent des messages asynchrones, ils doivent être actifs.
   - Il existe des associations directes ou indirectes entre leurs classes (vérification de cohérence machines/classes).

   (b) Si une machine reçoit un événement temporel, alors il y a une horloge qui avait été activée dans un état précédent de la même machine.
3. Si un objet a plusieurs machines, elles doivent être cohérentes entre elles (comment ? quelle cohérence ?).
4. Si les classes de deux machines sont liées par la relation de spécialisation, alors les machines doivent être compatibles (cette règle reste à définir précisément). De même, si leurs classes sont des sous-classes d'une classe définissant une machine, l'ensemble doit être cohérent, à niveau d'abstraction près.

Vérification entre machines et activités

Les activités participent à la description des états ou des transitions, elles sont relatives aux opérations. Peu de règles sont définies car toutes les actions ne font pas référence à des opérations, toutes les opérations ne font pas référence à des activités. Précisons quelques règles :

1. Seule une action liée à une activité d'un état peut être interrompue. Si une opération, décrite par un graphe d'activités, est associée à une transition, alors le graphe ne doit pas autoriser la réception d'événements.
2. Les actions engendrées par un graphe d'activités pour une opération invoquée dans une machine deviennent des actions de la machine et sont à mettre en cohérence avec les machines liées (point précédent).

Vérification croisée des diagrammes de types et d'instances

Dans cette partie, nous considérons les collaborations comme un tout. Nous ne distinguons pas les diagrammes de collaboration des diagramme de séquence.

Il faut bien noter que les collaborations sont des exemples d'interactions qui doivent respecter les spécifications des diagrammes de types. Tous les diagrammes d'interaction doivent être conformes aux diagrammes de types. L'inverse n'est pas vrai, certaines propriétés des diagrammes de types ne sont pas démontrables car les collaborations ne définissent pas TOUTES les instanciations possibles. Voici quelques exemples de règles difficiles à démontrer :

- Toute classe a des instances dans au moins une collaboration.
- Tous les événements ou actions de type envoi de message ou de signal d'une machine figurent dans au moins une collaboration (actions reçues/émises).
- Toutes les opérations d'une classe sont utilisées dans au moins un diagramme d'objet ou un diagramme états-transitions ou la description d'une autre opération.
- Tous les attributs sont accédés par au moins une opération (pré- post-conditions OCL).
- La concurrence intra-objets implique une agrégation des objets concurrents (cette règle issue d'OMT n'est pas toujours souhaitable).
- Tout événement d'un diagramme états-transitions apparaît dans un diagramme d'objets.
- Une classe abstraite n'a pas d'instances.

Voici une liste non exhaustive de règles de vérification entre collaborations et diagrammes de types :

1. Tout objet est défini par au moins une classe. Nous conseillons de ne pas avoir d'objets ou de classificateurs type sans nom de classe. Son utilisation est conforme à la spécification de la classe :
   (a) Son état (ses valeurs) est conforme à la définition de ses attributs.
   (b) Ses liens sont conformes à ses associations (classes, cardinalités, contraintes, propriétés).
   (c) Un objet composant ne peut communiquer qu'avec son composé ou avec d'autres composants (encapsulation forte).
   (d) Une classe abstraite n'a pas d'instances.
   (e) L'invariant de la classe est respecté, autant qu'une vérification puisse se faire.
   (f) Si l'objet est actif, sa classe doit l'être.
   (g) Si l'objet est multiple, sa classe doit être une sous-classe de la classe `Collection`.

   Si des règles associées au stéréotype de la classe sont formalisées en OCL, on peut envisager de les vérifier.
2. Si deux objets communiquent, alors il existe une navigation possible entre les classes de ces objets.
3. Pour chaque envoi de message, on vérifie les points suivants :
   (a) Les messages reçus par un objet sont définis par des événements dans le diagramme états-transitions de sa classe ou par des opérations de la classe s'il n'y a pas d'automate.
   (b) Les actions des envois de messages ou de signaux sont conformes à celles des opérations utilisées (profil, synchronisme).
   (c) Les actions des envois de messages ou de signaux sont conformes à celles des transitions (appel d'opération, signal, création, destruction), lorsqu'une machine existe chez le receveur (profil, synchronisme).

