Ingénierie de systèmes d'information : une approche de multi-modélisation et de méta-modélisation

Jose Celsio Freire Junior

THÈSE

présentée par

José Celso FREIRE JUNIOR

pour obtenir le grade de DOCTEUR

de l'UNIVERSITÉ JOSEPH FOURIER - GRENOBLE 1

(arrêté ministériel du 30 Mars 1992)

Spécialité : **Informatique**

Ingénierie des Systèmes d'Information : Une Approche de Multi-Modélisation et de Méta-Modélisation

---

Date de soutenance : 10 juillet 1997

Composition du jury :

Président :	Yves Ledru
Rapporteurs :	Henri Habrias
	Flavio Oquendo
Examinateurs :	Jean-Pierre Giraudin
	Monique Chabre-Peccoud

Thèse préparée au sein du

LABORATOIRE LOGICIELS SYSTÈMES RÉSEAUX - IMAG

À mes parents et à Néia

“Se as coisas são inatingiveis... ora !
Não é motivo para não querê-las...
Que tristes os caminhos, se não fora
A mágica presença das estrelas !”

Mario Quintana

Remerciements

Je tiens à remercier :

Mr. Yves Ledru, Professeur à l'Université Joseph Fourier de Grenoble, de m'avoir fait l'honneur de présider ce jury de thèse.

Mr. Henri Habrias, Professeur à l'Université de Nantes, d'avoir jugé ce travail. Je tiens particulièrement à le remercier pour avoir relu avec beaucoup d'attention une version préliminaire de ce manuscrit. Ses remarques et ses conseils judicieux ont permis d'améliorer significativement ce mémoire de thèse.

Mr. Flavio Oquendo, Professeur à l'École Supérieure d'Ingénieurs d'Annecy à l'Université de Savoie, d'avoir bien voulu apporter son jugement sur ce travail. Par ses remarques, il a contribué à la qualité de ce document.

Mr. Jean-Pierre Giraudin, Professeur à l'Université Pierre Mendès France de Grenoble, pour sa disponibilité, sa confiance, ses critiques constructives et les discussions fructueuses tout au long de ces quatre années de travail. Ses corrections et propositions lors de la rédaction ont été d'une grande importance pour la qualité de cette thèse. D'un point de vue personnel, je tiens à le remercier par ses nombreux encouragements ainsi que pour sa gentillesse.

Mme Monique Chabre-Peccoud, Maître de Conférences à l'Université Joseph Fourier. Depuis mon arrivée à Grenoble, elle m'a soutenu dans ma nouvelle vie. Outre cette aide personnelle, je la remercie pour ses conseils sur mon travail.

Mr. Michel Adiba Professeur à l'Université Joseph Fourier de Grenoble et Mme Christine Collet, Maître de Conférences à l'Université Joseph Fourier, pour m'avoir accueilli dans l'équipe Storm et ainsi m'avoir donné les moyens d'effectuer ce travail de recherche.

Mr. Paul Jacquet, Professeur à l'Institut National Polytechnique de Grenoble et Directeur du Laboratoire Logiciels Systèmes et Réseaux, pour l'accueil au sein de ce nouveau laboratoire.

Le CNPq/Brésil, et le Departamento de Engenharia Elétrica de l'Universidade Estadual Paulista - UNESP campus de Guaratinguetá, pour m'avoir soutenu lors de la préparation de cette thèse. Je tiens aussi à remercier les gens qui, au Brésil, m'ont soutenu. Je pense plus particulièrement à mes collègues d'UNESP, Amorim, Maysa et Lotufo.

Mes collègues du LSR, en particulier Claudia, Françoise, Javan, Rafael et Thierry, pour les tête-à-tête à la Kfet et dans les couloirs et François pour son aide avec les problèmes techniques. Je tiens à remercier tout particulièrement Agnès, pour les discussions, les suggestions ainsi que le soutien et l'énorme effort, sans jamais dire non, qu'elle a dédié à ce travail, en tant que "traductrice" de mon pauvre français vers la langue de Molière... Je ne pourrais pas oublier Hervé, qui pendant ces quatre années a participé en tant qu'ami à tous les problèmes que j'ai rencontrés.

Au delà de cette thèse en informatique, ces quatre années m'ont permis aussi d'élaborer une "thèse" à propos de la vie... Je tiens à remercier les amis qui pendant le temps que j'ai passé à Grenoble, m'ont soutenu et aidé dans cette recherche. Je pense plus particulièrement à Claudia et Ricardo, Ana Paula, Marilia et Paulo, Alessandra et Fabiano et Denise et Romeu.

Mes parents qui ont su être l'arc qui m'a lancé vers l'avenir et qui comme le vent, ont accompagné, en étant toujours complices, le parcours de leur flèche.

Néia, pour sa présence et son soutien indéfectible tout au long de ces années. Nous aussi, nous avons écrit un chapitre de notre "thèse", et je suis convaincu que cette expérience nous ouvre de belles perspectives.

Table des matières

Liste des figures

Liste des tableaux

Chapitre 1

Introduction

1.1 Contexte et Motivation

Dans le cadre de la modélisation des Systèmes d'Information - SI, de l'expression des besoins à la conception et à la réalisation de solutions logicielles, de nombreuses représentations souvent hétérogènes sont utilisées. Elles se différencient pour exprimer différents niveaux (externe, conceptuel, logique, etc) ou différents points de vue (statique/dynamique/ fonctionnel, comportement externe, contexte, architecture du logiciel, ...). Ces représentations constituent des modèles (ou un modèle) du système d'information. Ces modèles ont différents objectifs selon le moment où on les écrit ou celui où on les utilise : on parlera de modèle de spécification, de conception, d'implantation, etc. Les nombreux acteurs qui interagissent (décrivent, évaluent, utilisent, etc.) dans cette modélisation ont des compétences et souvent aussi des objectifs différents.

Les concepts et les notations utilisés par ces représentations sont variés. Certaines représentations s'appuient sur des langages libres ou un peu structurés, d'autres sur des langages précis et formels. Dans le domaine des systèmes d'information, ce sont essentiellement des représentations graphiques qui sont utilisées ; elles sont qualifiées de langages semi-formels.

Cette situation est toujours d'actualité dans le cadre des approches à objets avec des représentations simples (ex. OOA) ou plus complètes (ex. OMT). Pour certains aspects, des formalismes simples suffisent, mais d'autres aspects comme, par exemple, la correspondance, la transformation, la cohérence et la complétude de modèles nécessitent des approches et des langages plus formels.

Partant de la nécessité de l'hétérogénéité des représentations, il s'agit d'utiliser des notations et des raisonnements pour, à terme, disposer de modélisations plus complètes et plus rigoureuses. L'objectif de ce travail s'ouvre donc sur une double perspective : d'une part, faciliter l'utilisation d'approches formelles dans le domaine des systèmes d'information et, d'autre part, augmenter la qualité sémantique et la précision de représentations graphiques de manière à mieux répondre aux besoins des différents acteurs.

Le domaine abordé et les techniques à utiliser pour aboutir à l'objectif recherché rendent cette recherche difficile, car il faut conjuguer :

- une bonne expérience de la modélisation de systèmes d'information, dans la mesure où il faut appliquer ces techniques de manière conjointe au processus de modélisation de systèmes qui est, à lui seul, une activité complexe ;
- une connaissance des nouvelles méthodes de conception orientées objets, car ce sont ces méthodes qui sont utilisées lors de la modélisation des systèmes ;
- des notions suffisantes de techniques de spécifications formelles, car ces techniques sont utilisées afin de rendre plus formelles, plus précises et plus rigoureuses les modélisations créées avec les méthodes orientées objets ;
- une pratique des ateliers de génie logiciel, car l'un des résultats de l'application de ces différentes techniques vise la construction d'un atelier expérimental de modélisation ;
- une possibilité de définir un nouvel environnement de développement d'applications s'appuyant sur trois types de formalismes qui, jusqu'à maintenant, étaient pris en charge d'une manière non homogène.

1.2 Contribution de la Thèse

Après différentes études concernant principalement les méthodes orientées objet et les méthodes formelles, nous nous sommes plus particulièrement intéressés à l'analyse de différentes relations qui peuvent exister entre des modélisations informelles, semi-formelles et formelles utilisables conjointement. Ce travail formalise d'une manière partielle le problème de couplage entre des représentations utilisant des formalismes différents dans le cadre d'objectifs différents. Ces éléments nous ont conduit à la nécessité de disposer d'un méta-modèle.

Notre principale contribution porte sur la proposition et l'utilisation d'un méta-modèle. Ce méta-modèle est à la base de la représentation de méthodes multi-paradigmes et sert de support pour l'utilisation conjointe de trois approches de modélisation (informelle, semi-formelle, et formelle) d'une manière cohérente et assistée, soit dans la **méta-modélisation de méthodes**, soit dans la **multi-modélisation de systèmes d'information**.

Nous pouvons de plus utiliser le méta-modèle proposé au niveau de la comparaison de méthodes en construisant un modèle global (le méta-méta-modèle) pour représenter des modèles de méthodes (le méta-modèle) afin de les comparer.

Notre méta-modèle constitue le cœur d'un atelier expérimental de modélisation offrant une approche simplifiée qui englobe les différents types de modélisation. Ce méta-modèle peut être utilisé dès la phase de description d'une méthode jusqu'à l'utilisation d'un atelier spécifique généré et dédié à la spécification d'un système d'information selon une méthode choisie. De cette manière nous offrons un générateur d'ateliers pour des méthodes spécifiques. Le travail présenté contribue ainsi au domaine de la spécification des systèmes d'information.

Notre atelier expérimental de modélisation intègre un éditeur hyper-texte qui, à partir de l'utilisation du méta-modèle proposé, permet de produire des documents de spécification de systèmes combinant différentes formes de modélisation. À travers la construction de documents hyper-textes, nous abordons partiellement le problème du guidage d'un processus de modélisation et de développement de systèmes.

1.3 Organisation de la Thèse

Cette thèse est organisée de la manière suivante :

– Le chapitre 2 introduit le concept de modélisation par rapport à son utilisation en génie logiciel et décrit des ateliers de génie logiciel qui automatisent l'utilisation de méthodes. Ensuite, trois formes de modélisation (informelle, semi-formelle et formelle) sont détaillées. Pour la spécification semi-formelle, on présente des approches cartésiennes et systémiques ; la spécification formelle est étudiée à travers plusieurs approches (Z, VDM, B). La présentation de la notion de méta-modélisation termine ce chapitre.

– Dans le chapitre 3, un type particulier de modélisation semi-formelle est étudié. Il s'agit de la modélisation orientée objet de par son importance dans le milieu professionnel. Les concepts clefs (objet, classe, héritage, etc.) du monde objet sont introduits

ainsi que des démarches qui peuvent être utilisées avec la modélisation orientée objet. Puis, quatre méthodes orientées objets sont introduites et comparées ; elles sont détaillées en annexes.

– La proposition d'un nouvel atelier de modélisation constitue le chapitre 4. Dans ce but, nous décrivons des éléments nécessaires à intégrer aux nouveaux ateliers de modélisation. Ensuite, notre méta-modèle est décrit et illustré dans un contexte de méta-modélisation ainsi que de multi-modélisation. L'architecture de l'atelier développé est alors présentée avec ses deux niveaux d'usage. Enfin ce chapitre décrit l'étude sur des relations entre fragments de modélisations de formalismes différents.

– Le chapitre 5 est consacré à la présentation de l'implantation et des expérimentations réalisées avec l'atelier développé. L'implantation de l'atelier est présentée d'une part par rapport à chacun des outils dont il est composé et d'autre part vis-à-vis des niveaux possibles d'utilisation en méta-modélisation ou multi-modélisation. Quatre études de cas sont ensuite réalisées afin de démontrer les capacités de cet atelier expérimental. Ces études de cas portent sur la méta-modélisation et la multi-modélisation de méthodes orientées objets, sur leur comparaison et sur la spécification d'une partie de l'application STORM.

– Le chapitre 6 conclut ce travail en présentant nos apports par rapport à la modélisation des systèmes d'information. Il indique aussi les aspects techniques à étudier afin que le prototype d'atelier développé soit mieux intégré et utilisable.

– Des annexes complètent ce mémoire en décrivant des modèles et méthodes qui nous ont servi de référence pour analyser les types d'éléments que notre méta-modèle et notre atelier devaient supporter et combiner pour les enrichir.

Chapitre 2

La Modélisation

2.1 Modélisation dans le Génie Logiciel

Selon le Petit Robert, modéliser consiste à “établir le modèle de quelque chose”, “présenter sous forme de modèle”. La modélisation est donc l’établissement d’un modèle pour quelque chose. Si ce quelque chose est un logiciel, alors on peut se poser les questions suivantes : *qu’est-ce qu’un modèle* et *pourquoi des modèles pour les logiciels* ?

Par rapport à la première question, nous trouvons plusieurs acceptions du terme modèle. Selon le Dictionnaire Encyclopédique du Génie Logiciel de H. Habrias [Hab97], nous pouvons distinguer trois notions de modèle :

- celle de l’ingénieur qui a été énoncée par M. Minsky [Min68] : “Pour un opérateur `O`, un objet `M` est un modèle de l’objet `A` si `O` peut utiliser `M` pour répondre à des questions à propos de `A`” ;
- celle du mathématicien appliqué : on abstrait la réalité en utilisant des concepts mathématiques (ensembles, relations, graphes, ...) ;
- celle du logicien : la sémantique va être exprimée par une fonction d’identificateur vers des significations.

Plus généralement, on peut dire qu’un modèle est une simplification de la réalité, ce qui n’est pas un défaut car le travail sur un modèle exempté de détails inutiles et non pertinents est plus intéressant dans le processus de modélisation d’un Système d’Information.

La deuxième question portant sur les modèles et les logiciels est traitée ci-dessous.

2.1.1 Historique du Génie Logiciel

Le logiciel comme pièce de fond du monde informatique recouvre toute une industrie ; cette *industrie du logiciel* récente est basée sur le concept du *Génie Logiciel* ou, plus précisément, d'Ingénierie du Logiciel.

La première définition du génie logiciel ("Software Engineering") a été donnée par P. Naur dans une conférence de l'OTAN [NR69] :

> "L'établissement et l'utilisation des principes valides d'ingénierie pour obtenir d'une manière économique un logiciel fiable qui marche efficacement avec des machines réelles".

Une définition plus pragmatique est donnée par R. S. Pressman [Pre94] selon laquelle le génie logiciel englobe les trois points suivants :

- *les méthodes* : elles définissent le comment, c'est-à-dire ce qu'il faut faire pour produire un logiciel. En d'autres termes, c'est la technique pour la résolution des problèmes. Chaque méthode a besoin d'un *langage* pour la production d'une spécification, et donc à chaque méthode est attaché un langage propre.
- *les outils* : ils constituent le support pour les méthodes. Ces outils peuvent être réalisés ou non d'une manière automatique. Lorsque des outils sont intégrés d'une manière telle que l'information créée par un outil peut être utilisée par un autre, on peut dire qu'un système pour le développement du logiciel est établi [Pre94]. Ces systèmes composent ce qu'on appelle les Ateliers de Génie Logiciel - AGL ou CASE ("Computer Aided Software Engineering") ; ce point sera traité en détail dans la section 2.1.3.
- *les processus* : ils encadrent les méthodes et les outils en définissant la séquence selon laquelle les méthodes seront utilisées ainsi que les documents qui seront produits à partir du contrôle établi sur ce processus.

Il existe donc un processus de modélisation appliqué dans le génie logiciel pour la modélisation d'un SI. Ce processus exprime le passage du domaine d'un problème vers celui de sa solution informatique [Bar92]. Ce processus concrétisé par un *cycle de vie* a pour objectif l'amélioration des points suivants pour tout logiciel développé :

- l'*exactitude*, qui est l'évaluation qualitative de l'importance d'une erreur [Hab97], exprime le compromis entre le logiciel développé et les fonctions qu'il est censé implanter, d'une manière propre, cohérente et prévisible ;

– l'"*utilisabilité*", qui est une tentative pour quantifier la convivialité d'un logiciel [Pre94], exprime le compromis entre ce que voulaient les utilisateurs du système et la structure du produit développé, son implantation et sa documentation ;

– le *coût effectif* , qui est l'évaluation quantitative financière du logiciel, exprime le compromis entre la valeur que l'utilisateur voulait payer et le coût réel total du logiciel développé.

Le génie logiciel comprend donc l'utilisation de méthodes d'ingénierie pour atteindre les objectifs cités ci-dessus, lors de la production d'un logiciel. L'emploi du génie logiciel, qui aujourd'hui s'impose comme une solution naturelle, est apparu au début des années 60 comme une solution qui pouvait résoudre la "*crise du logiciel*" se manifestant entre autres par le dépassement du coût des logiciels par rapport au coût du matériel.

Le génie logiciel, en tant que processus, est détaillé dans la prochaine section.

2.1.2 Utilisation du Génie Logiciel

L'utilisation du génie logiciel dans la production d'un logiciel qui accomplit les besoins attendus en même temps qu'il satisfait les objectifs d'un processus d'ingénierie est implantée avec un processus par étapes. Ces différentes étapes forment le *cycle de vie d'un logiciel*. Plusieurs modèles décrivant ce processus ont été proposés. Ci-dessous, on en présente quelques uns en commençant par le modèle en cascade de W. Royce [Roy70] et en allant jusqu'au modèle en spirale de B. Boehm [Boe88].

Le modèle du **Cycle de Vie en Cascade** est présenté à la figure 2.1. Dans cette figure, on peut voir les différentes phases identifiables lors de l'implantation d'un logiciel. La phase 5 représentée avec une ombre ainsi que les flux représentés en pointillés ont été ajoutés au modèle comme réponse aux critiques apportées au modèle original présenté en 1970. Parmi ces critiques, on peut citer, par exemple, celle précisant que le modèle n'est pas utilisable avec tous les types de systèmes, et celle engendrée par la pratique et prouvant que les phases se recouvrent les unes par rapport aux autres, ce qui rend le modèle peu réaliste.

La figure 2.1, dans sa totalité, présente le modèle dans une forme qui est aujourd'hui plus ou moins acceptée dans la communauté mais qui, malgré cela, présente encore quelques problèmes : le grand nombre d'itérations rend difficile l'établissement de points d'arrêts pour la planification du développement, le développement parallèle est difficile, il n'y a pas vraiment un formalisme clair appliqué au développement et à la vérification et, l'étape de tests étant la dernière avant la livraison, les problèmes peuvent être découverts trop tardivement.

Une analyse plus approfondie est faite dans de nombreux ouvrages [Som92, MJ82].

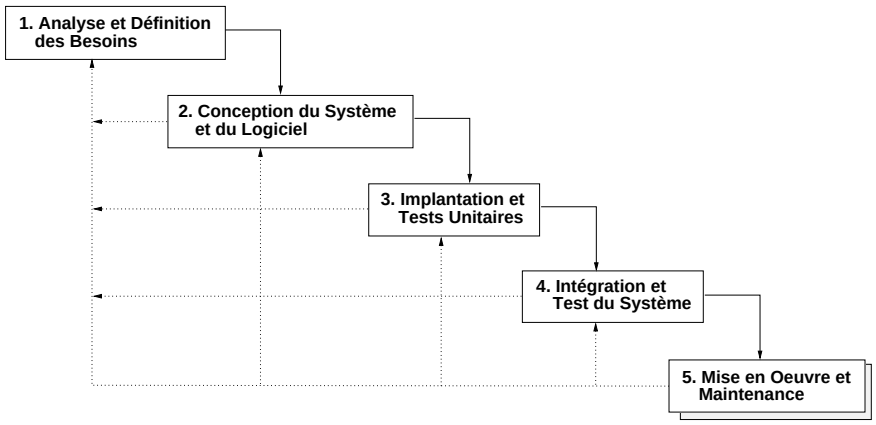


Figure 2.1 - Le Cycle de Vie en Cascade

Une évolution du cycle en cascade est le modèle du **Cycle de Vie en V** qui est présenté à la figure 2.2 [GMSB96]. Dans ce modèle le problème relatif aux tests est mis en évidence. Si on divise le modèle verticalement, on peut dire que le côté gauche présente les étapes en rapport avec la construction du système et que le côté droit présente les étapes qui traitent de la validation et de la vérification. Les étapes de test pour la validation et la vérification sont préparées par les premières étapes du cycle.

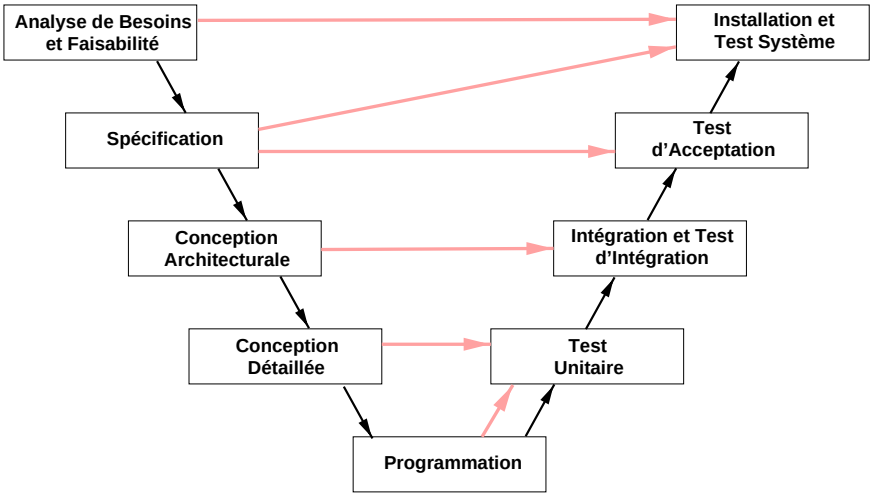


Figure 2.2 - Le Cycle de Vie en V [GMSB96]

Dans la figure 2.2, les flèches en gras lient des étapes qui, grosso modo, sont les mêmes que celles définies pour le modèle du cycle de vie en cascade. Les flèches ombrées indiquent qu'une partie des résultats produits par une étape particulière est utilisée par l'étape où arrive la flèche ; ainsi, par exemple, après l'étape de programmation, les tests unitaires doivent être décrits et prêts à être utilisés.

En plus, le modèle peut être divisé horizontalement en trois parties : la partie supérieure présente la partie du modèle à travers laquelle les utilisateurs peuvent intervenir sur le développement, la partie du milieu correspond à la solution architecturale proposée et dans la partie inférieure l'implantation est traitée.

Bien que le cycle en V propose une meilleure solution au processus de test, les autres problèmes cités pour le cycle en cascade persistent. Une solution apparue pour résoudre ces problèmes est celle qui propose l'utilisation d'un **Processus de Prototypage**, lequel utilise un processus de programmation exploratoire où un logiciel est développé pour être présenté aux utilisateurs et de cette manière corrigé. Le prototypage est donc un processus qui rend possible, au développeur, la création d'un modèle d'un logiciel qui doit être construit. Les étapes qui composent ce processus sont présentées dans la figure 2.3 [Pre94].

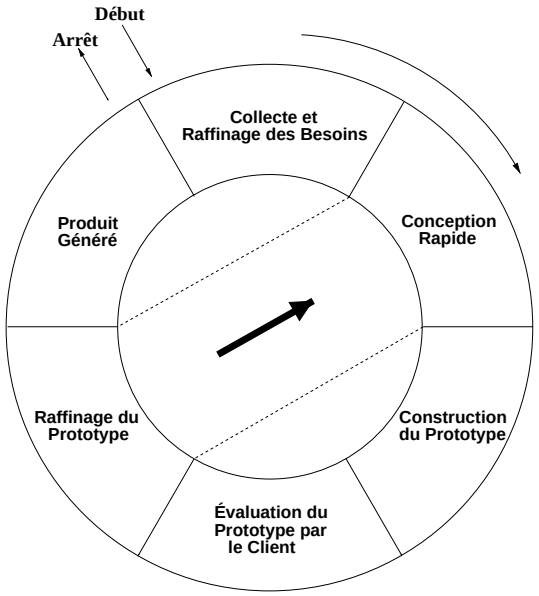


Figure 2.3 - Le Processus de Prototypage [Pre94]

Les problèmes de cette approche sont divers : la documentation est difficile, les systèmes produits peuvent ne pas être bien formés ce qui peut rendre la maintenance coûteuse et, enfin, les structures organisationnelles d'une entreprise peuvent avoir des difficultés d'adaptation pour travailler de cette manière.

Bien que cette approche puisse marcher pour une certaine catégorie de systèmes, une approche plus cohérente est l'intégration de l'approche par prototypage dans un processus de développement de logiciel.

Une proposition qui intègre le cycle de vie en cascade et le prototypage est celle du **Cycle de Vie en Spirale** proposé par B. Boehm [Boe88] (cf. figure 2.4), qui prend en compte les meilleures caractéristiques des deux approches citées. Ce cycle de vie est basé sur les risques : avant chaque cycle de la spirale, on effectue une analyse des risques et à la fin de chaque cycle, on estime par une procédure de révision si on passe au cycle suivant ou non.

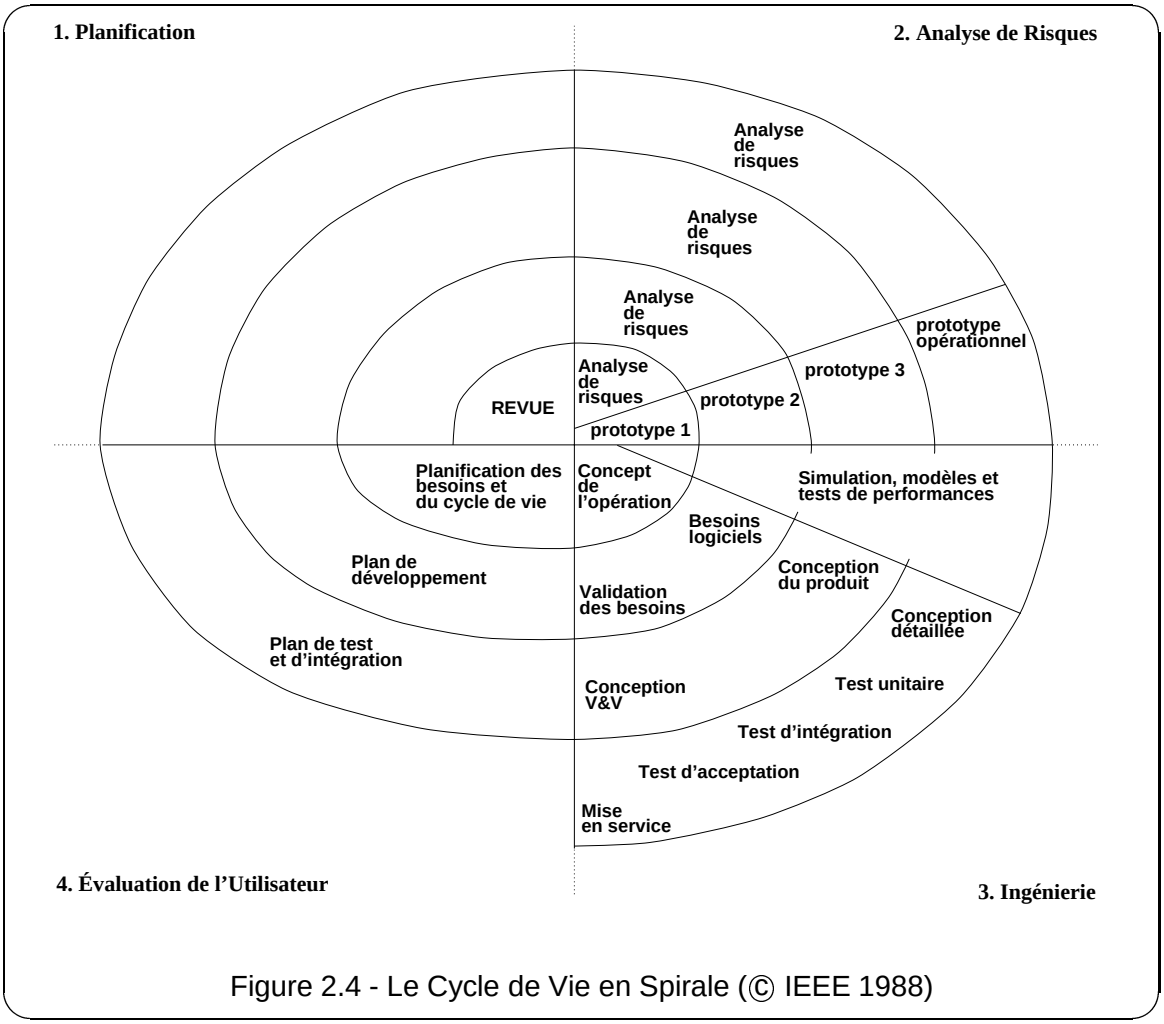


Figure 2.4 - Le Cycle de Vie en Spirale (© IEEE 1988)

La figure qui présente le cycle de vie en spirale est divisée en quatre parties qui représentent les quatre activités principales présentes dans ce cycle de vie :

1. *planification* : sert à déterminer les objectifs, les alternatives et les contraintes ;
2. *analyse de risques* : sert à évaluer les alternatives et à identifier et à résoudre les risques ;
3. *ingénierie* : sert à développer et à vérifier le produit ;
4. *évaluation de l'utilisateur* : estimation des résultats de l'ingénierie.

Les outils cités comme des composants du génie logiciel lorsqu'ils servent de support pour les méthodes, sont décrits dans la prochaine section.

2.1.3 Ateliers de Génie Logiciel

L'application des méthodes (traitées plus loin dans ce document) dans le Génie Logiciel est devenue un processus difficile, et, au fil des années, des outils automatiques ont été produits pour rendre plus pratique l'application de ces méthodes au processus de développement. Ces outils sont apparus au début des années 80 et ont été appelés *CASE* "Computer-Aided Software Engineering" ou Ateliers de Génie Logiciel - AGL.

Plusieurs définitions des AGL peuvent être données. On présente ici une définition qui fait ressortir l'aspect modélisation et qui a été donnée par R. J. Noran [Nor92] : "Un AGL est un ensemble intégré d'outils qui permet aux développeurs de logiciel de documenter et modéliser un système d'information dès la spécification initiale des besoins jusqu'au projet et son implantation, en passant par l'application de tests de cohérence, complétude et conformité aux spécifications proposées".

Plusieurs types de classification peuvent être appliqués aux AGL. Par exemple, on peut parler de générations d'AGL et la table 2.1 (empruntée de [Ham93]) présente cette classification.

Génération	Période	Caractéristiques
première	1972-1981	• Matériel de support cohérent ; • Outils hétérogènes facilitant la manipulation des couches de logiciels ; • Programmation traditionnelle ; • Orienté réalisation et gestion de code source.
deuxième	1981-1990 ?	• Matériel cohérent ; • Outils connectés par des ponts ou par un dictionnaire commun ; • Génération de code structuré ; • Supporte des méthodes structurées ; • Couverture de tout le cycle de vie (en principe) ; • Pas de gestion de projets ni d'assurance qualité.
troisième	1990 ?- ?	• Architecture répartie ; • Dictionnaire et interfaces normalisées ; • Méthodes paramétrables ; • Édition syntaxique/graphique ; • Génération de code réutilisable ; • Gestion de projets et assurance qualité intégrées.

Table 2.1 - Générations d'Ateliers de Génie Logiciel [Ham93]

Selon P. Oman [Oma90], un AGL doit être un outil qui implante des méthodes connues et bien définies. La définition de B. Boehm [Boe76] pour le génie logiciel explicite que tout le cycle de vie, de l'analyse des besoins jusqu'à la maintenance, fait partie du génie logiciel. En prenant en compte ces deux affirmations, on arrive à la classification donnée par A. Wasserman [Was89] selon laquelle les outils peuvent être classés en deux types d'architecture :

– *verticale* : ce sont des outils qui implantent une activité spécifique du cycle de vie, comme par exemple l'analyse ;

– *horizontale* : ce sont des outils (ou un "multi-outil" unique avec divers composants) qui supportent toutes les activités du cycle de vie ; ces outils ont comme principal problème l'intégration des différents composants afin que l'"outil global" ait des interfaces utilisateur cohérentes, qu'il puisse assurer l'échange de données entre les composants, qu'il soit portable et enfin qu'il puisse rendre possible l'intégration de données [SO90].

Pour qu'on puisse classer un AGL comme un outil d'architecture horizontale, il faut donc que celui-ci intègre plusieurs autres outils. Sur cette intégration, portent quelques propositions qui essayent de la normaliser. Dans ce texte, on ne traite pas ce point qui est détaillé dans [Eme94] et [OMG93].

Avec l'apparition de différentes méthodes et la permanente adaptation de celles-ci à des caractéristiques propres de chaque entreprise, l'intégration d'une méthode "particulière" dans un AGL a commencé à poser des problèmes quand il a fallu avoir des AGL adaptés aux besoins propres de chaque entreprise. La solution à ce problème est apparue avec les *Méta-AGL*, qui sont des AGL destinés à construire des AGL ; il faut d'abord utiliser les premiers pour créer les deuxièmes qui seront alors employés par le développeur de logiciel. De cette manière, on peut encore classer les AGL selon le niveau d'utilisation : *méta* ou *normal*. Naturellement, les entreprises qui travaillent avec les méta-AGL proposent leurs propres AGL pour différentes méthodes.

La dernière classification proposée dans ce texte a pour but la division des AGL en *commercial* et *de domaine public*. Un AGL commercial est vendu alors qu'un AGL du domaine public est fourni normalement avec des objectifs de recherche ou en tant que "shareware".

Une composition des classifications méta/normal et commercial/domaine public peut être proposée. La table 2.2 introduit ainsi quelques AGL présentés dans les sections suivantes. Pour une liste plus exhaustive des AGL voir [CS96, Sto96, Fer96].

	Méta	Normal
Commercial	Paradigm Plus GraphTalk	Rational Rose ObjectTeam/OMT
Domaine Public	Hardy MetaView	Object Domain TCM

Table 2.2 - Ateliers de Génie Logiciel

2.1.3.1 Paradigm Plus

L'outil Paradigm Plus [PLA96c], outre ses capacités de générateur d'AGL, intègre trois autres caractéristiques. L'utilisation de la méthodologie Modélisation des Composants d'Entreprise ("Entreprise Component Modeling" - ECM) [PLA96a] est possible pour la modélisation de processus commerciaux. ECM est intégrée à l'outil et en travaillant sur un dictionnaire de données, elle doit permettre le développement des applications commerciales d'une manière rapide et avec un haut degré de qualité.

Pour des applications fortement basées sur des systèmes de gestion de bases de données relationnels, des facilités sont offertes pour la réalisation d'un "mapping" entre des modèles de la conception vers ceux des modèles logiques de bases de données. Par exemple à travers ces modèles logiques, l'outil doit pouvoir générer des schémas relationnels physiques pour

plusieurs systèmes de gestion de bases de données relationnels comme Oracle, Sybase, DB2, Informix et Microsoft SQL Server.

Enfin, l'analyse et la conception orientée objets sont supportées par plusieurs méthodes : OMT, Fusion, Booch, OOCL, Martin et Odell, Shlaer et Mellor, Coad et Yourdon et UML. L'outil doit permettre, à travers une technique d'ingénierie par "aller-retour", un développement itératif de la conception vers l'implantation (génération des définitions de classes) et vice-versa pour les langages C++, Visual Basic, Smalltalk, Powerbuilder, PLATINUM Object Pro et Forte ; la génération simple de code doit être possible pour Java, Delphi et ADA. Avec un langage de script appelé "Photo Script", l'utilisation de l'outil comme méta-AGL est possible ainsi que la capacité d'intégration de Paradigm Plus avec d'autres outils.

Paradigm Plus est disponible pour les plates-formes Unix, OS2 et Windows ; il est produit et distribué par la Société PLATINUM Technology Inc.

2.1.3.2 GraphTalk

L'outil GraphTalk [Par93b] a été l'un des premiers outils de méta-modélisation sur le marché. Il est construit sur des méta-modèles et peut être utilisé avec d'autres outils à travers des "démons" écrits en C/C++. Une description plus détaillée est donnée dans la section 5.1.1.

GraphTalk est disponible pour les plates-formes Unix et Windows. Il a été conçu par la société Rank Xerox et est actuellement commercialisé par la société Continuum .

2.1.3.3 Hardy

L'outil Hardy [Sma96] est un outil pour créer des diagrammes. Son fonctionnement est basé sur une structure d'hyper-texte. Il intègre le système CLIPS 6.0 [Ril95], un environnement pour le développement de systèmes experts qui utilise un langage orienté objet basé sur des règles et qui a été développé par la NASA. L'utilisation d'un paradigme hyper-texte permet la construction de hiérarchies où les composants peuvent être des diagrammes, du texte ou simplement des liens. Un éditeur de diagrammes basé sur la structure d'hyper-texte fournit à Hardy sa caractéristique de méta-AGL.

Une bibliothèque des fonctions CLIPS offre soit des facilités pour la création des diagrammes, soit des facilités pour l'adaptation de l'interface utilisateur à des besoins particuliers. L'utilisation de cette même bibliothèque permet à Hardy de fournir des fonctions pour le traitement des événements du système, comme par exemple ceux produits par l'utilisation de la souris.

Hardy est disponible pour les plates-formes Unix et Windows. Il est produit et développé par l'Institut des Applications de l'Intelligence Artificielle de l'Université d'Edimburgh. Pour des finalités de recherche, la licence d'utilisation est gratuite.

2.1.3.4 MetaView

L'outil MetaView [Fin94] est un outil pour la conception d'AGL présentant quelques caractéristiques particulières : il peut créer et manipuler des modèles de méthodes en les combinant facilement ; il peut aussi définir des contraintes qui portent sur ces modèles de manière à pouvoir vérifier leur cohérence et complétude.

Les modèles d'une méthode, appelés Modèles de Description de Logiciels ("Software Description Models" - SDM) sont décrits à travers un méta-modèle décrit à son tour avec des diagrammes EARA/GE ("Entity - Aggregate - Relationship - Attribute with Graphical Extension). Ces diagrammes sont stockés dans un dictionnaire de données et ils peuvent être décrits avec un langage particulier appelé Langage de Définition d'Environnements ("Environments Definition Langage" - EDL).

Sur les modèles d'une méthode, l'outil permet d'appliquer des contraintes (des prédicats logiques). Ces contraintes peuvent être utilisées soit pour maintenir la cohérence des données du dictionnaire de données, soit pour assurer qu'une spécification de logiciel est complète par rapport à une méthode donnée.

MetaView est un outil qui n'est pas encore disponible en raison de son état actuel de développement. Il est développé par les groupes des Départements d'Informatique de l'Université d'Alberta et de l'Université de Saskatchewan au Canada.

2.1.3.5 Rational Rose/C++

L'outil Rational Rose/C++ est un AGL dédié à deux méthodes orientées objets spécifiques : OOD et OMT. Il s'appuie naturellement sur la modélisation orientée objet et sur une technique appelée Développement Itératif Contrôlé.

L'outil supporte tous les concepts proposés par la méthode Booch. La méthode OMT a un support pour ses trois modèles : objet, dynamique et fonctionnel, bien que le modèle fonctionnel soit représenté avec des scénarios et non avec la proposition originale de Rumbaugh. L'outil permet quelques vérifications sémantiques des modèles créés. L'implantation est dirigée vers le langage C++, mais il existe d'autres ateliers dirigés vers d'autres langages.

La technique de Développement Itératif Contrôlé est censée être, à travers une séquence d'itérations, un meilleur modèle de développement que celui de la cascade. A chaque itération les fonctionnalités du logiciel doivent s'accroître pour progresser vers la satisfaction des besoins spécifiés au départ ; à chaque itération, un "risque critique" est défini et un développement de l'analyse à la production d'une architecture initiale est réalisé en employant des tests, de manière à ce que le problème défini par le risque critique soit résolu.

L'outil permet l'utilisation d'une technique d'ingénierie par des "allers-retours" ce qui facilite le processus itératif de développement entre l'analyse et l'implantation avec le processus de ré-ingénierie annexe ; son utilisation pour la rétro conception est aussi possible. La personnalisation de l'interface homme-machine est possible ainsi que l'utilisation dans un environnement de développement parallèle multi-utilisateurs. Par rapport à ce dernier point, l'atelier offre plusieurs techniques pour la gestion des utilisateurs.

Rational Rose/C++ est disponible pour les plates-formes Unix et Windows. Il est produit et commercialisé par la société Rational Software Corporation.

2.1.3.6 ObjectTeam/OMT

L'outil ObjectTeam/OMT est un AGL dédié à la méthode OMT et basé sur un dictionnaire de données en ligne, lequel rend possible l'utilisation simultanée de l'atelier par plusieurs utilisateurs ; il fournit plusieurs techniques pour la gestion de ces utilisateurs. Une autre caractéristique est son système de versions intégré au dictionnaire de données. ObjectTeam/OMT fait partie d'un outil plus général appelé "ObjectTeam Enterprise" qui l'utilise comme outil de modélisation OMT pour de gros projets dans une entreprise ; il peut cependant être utilisé tout seul.

L'atelier prend en compte les trois modèles de la méthode OMT et en introduit d'autres. Le Modèle d'Objets est pris en compte avec des diagrammes de classes. Le Modèle Dynamique est pris en compte par des traceurs d'événements et par des diagrammes de transition d'états. Le Modèle Fonctionnel est pris en compte par des diagrammes de flux de données. Deux autres modèles sont intégrés à ces trois premiers : le Modèle de Communication de Classes utilisé pour représenter les échanges de messages entre les classes (diagramme de généralisation de messages) et pour permettre le groupement de plusieurs classes en sous systèmes (diagramme de communication de classes), et les Scénarios de Cas d'Utilisation qui, avec des traceurs d'événements et des diagrammes de communication de classes, sont utilisés pour modéliser les acteurs et les itérations dans la totalité d'un système.

L'outil offre la possibilité de génération de code en C++ ou Smalltalk, de réutilisation et de rétro conception de ce code. Il existe des facilités pour la définition de la documentation (générée par l'atelier) et pour la définition de l'interface homme-machine. L'intégration avec d'autres outils est possible à travers le dictionnaire de données.

ObjectTeam/OMT est disponible pour les plates-formes Unix et Windows. Il est produit et commercialisé par la société Cayenne Software Inc, fusion de la société Cadre Technology et de celle de B. Bachmann.

2.1.3.7 Object Domain

L'AGL Object Domain est dédié à l'analyse et à la conception orientée objet de la méthode Booch ; il fournit aussi la possibilité d'utilisation d'une partie de la méthode OMT. Par rapport à la méthode de Booch, l'outil permet la manipulation des six diagrammes de la méthode : diagramme de classes, d'objets, d'itération, diagrammes d'états avec des états emboîtés, diagramme de modules et diagrammes de processus. L'utilisation de la méthode OMT est restreinte aux diagrammes de classes et de transitions d'états.

L'outil possède un analyseur C++ qui peut réaliser de la rétro conception de code vers des classes. Avec un langage de scripts, les modèles objets peuvent être manipulés, la documentation peut être générée et le dictionnaire de données peut être utilisé hors du contexte de l'atelier.

Object Domain est un "shareware" disponible uniquement pour la plate-forme Windows. Il est développé et commercialisé par la société Object Domain Systems.

2.1.3.8 TCM

L'outil TCM ("Toolkit for Conceptual Modeling") n'est pas un AGL classique. Il ne s'appuie pas sur une méthode particulière mais fournit plutôt des outils pour représenter le modèle conceptuel d'un système logiciel à travers des diagrammes, tables et arbres. L'outil TCM sert aux activités de spécification et maintien des besoins d'un système.

Il offre des éditeurs pour la création de plusieurs types de représentation ; chaque éditeur est composé d'outils pour la création d'éléments spécifiques. La table 2.3 présente ces éditeurs ainsi que les éléments que les outils peuvent créer. La vérification des contraintes simples (ex : la duplication de noms) sur un seul document est possible, mais il est impossible de traiter des contraintes entre documents.

Éditeur	Outil
Diagramme Générique	TGD : Diagrammes Génériques
Vue de Données	TERD : Diagrammes Entités-Relations TCRD : Diagrammes de Classes-Relations
Vue Comportementale	TSTD : Diagrammes de Transitions d"États TRTD : : Graphes de Processus Récursifs TPSD : Diagrammes de Structures de Processus
Vue Fonctionnelle	TDFD : Diagrammes de Flux de Données TDCFD : Diagrammes de Flux de Contrôle et Données TSND : Diagrammes de Systèmes de Réseaux
Tables	TGT : Tables Génériques TTDT Tables de Décomposition de Transaction TTUT : Tables d'Utilisation de Transactions TFET : Tables Entité Type - Fonction
Arbres	TGTT : Arbres Textuels Génériques TFDT : Arbres de Décomposition de Fonctions

Table 2.3 - Éditeurs et Outils de TCM

Une caractéristique particulière de cet outil est la possibilité de création de diagrammes, tables et arbres génériques à travers les outils TGD, TGT et TGTT. En plus d'éléments préétablis, on peut en définir d'autres adaptés à des besoins particuliers. De cette manière avec une composition particulière d'éléments créés par les outils, on peut adapter TCM à une méthode spécifique. L'outil ne traite cependant pas les problèmes relatifs à la cohérence et à la complétude d'une modélisation par rapport à une méthode.

TCM est actuellement en développement à la Faculté de Mathématique et Informatique de l'Université d'Amsterdam. Il est disponible pour la plate-forme Unix ; c'est un logiciel gratuit.

La conduite de plusieurs étapes du cycle de vie, qui peut être faite soit d'une manière automatique avec des AGL, soit manuellement, est guidée par une ou plusieurs méthodes qui utilisent des modèles pour modéliser les systèmes. Les méthodes, leurs langages et les utilisations sont traités dans la section suivante.

2.1.4 Méthodes, Langages et Modélisation

Dans la section 2.1.1, les méthodes ont été introduites comme “composants” du génie logiciel. Dans cette section, les méthodes et les langages qui leur sont attachés ainsi que leur utilisation et le processus de modélisation sont détaillés.

Selon Rumbaugh [Rum95d], une méthode est composée de directives et de règles et a les composants suivants :

- un ensemble de concepts fondamentaux de modélisation pour capturer la connaissance sémantique d’un problème et de sa solution ;
- un ensemble de vues et de notations pour présenter la modélisation sous-jacente aux personnes qui les examineront et les modifieront ;
- un processus interactif pas-à-pas employé pour la construction des modèles et pour leur implantation ;
- une collection de suggestions et de règles qui conduisent à l’exécution du développement ; le concept de *méta-schémas* (“patterns”) est une tentative pour la représentation de l’expérience personnelle des développeurs d’une manière uniforme.

Les méthodes peuvent être classées chronologiquement dans quatre catégories par rapport à leur utilisation dans la modélisation de SI [Gir95] :

1. les *méthodes d’analyse* proposées dans les années 60 (CORIG [Mal71], etc.) qui ont démarré l’approche industrielle de l’informatisation ;
2. les *méthodes cartésiennes* (HIPO [Cor74], SA [De 78], SSA [GS79], SA/SD [YC79], SADT [RS77], etc.) sont la continuation des premières méthodes d’analyse associées à une approche fonctionnelle du SI par rapport aux opérations qui doivent être décomposées et à l’organisation du processus par étapes ; la dernière étape (l’implantation) est basée sur les règles de la programmation structurée ;
3. les *méthodes systémiques* (Merise [TRR83], IDA [BP83], Remora [RFB88], etc.) sont apparues dans les années 80 et marquent la rupture avec les méthodes antérieures pour privilégier une approche conceptuelle globale du SI basée sur une recherche des éléments pertinents du SI et de leurs relations ;
4. les *méthodes objets* (OOD [Boo94], OOA [CY91a], OOAD [MO95], O* [RLB92], OMT [RBP+91], Fusion [CAB+93], etc.) centrées sur une synthèse des méthodes

systémiques et cartésiennes combinent des spécifications détaillées basées sur la notion d'objet avec des spécifications globales basées sur les relations statiques et dynamiques entre les objets.

Les méthodes limitées à la structuration et à la construction de logiciels pourraient aussi être regroupées historiquement dans d'autres catégories peut être plus adaptées : *méthodes fonctionnelles* SA/SD [YC79], SADT [RS77], etc.), *méthodes structurées* (Merise [TRR83], JSD/JSP [Jac83], etc.), *méthodes modulaires* (HIPO [Cor74], etc.) et *méthodes objets* (OOD [Boo94], OMT [RBP+91], etc.). Naturellement, ce genre de classification est toujours délicat et ne sert que de point de repère général.

Par rapport à la notion de modèle avec l'optique du mathématicien, nous pouvons dire qu'un modèle est la représentation formelle d'un système avec un niveau de détail quelconque. Cette représentation est construite avec une collection de *concepts de modélisation* (par exemple pour OMT : classes, associations, états, etc.) [Rum95d]. Ce processus de construction d'un modèle constitue la *modélisation*.

La modélisation d'un SI est exprimée à l'aide d'un langage (ou notation) avec lequel on construit, examine et manipule les modèles d'un SI.

Un langage est un ensemble de symboles ou de signes permettant la communication. On peut distinguer les langages selon les trois formalismes suivants : informel, semi-formel et formel [FKV94]. Une autre classification est donnée par G. Winskel [Win93] : elle est basée sur la sémantique des langages.

Pour des raisons pratiques et d'usage, dans le cadre de la modélisation des SI, la classification et la combinaison de documents exprimés selon des langages, tantôt en langue naturelle (informel), tantôt graphiques (semi-formel) et/ou formels, sont préférables à l'intégration d'approches dénotationelles opérationnelles et/ou axiomatiques limitées essentiellement aux langages formels.

La table 2.4 inspirée de M. D. Fraser [FKV94] montre une définition des catégories de langages ainsi que des exemples de langages ou de méthodes utilisés.

Catégories de Langages			
Informel		Semi-Formel	Formel
“Simple”	Standardisé		
Langage qui n’a pas un ensemble complet de règles pour restreindre une construction.	Langage avec une structure, un format et des règles pour la composition d’une construction.	Langage qui a une syntaxe définie pour spécifier les conditions sur lesquelles les constructions sont permises.	Langage qui possède une syntaxe et une sémantique définies rigoureusement. Il existe un modèle théorique qui peut être utilisé pour valider une construction.
Exemples de Langages ou Méthodes			
Langage Naturel.	Texte Structuré en Langage Naturel.	Diagramme Entité-Relation, Diagramme SADT ou JSD.	Réseaux de Petri, Machines à états finis, VDM, Z.

Table 2.4 - Classification et Utilisation de Langages ou de Méthodes

Par rapport au langage ou à la méthode utilisé dans une modélisation, on classe cette modélisation comme étant formelle, semi-formelle ou informelle. Les sections suivantes traitent ces points.

2.2 Modélisation Informelle

Le processus de modélisation informelle utilise comme outil de modélisation un langage informel (le langage naturel) dans la construction d’une modélisation sans l’application de règles strictes. Ce genre de modélisation peut être justifié par au moins deux raisons [BGW78] :

– elle est concise, et par sa facilité de compréhension, elle permet des accords entre les personnes qui spécifient et celles qui commandent un logiciel ;

– elle représente une manière familière de communication entre les gens.

D’un autre côté, l’utilisation d’un langage informel rend la modélisation imprécise et parfois ambiguë. De plus, comme le raisonnement humain est le facteur principal pour l’analyse et la validation de la spécification [FKV91] il peut conduire à des erreurs de compréhension et de vérification.

Une solution qui peut être employée pour diminuer les erreurs est l’utilisation d’une *Modélisation Informelle Standardisée*, c’est-à-dire une modélisation qui utilise un langage

naturel, tout en introduisant des règles pour l'utilisation de ce langage dans la construction de la modélisation. Un tel type de modélisation garde les avantages de la modélisation informelle en la rendant moins imprécise et moins ambiguë.

La figure 2.5 empruntée à [Flu95] présente une modélisation informelle qui utilise un langage naturel, le français, et une structure qui rend cette spécification standardisée.

```
# Spécification en français structuré d'une procédure de recherche d'un livre

Procédure de Recherche
1. Obtenir des détails sur le livre
2. Lire les détails du 1er livre du catalogue (livre de test)
3. Comparer les détails du livre recherché avec les détails du livre de test
      3.1 Tous les détails concordent : le livre est trouvé
      3.2 Tous les détails concordent : mais le livre est emprunté
      3.3 Les détails diffèrent : lire les détails du prochain livre
4. Recommencer en 3 jusqu'à obtenir le résultat "le livre est trouvé"
                                    ou le résultat "le livre est emprunté"
                                    ou "la fin du catalogue est atteinte"
5. Indiquer le résultat de la recherche
      5.1 "le livre est trouvé" : indiquer où il se trouve
      5.2 "le livre est emprunté" : indiquer la date de son retour
      5.3 "la fin du catalogue est atteinte" : livre non disponible
```

Figure 2.5 - Spécification informelle en français structuré [Flu95]

Si on peut créer un ensemble de règles qui, à travers un texte structuré, induit la construction de la spécification, on peut dire qu'on utilise un *canevas de modélisation*. Celui-ci peut être employé comme une structure sur laquelle la modélisation sera construite ; cette structure peut même être composée d'éléments autres que le texte. Dans notre travail, nous proposons un canevas (cf. section 4.6) comme structure de modélisation intégrant des liens hyper-textes. La section suivante introduit le concept d'hyper-texte et ses possibilités d'utilisation.

2.2.1 Documents Hyper-textes

Les concepts généraux des systèmes hyper-textes ont été proposés pour la première fois par V. Bush en 1945 [Bus45] avec un système appelé Menex. Pendant presque vingt ans, ces systèmes ont attendu le développement des machines pour pouvoir se développer [Nie90]. Cependant, en 1962, Doug Engelbart a démarré un projet appelé “Augment” qui visait à accroître la productivité et la capacité de travail des personnes. Le “Système NLS” (“oN-Line Systems”) qui avait plusieurs caractéristiques d’un système hyper-texte, constituait une partie de ce projet.

Le terme “hyper-texte” a été proposé par T. Nelson en 1965 dans le cadre du projet Xanadu qui visait à englober la totalité des textes écrits dans le monde. Une partie de ce système a été implantée par la société Autodesk, mais le projet a été abandonné en 1992 [1]. En 1967 le premier système hyper-texte a été construit et mis en marche ; ce système s’appelait “Hypertext Editing System” et il a été construit à l’Université de Brown sous la direction d’Andries van Dam.

Le premier système hyper-texte, avec des éléments non textuels, est probablement le système “Aspen Movie Map” développé en 1978 par Andrew Lippman et d’autres chercheurs dans le “MIT Architecture Machine Groupe” pour simuler une visite de la ville d’Aspen au Colorado.

Ensuite la technologie a évolué jusqu’à connaître l’explosion actuelle, grâce surtout à l’Internet et au langage HTML qui en est actuellement à sa version 3.2 [Rag96].

On peut définir maintenant plus précisément un hyper-texte. Un hyper-texte est “une manière non linéaire de visualisation de l’information” [SK89]. On peut voir dans la figure 2.6 qu’un lecteur peut suivre dans un système hyper-texte (les flèches noires) un ordre non séquentiel par opposition à l’ordre séquentiel imposé par les documents textuels traditionnels (les flèches grises).

[1] Ted Nelson est actuellement au Japon où il travaille avec le professeur Yuzuru Tanaka à l’Université d’Hokkaido dans le développement d’une nouvelle version de Xanadu.

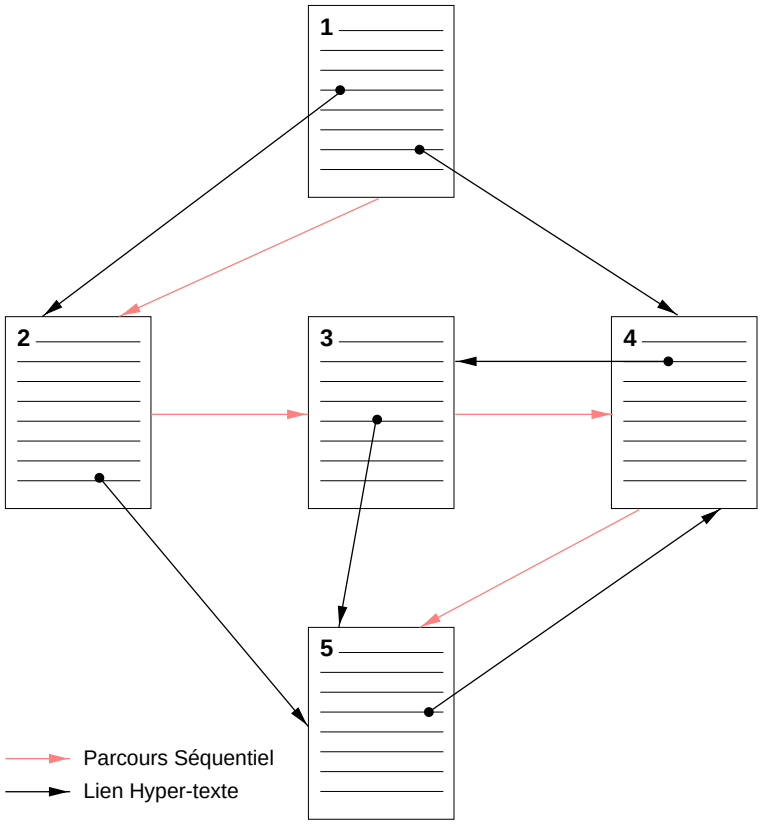


Figure 2.6 - Documents Hyper-textes

Un système hyper-texte est composé d'un ensemble de "pièces de texte" ; chaque unité d'information qui compose une pièce de texte est appelée un *nœud* (les ronds dans les feuilles de la figure 2.6) et chaque pointeur entre les nœuds est appelé un *lien* (les flèches noires de la figure 2.6) ; cet ensemble de nœuds et de liens forment un réseau. Le nœud de départ d'un lien est une *ancre*, qui à travers un lien pointe vers le *nœud destination*. Plusieurs systèmes hyper-textes implantent aussi la facilité de "marche arrière" ("backtrack") qui permet un retour dans un arbre de liens [Nie90].

Lorsque l'information contenue dans les nœuds n'est pas uniquement du texte, le terme *hyper-média* peut être employé ; cependant quelques auteurs (ex. J. Nielsen [Nie90]) ne différencient pas les deux termes. Nous utilisons le terme hyper-texte.

Dans le cadre du travail présenté dans cette thèse, les systèmes hyper-textes sont intéressants par leur possibilité d'être utilisés pour lier différents types de modélisation, en réalisant la connexion entre un document de spécification, le canevas et les différentes modélisations (cf. figure 2.7).

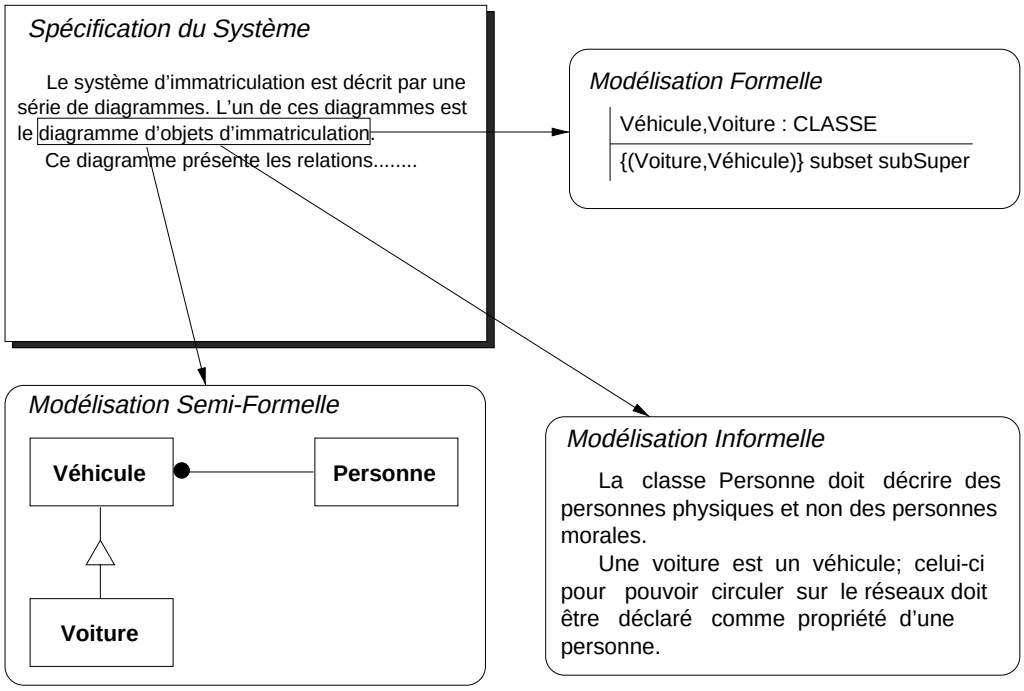


Figure 2.7 - Modélisation avec un Document Hyper-texte

Cette modélisation basée sur un canevas peut être implantée avec des logiciels qui ont soit des caractéristiques hyper-textes, soit des caractéristiques de structuration. Les logiciels LEdit, Thot et Intermedia en sont des exemples et sont présentés dans les sections suivantes. Ces logiciels peuvent être utilisés pour implanter le canevas de modélisation et, à partir de celui-ci, faire référence aux différents types de modélisation différents à travers une utilisation conjointe des caractéristiques de textes structurés et d'hyper-textes.

2.2.1.1 LEdit

L'éditeur LEdit [Par94] développé par la société Parallax est un constructeur d'éditeur guidé par une syntaxe. Les documents créés avec LEdit respectent une syntaxe pré-définie en BNF. La structure définie ne peut avoir comme composants que des éléments textuels qui respectent cette syntaxe rigide. Son utilisation comme hôte pour un système d'édition basé sur un canevas est limitée ; cependant son utilisation comme éditeur dirigé vers une grammaire particulière, comme par exemple la grammaire d'un langage formel, est possible et d'implantation facile. Une description plus détaillée est donnée dans la section 5.1.2.

2.2.1.2 Thot

L'éditeur Thot, développé dans le cadre du projet Opéra de l'unité de recherche INRIA Rhône-Alpes, permet la création, la modification et la consultation de façon interactive de documents qui respectent des modèles. Dans Thot, un document est représenté par sa *structure logique* dont les éléments forment des structures hiérarchiques qui rendent compte des relations d'inclusion et d'ordre entre les éléments. Cette structure logique est contrainte par un schéma de structure, qui spécifie principalement les types des éléments utilisables et les relations qui peuvent les relier. Cette structure est construite par l'éditeur Thot, sous le contrôle de l'utilisateur [Opé96].