(d) Si un message est envoyé par un objet, on doit trouver une action correspondant à cet envoi chez l'appelant, soit dans la description d'une opération ou dans son graphe d'activité, soit dans la machine associée à la classe de l'objet ou de l'une de ses super-classes.

(e) Si un résultat est envoyé par un objet, on doit trouver une action correspondant à l'invocation d'une opération. Le résultat est conforme au profil de l'opération et à sa post-condition.

(f) Si un envoi de message est gardé dans la collaboration, il en est de même dans la machine.

(g) Si un envoi de message est itératif dans la collaboration, il en est de même dans la machine, l'opération ou le graphe d'activité.

(h) Si un envoi de message est récursif dans la collaboration, il en est de même dans la machine, l'opération ou le graphe d'activité.

4. Dans un diagramme de collaboration, on vérifie la conformité de la structure avec le diagramme de classe :

   (a) Les objets ou classificateurs type correspondent à des classificateurs. S'ils ont des descriptions spécifiques, elles doivent être conformes à celles des classes (valeurs, invariants, stéréotypes, etc.).

   (b) Les liens ou associations type correspondent à des associations. Les caractéristiques des associations sont respectées par leurs occurrences (noms de rôles, cardinalités, contraintes, agrégation, composition, etc.).

   (c) Les relations autres que l'association type entre deux classificateurs type sont une copie identique de la même relation entre classe.

   (d) Les stéréotypes et contraintes sont cohérents. Difficile à vérifier.

5. L'ordre des envois de message ou de signaux est conforme à l'ordonnancement imposé par les machines des objets concernés. Les liens de précédence entre messages d'une collaboration sont conformes à l'ordonnancement imposé par les machines des objets concernés.

6. L'ordre des envois de message ou de signaux induits par un appel d'opération est conforme à l'ordonnancement imposé par le graphe d'activité de l'opération, ou à défaut de la liste des actions de l'opération.

7. Si un objet a plusieurs flots de contrôle concurrents dans un diagramme de séquence, alors sa classe est active et ses flots apparaissent dans la machine.

Les points suivants sont plus difficiles à vérifier :
- Cohérence des expressions OCL. Par exemple, vérifier que les conditions des gardes ne sont pas toujours fausses.
- Vérification de type, tenant compte du polymorphisme.
- Simulation des diagrammes de séquence à l'exécution.
- Vérification des règles de visibilité des attributs et des opérations.

5 Vérification en pratique

5.1 Introduction

Dans les sections 3 et 4, nous avons vu les principes de la vérification de modèles UML. Nous avons détaillé nombre de règles. Mais le travail n'est pas suffisant ni satisfaisant pour trois raisons :

1. Rien ne garantit (rien ne prouve) que cet ensemble de règles soit suffisant, ni même cohérent.
2. Rien ne garantit (rien ne prouve) que les modèles auxquels on va appliquer la vérification soient suffisants. La vérification va de pair avec le détail fourni dans les modèles. Si peu de choses sont indiquées, que peut-on vérifier ?
3. Le travail est fastidieux. Nous avons décrit près de 200 règles. Prendre chaque règle et la vérifier sur chaque diagramme concerné est une activité lourde et pénible. La vérification proposée risque d'être appliquée très partiellement.

Dans cette section, nous discutons des conditions de la vérification, nous étudions les besoins vis-à-vis du processus de développement et nous traçons les grandes lignes des solutions retenues.

5.2 Quels besoins en pratique ?

La vérification repose sur des descriptions complètes des modèles, une liste exhaustive et cohérente de règles de vérification, un ordre d'application de ces règles. En pratique, elle repose aussi sur une automatisation du processus, dans l'AGL qui permet de saisir les modèles puis de les exploiter. Ces conditions ne sont pas réunies pour UML, nous en examinons les raisons point par point.

Descriptions complètes des modèles

Une vérification de cohérence n'est pas possible si les diagrammes sont volontairement incomplets c'est-à-dire si "ce qui est noté dans un diagramme n'a pas besoin d'être noté dans l'autre, l'union des deux donnant le modèle global". Par exemple, si dans un diagramme états-transitions les opérations et les événements émis ne sont pas modélisés, il est difficile de vérifier la cohérence avec le diagramme de classes, les diagrammes d'objets et les autres diagrammes états-transitions. En résumé, des vérifications sont possibles si les diagrammes ont un niveau de détail suffisant.