L'éditeur Thot peut échanger des documents avec d'autres systèmes par l'intermédiaire d'un outil d'exportation paramétrable, ce qui le caractérise comme un système ouvert qui peut s'intégrer dans d'autres applications à travers son interface de programmation et son mécanisme d'appels externes [Opé96]. Une description plus détaillée est donnée dans la section 5.1.3.

L'utilisation de Thot comme hôte pour un système d'édition basé sur un canevas est possible à travers la combinaison d'un système structuré et de liens hyper-textes. Cette utilisation conjointe a déjà été étudiée pour Thot [QV92] et sera traitée dans la section 5.1.3.

2.2.1.3 Intermedia

Le logiciel Intermedia [Mey86] est un système de réseaux hyper-textes développé à l'Université de Brown. Il a été utilisé à l'origine dans l'enseignement.

Son utilisation permet la création de documents hyper-médias à travers la création de liens entre les documents qui doivent composer le document principal. Intermedia utilise la programmation orientée objet ainsi qu'un modèle client-serveur basé sur le système de permissions Unix pour permettre la création et la manipulation de liens hyper-textes.

La création de liens est basée sur un browser graphique à travers lequel est construit un document. Il offre un ensemble d'éditeurs pour la manipulation d'informations de différents formats (texte, graphique, animation et donnée) originaires d'un vidéo disque. Il offre aussi des outils pour le traitement linguistique du texte produit à travers une interface avec le "American Heritage Dictionary". Les liens hyper-textes peuvent être créés entre n'importe quel type de données traité par Intermedia. Un browser est aussi fourni pour la manipulation des informations sur les liens. Ces informations portent sur les ancres et les liens eux-mêmes et sont stockées dans un système de gestion de bases de données. La séparation entre les

données d'un document et les données des liens est une caractéristique de différenciation du produit [HKR+92].

Intermedia, par sa caractéristique "d'outil d'agrégation" de documents, ne se prête pas très bien à la fonction d'hôte pour un système d'édition de modélisations basé sur un canevas, à cause de l'absence d'une structure sur laquelle un document doit être construit. Cependant, ses caractéristiques, ses fonctionnalités d'éditeur hyper-texte et son interface sont remarquables.

2.3 Modélisation Semi-Formelle

Le processus de modélisation semi-formelle utilise comme outil de modélisation un langage textuel ou graphique pour lequel une syntaxe précise est définie ainsi qu'une sémantique ; cette sémantique assez faible permet une certaine dose de contrôle et l'automatisation de quelques tâches [And95].

La plupart des propositions semi-formelles s'appuient fortement sur un langage graphique. Cela peut se justifier par l'expressivité que peut avoir un modèle graphique bien développé ; le langage textuel est utilisé normalement comme appui aux modèles graphiques. La modélisation semi-formelle en utilisant un langage graphique peut produire des modèles de compréhension facile. Cependant, le manque de sémantique est un fort handicap pour ce genre de modélisation ; le problème existant pour les langages informels, de manque de précision par rapport à la compréhension de la modélisation et d'ambiguïté, persiste pour les langages semi-formels.

L'utilisation de contraintes dans les modèles graphiques vise à améliorer les problèmes cités ci-dessus par rapport aux modèles semi-formels. Un exemple d'utilisation de contraintes textuelles sur un modèle graphique concerne le travail de J. J. Odell [Ode93], où des contraintes structurelles sont définies sur des relations afin d'éclaircir leur sémantique dans le cadre d'une modélisation basée sur des modèles orientés objets. Une autre proposition est celle de S. Cook et J. Daniels [CD94a]. Dans ce travail, le diagramme d'objets de la méthode OMT [RBP+91] et les diagramme d'états de D. Harel [Har87] sont notés avec un sous ensemble du langage formel Z [Spi89] afin d'introduire des contraintes sur ces diagrammes.

Les modèles (semi-formels) orientés objets sont apparus pour "prendre la place" des modèles de l'analyse structurée qui ont vu leur utilisation se développer fortement avec les AGL de deuxième génération. À travers ces outils, plusieurs types de modélisation ont pu être utilisés de manière automatisée, ce qui en a développé leur utilisation.

Dans la section 2.1.4, les méthodes ont été classées en méthodes des années 60, méthodes cartésiennes, méthodes systémiques et méthodes objets. Ces méthodes orientées objets sont basées sur des langages semi-formels (ce qui leur donne une caractéristique de modélisation semi-formelle). Leur étude est reprise en détail au chapitre 3.

Approche Cartésienne

Le “Discours de la Méthode”, publié en 1637 par René Descartes, décrit les fondements du procédé de décomposition et pour cette raison les méthodes qui suivent une approche par décomposition fonctionnelle sont classées comme cartésiennes. Une méthode d’analyse suit une approche fonctionnelle quand son procédé d’analyse consiste à décomposer hiérarchiquement un problème à résoudre en fonctions jusqu’à l’obtention par affinements successifs de sous-fonctions suffisamment simples [Bru93].

La plupart des méthodes cartésiennes utilisent des modèles graphiques basés sur les flux de données ; ces sont les “Diagrammes de Flux de Données - DFD”. Un exemple est présenté dans la figure 2.8. Ce diagramme reprend l’exemple présenté avec un langage informel standardisé (cf. figure 2.5).

Les diagrammes de flux de données ont comme composants des *entités externes* qui sont des éléments externes au système et avec lesquels celui-ci communique, des *processus ou fonctions* qui représentent la partie du système qui transforme les entrées en sorties, des *répertoires de données* qui sont utilisés pour modéliser des données en attente et des *flux de données* qui sont utilisés pour représenter le mouvement des données dans le système. Les DFD peuvent être présentés avec plusieurs niveaux d’abstraction à travers la décomposition d’un processus selon un nouveau diagramme.

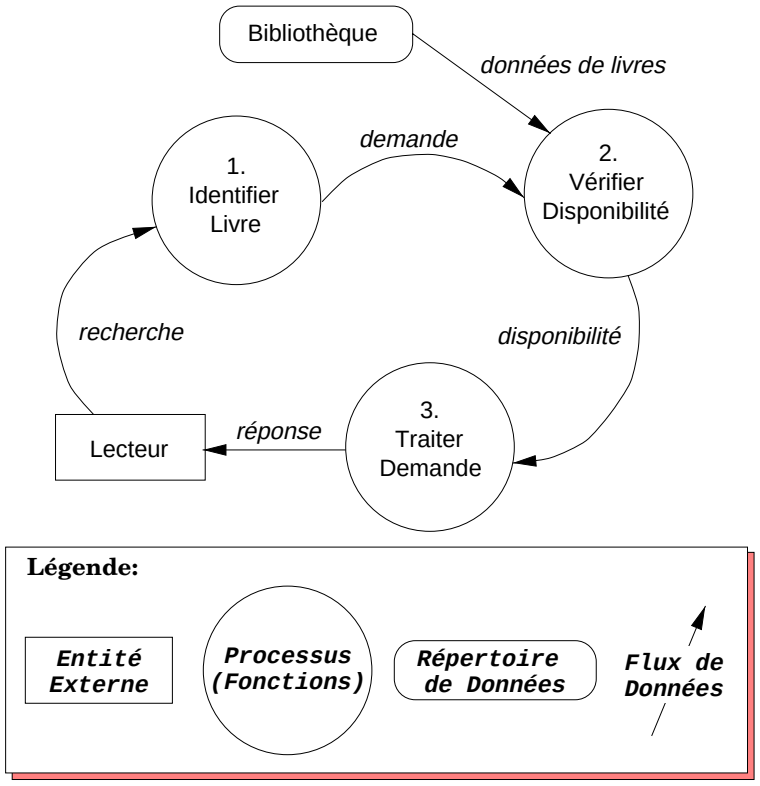


Figure 2.8 - Diagramme de Flux de Données

Approche Systémique

Les méthodes qui peuvent être classées comme systémiques sont basées sur la représentation de phénomènes pertinents de l'univers d'application à travers un modèle conceptuel. Elles appréhendent la réalité comme un ensemble d'entités qui ont des relations et des interactions entre elles et qui évoluent au cours du temps [Bru93].

Ces méthodes sont particulièrement adaptées à l'étude des systèmes vis-à-vis de leur environnement. Il est tout aussi intéressent de représenter les interactions entre composants d'un système, que les interactions entre ce système et son environnement. La plupart des méthodes systémiques utilisent pour la représentation du modèle conceptuel des modèles graphiques basés sur des "Modèles Entité-Relation - ER" [Che76]. La figure 2.9 présente le modèle ER conforme à l'exemple présenté en français structuré (cf. figure 2.5).

Les modèles ER ont comme composants des *types d'entités*, des *relations* et des *cardinalités*. Les types d'entités représentent des définitions en intention des ensembles d'entités du monde réel avec les caractéristiques suivantes : avoir une identification unique, accomplir un rôle dans le système en développement et pouvoir être décrits par une ou plusieurs valeurs. Les relations représentent un ensemble de connexions entre entités et peuvent être précisées par des cardinalités.

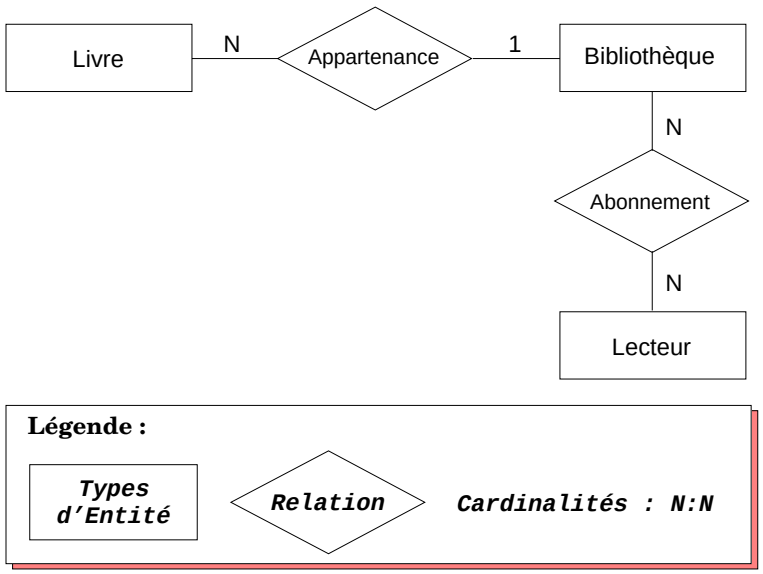


Figure 2.9 - Diagramme Entité-Relation

2.4 Modélisation Formelle

Une méthode formelle est un processus de développement rigoureux basé sur des notations formelles avec une sémantique précise ainsi que sur des vérifications formelles [Hab94]. Le principal avantage des spécifications formelles est leur capacité à exprimer une signification précise, en permettant de cette manière des vérifications de la cohérence et de la complétude d'un système [CD94b].

Ci-après, on montre deux tables qui font ressortir les caractéristiques des méthodes formelles dans le but d'en préciser les avantages et les inconvénients de leur utilisation. La première table (table 2.5) inspirée de l'article de A. Hall [Hal90a] décrit sept mythes des méthodes formelles. La deuxième table (table 2.6) décrit sept autres mythes à partir de l'article de J. P. Bowen et M. C. Hinchey [BH95].

Mythes		Commentaires
1	Les méthodes formelles peuvent assurer que le logiciel est parfait.	Il est impossible de faire une telle affirmation ; cependant l'utilisation des méthodes formelles est justifiée car elles offrent pour certains cas de meilleures garanties que n'importe quelle autre méthode et même s'il est possible de commettre des erreurs, ces erreurs sont plus facilement détectables grâce à ces méthodes.
2	Les méthodes formelles ne traitent que de la preuve de logiciels.	En réalité, les méthodes formelles traitent la spécification complète d'un système, car leur utilisation implique d'en écrire une spécification, de prouver des propriétés de cette spécification, d'implanter un logiciel à partir de cette spécification et de vérifier ce logiciel par rapport à la spécification.
3	Les méthodes formelles ne sont utilisées que pour les systèmes à sécurité critique ("safety-critical systems").	Les spécifications formelles peuvent aider à la spécification de n'importe quel type de système, mais son utilisation se justifie fortement lorsque le coût des échecs est élevé.
4	Les méthodes formelles nécessitent pour leur utilisation de mathématiciens de haut niveau.	L'utilisation des méthodes formelles porte plutôt sur l'écriture des spécifications, et dans ce cas, il faut "seulement" apprendre à utiliser le langage formel ; cependant la modélisation des choses du monde réel continue à être un sujet difficile. De plus, pour arriver à construire une modélisation, il faut entraîner les utilisateurs à l'utilisation de la méthode. Si un développement complet est requis, alors il faut que le développeur ait une bonne connaissance mathématique pour pouvoir réaliser les preuves.
5	Les méthodes formelles augmentent le coût du développement.	Lorsqu'une méthode formelle est utilisée, si la phase de spécification prend vraiment plus de temps et peut coûter effectivement plus cher, les bénéfices introduits par une spécification précise se font sentir lors des phases d'implantation, d'intégration et de tests de manière à ce que le coût global soit plus faibles.
6	Les méthodes formelles sont inacceptables pour les utilisateurs.	Par le fait que les méthodes formelles capturent ce que le système doit faire avant qu'il soit construit, elles aident l'utilisateur à savoir ce qu'il est en train de concevoir ; il faut cependant "traduire" ces spécifications soit en paraphrasant la spécification vers un langage naturel, soit à travers la démonstration de conséquences de la spécification, soit en animant celle-ci (par exemple avec un prototype).
7	Les méthodes formelles ne sont pas utilisées pour des systèmes réels de grande taille.	Elles sont utilisées dans des systèmes réels pour la spécification soit du logiciel, soit du matériel ; des exemples sont la spécification du matériel de l'usine nucléaire Sizewell-B en Grande-Bretagne, le système CICS d'IBM et les récents avions produits par Airbus.

Table 2.5 - Sept Premiers Mythes de la Spécification Formelle

Mythes		Commentaires
8	Les méthodes formelles provoquent des retards dans le processus de développement.	Bien que des projets ayant utilisé des techniques formelles aient présenté des retards, attribuer ce retard à l'utilisation des méthodes formelles n'est pas correct ; la cause est due plutôt au manque d'expérience à établir la durée de développement du projet, car il n'y a pas assez de données sur lesquelles on peut baser les estimations.
9	Les méthodes formelles manquent d'outils pour les supporter.	Si, par rapport aux outils CASE disponibles pour d'autres méthodes, cette affirmation peut être fondée, on ne peut pas tout de même nier l'existence d'outils de preuves pour quelques langages spécifiques existants ; il est vrai qu'un outil qui couvre toute la spécification n'existe pas, mais quelques uns montrent cependant la direction à suivre pour y arriver (ex : les outils B-Tool et Atelier-B).
10	Les méthodes formelles remplacent les méthodes traditionnelles de développement.	Si une méthode est vue comme un modèle de développement, des langages, des étapes bien définies ainsi que leur enchaînement, et un guide pour leur application, plusieurs méthodes formelles ne peuvent pas être appelées "méthodes", car elles font très peu attention au modèle de développement et au guidage de leur application ; une approche plus réaliste est l'utilisation conjointe de méthodes formelles et traditionnelles.
11	Les méthodes formelles ne s'appliquent qu'aux logiciels.	Les méthodes formelles sont appliquées aussi bien dans le développement du logiciel que du matériel ; un exemple est la vérification, à partir d'une spécification Z, de l'unité T800 (un transputer de point flottant).
12	Les méthodes formelles ne sont pas nécessaires.	Même si dans quelques spécifications, l'utilisation des méthodes formelles peut paraître peu intéressante, elles sont souhaitables, recommandées et même imposées, pour des systèmes où l'exactitude est très importante.
13	Les méthodes formelles n'ont pas de support dans la communauté informatique.	Aujourd'hui, le développement des méthodes formelles s'appuie sur le travail de plusieurs personnes, certaines même qui ne sont pas à l'origine de la méthode, ce qui montre leur diffusion ; d'autres points qui contredisent cette affirmation sont l'existence de plusieurs forums de discussion ainsi que l'intégration des méthodes formelles dans le curriculum universitaire.
14	Les personnes qui travaillent avec les méthodes formelles n'utilisent que des méthodes formelles.	De nombreuses personnes de la communauté formelle utilisent d'autres méthodes car elles reconnaissent que, pour certains systèmes, il est très difficile de les appliquer (ex : les interfaces homme-machine).

Table 2.6 - Sept autres Mythes de la Spécification Formelle

Ci-dessous, on propose un résumé des tables 2.5 et 2.6 par rapport aux mythes évoqués. Les méthodes formelles aident à trouver des erreurs et à diminuer leur incidence à travers une spécification complète qui peut être appliquée à n'importe quel type de système, logiciel ou matériel (mythes 1, 2, 3 et 11). Elles ne remplacent cependant pas des méthodes existantes et doivent être utilisées d'une manière conjointe (mythe 10). Si une connaissance mathématique solide est nécessaire pour réaliser des preuves, elle ne l'est pas pour spécifier (mythe 4). L'utilisation de méthodes formelles peut provoquer une diminution des coûts et elle n'est pas à l'origine des retards dans les projets (mythes 5 et 8).

Avec une traduction appropriée, les méthodes formelles peuvent aider dans la compréhension d'un système par un utilisateur (mythe 6) même pour de gros systèmes où elles sont aussi appliquées (mythe 7). En plus des outils de preuves déjà existants, les études pour la création d'autres outils qui couvrent toute la spécification, montrent l'intérêt de la communauté pour ces méthodes (mythes 9 et 13). L'utilisation des méthodes formelles, plus que souhaitable, commence à être imposée par des sociétés dans certains cas, même si elles ne sont pas utilisées dans tous les types de modélisation (mythes 12 et 14).

Dans le cadre de cette thèse, les commentaires portant sur les bénéfices introduits par l'utilisation des méthodes formelles, ainsi que la suggestion d'une utilisation conjointe avec d'autres méthodes sont plus particulièrement retenus.

La figure 2.10 empruntée à J. Flumet [Flu95] présente une modélisation formelle du même cas exemple déjà présenté d'une manière informelle dans la figure 2.5 et d'une manière semi-formelle dans les figures 2.8 et 2.9. Dans cet exemple, on utilise, pour la spécification, un langage mathématique avec des éléments de la théorie des ensembles et des éléments de la logique mélangés au texte en langage naturel. Le langage mathématique est le langage formel utilisé.

Plus précisément, on peut dire que le langage formel introduit une notation, son *domaine syntaxique*[2], un univers d'objets, son *domaine sémantique*[3] et une règle précise qui définit quels objets satisfont chaque spécification. Une spécification est une phrase écrite en terme des éléments du domaine syntaxique ; comme les langages de spécification ont des domaines syntaxiques et sémantiques différents, les méthodes formelles basées sur ces langages sont différentes [Win90].

[2]Le domaine syntaxique est défini en termes d'un ensemble de symboles (des constantes, des variables et des connecteurs logiques) et d'un ensemble grammatical de règles pour combiner ces symboles dans des phrases bien formées.

[3]Les exemples classiques de domaines sémantiques sont des algèbres, des séquences d'états et d'événements et des fonctions. Dans le premier cas, des langages de spécification basés sur des types abstraits de données sont utilisés, dans le deuxième cas, des langages pour des systèmes concurrents et distribués et dans le troisième cas des langages de programmation.

1. Un auteur *Aut*, un titre *Tit*, et le nom d'un usager *Emp* (qui emprunte un livre) sont formés par des lettres.
2. La date d'achat *Ach* et la date d'édition *Ed* du livre sont définies à l'aide d'une date.
3. Le numéro ISBN *Num* du livre est formé par des chiffres.
4. Un livre L est défini par un sextuplet :
   $L =< Aut_L, Tit_L, Num_L, Ach_L, Ed_L, Emp_L >$
5. Un catalogue C est définie comme un ensemble de livres.
6. Déterminer si un livre X est dans le catalogue s'écrit :
   $Si \exists L \in C, tq(Aut_l = Aut_X)$
   $et(Tit_L = Tit_X)$
   $et(Num_L = Num_X)$
   $et(Ed_L = Ed_X)$
   alors le livre existe.
7. Tester si le livre X est emprunté s'écrit :
   $Si(Emp_X \neq \text{ıȷ})$ alors le livre est emprunté.

Figure 2.10 - Spécification formelle en langage mathématique [Flu95]

Les méthodes de spécification formelle peuvent être classées selon trois catégories par rapport aux langages qu'elles utilisent [And95, Flu95] : (1) les spécifications algébriques ; (2) les spécifications orientées modèles ; (3) les spécifications hybrides. Ces trois catégories de spécifications sont présentées dans les sections suivantes avec des exemples. M. D. Fraser, K. Kumar et V. K. Vaishnavi [FKV94] introduisent une classification orthogonale à celle-ci : les méthodes utilisées pour spécifier les systèmes séquentiels et celles utilisées pour spécifier les systèmes concurrents. Ils introduisent de plus une classification selon le processus de formalisation qui peut être direct (transition directe informel/formel) ou transitionnel (transition informel/semi-formel/formel) et selon le support à la formalisation qui peut être assisté par ordinateur ou non.

Le grand nombre de méthodes de modélisation formelle existantes aujourd'hui rend impossible une présentation exhaustive de ces méthodes. Ci-après, en utilisant une table inspirée de [FKV94], on présente une compilation sommaire de méthodes formelles en les classant selon quelques unes des catégories introduites ci-dessus. Une liste plus complète des méthodes formelles est donnée par J. Bowen [Bow96a]. Trois méthodes (VDM, Z et B) sont détaillées dans les sections qui traitent de leur catégorie spécifique. Elles sont présentées pour des raisons différentes. Les deux premières car elles appartiennent à la catégorie qui présente le plus d'intérêt pour le travail de cette thèse : elles ont été choisies car elles sont les plus connues de cette catégorie et aussi parce que Z est utilisé explicitement dans notre expérimentation. La méthode B est décrite car elle introduit une approche combinée

qui supporte la totalité du cycle de vie d'un logiciel, ce que peut lui donner, selon J. P. Bowen et M. G. Hinchey [BH95], le statut de méthode formelle du futur.

Méthode	Description et Référence
VDM	La méthode VDM est une méthode de spécification orientée modèle utilisée pour spécifier des systèmes séquentiels. Elle intègre un ensemble de types de données à travers lesquels d'autres types peuvent être définis [Jon90].
Z	Z est un langage utilisé pour spécifier des systèmes séquentiels à travers des spécifications orientées modèles. Il est basé sur la théorie des ensembles et permet une modélisation à l'aide de structures appelées schémas [Spi89].
B	La méthode B est une méthode hybride développée par J. R. Abrial et d'autres ; elle est destinée à la modélisation de systèmes séquentiels, depuis la spécification jusqu'à l'implantation qui est basée sur des machines abstraites. Elle s'appuie sur l'outil B-Tool [Abr96] ainsi que sur l'outil Atelier-B.
Larch	Larch est un langage utilisé pour spécifier des systèmes séquentiels à travers des spécifications algébriques. Dans ce langage, il n'existe pas de types pré-définis ; Larch n'utilise pas non plus de symboles propres. L'utilisateur n'a pas besoin de connaître un vocabulaire spécial pour l'utiliser ; c'est lui qui introduit les symboles lorsqu'il définit des équations [GH83].
CSP	CSP ("Communicating Sequential Processes") est un langage utilisé pour spécifier des systèmes concurrents à travers des spécifications orientées modèles. Il a été développé à l'origine par C. A. R. Hoare. Avec CSP, l'information est transmise d'un processus à l'autre par un canal de communication. Cette communication s'établit si un processus est prêt à envoyer une information et si un autre est prêt à la recevoir sur le même canal. Il existe un outil pour la vérification de modèles qui a été développé par B. Roscoe : FDR [Hoa85].
Réseaux de Petri	Les Réseaux de Petri définissent un langage pour spécifier des systèmes concurrents avec une approche orientée modèle. Ils sont utilisés pour la spécification du temps et du parallélisme. Un réseau de Petri est basé sur deux objets fondamentaux, les places et les transitions. Les places peuvent contenir des marques dont l'évolution dans le réseau est décrite par les arcs qui lient les places et les transitions.
RAISE	La méthode RAISE ("Rigorous Approach to Industrial Software Engineering") est utilisée pour spécifier des systèmes concurrents en utilisant une approche hybride. Elle est composée d'une méthode de développement fondée sur un paradigme basé sur des raffinements successifs, et d'un langage de spécification appelé RSL qui a des caractéristiques inspirées des autres langages (VDM, CSP et ACT ONE). RAISE est supportée par un outil [HPP93].
LOTOS	LOTOS ("Language of Temporal Ordering Specifications") est une technique de spécification formelle qui utilise une approche hybride pour la spécification de systèmes concurrents ; elle comprend un langage pour la spécification de processus (similaire à CSP) et un langage de spécification algébrique appelé ACT ONE pour la spécification de données. LOTOS est supporté par des outils (simulation, exécution et vérification) et est devenu une norme ISO [GLO91] .

Table 2.7 - Méthodes Formelles

2.4.1 Spécifications Algébriques

Une spécification algébrique est une technique où un objet type est spécifié en termes de relations entre les opérations définies sur le type[4]. Les opérations sont définies à l'aide d'équations et peuvent être classées comme des *opérations de construction*, qui sont des opérations que créeent ou modifient des entités et comme des *opérations d'inspection*, qui déterminent des attributs d'une *sorte* (un ensemble d'objets) définie dans la spécification [Som92].

On peut dire qu'une spécification algébrique est composée d'une signature et d'un ensemble d'axiomes qui sont des formules logiques. La signature comprend les noms de domaines (les sortes) et les opérations définies pour leur profil. La signature permet la construction de toutes les valeurs des types de données à travers l'application des opérations [And95].

I. Sommerville [Som92] présente une notation (cf. figure 2.11.a) pour la représentation des spécifications algébriques. Cette notation est utilisée pour présenter un exemple d'utilisation de ce type de spécification (cf. figure 2.11.b).

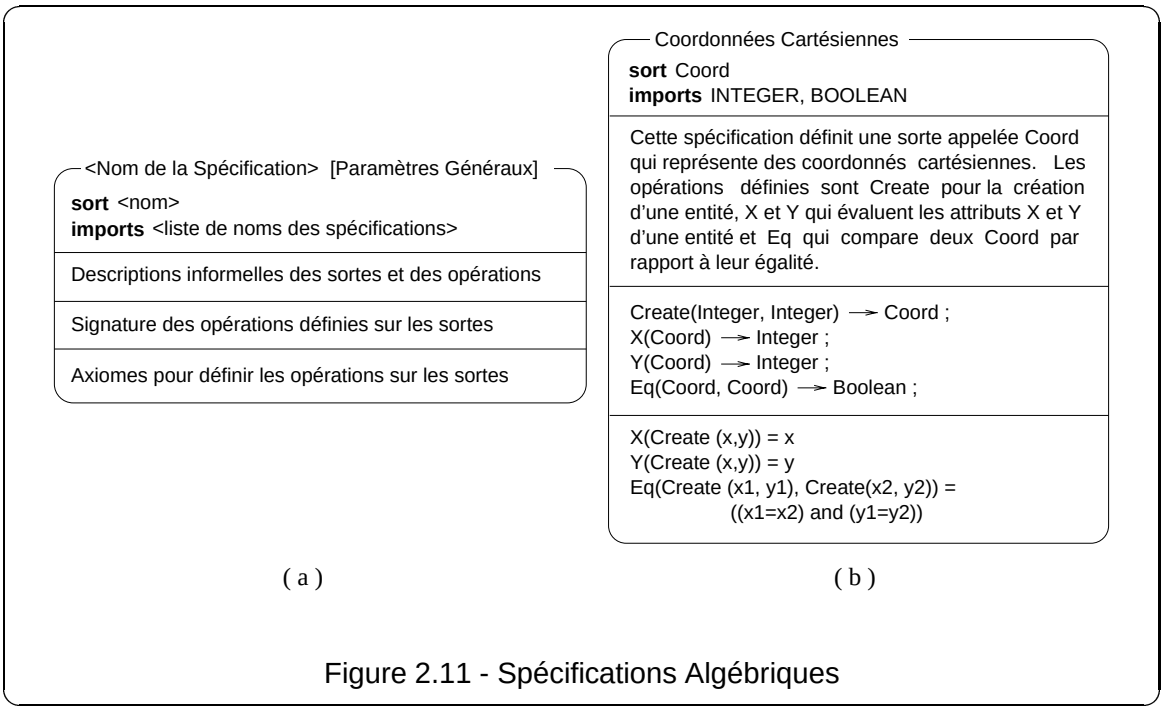


Figure 2.11 - Spécifications Algébriques

[4]dans ce texte, les objets types sont les types abstraits de données utilisés dans les langages de programmation ; on peut donc parler de Spécifications Algébriques de Types Abstraits de Données.

La figure 2.11.b montre la spécification d'un système de coordonnées cartésiennes avec des opérations pour la création, l'accès aux éléments X et Y, et la comparaison de coordonnées (cet exemple est emprunté à I. Sommerville [Som92]).

2.4.2 Spécifications Orientées Modèles

Une spécification orientée modèle est une méthode formelle qui repose sur la définition d'un modèle pour un système en utilisant des concepts mathématiques bien connus, comme les ensembles et les fonctions. La définition des opérations est basée sur la manière par laquelle elles affectent le modèle du système. Les spécifications orientées modèle sont riches en constructeurs et donnent des spécifications assez concises [Som92].

Les spécifications algébriques, par leur approche plus déclarative et abstraite que les spécifications orientées modèle, ne sont pas utilisées dans cette thèse dont le but est l'utilisation des langages formels pour la production de spécifications concises adaptés aux systèmes d'information. Dans ce contexte, les spécifications orientées modèle nous semblent plus appropriées. Les deux méthodes de spécification orientée modèle qui ont vu leur utilisation s'établir dans la communauté, sont les méthodes VDM et Z. Elles sont présentées respectivement dans les sections 2.4.2.1 et 2.4.2.2.

VDM, la méthode la plus ancienne, a été développé au laboratoire IBM de Vienne. La méthode Z a été développée à partir des travaux de J. R. Abrial. Une comparaison sommaire de ces deux méthodes montre que VDM insiste plus sur les fonctions et le λ-calcul alors que Z favorise les pré et post-conditions et la théorie des ensembles [And95]. Une comparaison plus approfondie est faite dans différents travaux [HJN93, Lin93]. D'autres comparaisons ont été réalisées entre méthodes : entre la méthode VDM et la méthode B [BR93] et entre les trois méthodes VDM, Z et B [BC94].

Dans cette thèse la méthode formelle utilisée est Z. Z et VDM ont toutes deux des caractéristiques intéressantes pour ce travail, mais le choix de Z est dû surtout à la plus grande lisibilité de ses spécifications grâce à l'utilisation de schémas, alors que VDM produit des spécifications textuelles structurées en modèles.

2.4.2.1 Méthode VDM

La méthode VDM ("Vienna Development Method") est une méthode de spécification formelle orientée modèle qui a ses origines dans les recherches réalisées au laboratoire IBM de Vienne sur les méthodes de spécification formelle dans les années 60.

VDM est composée d'un ensemble de techniques pour la spécification formelle des systèmes informatiques à travers un langage de spécification appelé VDM-SL ("VDM-Specification Language"). Elle introduit des règles pour le raffinement des données et des opérations ainsi qu'une théorie de preuve. Les raffinements établissent un lien entre la spécification des besoins et l'implantation. À travers la théorie de la preuve, les propriétés d'un système peuvent être vérifiées ainsi que la correction des décisions prises lors de la conception [Ver95].

Une spécification écrite avec le langage VDM-SL est un modèle mathématique construit avec des types de données simples comme les ensembles, les listes et les applications et avec des opérations qui peuvent changer l'état du modèle. Le langage VDM-SL est en cours de standardisation par l'ISO.

Dans VDM, les types de données définis sont les entiers, les booléens, les entiers relatifs, les réels et les chaînes de caractères, ainsi que les opérateurs qui leurs sont associés. Les autres types utilisés dans le langage doivent être composés à travers l'utilisation des constructeurs `ensemble`, `séquence`, `compose`, `produit` et `correspondance` : `compose` construit des types semblables aux structures du langage C, `produit` construit un type `compose` où les champs ne sont pas nommés et `correspondance` construit des types décrits par deux valeurs. Tous ces constructeurs ont des opérandes pour manipuler leurs éléments ainsi que des générateurs pour la création des éléments.

Avec VDM-SL, on peut aussi définir des `invariants des types` afin de restreindre leur ensemble de valeurs possibles. Cette restriction est réalisée soit à travers des fonctions booléennes, soit à travers des appels à d'autres fonctions, soit avec une combinaison des deux. Des états sont aussi définis avec VDM-SL. Un `état` est une collection de variables externes qui définissent la situation d'un système : un système démarre avec un état initial et change d'état par l'application d'opérations.

Les opérations dans VDM peuvent être définies de deux manières différentes, soit de manière *implicite*, soit de manière *directe* ou *explicite*. Avec le mode implicite, on définit ce que l'opération fait (le quoi) sans définir la façon selon laquelle elle est exécutée. Le mode direct définit le mode de calcul utilisé en donnant un algorithme. Les opérations définies implicitement sont appelées `fonction` si elles ne changent pas l'état du système.

Les spécifications VDM utilisent une approche basée sur des modules pour la structuration de la spécification. Les types de données, les invariants et les états sont organisés à l'intérieur d'un module composé de trois parties : une interface pour contrôler l'importation et l'exportation de définitions, un état et sa valeur initiale, ainsi que la définition des opérations. Des commentaires peuvent être introduits dans un module ; ils sont précédés par le caractère "–".

Cette section est une petite introduction à la méthode VDM ; pour une étude plus complète on peut voir [Jon90] d'où a été emprunté l'exemple présenté dans la figure 2.12 pour donner une idée du type de spécification produite avec VDM, et cela sans rentrer dans les détails des symboles utilisés. La figure 2.12 donne la spécification d'un *sac* en utilisant un module ; les mots en gras sont des mots clés du langage et ceux en italique sont des commentaires.

module *BAG*
parameters types X

– Description de l'Interface
exports
types
Bag
operations
$COUNT : X \xrightarrow{o} N$,
$ADD : X \xrightarrow{o}$

– Description des types et états
definitions
types
$Bag = X \xrightarrow{m} N_1$;
state
State **of** $b : Bag$
init $(mk\text{-}State(b_0)) \triangleq b_0 = \{\}$
end ;

– Desc. des fonctions et opérations
functions
$mpc : X \times Bag \to N$
$mpc(e, m) \triangleq$ **if** $e \in$ **dom** m
them $m(e)$ **else** 0

operations
$COUNT\,(e : X)\,c : N$
ext rd $b : Bag$
post $c = mpc(e, b)$

$ADD\,(e : X)$
ext wr $b : Bag$
post $b = \overleftarrow{b} \dagger \{e \mapsto mpc(e, \overleftarrow{b}) + 1\}$

end *BAG*

Figure 2.12 - Exemple de Spécification VDM [Jon90]

En regardant l'exemple de la figure 2.12, on peut dire que la notation utilisée par la méthode VDM est similaire à celle utilisée par la méthode Z (cf. section 2.4.2.2) avec une syntaxe additionnelle pour la description formelle de modules. La possibilité de définition d'une opération d'une manière directe est un avantage pour la description formelle d'algorithmes de logiciels.

La méthode VDM est supportée par une série d'outils. On peut citer par exemple "IFAD VDM-SL Toolbox" [IFA96] qui est composé d'un ensemble d'outils pour la vérification syntaxique et sémantique, l'analyse de tests, l'exécution et le débugage des spécifications VDM. Cet outil est développé par l'Institut d'Informatique Appliquée du Danemark. Un autre outil, du domaine public, est le parseur VDM [Rhi94] développé par l'Université Tech-

nologique de Braunsweis en Allemagne. Il existe aussi des styles LaTeX pour l'écriture de spécifications VDM. Pour une liste des outils ainsi que pour des références sur VDM on peut consulter l'article de P. G. Larson [Lar96].

2.4.2.2 Méthode Z

Z est un langage pour la spécification formelle de systèmes. Bien qu'il ne soit pas une méthode, on peut utiliser la notation Z pour produire des spécifications avec une approche orientée modèle. La notation Z a été introduite par J. R. Abrial à Grenoble, puis à EDF [Abr78] au cours des années 70, et développée par le "Programming Research Group" de l'Université d'Oxford pendant les années 1980-1985.

Le but principal recherché avec l'utilisation de Z est la spécification de systèmes. Cette spécification doit définir ce que le système doit faire sans dire comment le faire. Cependant des études[5] ont été réalisées pour raffiner des spécifications Z afin de produire, à partir d'un modèle, un autre modèle plus proche de l'implantation car Z n'introduit pas de notation pour décrire des algorithmes [Dil94]. Le but du travail présenté dans cette thèse étant la spécification de systèmes d'information, on ne s'intéresse pas à cet aspect algorithmique.

Une spécification Z traite un système comme des machines abstraites pour lesquelles l'*état interne* (un ensemble de variables) peut être modifié ou accédé seulement par les *opérations* associées à la machine [Led94]. La notation Z utilisée dans la spécification est basée sur la théorie des ensembles ainsi que sur la logique du premier ordre. Des notations sont proposées pour les différents termes d'une spécification : prédicats, ensembles, relations, fonctions, séquences, etc. Dans Z, il existe aussi une notation appelée Schéma qui permet la réutilisation du texte mathématique et la description à la fois des aspects statiques et dynamiques d'un système. Selon M. Spivey [Spi89], les aspects statiques comprennent les états que le système peut prendre et les relations d'invariance qui sont maintenues lorsque le système passe d'un état à un autre. Les aspects dynamiques décrivent les opérations potentielles, les relations liant les entrées et les sorties d'une machine abstraite et les modifications d'états se produisant.

Le langage Z est supporté par des outils pour la vérification de la syntaxe et des types des spécifications. Parmi ces outils on peut citer le "ZTC - Z Type Checker" développé

[5] Pour passer de la spécification à l'implantation, M. Spivey [Spi89] utilise une relation d'abstraction entre un état abstrait et un état concret du système ainsi que des raffinements (voir plus bas). S. King [Kin90] propose l'utilisation du calcul de raffinements introduit par C. C. Morgan ; dans le chapitre 7 du livre de J. B. Wordsworth [Wor92] l'utilisation des concepts présents dans le travail de Dijkstra est proposée et dans le chapitre 15 du livre de A. Diller [Dil94] l'utilisation des formules de la logique de Floyd-Hoare est introduite.

par X. Pia [Jia95] de l'Université DePaul et qui appartient au domaine public ainsi que le vérificateur de types "Fuzz" développé par M. Spivey. De plus, des polices pour l'écriture de spécifications Z sont disponibles pour Unix et Windows ainsi que des styles LaTeX. Pour une liste des outils ainsi que pour des références sur Z, on peut consulter l'article de J. Bowen [Bow96b].

Nous présentons la notation Z à travers un exemple de modélisation d'un agenda d'anniversaires tel qu'il est introduit par M. Spivey [Spi89]. La notation nécessaire à la compréhension de cet exemple est donnée à l'annexe F. Pour une description plus complète de Z, on peut consulter [Nic96] qui présente le langage Z tel qu'il a été proposé pour la standardisation à l'ISO.

Un Agenda des Anniversaires

Afin de manipuler les noms des personnes et les dates on doit définir des *types de base* pour le système :

$$[\mathit{NAME}, \mathit{DATE}]$$

Le premier schéma introduit décrit l'*espace d'état* du système.

BirthdayBook

$$\mathit{known} : \mathbb{P}\,\mathit{NAME}$$
$$\mathit{birthday} : \mathit{NAME} \pfun \mathit{DATE}$$

$$\mathit{know} = \mathrm{dom}\,\mathit{birthday}$$

Dans ce schéma, la *partie déclarative* (première partie d'un schéma) introduit l'ensemble des noms enregistrés (*known*) et une fonction partielle qui à partir d'un nom rend la date d'anniversaire correspondante (*birthday*). La deuxième partie d'un schéma, la *partie des prédicats* représente une relation vraie pour tout état du système et qui doit être maintenue à vraie quelle que soit l'opération appliquée. Cette relation est un *invariant du système*.

Après avoir défini l'espace d'état du système, il faut définir l'*état initial* pour celui-ci, ce qui est fait avec le schéma donné ci-dessous.

InitBirthdayBook

$$BirthdayBook$$

$$known = \varnothing$$

Ce schéma introduit un important concept du langage Z qui est l'inclusion de schémas. Ici, le schéma *BirthdayBook* est inclus dans le schéma *InitBirthdayBook* ce qui signifie que les variables et les prédicats définis dans *BirthdayBook* sont valables aussi dans *InitBirthdayBook* : comme *known* est vide, l'invariant $know = \mathrm{dom}\, birthday$ oblige que *birthday* soit également vide.

L'espace d'état et l'état initial étant définis, on peut passer à la définition des opérations. La première opération définie est l'ajout d'un nouvel anniversaire. Le schéma correspondant est donné ci-après.

AddBirthday

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$

$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

Avec ce schéma de nouvelles notations sont introduites. Ainsi un signe d'interrogation (" ?") suit une variable lorsqu'il s'agit d'une *variable d'entrée* : *name*? et *date*? vont recevoir le nom et la date dans l'opération. La notation $\Delta BirthdayBook$ veut dire que le schéma décrit une opération qui change l'état du système ; elle veut dire aussi que les variables du schéma *BirthdayBook* sont introduites dans le schéma *AddBirthday* avec la possibilité d'avoir des observations de l'état avant et après le changement. Cela introduit implicitement une autre notation : les variables qui représentent l'*état d'après* ont comme décoration un prime comme par exemple $birthday'$. La première ligne de la partie des prédicats introduit la *pré-condition*[6] pour le succès de l'opération à travers l'interdiction d'inclusion d'un nom connu du système. La deuxième ligne indique que lorsque la pré-condition est satisfaite, la nouvelle paire $name? \mapsto date?$ est ajoutée au système.

[6] M. Lemoine [Spi94] dans la traduction française du livre de Spivey, fait ressortir le fait que le terme pré-condition est une simple interprétation due au connecteur logique "et" qui lie les deux formules.

Le schéma ci-dessous introduit une opération pour trouver la date d'anniversaire d'une personne connue du système.

FindBirthday

$$\Xi BirthdayBook$$
$$name? : NAME$$
$$date! : DATE$$

$$name? \in known$$
$$date! = birthday(name?)$$

$\Xi BirthdayBook$ introduit les quatre variables (*name* et *date* avant et après) et indique qu'il s'agit d'une opération qui ne change pas l'état du système (implicitement on a $known' = known$ et $birthday' = birthday$). La variable *date* suffixée avec le signe d'exclamation (" !") indique qu'il s'agit d'une *variable de sortie* pour l'opération. La partie inférieure du schéma montre que *FindBirthday* prend un nom en entrée (*name*?) qui doit exister dans le système (la pré-condition) et fournit la date d'anniversaire correspondante comme sortie (*date*!).

Le schéma qui suit introduit l'opération la plus importante d'un système d'agenda des anniversaires : c'est l'opération qui fournit les personnes qui ont leur anniversaire à une date donnée.

Remind

$$\Xi BirthdayBook$$
$$today? : DATE$$
$$cards! : \mathbb{P}\, NAME$$

$$cards! = \{n : known \mid birthday(n) = today?\}$$

Ce schéma présente une opération qui ne change pas l'état du système (la convention Ξ). Dans ce schéma, il n'y a pas de pré-condition définie. Il donne comme sortie une variable (*cards*!) égale à l'ensemble de toutes les personnes qui ont leur anniversaire à la date *today*? ; pour cela le schéma utilise la définition d'ensemble en intention et la fonction *birthday* introduite avec $\Xi BirthdayBook$.

Le processus grâce auquel on passe des spécifications à l'implantation est appelé *raffinement*. Selon J. M. Spivey [Spi89], l'idée principale porte sur la description des données concrètes que le programme utilisera comme étant une représentation des données abstraites de la spécification, pour ensuite dériver les descriptions des opérations en terme de structures de données concrètes.

On présente ici le raffinement pour une partie de l'exemple développé (avec des structures de données abstraites) jusqu'à maintenant. L'agenda d'anniversaires est implanté à l'aide de deux tableaux déclarés par :

names : **array** $[1..]$ **of** *NAME* ;

dates : **array** $[1..]$ **of** *DATE* ;

Ces tableaux sont modélisés mathématiquement par les fonctions :

$$names : \mathbb{N}1 \rightarrow NAME$$
$$dates : \mathbb{N}1 \rightarrow DATE$$

Un élément du tableau défini par $names[i]$ correspond à la valeur $names(i)$ de la fonction, ce qui est déclaré de la façon suivante :

$$names' = names \oplus \{i \mapsto v\}$$ [7]

Un schéma pour décrire l'espace d'états avec des schémas concrets de données est développé ci-dessous. Ce schéma introduit les fonctions décrites ci-dessus ainsi que la variable *hwm* ("high water mark") pour contrôler le nombre d'éléments des tableaux. Le prédicat du schéma interdit la duplication d'éléments dans le tableau *names*.

*BirthdayBook*1

$$names : \mathbb{N}1 \rightarrow NAME$$
$$dates : \mathbb{N}1 \rightarrow DATE$$
$$hwm : \mathbb{N}1$$

$$\forall i, j : 1 \,..\, hwm \bullet$$
$$i \neq j \Rightarrow names(i) \neq names(j)$$

La prochaine étape vers l'implantation est l'établissement d'un pont entre les deux vues des états d'un système : la vue qui présente l'*espace d'états abstraits BirthdayBook* et celle qui présente l'*espace d'états concrets BirthdayBook*1. Ce pont est appelé *relation d'abstraction* par Spivey. Le schéma présenté ci-dessous introduit cette relation.

[7] L'opérateur $\oplus$ est l'opérateur *écrasement* de Z. Exemple d'utilisation : soient $f, g : \mathbb{N} \leftrightarrow \mathbb{N}$, $f == \{0 \mapsto 1, 1 \mapsto 2\}$ et $g == \{0 \mapsto 3, 2 \mapsto 4\}$; alors $f \oplus g = \{1 \mapsto 2, 0 \mapsto 3, 2 \mapsto 4\}$; la relation g "écrase" partiellement f : le couple $0 \mapsto 3$ de g "écrase" le couple $0 \mapsto 1$ de f.

Abs

$$
\begin{array}{|l}
\hline
BirthdayBook \\
BirthdayBook1 \\
\hline
known = \{i : 1 \mathinner{..} hwm \bullet names(i)\} \\
\forall\, i : 1 \mathinner{..} hwm \bullet \\
\quad birthday(names(i)) = dates(i) \\
\hline
\end{array}
$$

À travers le premier prédicat, une relation entre des personnes et des noms est établie. Le deuxième prédicat montre qu'une personne "*i*" qui s'appelle $name(i)$ a sa date d'anniversaire définie par $date(i)$.

Si on peut avoir différents états concrets pour le même état abstrait (cas de plusieurs implantations différentes) le contraire n'est pas vrai : un état concret ne représente qu'un état abstrait et un seul.

Après la définition de l'état concret, les opérations peuvent être définies par rapport à cet état. Ainsi l'opération *AddBirthday*1 est présentée ci-dessous.

*AddBirthday*1

$$
\begin{array}{|l}
\hline
\Delta BirthdayBook1 \\
name? : NAME \\
date? : DATE \\
\hline
\forall\, i : 1 \mathinner{..} hwm \bullet name? \neq names(i) \\
hwm' = hmw + 1 \\
names' = names \oplus \{hmw' \mapsto name?\} \\
dates' = dates \oplus \{hmw' \mapsto date?\} \\
\hline
\end{array}
$$

Ce schéma présente une implantation valide [Spi89] de *AddBirthday* pour deux raisons :

- lorsque l'opération *AddBirthday* est correcte dans un espace d'états abstraits quelconque, l'implantation *AddBirthday*1 est correcte dans l'état concret correspondant ;
- l'état final qui résulte de *AddBirthday*1 représente un état abstrait que *AddBirthday* pourrait produire.

Le schéma *AddBirthday*1 peut se traduire directement vers un langage de programmation :

procedure *AddBirthday*(*name* : *NAME* ; *date* : *DATE*) ;
begin
hwm : = *hwm* + 1 ;
names[*hwm*] : = *name* ;
dates[*hwm*] : = *date* ;
end ;

L'implantation d'autres opérations peut exiger la définition de nouveaux schémas pour de nouveaux états concrets ainsi que de nouvelles relations d'abstraction. Cependant la procédure de raffinement reste la même.

L'exemple ci-dessus donne une idée des spécifications produites avec Z. Quelques remarques peuvent être faites [Spi89] :

- les schémas permettent de décrire aussi bien l'état d'un système que les opérations qui peuvent être appliquées sur cet état ;
- les données du système sont manipulées à l'aide de types mathématiques classiques comme les ensembles et les fonctions ;
- les invariants d'états sont une information importante entre les états ;
- les opérations sont représentées comme des relations "entrée-sortie" ce qui n'impose pas une solution impérative d'implantation.

2.4.3 Spécifications Hybrides

Les spécifications hybrides ont des caractéristiques des spécifications algébriques et des spécifications orientées modèles. Des exemples de spécifications hybrides sont LOTOS [GLO91] et la méthode B. On présente B dans la section suivante.

2.4.3.1 Méthode B

B est une méthode de développement formelle développée à l'origine par J. R. Abrial avec la collaboration d'autres chercheurs du "Programming Research Group" de l'Université d'Oxford dans les années 80. Abrial travaillait alors pour British Petroleum dans un projet appelé "B-Technology".

La Méthode-B [Abr96] est composée d'une collection de techniques mathématiques utilisées pour la spécification, la conception et l'implantation, comme des machines abstraites, de logiciels critiques de sécurité. Elle comprend une notation AMN (Abstract Machine Notation) et la méthode, la notation AMN étant à la fois le langage de spécification et de conception.

Le processus de développement d'un logiciel avec la méthode B, de la spécification à l'implantation doit respecter les étapes suivantes [Lan96] :

1. *analyse des besoins* : un modèle d'analyse composé de modèles informels ou structurés doit être construit pour le système ;

2. *développement de la spécification* : une spécification formelle est produite. Pour y arriver, on doit (a) décomposer le modèle d'analyse en machines abstraites, (b) "animer" les machines abstraites pour vérifier la spécification par rapport aux besoins et à des scénarios de test et (c) générer et prouver les contraintes internes de cohérence ;

3. *conception* : un projet formel du système est généré, grâce à : (a) l'identification des composants décomposables de l'implantation (réutilisation ou bibliothèques), (b) le raffinement des composants choisis et (c) la génération et la preuve des contraintes de raffinement ;

4. *codification, intégration et tests* : le résultat de cette étape est le logiciel exécutable. Pour achever le processus, il faut d'abord (a) appliquer un générateur de code aux spécifications de plus bas niveau et (b) tester le résultat par rapport à des cas tests fondés sur les besoins initiaux.

Une analyse plus profonde du processus de développement présenté ci-dessus permet de dire que le concept le plus important du développement avec la méthode B est le *développement par couches* (cf. 3.a) où l'implantation d'un système est faite en utilisant d'autres spécifications. Quelques sous-étapes (ex : 2.b) doivent être conduites sous la direction d'un outil. Cela nécessite l'existence d'outils qui supportent la méthode : le "B-Toolkit" produit par la société B-Core qui permet l'animation des spécifications, la génération des contraintes de cohérence, le support de preuves et la génération du code ; l'autre outil "Atelier B" produit par la société Stercia Méditerranée en collaboration avec J. R. Abrial, est semblable à "B-Toolkit", mais sans l'animation [Lan96].

Les spécifications B sont construites sur la notation AMN qui a ses origines dans la théorie des ensembles utilisée dans Z mais elles utilisent des spécifications complètes (données

et opérations qui portent sur ces données) basées sur des éléments mathématiques élémentaires (logique des prédicats, ensembles, séquences, fonctions, etc.) alors que Z utilise des schémas. Les opérations sont présentées comme des transformations de prédicats à travers l'utilisation de la technique des substitutions généralisées, ce qui permet à B de spécifier des transformations d'états. Ces données et opérations sont encapsulées dans des modules qui peuvent être vus et utilisés par d'autres modules. Le concept de module B est très proche des classes Eiffel [Lan96].

Un exemple de spécification B (emprunté de K. Lano [Lan96]) utilisant la notation AMN pour spécifier un tableau est présenté dans la figure 2.13.

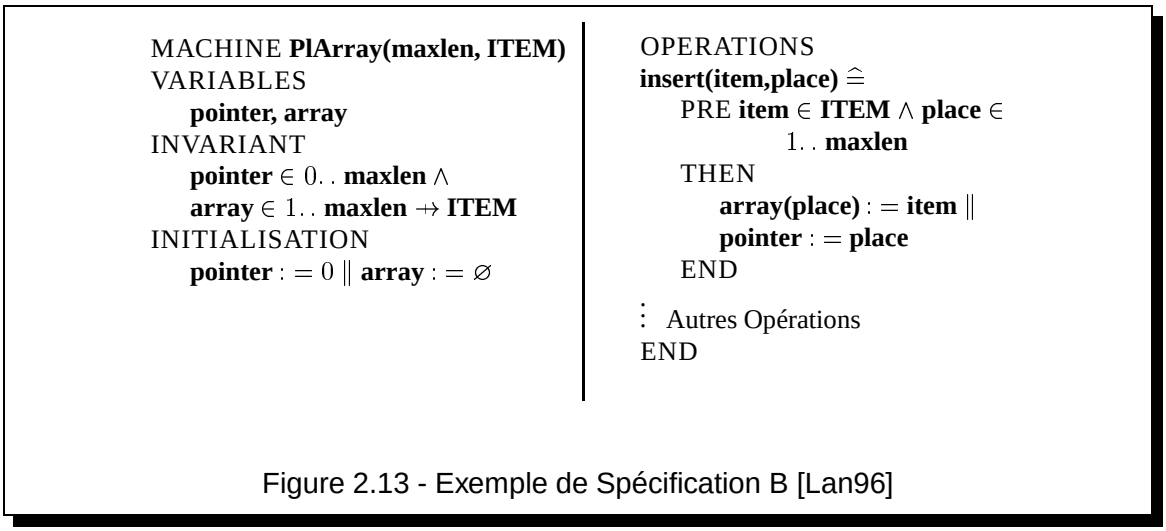


Figure 2.13 - Exemple de Spécification B [Lan96]

Les résultats obtenus par des entreprises (GEC Alsthom, Matra International) dans l'utilisation de la méthode pour des systèmes réels, ainsi que le support de la méthode par des outils robustes, permettent de conforter l'affirmation de J. P. Bowen et M. G. Hinchey [BH95] : "la méthode B est peut être la méthode du futur". Cependant, dans cette thèse, le but recherché de l'utilisation d'une méthode formelle est celui de la spécification de systèmes d'information. Les atouts de B au niveau d'une bonne couverture du cycle de vie ne constituent donc pas une différence essentielle. Nous avons personnellement préféré les notations Z par rapport aux notations B, car elles nous semblent plus "accessibles" pour des lecteurs non spécialistes de spécifications formelles mais néanmoins familiers avec la théorie des ensembles et l'approche base de données.

2.4.4 Spécifications Formelles et Objets

Les spécifications formelles introduisent des bénéfices dans le processus de développement du logiciel (cf. section 2.4). Selon B. Meyer [Mey88], les logiciels doivent être

extensibles, réutilisables et compatibles. Le concept qui peut répondre à ces besoins est la modularité. Par rapport à cette modularité, les méthodes doivent permettre une composition et une décomposition modulaires. Elles doivent fournir des possibilités pour leur compréhension de manière modulaire. Elles doivent permettre qu'un changement dans un module ne se diffuse pas dans tout le système. Elles doivent fournir un mécanisme de protection modulaire dans la mesure où un problème survenu lors de l'exécution du système doit rester confiné au module qui a causé le problème, ou dans le pire des cas aux modules voisins de celui-ci. Les spécifications formelles en plus des bénéfices qu'elles introduisent, doivent aussi répondre à ces besoins.

Selon K. Lano [Lan95], l'utilisation conjointe de méthodes formelles et orientées objets doit fournir une approche formelle pour le développement d'un système afin d'atteindre les critères de Meyer (au moins les quatre premiers). Il soutient que cette approche formelle est possible grâce aux caractéristiques suivantes des deux types de méthodes :

- les méthodes objets encouragent la création d'abstractions et les méthodes formelles introduisent les moyens pour décrire précisément ces abstractions ;
- les méthodes objets offrent des mécanismes de structuration et des disciplines de développement pour permettre l'utilisation des méthodes formelles avec des grands systèmes ;
- les méthodes formelles fournissent une manière précise pour représenter des concepts complexes des méthodes objets comme l'agrégation et les diagrammes d'états.

Cette utilisation conjointe est une réalité de nos jours. En fait, plusieurs méthodes proposent une utilisation conjointe des méthodes objets et formelles. Dans ce texte, comme Z est utilisé, ce sont les méthodes qui utilisent Z avec une orientation objet qui seront détaillées. Cependant, à titre de référence, on peut citer des méthodes telles que VDM++ [Dür94] qui est une extension objet de la méthode VDM, et la méthode PLUSS [Bid89] qui est un langage pour la production de spécifications structurées, ainsi que la méthode COLD [FJ92] destinée à l'écriture de spécifications algébriques par modèles abstraits et à leur implantation par des raffinements successifs. Pour plus de détails, on peut lire la thèse de P. André [And95] qui propose une classification sur l'intégration des méthodes formelles et objets (intégration conjointe, partielle et transactionnelle).

Par rapport à Z, on peut dire qu'au début de son utilisation conjointe avec une orientation objet, deux approches existaient : celle qui envisageait une utilisation du Z standard pour la définition des concepts du monde objet, et celle qui envisageait des ajouts sémantiques pour

rendre Z compatible avec les concepts objets. Aujourd'hui, la deuxième approche est en train de s'imposer. Les prochaines sections introduisent ces deux approches qu'on appelle Z Adapté aux Objets et Z Orienté Objet.

2.4.4.1 Z Adapté aux Objets

L'approche qu'on appelle Z Adapté aux Objets est basée sur les travaux de A. Hall [Hal90b, Hal94], où l'objectif recherché était :

- la partition d'une spécification en spécifications d'objets individuels pour leur combinaison ultérieure ;

- la définition d'une classe d'objets en termes d'autres classes avec lesquelles la première partage le comportement.

Un premier article [Hal90b] exploite le premier point en donnant des directives pour spécifier des objets et leurs états, pour utiliser des identificateurs d'objets pour y faire référence et pour exprimer leur individualité, pour exprimer l'état d'un système par les objets qu'il contient, pour spécifier les relations entre objets en termes des identificateurs d'objets ainsi qu'une méthode pour la définition des opérations selon un objet unique et pour le calcul de l'effet d'une opération sur l'état du système.

Dans un deuxième article [Hal94], A. Hall introduit les classes avec leur définition en *extension* (l'ensemble de toutes les instances) et en *intention* (la définition des propriétés des instances). Un schéma pour représenter le sous-typage (l'héritage) entre classes est donné ainsi que des directives pour le traitement des opérations par rapport à l'héritage. Les concepts principaux introduits dans ce travail sont présentés dans la section F.2 de l'Annexe F à travers un exemple.

Une utilisation des propositions de Hall est faite par J. Hammond [Ham94] dans la modélisation d'un petit système bancaire. D'abord une modélisation avec la méthode objet de S. Shlaer et S. Mellor est présentée, puis cette application est modélisée avec Z.

La vérification d'une spécification au niveau de la syntaxe et des types peut être exécutée avec des outils destinés à la vérification des spécifications Z standard, comme par exemple l'outil "Z Type Checker" [Jia95].

2.4.4.2 Z Orienté Objet

L'approche appelé Z Orienté Objet dans ce texte est fondée sur l'ajout de nouveaux concepts et sur l'extension de la sémantique Z de manière à inclure les concepts du monde objet, notamment à travers l'inclusion d'une notation pour représenter une classe qui doit encapsuler l'état, son initialisation et les opérations. Plusieurs propositions utilisent cette approche. On en introduit quelques unes (OOZE, Z++ et MooZ) ici et on présente avec plus de détails celle qui semble s'imposer : Object-Z [And95].

OOZE

OOZE ("Object-Oriented Z Environment") est un langage formel basé sur l'approche algébrique (à travers l'utilisation du langage de spécification algébrique OBJ3 [KKM87]) mais qui produit une spécification avec une "syntaxe-Z". Il a été développé à l'origine par A. Alencar et J. Goguen [AG91] à l'Université d'Oxford. Le langage peut être utilisé pour la description et la spécification de besoins.

OOZE a un modèle objet complet où les classes encapsulent les schémas d'états, l'initialisation et les opérations. La vérification syntaxique et des types ainsi que des facilités de preuve sont offertes avec l'environnement d'OBJ3.

Z++

Z++ est un langage de spécification et de conception orienté objets [LH94] qui a la caractéristique de ne pas utiliser les schémas Z pour la syntaxe, mais plutôt les machines abstraites de B. Il a en plus des constructeurs de classes et des mécanismes pour la spécification de la concurrence et du temps réel à travers la logique temps réel ("Real Time Logic"- RTL).

Z++ a ses origines dans le projet européen ESPRIT appelé REDO et il a été développé à l'Université d'Oxford en 1989. Le langage comprend des constructions orientées objets définies avec un style modulaire dans lequel le langage Z est utilisé. Un processus systématique pour la formalisation de spécifications semi-formelles OMT vers le langage Z++ a été aussi défini. Un prototype d'outil pour l'animation des spécifications Z++ a été construit. L'outil fuzz peut être aussi employé pour la détection des erreurs relatives au Z standard.

MooZ

Mooz ("Modular Object-Oriented Z") est un langage de spécification basé sur Z qui utilise une approche orientée objet [MC92]. Une spécification Mooz est composée

d'un ensemble de définitions de classes liées par une hiérarchie. Ces classes sont définies en grande partie comme une spécification Z structurée qui organise les éléments de base du modèle qui sont les schémas d'états, les états initiaux et les opérations ; une définition générique de classe peut être obtenue à travers l'introduction de types de base comme de paramètres [MC92].

Le langage Mooz a été proposé à l'origine par S. R. L. Meira et A. L. C. Cavalcanti et il est actuellement développé à l'Université Federal de Pernambuco au Brésil. Un prototype d'outil appelé ForMooz est disponible comme support pour la construction et le prototypage des spécifications Mooz. Un style existe pour l'écriture des spécifications en utilisant LaTeX.