Dans un processus itératif et incrémental, les seuls modèles complets sont ceux qu'on obtient à la fin du processus. La seule sémantique formelle est celle du langage de programmation cible. A chaque itération du processus unifié, les différents modèles du systèmes sont enrichis, sachant que les modèles d'analyse des besoins, d'analyse et de conception sont plus complets dans les premières phases (préparation, élaboration) tandis que les modèles de déploiement, d'implantation et de test sont plus riches dans les dernières phases (construction, transition). La vérification doit s'adapter à cette réalité : on travaille rarement avec des modèles complets.

Il est en effet souhaitable dans un modèle d'analyse de donner l'allure générale sans trop de détails. Ainsi, pour conserver un certain niveau d'abstraction, on peut omettre le profil des opérations, la description interne des opérations, ne définir que des interfaces pour certains paquetages ou certaines classes, etc.

L'usage de la vérification doit prendre en compte ces contraintes pratiques. On peut imaginer plusieurs niveaux de vérification, un filtrage de certaines règles, des priorités dans les erreurs relevées, des degrés de vérification, une qualité de la vérification, etc. Le problème reste ouvert, du point de vue de cet exposé.

Ensemble de règles structuré, complet et cohérent

La vérification n'est possible que si la procédure de vérification est bien établie : ordre et structuration des tâches, consignation des résultats, analyse et retour sur la modélisation. Même si nous avons structuré les règles en ensembles cohérents, il reste du travail à faire pour ordonner le processus de vérification.

La vérification n'est possible que si on dispose d'un ensemble correct, cohérent et complet de règles. Nous avons un ensemble de règles, mais rien ne prouve sa cohérence ou sa complétude. La correction peut être mise en doute par le fait que les règles soient formulées en langage naturel. Il nous faut un langage pour formuler et vérifier les règles.

Automatisation

Vérifier quelques centaines de règles prend du temps. Pour que la vérification soit opérationnelle, il faut l'outiller : outils de saisie et d'éditions des modèles, outils de vérification, outils de simulation, etc. La construction de ces outils repose sur une formulation claire et précise des concepts, des modèles et des règles de cohérence.

5.3 Quels moyens ?

En s'inspirant des techniques de compilation, la vérification dans le langage UML est découpée en deux parties : syntaxe et sémantique. La syntaxe abstraite (les structures réelles du langage, indépendamment du "sucre syntaxique") est celle définie par le métamodèle.

En compilation, toujours, les vérifications lexicales, syntaxiques et certaines vérifications sémantiques sont dirigées par la syntaxe. Dans UML, comme nous l'avons vu, bon nombre de règles lexicales, syntaxiques et de sémantique statique peuvent être formulées à partir du métamodèle.

La structuration des règles est induite par la syntaxe : elles sont rangées par élément de modélisation. Ces règles, pour peu quelles soient formalisées, sont décrites sous forme d'assertions OCL. Pour vérifier les règles, il faut disposer d'outils d'évaluation et de preuve pour OCL.

Principe 5.1 (Evaluation des règles) *Pour automatiser la vérification des règles, il faut les formuler dans le langage OCL et disposer d'outils pour le contrôle, la preuve et l'évaluation avec OCL. En résumé, il faut qu'OCL soit un langage formel à part entière et outillé (édition, évaluation, preuves).*

Un premier travail consiste, à partir de la description du langage OCL [Con99, WK98], à développer un environnement de développement autour d'OCL (éditeurs, compilateur, évaluateur, débogueur, etc.) et une interface avec le métamodèle.

Ce travail permettra alors d'étudier en profondeur le langage et de lui donner une sémantique formelle, condition indispensable pour vérifier la pertinence, la cohérence ou mieux la complétude des règles.

Principe 5.2 (Sémantique d'OCL) *Le principe 5.1 n'est applicable que si OCL dispose d'une sémantique formelle. La construction d'outils repose sur cette sémantique. Le typage d'UML rentre dans cette catégorie.*

Parmi les règles que nous avons proposées pour le modèle logique, nous avons vu que certaines nécessitaient une définition plus précise ou plus exacte des concepts UML. Pour que le langage UML soit exploitable pleinement, il faut qu'il ait une sémantique claire et non ambiguë : une sémantique formelle. Le langage OCL permet d'exprimer des propriétés des spécifications écrites en UML mais elles ne sont que complémentaires, ce sont des contraintes, des invariants. Ces propriétés sont des propriétés du système et non des modèles au sens où nous l'avons défini dans la section 3.1. Le langage OCL n'est pas suffisant pour cela, UML inclut bien d'autres concepts.