Object-Z

Le langage Object-Z [DKRS91] est une extension du langage de spécification formel Z afin de l'adapter aux concepts de l'approche objet. Le but principal recherché est de rendre la spécification de grands systèmes plus claire à travers l'utilisation de la structuration. Cette structuration se traduit par un nouveau schéma appelé classe qui regroupe un ensemble cohérent de schémas d'états ainsi que la définition de l'état initial et des opérations. L'utilisation de ces classes est réalisée soit à travers l'instanciation de celles-ci, soit à travers l'héritage. Grâce à des invariants historiques (présentés avec une logique temporelle) une interprétation comportementale est rendue possible, les historiques étant représentés comme une séquence d'événements (des appels d'opérations) que subit un objet.

Object-Z a été développé à l'origine par R. Duke et G. Rose à l'Université de Queensland en Australie. Il est supporté par un vérificateur de types appelé "Wizard" développé aussi à Queensland. En utilisant un style, les spécifications Object-Z peuvent être écrites en LaTeX.

Le schéma de classe introduit par Object-Z est présenté dans la figure 2.14 avec les types de composants possibles. Dans ce schéma la première déclaration introduit les sur-classes de la classe si elle en a ; la deuxième introduit les types et les constantes, avec la même syntaxe que dans Z. Les trois dernières déclarations introduisent un schéma d'états et un schéma d'état initial ainsi que des schémas éventuels d'opérations. Un invariant historique peut être donné.

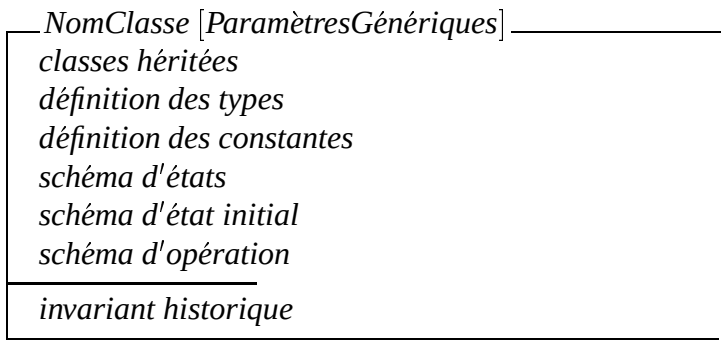


Figure 2.14 - Schéma d'une Classe dans Objetct-Z

Dans un schéma de classe, le schéma d'états n'est pas nommé. Les parties déclarative et prédicative du schéma d'états contiennent respectivement les variables d'états et les invariants d'états. Les variables et les invariants d'états sont les *attributs* de la classe. Ils sont implicitement inclus dans les parties déclarative et prédicative du schéma d'état initial ainsi que dans chacune des opérations, qui ont en plus les variables et les prédicats avec le prime (l'état d'avant) inclus dans leurs schémas.

Le schéma d'état initial est nommé avec le mot clé *INIT*. Pour les opérations une Δ-liste est ajoutée au schéma standard Z qui représente une opération pour représenter les variables qui peuvent changer lorsque l'opération est appliquée à un objet de la classe. Les attributs et les opérations sont appelés les *caractéristiques* de la classe. Les invariants historiques servent à contraindre le comportement des objets de la classe. Des commentaires textuels peuvent être ajoutés à l'intérieur des schémas de classes.

La notation proposée par Object-Z est utilisée dans ce travail pour modéliser formellement les classes d'un modèle. Ce choix s'appuie sur les bénéfices d'utilisation d'un langage formel (cf. section 2.4). Le langage Z étant l'approche formelle choisie dans ce travail, un "Z objet" est un choix naturel. Dans ce contexte, grâce à sa lisibilité et au fait qu'il commence à s'imposer comme standard pour la modélisation orientée objets avec Z, Object-Z a été retenu comme meilleure solution.

Ci-dessous, pour donner une idée du type de modélisation produite avec le langage Object-Z, on présente un exemple de schéma de classe pour une forme géométrique. Afin de mieux présenter le langage, un exemple de hiérarchie de formes emprunté à R. Duke, P. King, G. Rose, et G. Smith [DKRS91] est présenté en détails dans la section F.3 de l'Annexe F.

Forme

$couleur : Couleur$

$perim : \mathbb{R}$

$perim > 0$

Cette classe a deux constantes *couleur* et *perim*.

$x, y : \mathbb{R}$

INIT

$x = y = 0$

Transférer

$\Delta(x, y)$
$dx?, dy? : \mathbb{R}$

$x' = x + dx?$
$y' = y + dy?$

2.5 Méta-Modélisation

Selon le dictionnaire Robert, le terme méta vient du grec et signifie la succession, la transformation, le changement. Dans les néologismes scientifiques (méta-langue, méta-mathématique) méta signifie ce qui dépasse, englobe, un objet de pensée, une science.

En prenant en compte cette définition, on peut dire qu'un méta-modèle est le modèle d'un modèle. Plus précisément, un méta-modèle est un langage permettant de modéliser des systèmes particuliers, à savoir des modèles [Ous97] et la méta-modélisation est donc le processus de création d'un méta-modèle, tout comme la modélisation est le processus de création d'un modèle.

La figure 2.15 présente ces processus. En (a) est représentée la modélisation d'un système S ; lorsqu'on modélise un modèle, (b) on méta-modélise pour créer le modèle d'un modèle ou le méta-modèle ; si ce méta-modèle est lui-même modélisé (à travers une méta-méta-modélisation (c)) on produit le modèle du modèle d'un modèle ou le méta-méta-modèle. Dans les ateliers courants, on se limite à un seul niveau de méta-modélisation. Les méta-méta-modèles sont utiles lors de la transformation de modèles de méthodes.

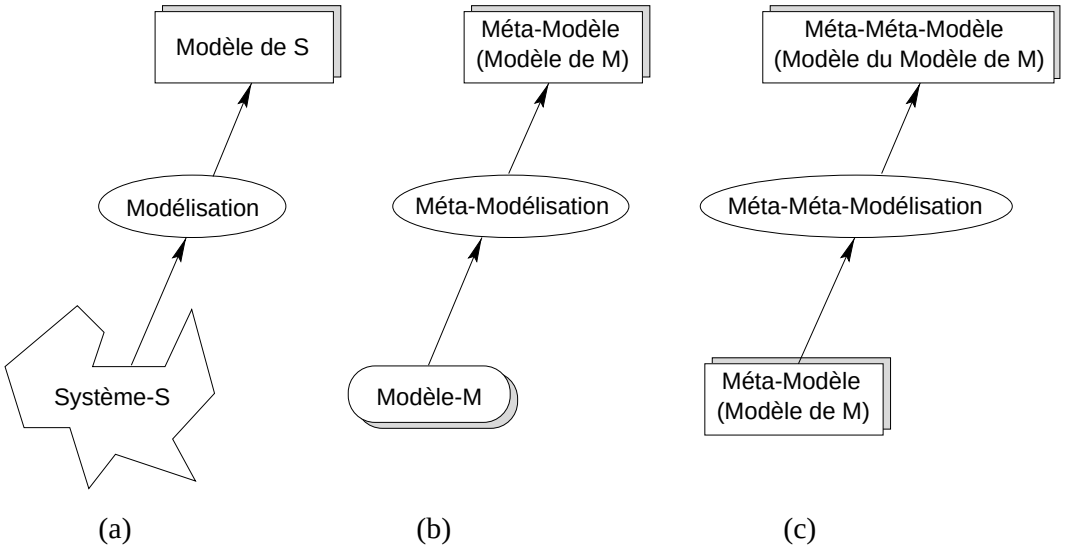


Figure 2.15 - Méta-Modélisation [Ous97]

Dans ce travail, un méta-modèle est proposé. La raison principale pour l'utilisation d'un méta-modèle repose sur la possibilité de définition d'un référentiel dans lequel des modèles informels, semi-formels et formels sont représentés conjointement, afin de rendre une modélisation basée sur un modèle défini à l'aide de ce méta-modèle plus claire et moins ambiguë.

Plusieurs résultats sont introduits par l'utilisation d'un méta-modèle. Un méta-modèle fait ressortir la capacité d'expression des méthodes (la richesse des concepts utilisés pour établir des spécifications précises des systèmes d'information et leurs relations) et la comparaison entre méthodes est rendue plus facile. Lorsque des méthodes sont représentées à travers un même méta-modèle, on peut les comparer et introduire un mécanisme pour la traduction des modèles d'une méthode vers des modèles équivalents d'une autre méthode en utilisant un méta-méta-modèle général.

Il faut noter aussi que l'utilisation du niveau méta-modélisation est indispensable pour coordonner des modélisations hétérogènes(aspect multi-modélisation).

L'utilisation d'un méta-modèle a été proposée dans la littérature comme l'une des solutions possibles pour faciliter la comparaison des méthodes orientées objets [EG93, HvdGB93]. Dans la section 5.3.2 nous présentons un cadre pour effectuer cette comparaison en utilisant un méta-modèle.

2.6 Conclusions

Dans ce chapitre, après l'introduction des concepts du génie logiciel, des ateliers qui implantent l'utilisation de ces concepts ont été introduits. Ces ateliers étant basés sur des méthodes elles-même basées sur des langages, ces derniers ont été présentés.

L'objectif de ce travail étant une proposition de modélisation qui utilise différents types de formalismes, les langages informels, semi-formels et formels ont été ensuite introduits.

Il faut noter que la distinction entre spécification semi-formelle et spécification informelle n'est pas toujours aisée. En prenant l'exemple du formalisme graphique IA-NIAM [Hab88] on peut dire qu'il permet une spécification tout à fait formelle car ce formalisme s'appuie sur un modèle rigoureux et bien défini : les ensembles, les relations binaires entre ensembles, etc., contrairement à d'autres modèles qualifiés de semi-formels, comme par exemple OMT [RBP$^+$91], car la définition de leurs concepts est trop ambiguë.

L'utilisation des méthodes formelles se fait chaque jour plus présente et en même temps nécessaire ; mais cette utilisation doit se faire d'une manière conjointe avec d'autres formalismes pour que les résultats soient encore meilleurs. Ainsi Object-Z a été choisi pour son utilisation comme composant formel d'un processus de méta-modélisation qui est aussi introduit.

Le langage formel Object-Z est une extension du langage formel Z avec une sémantique supplémentaire pour la représentation des classes, de l'héritage, de l'instanciation, du polymorphisme et des invariants temporels. Les relations entre classes sont représentées par d'autres classes qui ont, dans leurs schémas d'états, des instances (variables dont le type est une classe) de classes inter-reliées.

Chapitre 3

La Modélisation Orientée Objet

3.1 Concepts du Monde Objet

Depuis le début des années 80, le “Paradigme Objet” a attiré sur lui l’attention du monde du logiciel jusqu’à être considéré comme l’une des plus grandes évolutions depuis l’apparition de la programmation structurée des années 70. Dans les sections suivantes, nous en présentons les concepts essentiels car ils font partie des bases des modèles que nous souhaitons supporter dans notre atelier.

3.1.1 Objets

Une définition abstraite d’un objet, donnée par le Dictionnaire “Le Petit Robert” est : “tout ce qui se présente à la pensée, qui est occasion ou matière pour l’activité de l’esprit”. L’application de cette définition abstraite au domaine informatique est le point clé des méthodes orientées objet. A travers la *Modélisation Conceptuelle*, on peut comprendre pourquoi. La modélisation conceptuelle consiste à utiliser une abstraction du *Domaine d’Application* avec identification des parties des applications qui sont les moins volatiles aux changements à travers la modélisation des entités et des phénomènes importants des applications. La modélisation conceptuelle est conditionnée par deux aspects :

– *abstraction* : c’est un processus qui permet au programmeur, en regardant la réalité, d’en capturer et d’en abstraire, sous forme d’entités, la structure et les actions qu’il pense être essentielles pour l’application ; “une abstraction est une représentation simplifiée d’un système, qui accentue quelques éléments ou propriétés, en même temps qu’elle en supprime d’autres” [LT77] ;

– *représentation* : c'est le mécanisme utilisé pour le montage des conventions qui seront utilisées pour la représentation du modèle conceptuel. L'avantage des abstractions manipulées par le logiciel (les objets) est de permettre une traduction directe des "choses du monde réel". C'est cela qui représente l'intérêt premier des méthodes orientées objet.

Au niveau informatique, on peut définir un objet comme un être relativement indépendant composé de :

– *un état interne*, c'est-à-dire, une mémoire interne (les attributs) où les valeurs peuvent être stockées et modifiées pendant la vie de l'objet ;

– *un comportement*, c'est-à-dire, un ensemble d'actions appelées opérations, à travers lesquelles on change l'état interne d'un objet et à travers lesquelles l'objet répond aux demandes que lui font les autres objets.

L'ensemble état interne et comportement compose le concept d'objet au niveau informatique et on parle d'*encapsulation*. Le mécanisme utilisé pour changer l'état interne des objets est basé sur des messages. En effet pour changer l'état interne d'un objet, il faut que cet objet reçoive un ordre (le message) d'un autre objet qui lui demande d'exécuter une opération qui réalise un service demandé. Les messages sont traités plus loin dans ce texte.

3.1.2 Classes

Il y a deux catégories principales d'objets : les classes et les instances. Une classe est un terme général qui indique une classification ; plus particulièrement, dans le paradigme objet, une classe est un patron pour une catégorie d'items structurellement identiques et un mécanisme pour créer ces items, les instances, en se basant sur le patron. Une classe rassemble toutes les caractéristiques tant statiques (attributs) que dynamiques (opérations) de ses instances.

Dans le domaine des objets, par rapport à la terminologie employée, deux questions se posent : un objet est-il une classe ? Une classe est-elle un objet ? La réponse à ces questions dépend du nombre de niveaux d'instanciation présents dans le système (un, deux, trois, etc.) et de la réification du modèle utilisé.

Dans les systèmes avec un seul niveau d'instanciation, appelés *systèmes à hiérarchie simple*, tous les objets peuvent être vus comme des classes et toutes les classes peuvent être

vues comme des objets. Il existe un seul type d'objets, donc pour ces systèmes les objets sont des classes et les classes sont des objets ; plus généralement, tout est objet.

Dans les systèmes avec deux niveaux d'instanciation, tous les objets sont des instances de classes, mais ces instances ne sont pas des classes accessibles (cf. langage C++). On a donc deux types d'objets : les *Objets* et les *Classes*. Pour ces systèmes, les classes ne sont pas des objets, et les objets ne sont pas des classes.

Dans les systèmes avec trois niveaux d'instanciation, tous les `objets` sont des instances de `classes` et toutes les `classes` sont des instances d'une classe appelée `méta-classe`. Une `méta-classe` est donc une classe dont les instances sont elles aussi des classes. Comme la `méta-classe` est elle aussi une classe, elle est donc une instance d'elle-même. Il y a deux types distincts d'objets : les *Objets* et les *Classes*, avec une classe distinguée, la *Méta-Classe*.

Dans les systèmes avec cinq niveaux d'instanciation, (cf. Smalltalk) chaque donnée est représentée par un objet ; chaque `objet` est une instance d'une classe particulière, laquelle décrit la structure et les opérations de toutes ses instances. Les `classes` peuvent être considérées comme des gabarits d'objets (la `classe Class`), lesquels sont capables de créer des instances d'eux mêmes (les `classes`) ; elles sont aussi des objets. Les `classes Class` sont décrites par une autre classe appelée `méta-classe` . Les `méta-classes` sont elles aussi des objets et sont des instances d'une autre classe appelée `méta-classe Class`. Pour éviter une régression infinie, les `méta-classes` et les `méta-classes Class` ont une relation d'instanciation réciproque. Les cinq niveaux sont donc : `objet`, `classe`, `classe Class`, `méta-classe` et `méta-classe Class` et il y a trois types différents d'objets : les *Objets*, les *Classes* et les *Méta-Classes*. Toutes les classes sont dérivées d'une classe appelée Objet laquelle définit le comportement commun pour tous les objets.

Dans les systèmes avec trois ou cinq niveaux, les classes sont des objets, car elles sont des instances des méta-classes ; un objet est aussi une classe, mais seulement si cet objet est une instance d'une méta-classe.

Un concept similaire à celui des méta-classes présent dans les méthodes est celui de *Classes Paramétrées*. Une *classe paramétrée* est un gabarit pour une classe dans lequel des items spécifiques ont été identifiés comme étant nécessaires pour créer des *Classes Non-Paramétrées* basées sur le gabarit. Une classe paramétrée peut être vue comme une classe avec des définitions en blanc. On ne peut pas instancier directement une classe paramétrée ; il faut d'abord remplir les paramètres définis en blanc dans la classe paramétrée, ce qui donnera une classe non-paramétrée, pour pouvoir ensuite instancier cette classe.

Un troisième type de classe en relation avec le concept d'instanciation est la *Classe Abstraite*, qui ne peut pas être instanciée et qui englobe des concepts cohérents mais aussi incomplets, dans le but de fournir leurs caractéristiques, via l'héritage, à des classes spécialisées.

Plus généralement, en modélisation de SI, il est possible de distinguer plusieurs types de classes :

- *des classes abstraites* pour faciliter la description de propriétés générales ;
- *des classes dérivées* utilisables pour exprimer des points de vue spécifiques ;
- *des classes utilitaires* abstraites ou instanciables mais qui sont secondaires dans la spécification du SI.

3.1.3 Héritage

L'héritage est une manière naturelle de modéliser le monde réel (le domaine d'application) de façon à fournir un modèle concis pour l'analyse, la conception et, parfois même, l'implantation orientées objets.

Ce processus de modélisation est implanté sur le paradigme objet à travers une structure hiérarchique de classes, basée sur une relation transitive et antisymétrique *entre classes* qui s'appelle héritage. Cette structure hiérarchique peut être employée pour représenter un héritage simple (une classe n'hérite que d'une seule classe) ou un héritage multiple (une classe hérite d'une ou plusieurs classes) et dans ce cas on a une structure du type treillis.

Deux classes différentes peuvent partager partiellement la description de leurs états et de leurs comportements. Pour éviter les redondances de description et pour faire ressortir les différences entre ces classes dans une structure hiérarchique d'héritage, la notion de sur-classe est introduit e (classe-mère ou classe-base selon la méthode) pour rassembler les descriptions communes, et de sous-classe (classe-fille ou simplement classe selon la méthode) pour décrire les spécificités. Un exemple est la sur-classe Véhicule qui a comme sous-classes Voiture et Bateau. Le processus de création de sur-classes est appelé *généralisation* alors que celui de création de sous-classes est appelé *spécialisation*.

Quand on crée les sous-classes, on emploie le concept d'héritage. L'héritage est donc la propriété qui permet la construction d'une classe en partant d'autres classes. Les nouvelles classes vont hériter de l'état et du comportement des sur-classes associées ; on peut cependant changer quelques propriétés indésirables de cet héritage.

L'héritage doit être étudié du double point de vue ensembliste ou extensionnel et descriptif ou intentionnel. Du point de vue ensembliste, il doit y avoir inclusion ensembliste des instances d'une sous-classe dans ses sur-classes. Cette inclusion ensembliste peut être contrainte : disjonction ou recouvrement, couverture totale ou partielle et partition. En ce qui concerne l'aspect descriptif, l'ensemble des descriptions d'une sur-classe doit être applicable aux instances de ses sous-classes. Cependant, une opération d'une sous-classe peut, sous le même nom, réaliser des actions formellement différentes mais équivalentes (ex : une opération `jouer` qui présente, soit une image, soit une vidéo, selon le type d'objet passé comme paramètre) à une opération définie dans sa sur-classe. Cette propriété est appelée polymorphisme et est décrite ci-dessous.

3.1.4 Polymorphisme

Le polymorphisme est la propriété qui permet à des opérations d'être vues sous différentes formes. Il y a deux types de polymorphisme :

- *surcharge* : le nom d'un opérateur ou d'une fonction peut être utilisé dans plusieurs situations, c'est-à-dire, un opérateur ou une fonction peuvent porter sur des données entières, graphiques, etc. Le code qui implante effectivement l'opérateur ou la fonction est définie par les paramètres qui lui sont passés. L'opération qui doit être surchargée est définie dans une sur-classe et son corps est défini dans la sous-classe qui surcharge l'opération.

- *polymorphisme paramétrique* : une variable utilisée dans le corps d'une opération peut porter sur des objets de types différents. Par rapport à la surcharge, le polymorphisme paramétrique, ou simplement polymorphisme, présente l'avantage de permettre l'écriture du corps d'une opération applicable à toute une catégorie de types [Bru93].

Le mécanisme qui supporte les deux formes de polymorphisme est la *Liaison Dynamique* ou *Résolution Tardive*. Par ses caractéristiques, le polymorphisme permet de ne pas connaître au moment de la compilation l'opération à exécuter par un objet à la réception d'un message. C'est la liaison dynamique qui permet la détermination de l'opération appelée au moment de l'exécution.

3.1.5 Message

Dans le monde des objets, la représentation de la communication entre un utilisateur et un objet ou entre les objets est basée uniquement sur l'envoi de messages. Un message est composé du nom de l'objet qui le reçoit, d'un sélecteur qui représente le nom de l'opération qui sera exécutée et de paramètres définis par la signature de l'opération cible.

Après l'envoi d'un message par l'objet émetteur, c'est à l'objet récepteur de définir quelle opération sera exécutée selon l'interface de sa classe ou d'une de ses sur-classes. De cette manière, des messages de "même signature" et de même sélecteur peuvent porter sur des opérations différentes (le concept de surcharge).

Dans la plupart des langages, les messages sont traités de manière procédurale, c'est-à-dire, qu'un message est traité par l'objet qui le reçoit et c'est seulement après ce traitement que le contrôle est rendu à l'objet (ou l'utilisateur) qui a envoyé le message. Cette approche pose un problème lorsque le parallélisme et la synchronisation sont des variables importantes.

3.1.6 Relations

Les classes et les objets établissent le comportement d'un système à travers leurs différents types de relations. Dans cette section nous présentons alors quelques uns des différents types de relations.

3.1.6.1 Instanciation

La relation d'instanciation est une relation existant entre un "patron" et quelque chose qui est défini à partir de ce patron. Des exemples sont la relation existant entre les méta-classes et les classes ou entre les classes et les objets. De cette manière par exemple, par rapport à une classe, un objet est créé à partir des descriptions contenues dans la classe, laquelle définit le comportement et les informations que l'instance contiendra. Une étude sur les différents niveaux d'instanciation a été présentée à la section 3.1.2.

3.1.6.2 Association

Une association décrit une connexion entre classes ou entre objets. Quelques méthodes (OMT, OOD) font une distinction entre les associations entre classes et les associations entre objets ; dans ce cas, une restriction existe : l'existence d'une association entre objets

est soumise à l'existence d'une association d'un type donné entre les classes dont les objets sont des instances.

Les associations entre objets peuvent avoir des attributs qui les caractérisent. Parmi ces attributs, on peut citer la multiplicité d'une association, les rôles des composants dans l'association, une clé pour l'association et même des attributs plus complexes décrits à travers des classes (classes d'associations).

3.1.6.3 Utilisation

La relation d'utilisation apparaît dans les méthodes orientées objets pour représenter des relations du type client/serveur : le client dépend du serveur pour accomplir ses responsabilités [Boo94] ; il *utilise* les services du serveur. Cette relation est de type envoi de messages.

3.1.6.4 Agrégation

La relation d'agrégation est un type de relation entre un "*tout*" et ses "*parties*" où le *tout* est composé de *parties* ; c'est une relation transitive et antisymétrique [Bla93]. Ce type de relation définit un lien entre un *composé* et ses *composants* et est appelé aussi *Relation de Composition*.

La relation d'agrégation est exploitée en détail par J. Odell dans [Ode94]. L'Annexe D dans sa section D.3 page 235 présente, l'analyse approfondie de J. Martin et J Odell en matière de composition [MO95]. Pour résumer on peut dire que ces auteurs présentent le type d'une agrégation comme étant défini par les valeurs des trois propriétés suivantes :

- *configuration* : un composant a une relation fonctionnelle particulière ou structurelle avec un autre composant ou avec l'objet qu'il constitue ;
- *homéomérique* : les composants sont de même type que le composé ;
- *invariance* : les composants ne peuvent pas être séparés du composé sans destruction de celui-ci.

En partant de ces trois propriétés, ces auteurs proposent six types de composition : *objet composant-objet composé, objet-matière, objet-partie, zone-endroit , membre-collection* et *membre-association*. Avec ces six types de relation d'agrégation, on arrive à représenter un grand nombre de situations de modélisation dont l'agrégation est l'objet.

3.1.7 Démarche Orientée Objet

L'orientation objet est une manière de penser un problème en utilisant des modèles organisés autour des concepts du monde réel [EG93]. Avec une démarche orientée objet, on doit donc arriver à un processus qui rend plus claire l'image qu'on a d'un système ; elle créée aussi une manière simple pour traduire cette image du système vers les systèmes logiciels [Ner92].

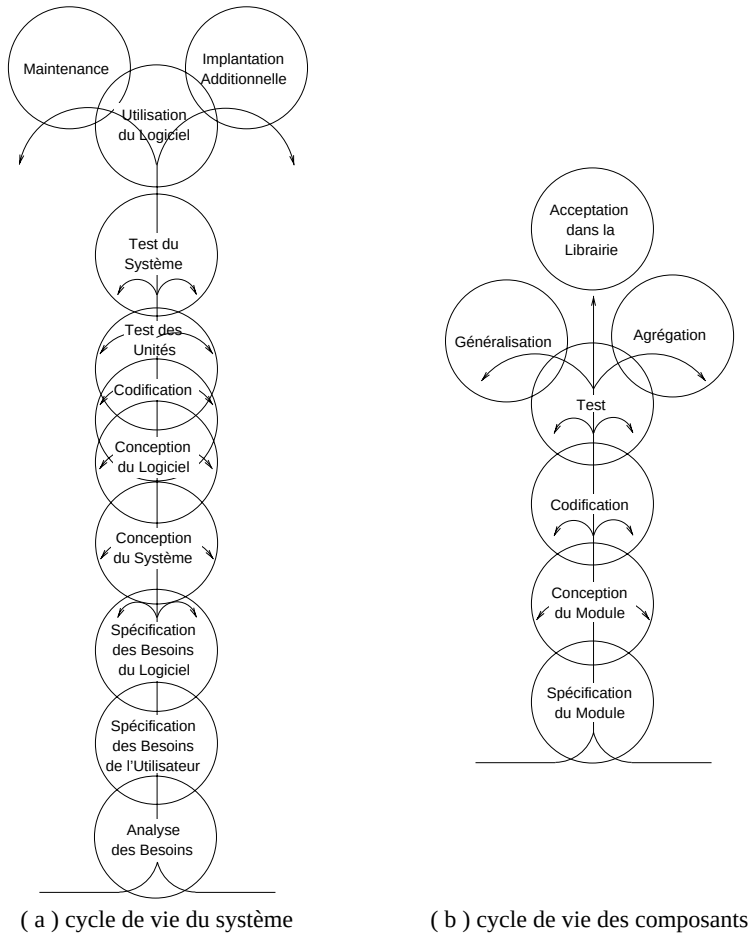
Dans le domaine du génie logiciel, il y a un point sur lequel tous les spécialistes s'accordent : pour créer un logiciel, il faut prendre en compte le concept de *Cycle de Vie* du logiciel. Avec une démarche générale traditionnelle, ce cycle de vie peut être visualisé comme un chemin qui débute avec une étape d'analyse, passe par une étape de conception pour se finir par une étape d'implantation. Si, d'un côté, on peut dire que l'un des principaux avantages de l'utilisation d'une analyse et d'une conception de systèmes orientées objet est de permettre une migration homogène de ces phases vers celle d'implantation, c'est à dire, de faciliter la communication entre les phases par l'utilisation d'un langage commun, d'un autre côté, un problème a été remarqué par Fichman [FK92], Champeaux [CF92] et Monarchi [MP92] : la frontière entre les phases d'analyse et de conception est très mince ; on ne peut plus parler de phases bien définies et séparées.

Le cycle de vie dans un développement orienté objet étant très lié à la méthode utilisée, on ne peut pas présenter *le cycle de vie général* des méthodes orientées objet. Cependant, ci-dessous nous présentons quelques propositions pour ce cycle de vie.

Le *Cycle de Vie en Fontaine* proposé par B. Henderson-Sellers et J. M. Edwards [HSE90] est une proposition intéressante. Ce cycle présente les phases du cycle de vie orienté objet ainsi que le recouvrement existant entre elles. Il présente aussi l'interaction entre ces différentes phases avec l'analogie d'une fontaine (cf. figure 3.1.a). Le cycle de vie est représenté de bas en haut et de cette manière les phases les plus importantes, celles de spécification, sont placées à la base.

L'idée présente dans ce cycle est que lorsqu'un vase (une phase) est plein, on peut passer soit au vase supérieur (la phase suivante) soit redescendre au vase inférieur pour compléter une activité non terminée dans une phase antérieure ; dans ce cas, le cycle reprend à partir de cette activité vers le haut de la fontaine. Les phases de maintenance et d'implantation additionnelles renvoient le cycle de vie vers le bas de la fontaine.

La construction d'un système étant basée sur les classes, les auteurs proposent aussi un cycle de vie pour le développement de ces classes (cf. figure 3.1.b). Ce cycle peut aussi être appliqué au développement d'un ensemble de classes appelé regroupement (“ cluster”).



(a) cycle de vie du système (b) cycle de vie des composants

Figure 3.1 - Cycle de Vie en Fontaine [HSE90]

L'un des principaux bénéfices de l'utilisation du paradigme objet étant la réutilisation, on présente aussi une proposition de cycle de vie orienté objet qui fait ressortir cet aspect. Par rapport au cycle de vie de B. Henderson-Sellers et J. M. Edwards, celui proposé par J. M. Nerson [Ner92] est plus concis. Il ne présente pas une liste extensive des phases du développement (cf. figure 3.2) mais fait ressortir les différences de niveau de modélisation (monde réel et informatique). Dans les deux niveaux, les activités s'appuient fortement sur la réutilisabilité, soit à partir de la réutilisation de “*cadres*” dans le modèle du monde réel, soit à partir de la réutilisation directe de classes dans le modèle informatique.

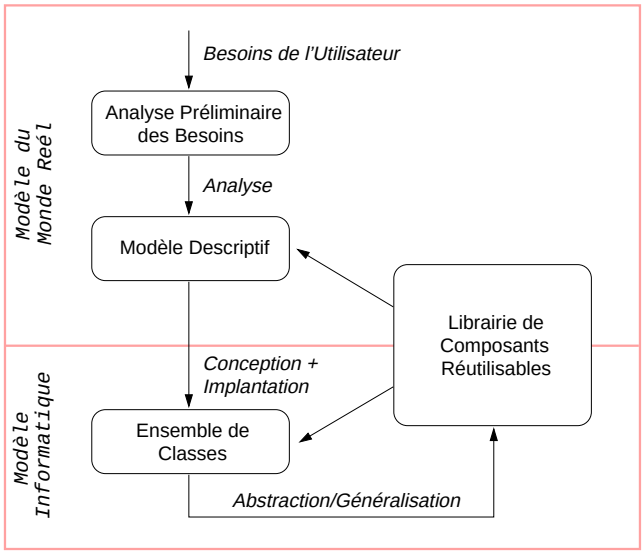


Figure 3.2 - Cycle de Vie Orienté Objet - J. M. Nerson [Ner92]

Dans les cycles de vie présentés ci-dessus, les trois étapes d'une démarche orientée objet sont présentes au dessus des phases élémentaires proposées par les cycles.

L'*Analyse Orientée Objet* est un processus qui établit une liaison directe entre les choses réelles du domaine du problème et son abstraction appelée modèle conceptuel ou spécification de l'application. L'analyse dresse un "pont" entre les utilisateurs, spécialistes du domaine d'application et les développeurs, spécialistes des solutions de problèmes génériques.

La *Conception Orientée Objet*, de son côté, doit organiser la solution informatique ; elle s'appuie sur les spécifications établies dans la phase d'analyse, avant de passer à une implantation logicielle et matérielle exploitable.

L'*Implantation Orientée Objet* consiste à réaliser un logiciel à l'aide de langages de programmation orientés objet. Ces langages sont à la base de toutes les propositions de développement orienté objet par leurs caractéristiques de réutilisation du code déjà développé.

La phase critique reste l'analyse des besoins des applications. La solution qui consiste à adopter une démarche qui commence avec l'analyse de systèmes orientée objet, passe par la conception orientée objet pour arriver a l'implantation avec les langages orientés objets, de manière à ce que tout le processus de création utilise un même paradigme, permet une modélisation plus stable et qui bénéficie de cohésion et de cohérence sur tout le cycle de vie.

3.2 Méthodes Orientées Objet

Le *Paradigme Objet* est apparu à la fin des années 80 comme l'élément qui pourrait révolutionner le processus de développement de logiciels à l'aide de langages et de méthodes orientés objet. Ces méthodes, comme les méthodes de l'analyse structurée, sont apparues après les langages ; ainsi après l'apparition des langages objets, les méthodes de conception orientées objets ont été proposées suivies des propositions des méthodes d'analyse orientées objets.

L'origine principale des méthodes de conception orientées objet est matérialisée par les travaux de G. Booch sur la conception de logiciels en langage Ada [Boo81, Boo82]. On peut parler de *Méthodes de Première Génération*. Booch préconise une structuration du logiciel en forme de treillis où les arcs sont représentés par des liens d'utilisation et les nœuds par des objets. Le lien d'héritage a été ajouté par la suite au lien d'utilisation. Ensuite les méthodes d'analyse sont apparues, avec l'objectif de fournir des techniques objet dès la phase de modélisation du domaine d'application.

Les méthodes pour l'étude et le développement de logiciels appliquées en génie logiciel étant un sujet permanent de recherche dans le monde de l'informatique, on peut remarquer que, depuis le début des années 90, plusieurs méthodes d'analyse et de conception ont été proposées. Ce sont les *Méthodes de Seconde Génération* ; parmi celles-ci quelques unes ont effectivement conquis une place dans le Monde des Objets.

Parmi elles, on peut citer les propositions de Berard [BSE92b, Ber93], Booch [Boo94], Coad et Yourdon [CY91a, CY91b, You94], Colbert [Col89] , Coleman et all [CJD93], Embley et Kurtz [EKW92], Martin et Odell [MO95], Rumbaugh et all [RBP$^+$91], Shlaer et Mellor [SM88, SM92] et Wirfs-Brock [WBWW90] par exemple.

Comme l'analyse structurée dans la décennie 70, l'analyse et le développement de systèmes orientés objet a encore un long chemin à parcourir pour s'établir comme une solution définitive. Cependant, les bénéfices de l'utilisation des méthodes orientées objet ont été déjà largement présentés dans la littérature ces dernières années. Prenant en compte ce fait, plusieurs entreprises se sont lancées dans l'utilisation des méthodes orientées objet.

Les méthodes citées ci-dessus ne constituent pas une liste exhaustive des méthodes existantes. Devant la multiplicité des modèles qui sous-tendent ces méthodes, on peut se demander comment choisir la méthode la plus utile dans le domaine où l'on veut l'employer. Ce problème de choix est essentiel pour les entreprises qui envisagent de migrer vers les méthodes orientées objets car les critères de choix sont nombreux : domaine d'application, formation du personnel, migration à partir d'une méthode, etc.

Nous avons choisi quatre méthodes jugées comme les plus représentatives : les méthodes de P. Coad et E. Yourdon, de G. Booch, de J. Rumbaugh et de son équipe, et de J. Martin et J. Odell.

La méthode de P. Coad et E. Yourdon a été choisie pour sa simplicité (elle dispose des concepts de base du paradigme objet) mais aussi pour son objectif affirmé de couvrir tout le cycle de vie des applications. Les méthodes de G. Booch et J. Rumbaugh ont été choisies pour leur richesse de concepts et de détails par rapport au paradigme objet. Ces deux méthodes sont présentées, malgré leurs ressemblances, car la méthode de Booch est orientée vers l'implantation alors que celle de Rumbaugh est orientée vers l'analyse. La méthode de J. Martin et J. Odell a été choisie pour son approche non usuelle par rapport aux trois autres.

3.2.1 Méthode OOA

3.2.1.1 Introduction à OOA

La méthode originale proposée par P. Coad et E. Yourdon [CY91a, CY91b] permet un développement pas à pas de la modélisation d'un système orienté objet en utilisant des abstractions du domaine du problème pour créer un modèle unique du système en cinq couches (Sujet, Classe et Objets, Structure, Attribut et Service). Le modèle créé lors de ce développement est utilisé pour décrire les fonctions du système à travers un modèle entité/relation auquel ont été ajoutées des caractéristiques typiques des langages/bases de données orientées objets comme les messages et les identités d'objets.

Cette approche par couche est questionnée par E. Yourdon [You94]. L'auteur réaffirme que cette approche avec un modèle unique est utile pour des applications qui ont une caractéristique d'orientation vers les données. Pour d'autres types d'applications, une approche multi-modèles serait une meilleure solution. Ce texte présente cette dernière proposition comme étant la méthode OOA, même si P. Coad n'est pas co-auteur du dernier ouvrage.

Quoiqu'on puisse avoir l'impression d'un changement par rapport à la méthode originale, dans [You94] ce sont plutôt les couches qui sont représentées comme étant des modèles. Les modèles[1] proposés sont présentés ci-dessous. Les concepts proposés par la méthode, la notation associée ainsi que la procédure de développement sont présentés en Annexe A.

– *Classe et Objets* : cette partie de l'analyse porte sur la découverte des classes et objets qui composeront le modèle d'analyse. Cette découverte selon l'auteur dépend de la perspective avec laquelle on regarde un système : perspective basée sur les données,

[1] dans [You94] le terme utilisé n'est pas modèle, mais on l'emploie dans ce texte pour garder une cohérence avec l'ensemble des méthodes présentées.

sur les fonctions ou sur le comportement. Pour les classes et objets, une notation pour leur représentation est proposée, des techniques pour trouver les objets dans le domaine du problème selon chacune des perspectives sont données et une technique pour grouper ces classes et objets dans des sous-systèmes est proposée ;

– *Structures* : cette partie s'occupe de mettre en évidence les classes dans des structures d'héritage ainsi que dans des structures du type composé-composant ;

– *Relations* : dans cette partie, les relations entre classes et entre objets sont introduites. Ces relations sont des représentations statiques de règles du genre : "une personne peut avoir une seule voiture et une voiture ne peut appartenir qu'à une seule personne". Ces relations ne sont pas dynamiques. La méthode présente des relations binaires et d'ordre supérieur ;

– *Attributs* : dans cette partie, le modèle de l'analyse est enrichi avec les attributs. Ces attributs décrivent des données qui sont cachées dans des classes et des objets ; ils sont traités par les services de la classe. La méthode présente une heuristique pour la découverte des attributs ainsi que pour le placement de ces attributs dans les hiérarchies de classes ;

– *Comportement de l'Objet* : cette partie modélise le comportement intrinsèque des objets à travers des diagrammes qui présentent les états des objets comme par exemple des diagrammes d'histoire de l'objet ;

– *Services* : cette partie s'occupe de la modélisation du côte dynamique des systèmes. Pour cela les services et les messages sont utilisés. Une heuristique pour la découverte des services ainsi qu'une technique pour documenter les messages sont proposées.

3.2.1.2 Évaluation - OOA

L'Analyse Orientée Objet de Coad et Yourdon est une méthode où l'on peut trouver tous les principes de base du paradigme objet : les classes, les instances, l'héritage, l'encapsulation et la communication entre objets. La procédure de recherche des classes et objets ainsi que celle pour le placement des attributs dans les hiérarchies de classes sont très intuitives.

La méthode ne supporte pas la notion de rôle dynamique des objets bien qu'elle essaie de résoudre cela avec des diagrammes qui représentent les états des objets ; ceux-ci cependant ne sont introduits qu'après l'identification des objets.

La notation utilisée dans la conception du système est la même que celle utilisée dans l'analyse. Si, d'un côté, cette approche ne dirige pas le développeur vers un langage ou un système d'exploitation particulier, d'un autre côté, le développeur n'a pas un modèle détaillé qui puisse le guider dans le développement de ce système.

3.2.2 Méthode OMT

C'est une méthode proposée par J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy et W. Lorensen dans le cadre de la société General Electric [RBP+91]. La Technique de Modélisation d'Objets OMT ("Object Modeling Technique") est basée sur trois points de vue ou trois modèles : le *Modèle des Objets*, le *Modèle Dynamique* et le *Modèle Fonctionnel*.

3.2.2.1 Modèle des Objets

Le Modèle des Objets présente les aspects statiques, les structures et les données d'un système en fournissant l'environnement des deux autres modèles ; le Modèle des Objets est le plus important des trois modèles. Les diagrammes présents dans ce modèle sont de deux types :

- *Diagrammes de Classes* : ils montrent les descriptions génériques des systèmes possibles, en agissant comme un patron, un gabarit qui décrit plusieurs instances des données ; le diagramme des classes décrit les classes d'objets, leurs hiérarchies et leurs relations et il peut être composé uniquement d'instances de classes avec leurs relations.
- *Diagrammes d'Objets* : ils montrent des implantations possibles d'un système avec leurs relations particulières. Il peut être composé d'instances de classes et d'instances d'objets avec leurs relations.

3.2.2.2 Modèle Dynamique

Le Modèle Dynamique représente les aspects du système affectés par le temps et par l'ordonnancement de l'exécution des opérations à travers la description de l'évolution temporelle des objets d'un système, par rapport aux changements qu'ils subissent comme réponse à des interactions avec d'autres objets, internes ou externes au système, en utilisant un diagramme appelé *Diagramme d'États* (un automate de Mealy) qui est basé sur le travail de D. Harel [Har87].

Le Diagramme d'États est donc utilisé pour montrer le rapport entre des événements et des états en décrivant le comportement des instances d'une classe particulière. C'est un graphe où les nœuds sont des états et les arcs entre ces nœuds sont des transitions d'états. Lorsqu'un événement est reçu, l'état suivant dépend de l'état courant et de l'événement qui a été reçu. Les Diagrammes d'États peuvent comporter des boucles et avoir un départ et une fin précis.

Le lien entre le Diagramme d'États et le Diagramme d'Objets est basé sur les événements qui correspondent à des opérations à déclencher définies dans les Diagrammes d'Objets. Le lien entre le Diagramme d'États et le Modèle Fonctionnel est basé sur les opérations qui représentent les fonctions du Modèle Fonctionnel.

Généralement, les Diagrammes d'États sont utilisés pour des objets qui reçoivent et qui traitent des événements externes selon l'état de l'objet ; ces objets sont appelés *contrôleurs* : ils contrôlent et maintiennent le système.

3.2.2.3 Modèle Fonctionnel

Le Modèle Fonctionnel décrit les opérations d'un système. Plus particulièrement il décrit comment l'exécution d'une opération change la valeur des attributs des objets d'un système.

Les opérations peuvent être étudiées de deux manières différentes : comme une boîte noire, ou à travers leur implantation. L'aspect "boîte noire" s'occupe des résultats des opérations sur les valeurs du système et il est non procédural ; il est représenté par la description des états "avant" et "après" l'exécution d'une opération dans le système. L'aspect "implantation" est procédural et il est représenté par la description du flux de contrôle entre les opérations subordonnées dans plusieurs objets qui ont rapport avec le changement d'état.

Le lien entre le Modèle Fonctionnel et le Diagramme d'Objets est basé sur les fonctions qui correspondent aux opérations définies dans les Diagrammes d'Objets.

Le Modèle Fonctionnel comprend toujours des *Descriptions des Opérations* et optionnellement des *Diagrammes de Flux de Données Orientées Objet*, des *Diagrammes d'Interaction d'Objets*, des *Diagrammes d'Interaction Concurrents*, du *Pseudo-code* et du *Code*.

Les principaux concepts de chacun des modèles ainsi qu'un résumé de la procédure de développement sont donnés en Annexe B.

3.2.2.4 Évaluation - OMT

La méthode OMT est une approche entité-relation des données à laquelle ont été ajoutés de nouveaux concepts et de nouvelles constructions, ce que lui donne une granulation fine pour bien exprimer différentes situations de modélisation. Peut être grâce à cette granularité, la méthode semble produire de meilleurs objets abstraits que d'autres méthodes orientées objet [FTAF94].

La méthode permet l'héritage multiple, recommande l'encapsulation de l'information, encourage la conception avec un regard vers la réutilisation et utilise une vraie approche objet. Cependant quelques problèmes existent avec la méthode.

La méthode, bien que présentée par les auteurs comme une méthode complète qui couvre tout le cycle de vie du logiciel, ne traite pas les phases de test, vérification, validation et maintenance. Plusieurs décisions importantes sur la conception d'un système ne sont pas appuyées par des étapes dans la méthode et parfois ne sont même pas citées.

Bien que l'influence des langages et bases de données orientés objets sur la méthode soit claire, les opérations ne sont pas suffisamment intégrées dans le modèle statique ; cela provoque une séparation entre le modèle statique (objets) et le modèle dynamique (qui présente des détails plus orientés conception qu'orientés besoins d'analyse). L'identification du comportement des objets et du polymorphisme des opérations est réalisée trop tardivement dans la procédure de développement.

3.2.3 Méthode OOD

La méthode OOD a son origine dans les travaux initiaux de G. Booch [Boo83] basés sur les travaux de R. Abbot [Abb83]. Elle a été développée d'une manière continue et dans la version actuelle [Boo94], G. Booch a mis à jour son travail antérieur [Boo91] en ajoutant le travail d'autres auteurs comme I. Jacobson, J. Rumbaugh, P. Coad et E. Yourdon et S. Shlaer et S. J. Mellor ; il y ajoute aussi l'expérience des développeurs de logiciels. La notation a été améliorée, les diagrammes d'états sont maintenant basés sur les propositions d'Harel [Har88] et plusieurs exemples en C++ illustrent les concepts proposés.

La méthode est présentée avec les notions de concepts essentiels et de concepts avancés qui composent une série de modèles (cf. figure 3.3). Ces modèles sont utilisés selon deux points de vues différents : une vue logique/physique et une vue statique/dynamique. Ces deux points de vues, qui sont détaillés dans les sections qui suivent, sont utilisés pour représenter les produits de l'analyse et de la conception. Les diagrammes qui composent ces modèles peuvent être vus comme des projections de ceux-ci.

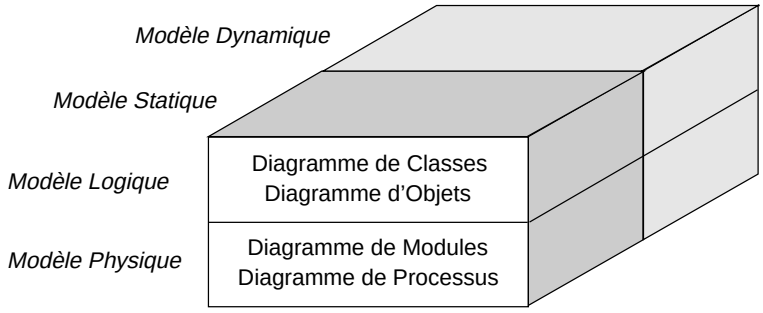


Figure 3.3 - OOD - Les Modèles de la Méthode [Boo94]

3.2.3.1 Modèles Logique/Physique

Le Modèle Logique décrit les abstractions du domaine du problème ainsi que l'architecture du système. Le Modèle Physique décrit les logiciels et matériels qui font partie du système.

La méthode propose que, pendant l'analyse, on détermine le comportement du système en utilisant des scénarios représentés dans le modèle logique par des *Diagrammes d'Objets*. Il faut aussi déterminer les rôles et les responsabilités de ces objets, et pour cela la méthode propose l'utilisation des *Diagrammes de Classes*.

Ensuite, il faut déterminer, pendant la conception, les classes du système ainsi que leurs relations ; établir la manière par laquelle les objets du système collaborent les uns avec les autres ; déterminer où les classes et les objets seront déclarés ; déterminer l'allocation des processus dans les processeurs ainsi que la manière par laquelle les différents processus d'un processeur seront organisés. Pour atteindre ces objectifs, la méthode propose l'utilisation des *Diagrammes de Classes*, des *Diagrammes d'Objets*, des *Diagrammes de Modules* et des *Diagrammes de Processus*.

3.2.3.2 Modèles Statique/Dynamique

Les systèmes ont des aspects statiques et dynamiques. Dans la méthode, cette division est représentée par les modèles statique et dynamique qui sont orthogonaux aux modèles logique et physique. Pour le modèle logique, il existe une partie statique et une partie dynamique ; pour le modèle physique il n'existe qu'une partie statique.

Le modèle statique décrit les aspects statiques du domaine du problème. Dans le modèle logique, il est représenté par les *Diagrammes de Classes* et par les *Diagrammes d'Objets*. Dans le modèle physique, il est représenté par les *Diagrammes de Modules* et les *Diagrammes de Processus*.

Le modèle dynamique décrit les aspects dynamiques du domaine du problème. Il est représenté dans le modèle logique avec les *Diagrammes de Transitions d'États* et les *Diagrammes d'Interaction*.

Les principaux concepts de chacun des diagrammes, leurs notations ainsi qu'un résumé de la procédure de développement sont donnés en Annexe C.

3.2.3.3 Évaluation - OOD

La méthode propose un grand nombre de concepts, ce qui constitue l'un de ses atouts. Ces concepts peuvent être utilisés à différents niveaux pour produire des spécifications avancées ou non. Pour cela, sont proposés des concepts de base et des concepts avancés. Le modèle développé peut être vu selon différentes perspectives (cf. figure 3.3) et, de cette manière, sa compréhension peut être améliorée.

Par rapport aux diagrammes proposés, on peut dire que les diagrammes de transitions d'états basés sur la proposition d'Harel [Har88] introduisent de puissantes techniques de modélisation.

La richesse des diagrammes, dûe à la variété de concepts, est très dirigée vers des réalisations selon des langages de programmation orientés objets. C'est pourquoi on peut dire que la méthode est véritablement orientée conception.

La méthode n'est pas proposée comme un processus de développement dans le sens où il n'y a pas une vraie sémantique pour le développement ; c'est plutôt une collection de techniques et d'heuristiques qui peuvent être utilisées lorsqu'on développe un système orienté objet [JCJO92].

3.2.4 Méthode OOAD

La méthode OOAD de J. Martin et J. Odell [MO95] est basée sur une représentation relationnelle d'un modèle de données binaire étendu et apporte des informations spécifiques et complètes dans la description du comportement des objets.

Les classes et les objets sont dérivés des concepts et des relations du domaine du problème, alors que les fonctions (applications, selon la méthode) sont développées en accord avec les notions de logique et de théorie des ensembles.

Par rapport au domaine de l'application et à la perspective selon laquelle on regarde ce domaine, un objet peut être membre de plusieurs classes et peut aussi changer d'appartenance à une classe au cours du temps.

La figure 3.4 présente les concepts et diagrammes proposés par J. Martin et J. Odell pour le développement d'un système, de l'analyse à l'implantation. Dans [MO95] les auteurs présentent le cœur de la méthode pour spécifier l'analyse du système ; les phases de conception et d'implantation introduites dans [MO95] sont développées dans [MO96].

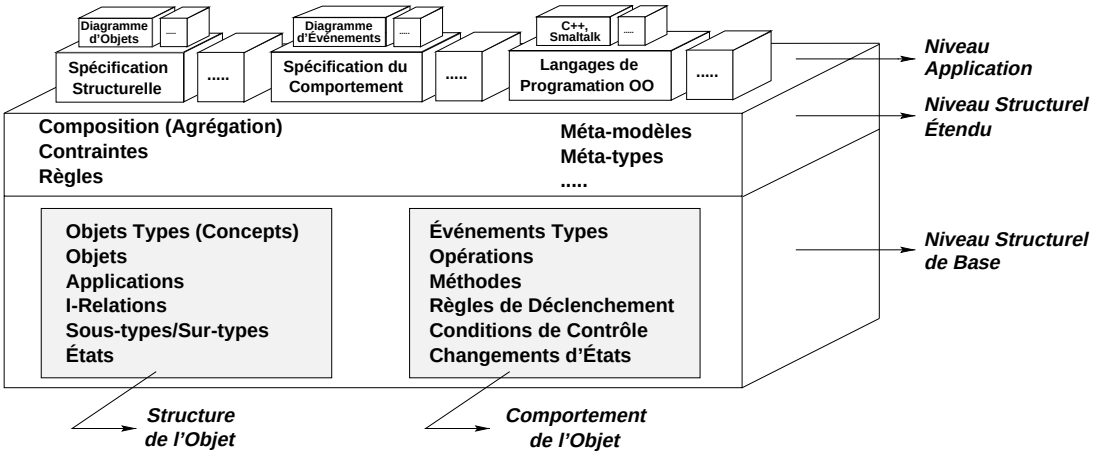


Figure 3.4 - OOAD-MO - Les Concepts, les Modèles et les Niveaux [MO95]

3.2.4.1 Évaluation - OOAD

La méthode proposée par J. Martin et J. Odell est une méthode moins proche des modèles habituels ; elle est construite à partir d'une vue particulière de l'analyse. Cette approche présente une grande cohésion entre ses techniques et fait attention à la sémantique. Elle définit un système à partir de trois vues : une *Vue Structurelle* ou de données, une *Vue Comportementale* ou dynamique et une *Vue Architecturale* ou fonctionnelle.

Pour la vue structurelle, les *Diagrammes de Relations Binaires* sont utilisés comme des diagrammes d'objets, et les *Diagrammes Entité-Relation-Attribut* comme les diagrammes des applications des objets. La méthode apporte, entre autres, une manière élégante de traiter le sous-typage, les associations dérivées et la classification dynamique (les objets peuvent changer de classe pendant leur vie).

La vue comportementale est définie avec des *Diagrammes d'Événements* qui peuvent exprimer le comportement du système dans un diagramme compact. La vue architecturale est définie avec les *Diagrammes de Flux d'Objets*. Ces diagrammes sont intéressants pour une analyse stratégique d'une entreprise. La vue architecturale montre une caractéristique frappante de la méthode : une grande préoccupation pour les applications commerciales.

La procédure de développement n'est pas vraiment présentée dans [MO95]. On trouve simplement des lignes générales pour la traduction des modèles issus de l'application de la méthode vers des langages de programmation orientés objet. Le processus complet, appelé COE "Corporate Object Engineering" est présenté dans [MO96].

3.2.5 Comparaison entre les Méthodes Orientées Objets

Étant donné le grand nombre de méthodes orientées objet diffusées, plusieurs travaux portent sur la comparaison de ces méthodes de manière à aider l'utilisateur dans le choix de l'une de ces méthodes.

Parmi ceux-ci, le travail de R. G. Fichman et C. F. Kemener [FK92] présente d'abord une analyse textuelle critique des techniques utilisées par les méthodes. Cette analyse porte sur des méthodes d'analyse structurée, des méthodes d'analyse orientée objet et des méthodes de conception orientée objet. La comparaison est faite en utilisant des tables. L'analyse textuelle critique est aussi l'outil de base employé par D. Champeaux et P. Faure [CF92]. Ils présentent les caractéristiques générales des méthodes orientées objet puis montrent les caractéristiques particulières de plusieurs méthodes en terminant par une présentation, à l'aide d'une table, des caractéristiques générales présentes ou non dans chacune des méthodes étudiées.

Dans le travail de D. E. Monarchi et G. I. Puhr [MP92], l'analyse textuelle critique est aussi employée pour mettre en évidence des aspects généraux de plusieurs méthodes puis pour les comparer globalement à l'aide de tables. Le travail se poursuit par une analyse des points forts et des points faibles du domaine de l'orientation par objets.

Une autre approche similaire est celle du rapport produit par Berard Software Engineering Inc. [BSE92a]. Après avoir cité des éléments du texte original des méthodes, celles-ci sont analysées par une présentation textuelle des concepts, des notations, des procédures de développement, de la pragmatique de développement, du support pour le génie logiciel et de la possibilité de commercialisation. Ces thèmes sont ensuite comparés à l'aide de tables.

Dans cette section, nous établissons une comparaison entre les méthodes présentées aux chapitres précédents. Cette comparaison est fortement basée sur le dernier travail cité [BSE92a] à travers l'utilisation du cadre proposé. Cependant des différences existent car ici les comparaisons sont établies pour des versions plus récentes des méthodes.

3.2.5.1 Problèmes de Comparaison de Méthodes

La comparaison de méthodes orientées objet est une tâche difficile car les méthodes sont composées de descriptions informelles où chaque méthode a sa propre définition de concepts, de techniques et de notations [HvdGB93].

Un autre point déjà cité dans la section 3.1.7, et qui rend difficile cette comparaison est la faible séparation entre les phases d'analyse et de conception. Un autre problème concerne l'orientation prise par les méthodes : quelques unes sont franchement orientées vers la conception, alors que d'autres sont orientées vers l'analyse. Ces deux aspects introduisent des difficultés parce que les mêmes concepts peuvent être présentés dans des contextes différents et les comparer n'est alors pas trivial.

Dans la suite, nous abordons la comparaison de méthodes avec des tables qui montrent les concepts présentés par chaque méthode. Nous espérons ainsi mettre en évidence les différents aspects de chaque méthode sans entrer dans le problème du choix de la meilleure méthode.

3.2.5.2 Concepts des Méthodes

Ci-dessous, on résume la façon dont les méthodes présentent les concepts d'objet (cf. table 3.1), de classe (cf. table 3.2), d'attribut (cf. table 3.3) et d'opération (cf. table 3.4), ainsi que la façon dont elles utilisent des concepts qui ont rapport avec ces concepts clés.

Méthode	Définition
OOA	Une abstraction de quelque chose du domaine d'un problème ou de son accomplissement qui reflète la capacité d'un système à maintenir les informations et/ou à interagir sur cette abstraction ; c'est une encapsulation de la valeur des attributs et de ses services exclusifs.
OOD	Une chose dont on peut faire quelque chose. Les objets ont un état, un comportement et une identité. La structure et le comportement des objets similaires doivent être définis dans leur classe commune. Les termes instance et objet sont similaires.
OMT	Un objet est une abstraction d'un ensemble des choses réelles tel que toutes les choses de cet ensemble - les instances - aient les même caractéristiques et aussi que toutes les instances soient assujetties et en accord avec un même ensemble de règles et de comportement.
OOAD	C'est quelque chose sur quoi on peut appliquer un concept : c'est une instance d'un concept. Les termes objet et instance sont interchangeables.

Table 3.1 - MOO - Définitions du Concept Objet

Méthode	Définition
OOA	Une description d'un ou plusieurs objets, à travers un ensemble uniforme d'attributs et de services ; elle peut aussi contenir une description d'une façon de créer de nouveaux objets dans cette classe. le terme classe-et-objets signifie "une classe et les objets de cette classe".
OOD	Un ensemble d'objets qui partagent une structure et un comportement commun. Les termes classe et type peuvent parfois être interchangeables.
OMT	Une classe d'objets décrit un groupe d'objets avec des propriétés similaires (les attributs), un comportement commun (les opérations), et des relations communes avec d'autres objets et une sémantique commune.
OOAD	Une implantation d'un concept ou d'un objet type. C'est la construction la plus employée pour définir des types abstraits de données dans les langages de programmation orientés objet ; en mathématique, la définition d'une classe est similaire à celle d'un ensemble et c'est cette définition qui est utilisée par les langages de programmation orientés objet.

Table 3.2 - MOO - Définitions du Concept Classe

Méthode	Définition
OOA	Une information d'état qui a une valeur propre pour chaque objet d'une classe ; cette valeur ne peut être manipulée que par les services de cette classe.
OOD	Exprime une propriété particulière d'une classe ; il faut que les attributs aient un nom, appartiennent à une classe et de manière optionnelle ils peuvent avoir une valeur par défaut.
OMT	Valeur d'une donnée maintenue par un objet dans une classe ; chaque attribut a une valeur propre à chaque instance d'objet dans une classe d'objets.
OOAD	Une association identifiable qu'un objet a avec un autre objet ou avec un ensemble d'objets représenté dans son objet type. Normalement les attributs s'appliquent vers des types d'objets montrables comme un numéro et une chaîne de caractères. Chaque attribut est une instance d'un attribut type.

Table 3.3 - MOO - Définitions du Concept Attribut

Méthode	Définition
OOA	Un service est un comportement spécifique dont un objet est responsable.
OOD	Un service accompli par les objets d'une classe. Une action qu'un objet réalise sur un autre objet pour obtenir une réaction. Les termes méthode et opération sont normalement interchangeables.
OMT	Une opération est une fonction ou une transformation qui peut être appliquée à un ou plusieurs objets d'une classe. Tous les objets d'une classe partagent les mêmes opérations.
OOAD	C'est un processus qui peut être requis comme une unité et qui exécute, pas à pas, deux fonctions de base : l'interrogation d'un objet ou le changement de l'état d'un objet.

Table 3.4 - MOO - Définitions du Concept Opération

Afin de pouvoir "Gérer la Complexité" [SM92] c'est-à-dire, organiser la présentation des modélisations, les méthodes introduisent de nouveaux concepts (cf. table 3.5). On peut remarquer que les méthodes de G. Booch et de J. Rumbaugh vont un peu plus loin dans ce domaine.

Méthode	Concept Utilisé
OOA	*Sujet* : une manière de contrôler la partie du système qui selon la préoccupation du lecteur peut être vue, en guidant le lecteur (analyste, spécialiste dans le domaine du problème, gérant, client) à travers un modèle vaste et complexe. Les sujets sont utiles aussi pour organiser le travail initial pour de grands projets.
OOD	*Catégories de Classes* : utilisées pour fractionner le Modèle Logique d'un système. Les catégories de classes sont composées de classes et d'autres catégories de classes.
	Sous-système : concept utilisé pour partager le Modèle Physique d'un système. Un sous-système est un agrégat qui contient des modules et d'autres sous-systèmes.
OMT	*Sous-système* : c'est un sous-ensemble, une partie d'un modèle entier ; il existe comme un outil d'organisation qui n'a aucun sens sémantique. Les sous-systèmes peuvent être emboîtés et les sous-systèmes de plus bas niveaux sont appelés *Modules*. Chaque classe a son module "maison" où ses détails son déclarés.
	Diagramme d'Interface de Sous-Systèmes : c'est la représentation employée pour présenter les sous-systèmes avec leurs classes et relations publiques.
OOAD	La méthode n'introduit pas de nouveaux concepts pour l'administration de la complexité.