Principe 5.3 (Sémantique d'UML) *Pour que le langage UML soit une norme, il faut qu'il ait une sémantique formelle, seule référence possible pour ses diverses implantations et la qualité des modèles (fiabilité, cohérence, portabilité).*

Pour que le langage soit optimal, il faudrait aussi pouvoir définir un "langage" pour les stéréotypes, qui permette pour chaque stéréotype de fixer la définition, les contraintes et les propriétés à vérifier. Ainsi, on peut imaginer un noyau de base, commun, et des modules ou des composants correspondant à des extensions de la notation sous forme de stéréotypes. L'idée étant d'avoir un langage qui, bien qu'adaptable, reste cohérent dans son esprit et ses outils. La notion de module existe déjà dans le métamodèle au travers des paquetages généraux (`Foundation`, `Behavioral elements`, `Model Management`, etc.). On peut s'en inspirer pour définir les extensions de stéréotypage.

Principe 5.4 (Sémantique modulaire d'UML) *Pour que le langage UML soit configurable mais toujours formel, il doit être modulaire et incrémental. En disposant d'un langage qui permette d'intégrer de nouveaux modules, pour les stéréotypes notamment, UML devient un framework de modélisation à objet.*

Enfin, pour que la vérification puisse s'intégrer avec le processus de développement, il faut pouvoir associer des degrés de vérification au niveau d'itération atteint. On accordera plus d'importance à certaines propriétés du système et des modèles dans les premières itérations et plus d'importance à d'autres dans les dernières itérations. La vérification va de pair avec le niveau de détail exigé.

Principe 5.5 (Processus de vérification) *Pour une approche pragmatique de la vérification, on a besoin de définir un processus de vérification, comme le processus unifié définit un processus de test.*

5.4 Perspectives

Chaque principe de la section précédent constitue un axe de travail et de recherche qu'il faut développer. Trois axes peuvent être menés en parallèle :

1. Le langage OCL (principes 5.1 et 5.2). Pour l'utilisation courante d'UML, les travaux sur OCL sont à la fois les plus simples et les plus urgents.
2. Le langage UML (principes 5.3 et 5.4). La vérification sûre ne sera possible que lorsqu'UML disposera d'une sémantique formelle et complète. La sémantique des stéréotypes est secondaire et dépend de la sémantique de base.
3. Le processus de vérification (principe 5.5). Ce point est relativement indépendant, il consiste à intégrer les principes de la vérification dans le processus unifié.

Nous consacrons la section suivante au point numéro 2, la formalisation d'UML.

6 Formalisation d'UML

Dans cette section, nous discutons différentes approches de la formalisation d'UML. Il s'agit d'un point de vue global de la situation avec un angle historique et non d'un état de l'art qui nécessiterait plus d'investigations.

UML ne répond pas complètement aux attentes de ses utilisateurs pour différentes raisons :

1. Le langage ne couvre pas tous les concepts. Certains modèles à objets, utilisés avant UML, ont des concepts qui ne sont pas représentés dans UML. Pour répondre à cette attente, UML propose l'enrichissement de chaque concept par des stéréotypes. La définition du concept est donc propre à son concepteur, elle n'est plus normalisée. Une autre solution est de fournir des notations optionnelles (paquetages supplémentaires dans la notation).
2. Le langage a des définitions générales qui peuvent être interprétées différemment selon les modèles. Pour coller à un maximum de langages de modélisation et de programmation,

la norme reste à un niveau d'abstraction élevé, ce qui induit différentes sémantiques. Ainsi, selon JACOBSON [RJB99b], la sémantique réelle d'un modèle UML est celle de son implantation dans un langage de programmation.