Table 3.5 - MOO - Gestion de la Complexité

3.2.5.3 Notations des Méthodes

Toutes les méthodes ont leur propre notation pour modéliser un système. Nous présentons maintenant les notations employées par chaque méthode pour représenter les concepts du paradigme objet. La table 3.6 présente les concepts statiques et la table 3.7, les concepts dynamiques.

Pour les concepts statiques, on analyse comment les méthodes représentent les concepts d'agrégation (composition d'objets), spécialisation (spécification d'un objet comme généralisé ou spécialisé à partir d'un autre), de communication (envoi et réception des messages), et d'interface de modules (implantation physique des objets).

Méthode	Agrégation	Spécialisation	Communication	Interface de Module
OOA	Composé-Composant	Gén-Spéc	Message	
OOD	Composition	Héritage	Diagramme d'Interaction	Module
OMT	Agrégation	Héritage	Scénario	
OOAD	Composition	Héritage	Scénario	

Table 3.6 - MOO - Concepts Statiques

Pour les Concepts Dynamiques, on analyse comment les méthodes représentent les changements d'états et la temporisation, concepts très importants pour modéliser le comportement d'un système.

Méthode	Changement d'États	Temporisation
OOA	Diagramme d'États	
OOD	Diagramme de Transition d'États	Diagramme d'Interaction
OMT	Diagramme d'États	Traceur d'Événements
OOAD	Changements d'États	Spécification basée sur des Scénarios

Table 3.7 - MOO - Concepts Dynamiques

3.2.5.4 Procédure de Développement des Méthodes

Dans ce paragraphe, nous présentons comment les documentations du cycle de vie sont prises en compte par chaque méthode, quels aspects du cycle de vie sont couverts et la démarche sous-tendue par chaque méthode. La table 3.8 montre si les documentations générées sont mentionnées en utilisant la cotation suivante :

- **0** : non mentionné ;
- **1** : mentionné mais il n'y a pas des détails ;
- **2** : mentionné et une définition est donnée ;
- **3** : mentionné, une définition est donnée, ainsi qu'un exemple ;
- **4** : mentionné, une définition est donnée, ainsi qu'un exemple, et le processus est défini ;
- **5** : mentionné, une définition est donnée, ainsi qu'un exemple, et le processus et l'heuristique sont définis.

Méthode	Spécification des Besoins	Spécification du Projet	Cas de Test	Spécification d'Objet/ Classe	Spécification du Sous-Système
OOA	2	2	0	5	0
OOD	1	5	2	4	4
OMT	2	2	0	5	3
OOAD	0	0	0	0	0

Table 3.8 - MOO - Documentation du Cycle de Vie

La table 3.9 présente les aspects du cycle de vie couverts par les méthodes en utilisant la même cotation que ci-dessus.

Méthode	Modélisation d'une Entreprise	Analyse du Domaine	Analyse de Besoins	Projet	Implantation	Test
OOA	0	1	5	5	3	3
OOD	0	4	2	5	4	2
OMT	0	0	5	5	3	2
OOAD	1	0	3	5	1	0

Table 3.9 - MOO - Phases du Cycle de Vie

La table 3.10 détermine, pour chaque méthode, un degré de précision selon différents aspects du processus de développement d'un système. Cette précision est définie selon l'échelle suivante : *1* signifie aucune ou peu de règles et *3* signifie cinq règles ou plus.

L'échelle utilisée n'est pas uniforme : si deux méthodes ont une même note, 2 par exemple, elles peuvent présenter les concepts d'une façon plus ou moins complète l'une par rapport à l'autre.

Méthode	Identification des Classes	Identification des Opérations	Placement des Opérations	Identification des Sous-Systèmes	Identification des États
OOA	3	3	0	0	1
OOD	3	3	3	3	3
OMT	3	3	1	2	3
OOAD	3	3	1	0	3

Table 3.10 - MOO - Phases du Cycle de Vie

3.2.5.5 Réflexion sur la Comparaison de Méthodes

Toutes les méthodes décrites utilisent les concepts présents dans le paradigme objet. Le nombre de concepts proposés par chaque méthode pour la modélisation d'un système est très différent. Alors que P. Coad et E. Yourdon proposent environ dix concepts, ce nombre est plus important dans les autres méthodes. Bien qu'un plus grand nombre de concepts puisse faciliter la modélisation, il peut aussi la compliquer. Les présentations proposées par G. Booch et J. Rumbaugh, avec la notion de concepts de base et de concepts avancés, essaient de résoudre le problème causé par le grand nombre de concepts proposés introduits par les méthodes.

Exceptée la méthode de G. Booch, toutes les autres sont orientées plutôt vers l'analyse. La méthode de G. Booch est plus orientée vers l'implantation et, avec ses Diagrammes de Modules, permet une représentation de cette implantation.

La méthode de J. Martin et J. Odell s'inspire d'un modèle relationnel. Elle introduit une approche différente des techniques plus traditionnelles en utilisant un "modèle objet complet" (tous les concepts proposés dans la méthode sont dérivés d'un Objet Type appelé Concept).

La comparaison réalisée confirme l'affirmation de Hong [HvdGB93] concernant les problèmes pour comparer les méthodes composées essentiellement de descriptions informelles spécifiques pour les concepts, les techniques et les notations.

La comparaison à l'aide de tables offre un cadre suffisant pour une première analyse comparative entre différentes méthodes. Cependant, pour une analyse plus profonde, où une comparaison plus rigoureuse des concepts proposés est envisageable, d'autres méthodes de comparaison sont souhaitables.

L'utilisation d'un méta-modèle comme outil de comparaison est l'une des manières pour s'attaquer au problème de comparaison des méthodes avec un regard plus rigoureux. Dans ce travail nous proposons un méta-modèle (cf. section 4.2) qui peut être utilisé dans la comparaison des méthodes orientées objet. Dans ce méta-modèle, chaque représentation d'une vue est composée d'une partie graphique comme dans [HvdGB93, EG93], associée à une partie formelle basée sur le langage de spécification Z [Spi89] et des commentaires complémentaires ou explicatifs en langue naturelle.

L'idée d'utiliser un langage formel a déjà été employée par P. Atzeni et R. Torlone [AT93] qui proposaient de décrire les modèles à travers un langage structuré pour établir un méta-modèle textuel permettant de gérer et de traduire les modèles. Une autre proposition dans cet axe a été faite par A. H. M. Hofstede, H.A. Proper et T. P. Weide [tHPvdW93] ; ils ont proposé un langage pour la description textuelle de modèles de systèmes d'information.

Dans la section 5.3.2, nous présentons un cadre qui utilise le méta-modèle proposé dans la comparaison de modèles de méthodes, ainsi qu'une analyse qui porte sur des travaux qui utilisent cette approche.

Dans les chapitres suivants, nous présentons nos propositions en matière de méta-modèle et d'atelier de modélisation. Nous présentons aussi des exemples où l'atelier développé est utilisé.

3.3 Conclusions

D'une manière générale, les plus grands bénéfices de l'utilisation du paradigme objet peuvent être exprimés par les facilités qui sont offertes pour la réutilisation de résultats de travaux antérieurs. Ce paradigme est bien adapté à modéliser le monde réel à travers les abstractions. L'ensemble analyse, conception et implantation est extrêmement conséquent. Les trois activités utilisent les mêmes concepts de base et génèrent donc un processus de développement homogène. Avec l'utilisation de l'encapsulation, les systèmes produits sont plus résistants aux changements et aux évolutions du système.

Le paradigme objet étant utilisé à travers les méthodes, on peut vouloir aborder la question d'orientation de celles-ci vers l'analyse ou la conception. Un travail qui proposait une solution à ce problème avec le couplage de différentes méthodes a été fait par D. Coleman [CAB+93, CJD93] avec la méthode Fusion. Cette méthode est présentée comme une *Méthode Systématique pour le Développement des Logiciels Orientés Objet*. Elle intègre des aspects de la méthode de G. Booch orientés vers l'implantation, des aspects de la méthode de J. Rumbaugh orientés vers l'analyse, des aspects de la méthode CRC orientés vers la communication entre objets et quelques aspects de la méthode de I. Jacobson (les cas d'utilisation). La méthode Fusion est un processus pour conduire un développeur de l'analyse des besoins jusqu'au code. Les phases, ainsi que les diagrammes utilisés dans ces phases, sont présentés dans la table 3.11 [PLA96b].

Phase	Modèles	Caractéristiques
Besoins	Modèle de Cas d'Utilisation	un par acteur.
	Diagramme de Scénarios	un par cas d'utilisation avec des flux d'événements complexes.
Analyse	Modèle d'Objets : - Diagrammes d'Objets	 plusieurs par système
	Interface du Système : - Modèle d'Objets du Système - Diagramme de Scénarios - Graphe d'Interface du Système - Dictionnaire de Données	 plusieurs par système un ou plus par scénario un par système un par système
	Modèle d'Interface : - Modèle d'Opérations - Diagramme de Scénarios	 un par système un par scénario
Conception	Graphe d'Interaction d'Objet Graphe de Visibilité Description de Classes Graphe d'Héritage	un par chaque opération un par classe un par classe un par classe avec héritage
Implantation	Diagramme d'États Génération du Code	un ou plus par cycle de vie

Table 3.11 - Phases et Modèles de la Méthode Fusion

Actuellement, G. Booch, I. Jacobson et J. Rumbaugh se sont unis dans la société Rational Software Corporation pour proposer une nouvelle méthode [Rat97] qui prend en compte des éléments de chacune de leurs méthodes (respectivement OOD, OOSE et OMT), tout en regardant des concepts proposés dans d'autres méthodes. De cette manière, est né le *Langage de Modélisation Unifié* - LMU ("Unified Modeling Language"-UML) [Mul97] pour le développement de systèmes (et non seulement logiciels) orientés objet. Le LMU est une

méthode de troisième génération pour spécifier, visualiser et documenter les artefacts d'un système orienté objet en construction [Rat96a]. Par ses caractéristiques, il est particulièrement approprié à des systèmes temps réel. La méthode LMU est basée sur trois modèles présentés dans la table 3.12. Par ses caractéristiques de méthode nouvelle et en développement, les considérations de cette table sont fortement basées sur [Rat96b].

Modèle	Diagramme	Objectif
Classes et Objets	Classes	présenter les classes les plus importantes du système ainsi que leurs relations.
	Collaboration	présenter des instances spécifiques des classes qui coopèrent pour accomplir un objectif.
Dynamique	Cas d'Utilisation	décrire la manière par laquelle un système peut être utilisé ; ils fournissent une vue de haut niveau des fonctionnalités prévues pour le système.
	Scénarios	ils sont des instances des Cas d'Utilisation et peuvent être représentés de deux manières : les *Diagrammes de Séquence* qui font ressortir l'aspect séquentiel des échanges de messages et les *Diagramme de Collaboration* qui font ressortir les aspects structuraux des échanges de messages.
Comportemental	Machines d'États	elles sont représentées avec une extension des diagrammes d'états d'Harel [Har87].

Table 3.12 - Modèles et Diagrammes de la Méthode LMU

La méthode propose aussi des concepts avancés pour la modélisation des systèmes, lesquels peuvent être utiles dans des situations spécifiques. Ces concepts sont les suivants :

– *Stéréotype* : c'est un concept utilisé pour la méta-classification d'un élément dans la méthode. Plusieurs types de stéréotypes sont proposés et d'autres peuvent être ajoutés par l'utilisateur ;

– *Conditionnement Logique pour Gros Systèmes* ("Large-Scale Logical Packaging") : c'est un concept utilisé pour le groupement des entités à travers les paquets ;

– *Modélisation Physique* : c'est un concept utilisé à travers les Diagrammes d'Utilisation qui présentent l'organisation du matériel et l'attachement du logiciel à ceux-ci.

Un autre aspect des méthodes peut être remarqué : leur manière d'introduire et d'utiliser les concepts. On peut parler des méthodes dites "pures", c'est-à-dire, n'utilisant que des concepts orientés objets pendant tout le développement d'un système. Elles introduisent des notations et des approches différentes des techniques plus traditionnelles. De l'autre côté, se situent les méthodes "Orientées Objets", qui utilisent de nouvelles idées appliquées à de vieux concepts [MP92]. Entre ces deux extrêmes, se situent des méthodes qui essaient d'établir un couplage entre le paradigme objet et le cycle de vie traditionnel.

Chapitre 4

Un Nouvel Atelier de Modélisation

4.1 Les Besoins des Nouveaux Ateliers

C. Rolland [Rol92] introduit un certain nombre de points clefs qui peuvent rendre l'utilisation des AGL plus effective. On exploite ci-dessous quelques uns de ces points. On peut parler, par exemple, du besoin d'intégration de ces outils : les données doivent être intégrées dans un dictionnaire unique et des modifications sur une modélisation doivent se répercuter sur toutes les données présentes. Les outils doivent être indépendants des méthodes ; cette indépendance peut se traduire, par exemple, par l'utilisation d'un méta-modèle qui décrit les méthodes. Les nouveaux AGL doivent offrir des possibilités de restructuration, de rétro-ingénierie et de ré-ingénierie. Ils ne peuvent négliger ni l'étape d'analyse des besoins, ni le guidage du processus de modélisation et ils doivent employer l'approche objet.

Le processus de modélisation des SI passe toujours par l'utilisation de modèles qui sont basés sur des approches variées : semi-formelle, formelle et informelle. Il existe alors une relation directe entre le type d'approche utilisée et l'AGL.

Les méthodes semi-formelles et informelles ont déjà fait la preuve de leur utilité à travers les différents modèles des méthodes qui les utilisent pour modéliser des SI. Leur utilisation est même indispensable au processus de modélisation des SI. Les méthodes formelles, en raison de l'amélioration de la précision des modélisations (vérifications de cohérence et de complétude) qu'elles apportent, sont de plus en plus présentes dans la modélisation des SI. L'analyse faite par J. P. Bowen et M. C. Hinchey [BH95] (présentée à la section 2.4) suggère une utilisation conjointe de ces trois approches dans la modélisation des SI. Cette approche est au centre de nos propositions.

Une analyse de la raison pour laquelle les Ateliers de Génie Logiciel sont “utilisés avec modération” est présentée par J. Iivari [Iiv96]. Elle montre que l’un des problèmes concerne le coût de ces outils ainsi que la formation nécessaire à leur utilisation. Un autre problème est lié à la complexité de tels outils ; cette complexité qui peut conduire à une certaine résistance à leur utilisation due à une mauvaise compréhension des avantages de l’outil. Par ailleurs, cette analyse montre qu’un soutien solide des chefs de projet, soit pour l’utilisation, soit pour l’entraînement, conduit à une augmentation de l’utilisation des AGL. Dans ce travail, une conclusion très importante, corroborée par d’autres auteurs, est liée aux avantages induits par l’utilisation des AGL : ils augmentent la productivité et la qualité des systèmes d’information et du processus de développement d’un logiciel. Cette conclusion est aussi l’une des raisons qui conduisent les entreprises à “imposer” l’utilisation de tels outils car cet argument est généralement accepté par les utilisateurs.

Une analyse de ces deux travaux [Rol92, Iiv96] permet de dire que, même s’il existe des résistances internes dans les entreprises à l’utilisation des AGL, d’excellents résultats sont obtenus lorsque cette utilisation devient effective. Ces résultats peuvent encore être améliorés en faisant évoluer les caractéristiques des ces outils.

Dans le cadre notre travail, nous retenons, parmi les points cités par C. Rolland, les nouveaux besoins quant aux possibilités d’utilisation/ré-utilisation de “différentes approches différentes” à travers un processus de méta-modélisation.

La prochaine section présente un méta-modèle qui cherche à répondre à ces besoins. Nous décrivons un atelier qui utilise ce méta-modèle ainsi que les différents niveaux auxquels il peut être utilisé. Nous présentons aussi une analyse des relations possibles entre fragments de différents types de modélisation, ainsi qu’un canevas de modélisation pour guider ce processus.

4.2 Le Méta-Modèle Proposé

Dans le but d’atteindre les caractéristiques attendues (cf. section 4.1) du nouvel atelier de génie logiciel, nous proposons l’utilisation d’un méta-modèle. Ce méta-modèle intègre les approches informelle, semi-formelle et formelle (cf. figure 4.1) et peut être utilisé à deux niveaux :

– *Méta-Modélisation* : représentation de modèles utilisés par des méthodes de modélisation de systèmes ;

– *Multi-Modélisation* : représentation des modèles issus de l'utilisation de ces méthodes pour la spécification d'un système.

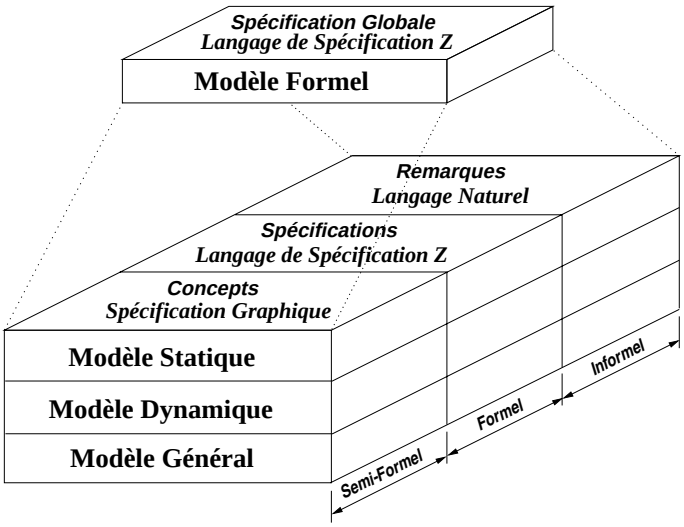


Figure 4.1 - Architecture du Méta-modèle

Il faut noter que la multi-modélisation d'un SI se situe à différents niveaux :

– plusieurs modèles à coordonner selon plusieurs facettes (statique, fonctionnelle, évolution, ...) pour appréhender la complexité d'un SI ;

– plusieurs modèles à coordonner selon plusieurs paradigmes (formel, semi-formel et informel).

L'emploi du méta-modèle est différent selon l'utilisation désirée. Une description de cet emploi, dans chaque type d'utilisation, est présentée à la section 4.4.

Le méta-modèle proposé utilise le concept de vues composées d'un ou plusieurs *schémas*. Le nombre de schémas d'une vue du méta-modèle est déterminé par la complexité de la méthode ou du modèle représenté.

Les sous-sections suivantes présentent chacune des vues de la figure 4.1 et expliquent le concept de schémas. Le méta-modèle constitue le noyau sur lequel l'atelier de modélisation est bâti. C'est autour de ce méta-modèle que sont coordonnés les composants de l'atelier décrits dans la section 4.3.

4.2.1 Vue Modèle Formel

La vue Modèle Formel est employée pour la déclaration formelle de tous les concepts de la spécification. Cette vue rend compatible le méta-modèle avec la syntaxe du langage formel utilisé. La notation utilisée dans cette vue dépend du niveau auquel l'atelier est utilisé (cf. section 4.4). Dans la section 4.2.5.1, sans rentrer dans les détails de cette utilisation à différents niveaux, nous présentons les deux types de spécifications globales utilisées : l'une pour la méta-modélisation et l'autre pour la multi-modélisation.

4.2.2 Vue Modèle Statique

La vue Modèle Statique est utilisée pour la représentation des concepts et des relations entre ces concepts figurant dans les méthodes pour donner une image statique d'un système. Ainsi, au niveau méta-modélisation, on rencontre des concepts de base d'une méthode (modèle d'objets avec ses composants classe, objet, etc), les relations entre ces concepts de base (instanciation, caractérisation statique, etc.) et plusieurs types de relations spécifiques (héritage, composition, etc.). Au niveau multi-modélisation, ce sont des instances des concepts de base décrits au niveau méta-modélisation qui sont présents. Dans la section 4.2.5.2, après avoir introduit la notation utilisée dans cette vue, nous présentons un exemple.

4.2.3 Vue Modèle Dynamique

La vue Modèle Dynamique contient les schémas employés pour représenter les concepts et relations entre ces concepts utilisés par une méthode pour la modélisation de la dynamique d'un système. Ainsi, par exemple, au niveau méta-modélisation, les modèles des diagrammes d'états ou des diagramme d'interactions sont introduits. Au niveau multi-modélisation des instances des concepts présents au niveau méta-modélisation sont utilisées. Dans la section 4.2.5.2, après avoir introduit la notation utilisée dans cette vue, nous présentons un exemple.

4.2.4 Vue Modèle Général

La vue Modèle Général est utilisée pour la représentation synthétique des relations entre des concepts des Modèles Statiques et Dynamiques. Cette vue introduit des relations sémantiques entre les composants des modèles de méthodes. Ces relations sont utilisables au niveau multi-modélisation. Une relation peut, par exemple, représenter le fait qu'une `classe`

(un concept d'un Modèle Statique) est toujours attachée à un `diagramme d'états` (un concept d'un Modèle Dynamique).

4.2.5 Les Schémas

Les vues du méta-modèle proposé utilisent l'approche des schémas avec différents niveaux de détails : les vues sont composées d'un ou plusieurs schémas.

La vue Modèle Formel, par sa caractéristique de spécification formelle globale, n'est composée que d'un schéma unique, le schéma Spécification Globale, qui représente la vue elle-même (cf. figure 4.1).

Les vues Modèle Statique, Modèle Dynamique et Modèle Général sont composées des schémas de trois types différents : Concepts, Spécifications et Remarques (cf. figure 4.1).

L'intégration des approches informelles, semi-formelles et formelles dans le méta-modèle est réalisée par de tels schémas. Ces schémas implantent l'intégration des approches dans les vues. Les quatre types de schémas sont décrits dans les sous-sections suivantes.

4.2.5.1 Spécification Globale

Le schéma Spécification Globale (*S_Global*) introduit les concepts formels qui seront utilisés dans la spécification. Le schéma S_Global est en réalité un squelette, où les types, les relations entre ces types ainsi que les axiomes utilisés dans une spécification sont déclarés. Ces types, relations et axiomes sont utilisés lorsque ce squelette est complété automatiquement, au fur et à mesure que la modélisation semi-formelle est construite.

La description de la cardinalité des relations entre types est implantée avec l'utilisation de fonctions et de relations mathématiques. Ainsi pour une relation donnée par $A \leftrightarrow B$, on représente sa cardinalité par $XY : WZ$, où XY sont les cardinalités minimales et maximales sur le domaine et WZ les cardinalités minimales et maximales sur l'image.

Dans la figure 4.2 nous présentons toutes les cardinalités possibles pour les relations entre les types du schéma S_Global ; la notation de cardinalité utilisée est celle de la plus part des méthodes orientées objet. Avec une approche basée sur les cardinalités, nous pouvons en extraire vingt cinq types de relations possibles. La première ligne de la figure 4.2 introduit quinze de ces types (cinq pour chaque cadre). Les lignes 2, 3 et 4 de la figure 4.2 introduisent dix types de relations.

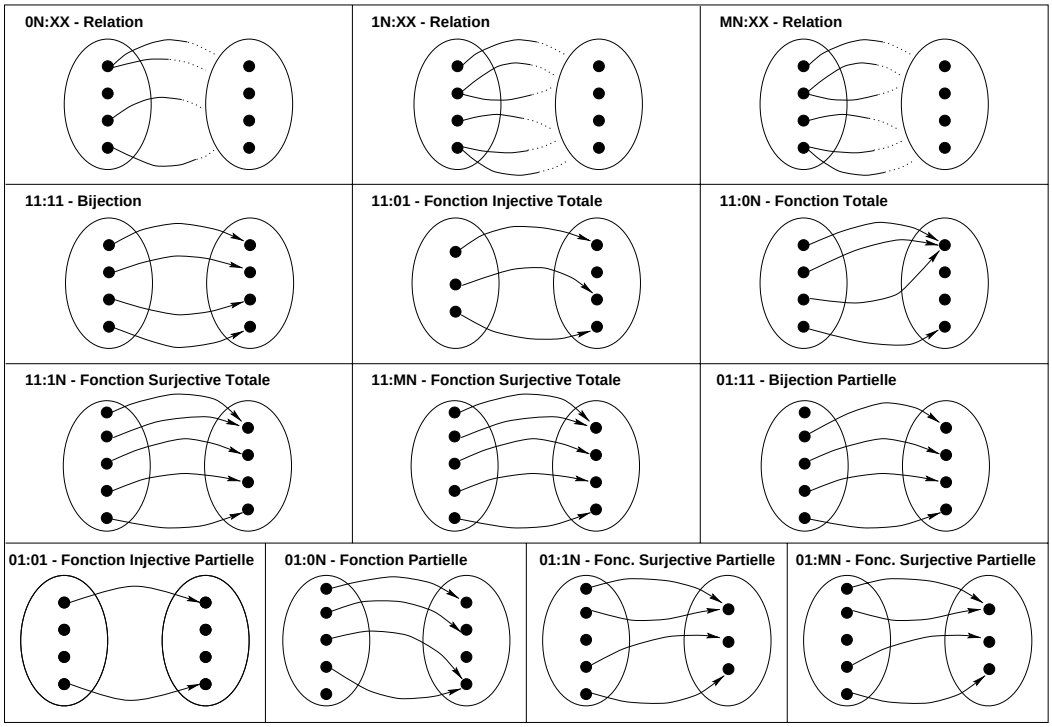


Figure 4.2 - Cardinalités, Relations et Fonctions

Une autre approche, centrée sur la représentation fonctionnelle des associations binaires, a été réalisée par P. Dardailler, C. Delobel et J. P. Giraudin [DDG85]. Dans cette étude, les combinaisons possibles d'association entre deux ensembles sont réduites à seize, une représentation syntaxique pour les associations est introduite ainsi qu'une proposition pour la représentation de la "richesse" des associations est dégagée. Par richesse, les auteurs entendent la représentation de l'aspect monovalué ou multivalué ainsi que l'aspect partiel ou total des associations. La richesse est associée à un nombre : la valeur la plus petite indique l'association la plus contrainte.

Prenant en compte cette étude, dans la table 4.1 nous présentons alors, avec une autre organisation, les cardinalités possibles pour les relations entre les types du schéma S_Global. Elles sont présentées en ordre décroissant de leur richesse [DDG85]. Comme le langage formel utilisé est Z, dans cette table nous montrons les types de relations existants dans Z, ainsi que les cardinalités associées à chacun des types de relation. Comme dans l'atelier développé (cf. chapitre 5) nous utilisons une notation textuelle pour représenter les relations et les fonctions. Dans la table 4.1, deux notations sont présentées : celle qui utilise les symboles graphiques offerts par LaTeX et la notation textuelle. Nous pouvons remarquer que deux fonctions différentes peuvent avoir la même richesse.

Richesse	Cardinalité	Type	Notation	
			LaTeX	Textuel
1	11 : 11	Bijection	$\rightarrowtail\!\!\!\!\twoheadrightarrow$	>->>
2	11 : 01	Fonction Injective Totale	$\rightarrowtail$	>->
	01 : 11	Bijection Partielle	$\nrightarrow\!\!\!\!\twoheadrightarrow$	>+>>
3	11 : 1*N*	Fonction Surjective Totale	$\twoheadrightarrow$	->>
	11 : *MN*	Fonction Surjective Totale	$\twoheadrightarrow$	->>
4	11 : 0*N*	Fonction Totale	$\rightarrow$	–>
5	01 : 01	Fonction Injective Partielle	$\nrightarrow\!\!\!\!\rightarrowtail$	>+>
6	01 : 1*N*	Fonction Surjective Partielle	$\nrightarrow\!\!\!\!\twoheadrightarrow$	+>>
	01 : *MN*	Fonction Surjective Partielle	$\nrightarrow\!\!\!\!\twoheadrightarrow$	+>>
7	01 : 0*N*	Fonction Partielle	$\nrightarrow$	+->
8	0*N* : *XX*	Relation	$\leftrightarrow$	<->
	1*N* : *XX*	Relation	$\leftrightarrow$	<->
	MN : *XX*	Relation	$\leftrightarrow$	<->

Table 4.1 - Relations, Fonctions Z et Cardinalités

Ils existent deux types différents de schémas S_Global : un pour les méta-modélisations et l'autre qui est utilisé dans les multi-modélisations.

Le schéma S_Global utilisé lors de la méta-modélisation est présenté dans la figure 4.3.

$[\, MODÈLE, CONCEPT, RELATION, PROPRIÉTÉ \,]$

SG_MétaModélisation

$nom\_modèle : \mathbb{P}\, MODÈLE$
$nom\_concept : \mathbb{P}\, CONCEPT$
$nom\_relation : \mathbb{P}\, RELATION$
$nom\_propriété : \mathbb{P}\, PROPRIÉTÉ$
$desc\_liens : domaine < type\_lien > image$

---

prédicat

Figure 4.3 - Schéma S_Global - Méta-Modélisation

Le schéma S_Global au niveau méta-modélisation utilise Z [Spi89] comme langage formel. La figure 4.3 correspond au squelette proposé par ce schéma. On y trouve les types de base `MODÈLE, CONCEPT, RELATION` et `PROPRIÉTÉ` qui sont les types utilisés dans la spécification. Ces types de base sont utilisés au niveau formel par les schémas S_Spécifications (cf. section 4.2.5.3). Les concepts semi-formels utilisés dans les schémas S_Concepts

sont la représentation graphique de ces types (cf. section 4.2.5.2). Les relations entre ces types sont aussi représentées ; le `type_lien` de la figure 4.3 peut prendre l'une des possibilités présentées dans la table 4.1.

Dans ce travail, la *multi-modélisation* s'applique toujours à des systèmes orientés objets ; pour cette raison, le schéma S_Global d'une multi-modélisation est du type de celui présenté dans la figure 4.4.

$[CLASSE]$

$$
\begin{array}{|l}
Classe : \mathbb{P}\, CLASSE \\
classe : CLASSE \\
subSuper : CLASSE \leftrightarrow CLASSE \\
compnt : CLASSE \leftrightarrow CLASSE \\
\hline
subSuper^{+} \cap \mathrm{id}\, classe = \varnothing \\
\mathrm{dom}\, subSuper \subset Classe \\
\mathrm{ran}\, subSuper \subset Classe \\
compnt^{+} \cap \mathrm{id}\, classe = \varnothing \\
\mathrm{dom}\, compnt \subset Classe \\
\mathrm{ran}\, compnt \subset Classe
\end{array}
$$

Déclaration des Relations

$$
\begin{array}{|l}
\hline
ClassesRelations \\
nom\_classe : CLASSE \\
desc\_relations : domaine < type\_relation > image \\
\hline
prédicat \\
\hline
\end{array}
$$

Déclaration des Structures d'Héritage
Structure d'Héritage : nom_structure

$$
\begin{array}{|l}
surClasse, sousClasse : CLASSE \\
\hline
(surClasse, sousClasse) \in subSuper
\end{array}
$$

Déclaration des Structures d'Agrégation
Structure d'Agrégation : nom_structure

$$
\begin{array}{|l}
composé, composant : CLASSE \\
\hline
(composé, composant) \in compnt
\end{array}
$$

Figure 4.4 - Schéma S_Global - Multi-Modélisation

Le Modèle Formel, donné par le schéma S_Global de la figure 4.4, a comme but, dans un processus de multi-modélisation, de présenter une vue formelle globale du système. C'est ici que des définitions sont introduites : déclaration des structures d'héritage et d'agrégation et déclaration des classes et des relations entre elles. Le squelette présenté dans la figure 4.4 est utilisé par l'éditeur syntaxique (cf. section 5.2.1.3) lors de la construction automatique du Modèle Formel.