3. Le langage est constitué de différents modèles qui peuvent ne pas être cohérents. UML propose neuf types de combinaisons, appelées **diagrammes**. Ces diagrammes décrivent des aspects complémentaires mais non disjoints du système. Ils ne répondent pas tous à la même préoccupation, certains diagrammes sont plus proches de l'expression des besoins du système tandis que d'autres sont dédiés à l'implantation du système.
4. Le langage est défini par un métamodèle. Bien que présenté comme un outil pour le développement d'outils sur UML, le métamodèle sert aussi à décrire le langage UML, ce qui ne facilite pas la compréhension du langage par un non spécialiste. Il y a confusion possible entre le modèle objet d'UML et le modèle de son langage.
5. Il manque une sémantique formelle des notations. La sémantique formelle est la base d'une compréhension unique et non ambiguë des concepts. Sans sémantique formelle, il n'est pas possible de comparer deux modèles UML réalisés par deux auteurs différents et/ou avec des outils différents. Sans sémantique formelle il est difficile d'implanter des outils de vérification fiables.

Ces problèmes sont liés intrinsèquement aux problèmes de toute normalisation :

1. Choix des concepts du langage. La question essentielle est : le modèle est-il un modèle qui englobe tous les autres ou un modèle minimal, commun aux autres langages ? Plus on ajoute de concepts, plus il y a de risques d'interférence entre les concepts et donc d'incohérence.
2. Pressions des acteurs. Les fournisseurs de méthodes et d'outils souhaitent évidemment que le modèle normalisé soit le plus proche de leurs propres propositions pour faciliter leur adaptation au standard.

Deux approches opposées ont été choisies à l'OMG : un modèle commun pour l'interopérabilité des langages à objets (*Interface Description Language*, IDL dans le cadre de CORBA) et un modèle unifié pour la description de modèles à objets UML. Le modèle commun est un ensemble de règles minimales et bien définies. Le modèle unifié est plus ambitieux, il veut couvrir l'ensemble des autres modèles.

Le modèle UML est le résultat d'un compromis ambitieux sur des méthodes et les langages de programmation du marché. Le compromis porte sur différents aspects : couverture du cycle de développement (un seul modèle de l'analyse des besoins à la réalisation), couverture des applications (toutes sortes d'applications), couverture des langages (toutes sortes de langages à objets), couverture des préoccupations (modélisation conceptuelle, répartition des processus, persistance...). Une telle richesse se paie par des risques de discordance et d'incohérence. En résumé, le reproche principal fait à UML est son manque de sémantique précise (formelle).

La formalisation des modèles des méthodes d'analyse et conception n'est pas un problème nouveau. D'une part, ces modèles viennent en général de la programmation ou de théories diverses des mathématiques, qui sont des bases relativement solides. D'autre part, la construction des outils et la mise en pratique des méthodes ont toujours nécessité des bases saines pour une compréhension acceptée par tous. Par exemple, on peut citer la logique, l'algèbre relationnelle, le calcul fontionnel et la réécriture, les algèbres de processus, les automates, les réseaux de PETRI, etc.

Dans les méthodes traditionnelles, la sémantique formelle s'est toujours heurtée au fait qu'on disposait de modèles, issus de théories différentes, qui ne se combinaient pas forcément bien. Il s'agit du problème, toujours actuel, des spécifications hétérogènes. Ainsi, des travaux ont essayé de combiner les approches par modèles abstraits (Z, VDM) avec les approches à processus (CCS, CSP), les approches processus et fonctionnelles (ou algébriques) (ACT1,

Lotos) ou objet (OBJ, Maude), structurelles et processus (SDL, RSL), modèle et algébriques (Cold).

Dans les méthode à objets, le besoin de formalisation est surtout apparu avec OMT. Il est né du fait qu'on a trois axes de modélisation qu'il faut rendre cohérents [HC91, ABR95, BC95]. Mais cela restait un problème annexe du point de vue des fournisseurs d'outils et de méthodes, même si des méthodes comme Synthropy [CD94] visent à renforcer les aspects formels[5]. En parallèle, plusieurs travaux issus de la communauté des méthodes formelles visaient à développer des méthodes formelles à objets. Les références suivantes donnent un bon aperçu de ces travaux [CLLF93, LH92, GK96, AR97, AR96]. Ont ainsi vu le jour des langages comme Object-Z, OOZE, Z++, VDM++, MooZ, Troll, Maude, etc. Ces langages sont issus des communautés autour de Z et autour des spécifications algébriques.

Le vrai besoin de formalisation s'est fait sentir avec UML. En effet, pour se prévaloir du titre de norme, UML se doit d'être un langage rigoureux et bien fondé. De nombreux travaux ont été lancés, ce qui a redonné une nouvelle jeunesse aux communautés des méthodes formelles, restées en marge du développement industriel. Ces communautés ont trouvé des débouchés à leurs travaux. Un groupe a même été formé pour cela, le groupe *Precise UML*[6] [EFLR98, EBF+98].

Il y a actuellement bon nombre de propositions en ce sens. Au départ, il s'agissait surtout de *remakes* de formalisation du modèle E-A-P (Entité-Association-Propriétés), c'est-à-dire la formalisation du diagramme des classes. Les premiers travaux n'étaient guère innovants car ils se bornaient aux classes, attributs, associations et spécialisations. La prise en compte des relations de spécialisation, des agrégations et des compositions a mis en évidence des failles dans la sémantique informelle d'UML. Assez curieusement, nous n'avons pas trouvé, mais notre science est loin d'être infuse en ce domaine, beaucoup de travaux reprenant les modèles des méthodes formelles à objets, surtout en ce qui concerne les spécifications orientées modèles (Z, VDM). Nombre de travaux reprennent en effet directement Z pour une formalisation d'UML. Nous expliquons cet état de fait par deux raisons : premièrement il s'agissait de travaux essentiellement théoriques et deuxièmement ces langages sont insuffisamment outillés, l'outil étant l'aiguillon qui permet à une idée de passer dans la pratique courante. Cela explique aussi pourquoi nombre d'approches de ce courant sont maintenant basées sur B, qui propose un environnement complet et fiable, et qui s'enrichit toujours de nouvelles extensions utilisables en objet.

La période actuelle pourrait être qualifiée de *brain storming*. En effet, chaque groupe, selon son expérience et ses "origines" formelles, propose une formalisation partielle d'UML. La première étape fut celle des formalisations individuelles de diagrammes (essentiellement le diagramme des classes, nous l'avons déjà dit). Elle n'est ni satisfaisante ni terminée pour l'instant. Elle met en évidence un certain nombre de difficultés concernant la sémantique des objets et des relations ou leur traduction dans tel ou tel support formel. Nous constatons différentes sémantiques des concepts objets, induites par les théories formelles sous-jacentes. La seconde génération des propositions essaie de globaliser le modèle objet en prenant en compte les aspects dynamiques ; le modèle objet devient un modèle dans lequel la communication prend plus d'importance [BHH+97, WK00, GPP98, Reg02, RCA01]. Ce n'est plus seulement une unité de stockage ou de calcul fonctionnel, mais une unité de coordination. Les concepts de la programmation concurrente à objets ne peuvent être ignorés dans la sémantique d'UML. Il n'y a pour l'instant pas de consensus sur un modèle objet formel.

Une autre approche de la formalisation concerne la formalisation du métamodèle. Si une sémantique formelle d'UML est donnée, comme le métamodèle est exprimé en UML, alors il a une sémantique formelle. A défaut, on peut donner une sémantique formelle du métamodèle, qui est plus simple, et élaborer à partir de là une sémantique pour UML.

[5] Syntropy s'est inspiré notamment de Z. Le langage OCL s'est lui inspiré de Syntropy !

[6] `http ://www.cs.york.ac.uk/puml/`

7 Conclusion

Dans ce chapitre, nous avons exploré le domaine de la vérification avec UML. Nous avons montré l'importance de cette activité mais aussi les nombreuses difficultés rencontrées. Les difficultés sont dues à la complexité (nombre et variétés des diagrammes, des modèles, des concepts), à l'intégration dans le processus de développement unifié (modèles incomplets et incrémentaux), au manque de règles et d'outils de vérification. Enfin, il manque une sémantique formelle à UML et OCL pour définir clairement les règles et mettre en œuvre efficacement les contrôles.

La tâche est conséquente et dépend de nombre d'acteurs. Une première étape est la formalisation du langage OCL. Elle permet ensuite de définir clairement un certain nombre de règles dans le méta-modèle et de prévoir l'intégration de la vérification dans le processus unifié. L'aboutissement ne peut avoir lieu que lorsqu'une sémantique précise et non ambiguë d'UML aura été adoptée. Les extensions du langage (stéréotypes, temps-réel) pourront alors être développées comme incréments sémantiques d'UML.