Dans le schéma S_Global utilisé dans la multi-modélisation, les structures d'héritage et d'agrégation sont représentées selon la proposition initiale de A. Hall [Hal90a, Hal94] rappelée ci-dessous.

A. Hall [Hal94] a proposé de déclarer des structures d'héritage par l'utilisation d'une relation entre classes donnée par la description axiomatique suivante :

$[\,CLASSE\,]$

$$
\begin{array}{|l}
Classe : \mathbb{P}\, CLASSE \\
classe : CLASSE \\
subSuper : CLASSE \leftrightarrow CLASSE \\
\hline
subSuper^{+} \cap \mathrm{id}\, classe = \varnothing \\
\mathrm{dom}\, subSuper \subset Classe \\
\mathrm{ran}\, subSuper \subset Classe
\end{array}
$$

La relation subSuper donne la sur-classe immédiate d'une classe. La fermeture transitive déclarée pour cette relation exprime le fait que dans une structure d'héritage, une classe ne peut pas être la sur-classe d'elle-même. En utilisant cette description axiomatique, la définition d'une structure d'héritage est présentée de la manière suivante :

$$
\begin{array}{|l}
surClasse, sousClasse : CLASSE \\
\hline
(surClasse, sousClasse) \in subSuper
\end{array}
$$

Pour pouvoir utiliser cette proposition de A. Hall, nous introduisons les définitions axiomatiques du début de la figure 4.4. Dans le schéma S_Global, les structures d'héritage sont déclarées dans la partie `Déclaration des Structures d'Héritage`. Les structures d'agrégation sont introduites de même manière dans le schéma S_Global par la partie `Déclaration des Structures d'Agrégation`. La représentation des relations

entre objets est faite en utilisant les relations Z. Les relations entre classes sont présentées alors dans la partie `Déclaration des Relations` avec les schémas `ClassesRelations` du schéma S_Global, plus particulièrement avec `desc_relations` pour lequel le `type_relation` est la représentation de l'un de types définis à la table 4.1.

4.2.5.2 Concepts - Les Schémas Semi-Formels

C'est avec les Schémas Concepts (*S_Concepts*) que le méta-modèle proposé utilise l'approche semi-formelle. Cette approche est essentiellement graphique ; sa notation dépend du niveau auquel l'atelier est utilisé. Au niveau méta-modélisation une notation (cf. figure 4.5) basée sur les diagrammes E-R-A est utilisée pour la représentation des méthodes.

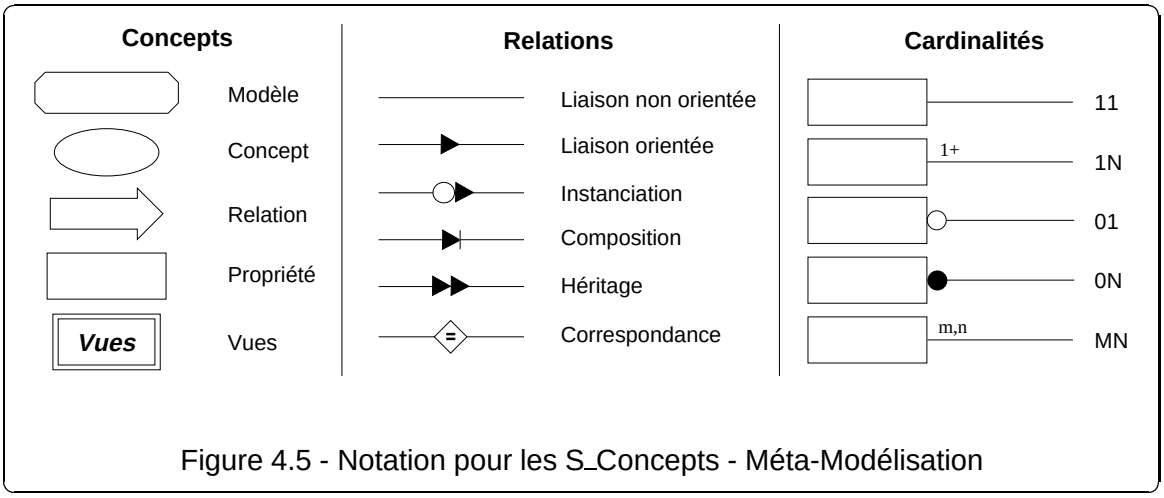


Figure 4.5 - Notation pour les S_Concepts - Méta-Modélisation

Les **Concepts** de la figure 4.5 représentent les composants graphiques avec lesquels le modèle semi-formel est construit. Ce sont les représentations graphiques des types de base de la vue Modèle Formel présentés dans la figure 4.3. Les concepts `Modèle` sont utilisés pour la représentation des modèles proposés par les méthodes (ex : modèle d'objets, modèle fonctionnel, etc.). Les concepts `Concept` permettent de représenter des concepts des méthodes (classe, état, etc.). Les concepts `Relation` modélisent les différents types de liens existants entre des composants d'une méthode. Les concepts `Propriété` sont utilisés pour représenter des caractéristiques possédées par les trois autres concepts. Finalement les concepts `Vues` sont utilisés pour introduire la notion de vues dans une modélisation. Il faut noter que le terme propriété n'est pas utilisé ici ni dans le sens des spécifications algébriques ni dans l'acception des textes mathématiques (axiomes). Il désigne des caractéristiques des concepts de type `Modèle, Concept` et `Relation`.

Les **Relations** de la figure 4.5 sont utilisés pour représenter l'aspect sémantique interne des différents liens possibles entre les Concepts. Ainsi les liens de type `Liaison non orientée` sont utilisés pour représenter des liens où le sens de parcours du lien n'a pas d'importance ; un exemple est donné par les liens L_1 et L_2 de $C_1 \overset{L_1}{\longleftrightarrow} R_1 \overset{L_2}{\longleftrightarrow} C_2$ où une relation (R_1) est déclarée entre deux concepts de type `Concept` (C_1, C_2) afin de définir la notion d'association entre classes. Les liens de type `Liaison orientée`, par contre, sont utilisés lorsque le sens de parcours est important ; un exemple est donné par les liens L_3 et L_4 de $C_3 \overset{L_3}{\longrightarrow} R_2 \overset{L_4}{\longrightarrow} C_4$ où une relation (R_2) est déclarée entre deux concepts de type `Concept` (C_3, C_4) afin de définir la notion d'instanciation entre classes (méta-classe C_3 et classe C_4).

Les relations `Instanciation`, `Composition` et `Héritage` sont utilisées pour représenter respectivement des liens d'instanciation, d'agrégation et d'héritage définis entre les types de Concepts d'une méthode.

Les relations `Correspondance` sont utilisées dans la vue Modèle Général pour exprimer des liens sémantiques entre concepts de modèles différents ; un exemple est donné par le lien L_5 de $C_1 \overset{L_5}{\longleftrightarrow} M_1$ qui déclare qu'un concept de type `Concept` (C_1) est attaché à un autre concept de type `Modèle` (M_1) afin de définir que chaque C_1 peut être associée à un M_1 (par exemple le diagramme d'états).

Les **Cardinalités** de la figure 4.5 représentent les différents types de cardinalités possibles pour les liens donnés par les Relations. Cet ensemble de cardinalités ($11, 1N, 01, 0N$ et MN) est exploité pour introduire la représentation formelle des liens entre les types d'une spécification formelle, selon la proposition de la figure 4.2 et de la table 4.1.

Nous présentons ci-dessous deux exemples de schémas S_Concepts. Le premier de ces exemples (cf. figure 4.6) montre le schéma S_Concept pour la vue *Modèle Statique* du méta-modèle proposé (cf. section 4.2.2).

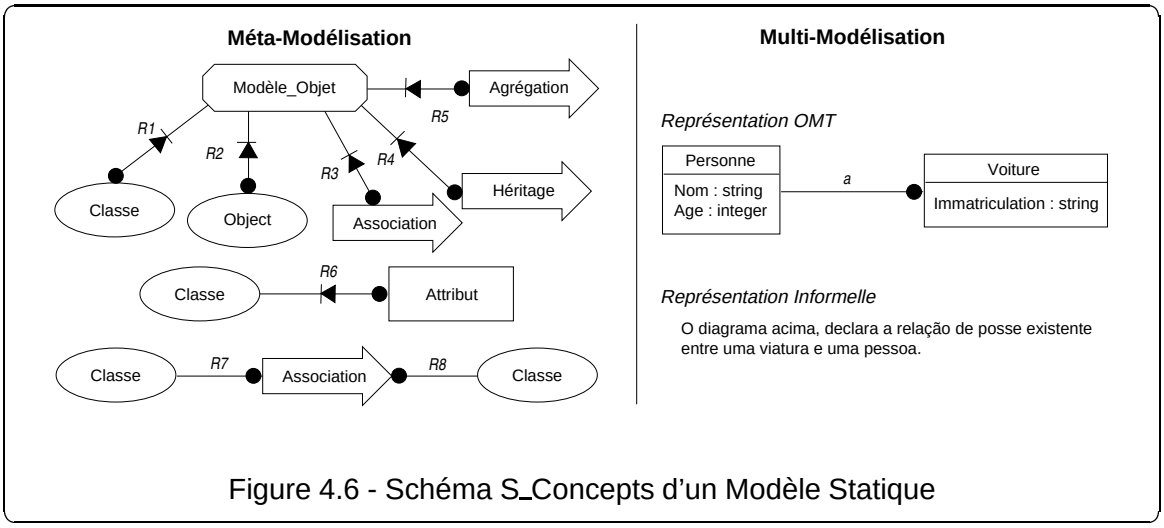


Figure 4.6 - Schéma S_Concepts d'un Modèle Statique

Dans le schéma S_Concept de la figure 4.6, un concept de type `Modèle` appelé “Modèle_Objet” a cinq relations de type `Composition` avec les concepts de type `Concept` “Classe” et “Objet” ainsi qu’avec les concepts de type `Relation` “Association”, “Héritage” et “Agrégation”. De plus, une “Classe” a une relation de type `Composition` avec un concept de type `Propriété` appelé “Attribut”. La sémantique de la représentation du “Modèle_Objet” de la figure 4.6 est d’une part sa composition de “$0, N$” ‘Classe”, “Objet”, “Association”, “Héritage” et “Agrégation”, d’autre part le fait qu’une “Classe” peut avoir des “Attributs” et enfin la possibilité de lier deux “Classes” par une relation de type `Lien non orienté` appelé “Association”.

Le deuxième exemple donné dans la figure 4.7 représente la vue *Modèle Dynamique* du méta-modèle proposé (cf. section 4.2.2). Un concept de type `Modèle` appelé “Diagramme_État” a quatre relations de type `Composition` avec les concepts de type `Concept` “État”, “État_Initial” et “État_Final” et avec le concepts de type `Relation` appelé “Transition_État”. La “Transition_État” a une propriété appelée “Condition”. La sémantique est similaire à celle décrite pour les composants la figure 4.6.

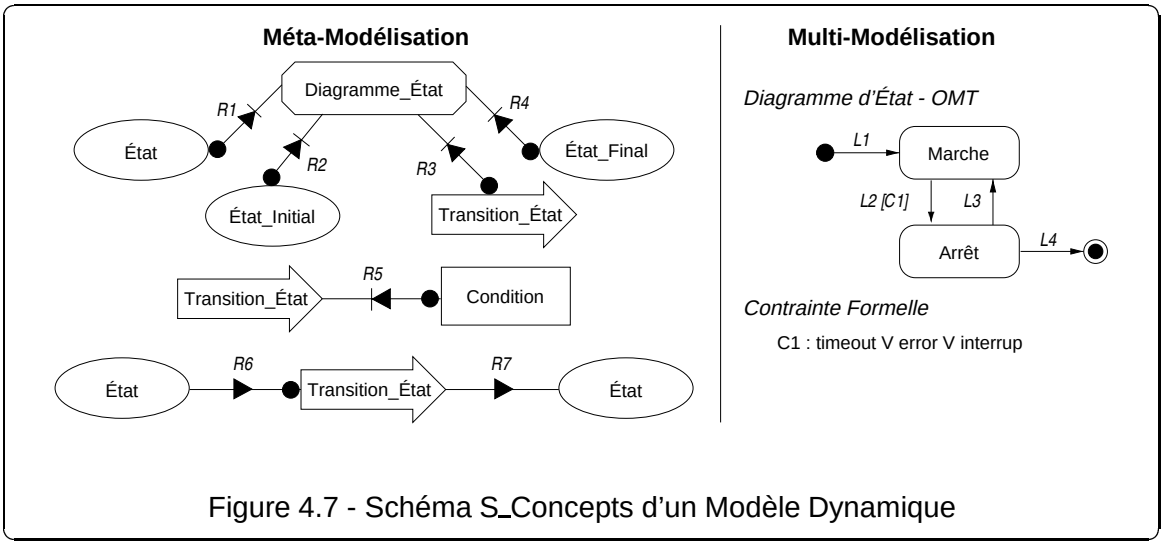


Figure 4.7 - Schéma S_Concepts d’un Modèle Dynamique

Les schémas S_Concepts au niveau multi-modélisation utilisent la notation propre à la méthode modélisée dans la méta-modélisation (voir section 4.4). À titre d’exemple les parties multi-modélisation des figures 4.6 et 4.7 en présentent des exemples. Ainsi dans la figure 4.6 deux classes (Personne et Voiture composants de `Modèle_Objet`) sont liées par l’`Association` “a”. Dans la figure 4.7 un modèle (`Diagramme_État`) est présenté. Ce diagramme d’états est composé de deux `États` (Marche et Arrêt), d’un `État_Initial` et d’un `État_Final` pour lesquels des `Transitions_États` sont définies.

L’approche multi-modélisation est donnée par plusieurs représentations. Ainsi dans la figure 4.6 nous avons à la fois une représentation semi-formelle et une représentation informelle, et dans la figure 4.7 une représentation semi-formelle et une représentation formelle.

4.2.5.3 Spécifications - Les Schémas Formels

Les Schémas Spécifications (*S_Spécifications*) permettent au méta-modèle proposé d'introduire l'utilisation de l'approche formelle dans les vues Modèle Statique, Modèle Dynamique et Modèle Général. Les schémas S_Spécifications permettent de décrire des aspects et des contraintes sur une méthode ou un modèle non explicitées avec les schémas S_Concepts. Comme pour la vue Modèle Formel, la notation utilisée dépend du niveau d'utilisation de l'atelier : méta-modélisation ou multi-modélisation.

Au niveau méta-modélisation, le langage Z [Spi89] est utilisé. Les schémas S_Spécifications complètent le squelette de la vue Modèle Formel (cf. figure 4.3). Ainsi les composants d'une méthode sont définis par rapport aux types de base et les relations existantes entre ces concepts sont déclarées. À ce niveau le schéma S_Spécifications et le schéma S_Global sont donnés par le même schéma. On peu dire que le schéma S_Spécifications est le schéma S_Global complété.

La figure 4.8 présente le schéma S_Spécification qui décrit le méta-modèle statique de la figure 4.6. Le "remplissage" du squelette du Modèle Formel est réalisé automatiquement au fur et à mesure que les schémas S_Concepts sont construits. La technique utilisée pour cette construction automatique, est détaillée au chapitre 5.

$$[\mathit{MODÈLE}, \mathit{CONCEPT}, \mathit{RELATION}, \mathit{PROPRIÉTÉ}]$$

Méthode_X

$$\mathit{Modèle\_Objet} : \mathbb{P}\, \mathit{MODÈLE}$$
$$\mathit{Classe}, \mathit{Objet} : \mathbb{P}\, \mathit{CONCEPT}$$
$$\mathit{Association}, \mathit{Héritage}, \mathit{Agrégation} : \mathbb{P}\, \mathit{RELATION}$$
$$\mathit{Attribut} : \mathbb{P}\, \mathit{PROPRIÉTÉ}$$
$$R1 : \mathit{Modèle\_Objet} \leftrightarrow \mathit{Classe}$$
$$R2 : \mathit{Modèle\_Objet} \leftrightarrow \mathit{Objet}$$
$$R3 : \mathit{Modèle\_Objet} \leftrightarrow \mathit{Association}$$
$$R4 : \mathit{Modèle\_Objet} \leftrightarrow \mathit{Héritage}$$
$$R5 : \mathit{Modèle\_Objet} \leftrightarrow \mathit{Agrégation}$$
$$R7 : \mathit{Classe} \leftrightarrow \mathit{Attribut}$$
$$R7 : \mathit{Classe} \leftrightarrow \mathit{Association}$$
$$R8 : \mathit{Association} \rightarrow \mathit{Classe}$$

---

prédicat

Figure 4.8 - Schéma S_Global - Méta-Modélisation

Dans la figure 4.8, chacun des concepts présents dans la figure 4.6 est typé. Les relations existantes entre ces concepts sont définies en utilisant les types de liens autorisés présentés dans la table 4.1.

Au niveau multi-modélisation, de la même manière qu'au niveau méta-modélisation, les schémas S_Spécifications sont construits automatiquement lors de la définition des schémas S_Concepts. Les schémas S_Spécifications utilisent Object-Z [Hal90a, Hal94] comme langage formel car ces schémas de multi-modélisation sont utilisés exclusivement pour décrire les composants (les classes) d'une modélisation. La vue globale est toujours donnée par le schéma S_Global. C'est dans le schéma S_Global que les structures d'héritage et d'agrégation ainsi que les relations entre classes présentes dans une modélisation sont décrites.

Dans la figure 4.9, nous présentons le schéma S_Global et le schéma S_Spécification pour la multi-modélisation de la figure 4.6. Le schéma S_Global, sans la partie de définition axiomatique, introduit la définition du lien "`a`" entre les classes `Personne` et `Voiture`. Ces classes sont déclarées dans le schéma S_Spécification avec le schéma de classes proposé par le langage Object-Z.

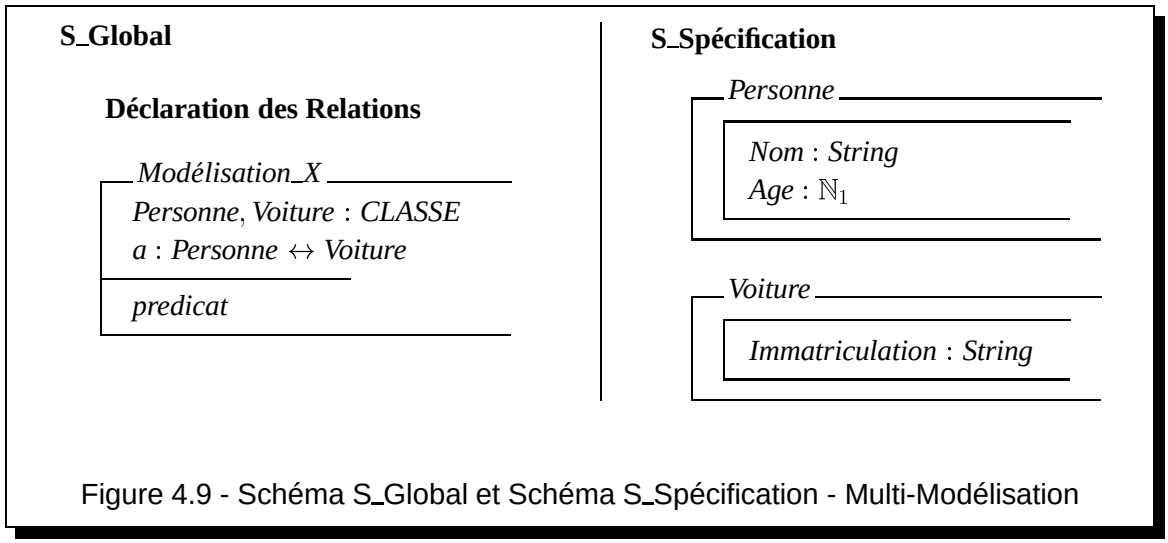


Figure 4.9 - Schéma S_Global et Schéma S_Spécification - Multi-Modélisation

4.2.5.4 Remarques - Les Schémas Informels

C'est avec les Schémas Remarques (*S_Remarques*) que le méta-modèle proposé utilise l'approche informelle. Les schémas S_Remarques montrent les aspects intéressants d'une méthode ou d'un modèle et clarifient les notations employées dans chaque schéma en utilisant le langage naturel. Les schémas S_Remarques peuvent être utilisés aussi pour décrire textuellement certains aspects qui ne peuvent être capturés ni avec les schémas S_Concepts, ni avec les schémas S_Spécifications.

4.3 L'Architecture de l'Atelier

L'atelier est construit sur trois outils existants : les méta-outils GraphTalk et LEdit et l'éditeur syntaxique Thot. Il est composé de deux modules principaux, le *Gestionnaire de Modèles* et le *Gestionnaire de Documents* (cf. figure 4.10). Les sous-sections suivantes décrivent chacun de ces modules.

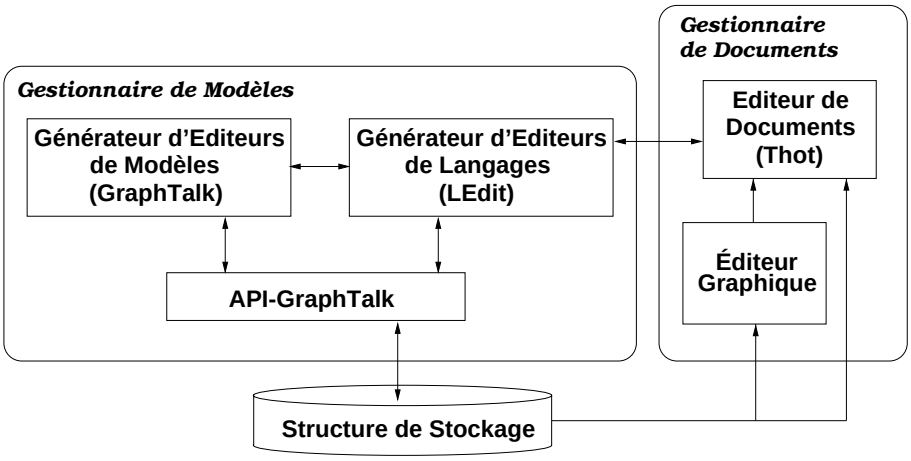


Figure 4.10 - Architecture de la Plate-forme

4.3.1 Gestionnaire de Modèles

Le Gestionnaire de Modèles permet la création et la maintenance de tous les schémas utilisés dans la description d'une méthode spécifique ou d'un système d'information avec l'atelier (méta-modélisation ou multi-modélisation). Il assure aussi la cohérence interne entre les trois types de spécifications (informel, semi-formel et formel). La manière dont cette cohérence interne est assurée est décrite au chapitre 5.

Le Gestionnaire de Modèles s'appuie sur les méta-outils *GraphTalk* et *LEdit* ainsi que sur les facilités qu'ils offrent, comme l'*API-GraphTalk* qui, à travers l'utilisation de "démons", permet la création et la maintenance d'une *Structure de Stockage*. Cette structure de stockage maintient les relations entre les éléments (et leurs caractéristiques) présents dans les schémas S_Concepts d'une modélisation.

Le Gestionnaire de Modèles peut être utilisé à deux niveaux différents : méta-modélisation et multi-modélisation (cf. section 4.4). Au niveau méta-modélisation le Gestionnaire de Modèles est composé de deux ateliers associés, `A2M` et `A2M'` et au niveau multi-modélisation

il est composé de l'atelier AM. Ces ateliers générés par GraphTalk/Ledit répondent à des besoins particuliers de chacun des niveaux. L'implantation de chacun de ces ateliers est détaillée dans le chapitre 5 ainsi qu'une description des outils GraphTalk et Ledit. Les fonctionnalités essentielles de ces ateliers, A2M, A2M' et AM, sont décrites à la section 4.4.

4.3.2 Gestionnaire de Documents

Le Gestionnaire de Documents permet la création et la maintenance d'une spécification informelle standardisée qui regroupe les trois catégories de modélisations employées. Il s'appuie sur l'éditeur syntaxique générique *Thot*. L'éditeur construit avec Thot est basé sur une spécification informelle standardisée : le Canevas de Modélisation.

Ce Canevas de Modélisation (cf. section 4.6) est composé d'un *texte descriptif structuré* qui porte sur les schémas produits par le Gestionnaire de Modèles. Pour pouvoir créer ce texte descriptif structuré, il faut d'abord modéliser le système avec le Gestionnaire de Modèles. Ce texte descriptif est composé d'un texte en langage naturel contenant des références (ancres, pointeurs) vers des éléments de schémas issus du Gestionnaire de Modèles.

La cohérence entre ce texte descriptif et les schémas du Gestionnaire de Modèles est assurée grâce au concept de type Vues présenté dans la figure 4.5. Après la modélisation d'un modèle ou d'un système, on doit définir les vues qui portent sur cette modélisation pour assurer la cohérence entre les deux gestionnaires. Si aucune vue n'est définie, la totalité de la modélisation est considéré comme une vue unique.

Pour introduire un élément dans le Gestionnaire de Documents, celui-ci, à partir de la Structure de Stockage, présente une liste des vues déclarées dans la modélisation construite avec le Gestionnaire de Modèles. Après avoir choisi l'une des vues déclarées, le Gestionnaire de Documents importe les composants semi-formels, informels et formels correspondant aux éléments définis dans le Gestionnaire de Modèles pour cette vue. Cette importation provient de deux environnements :

– LEdit : les "composants formels"sont extraits des fichiers générés directement par LEdit, soit à partir des schémas S_Spécifications, soit à partir du schéma S_Global ;

– Structure de Stockage : les "composants semi-formels" et les "composants informels" sont extraits à partir des composants définis dans les vues de la structure de stockage ; les composants informels directement, à travers de démons qui à partir de leur référence dans une vue les récupèrent dans le Gestionnaire de Modèles, et les composants semi-formels, donc graphiques, à travers l'*Éditeur Graphique*. Cet Éditeur Graphique

doit, à partir des données de la structures de stockage construire une représentation graphique "représentable" dans Thot.

Cette spécification informelle standardisée peut représenter par exemple un Document de Spécification Globale [AFN94] à générer pour un projet donné.

4.4 Les Niveaux d'Utilisation

L'atelier peut être utilisé pour spécifier un système d'information spécifique ou les modèles d'une méthode. L'utilisation de l'atelier pour la spécification d'une méthode se situe au niveau *Méta-Modélisation*. Son utilisation pour la spécification d'un SI, en utilisant la modélisation d'une méthode préalablement effectuée, se situe au niveau *Multi-Modélisation*.

Pour accomplir la tâche de méta-modélisation, il y a deux ateliers différents : A2M et A2M'. La multi-modélisation est réalisée par l'atelier AM. La figure 4.11 présente d'une manière schématique ce que chacun des ateliers produit, ainsi que l'interaction entre ces ateliers.

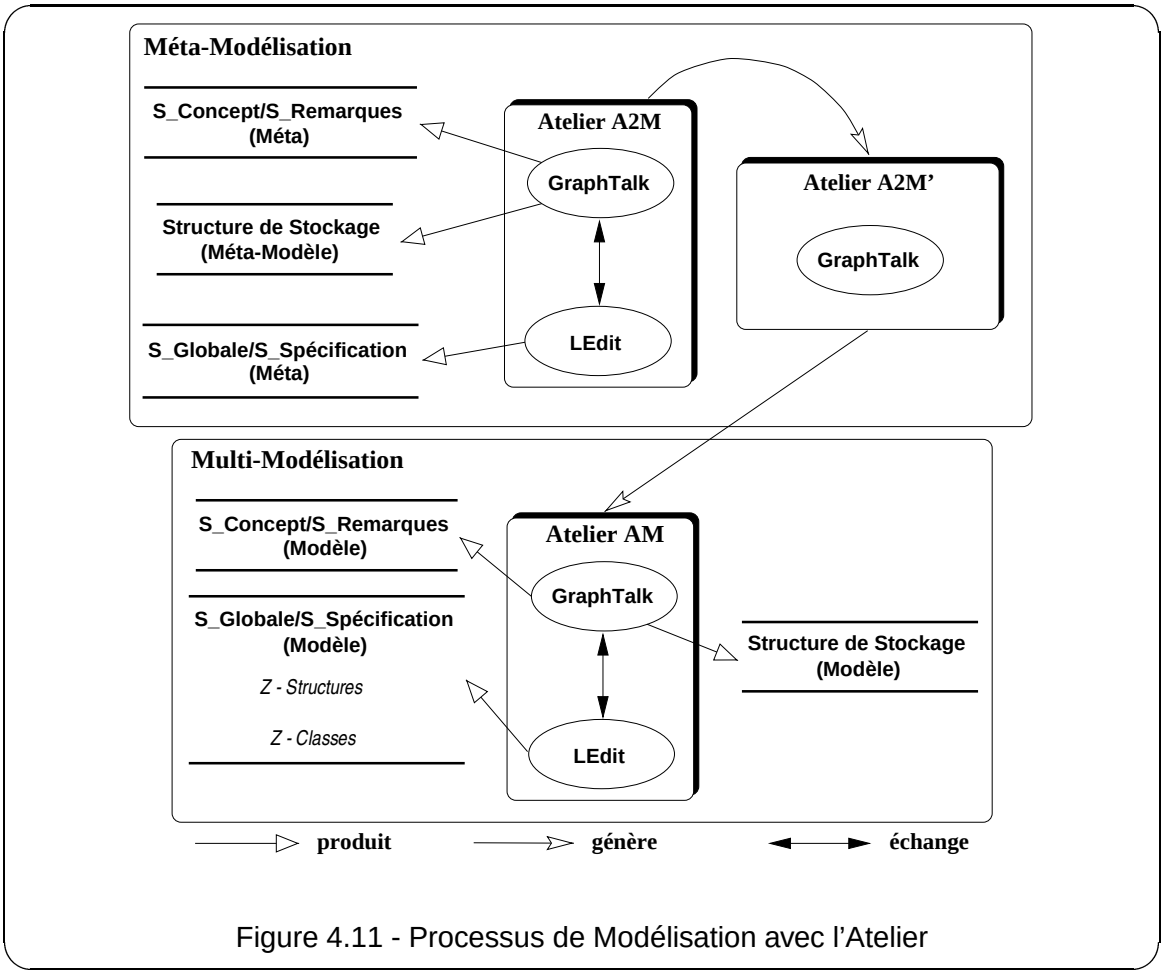


Figure 4.11 - Processus de Modélisation avec l'Atelier

Dans le niveau méta-modélisation, deux étapes sont nécessaires pour créer le méta-modèle d'une méthode avec deux ateliers différents. Le niveau multi-modélisation, quant à lui, avec son atelier, utilise les définitions produites au niveau méta-modélisation. Conformément dans la figure 4.11, nous présentons ci-dessous avec plus de détails ces deux niveaux d'utilisation.

La construction de ces trois ateliers successifs (A2M, A2M' et AM) constitue une démarche spécifique d'utilisation des méta-outils initiaux GraphTalk et LEdit.

4.4.1 Niveau Méta-Modélisation

Le premier des deux ateliers utilisés au niveau méta-modélisation, l'atelier A2M, s'appuie sur les trois vues (Statique, Dynamique et Général) du méta-modèle proposé. Ces vues sont utilisées pour produire une première version d'un modèle d'une méthode. Les schémas S_Concepts sont donnés par des diagrammes semblables aux diagrammes E-R-A qui représentent les concepts et les relations entre ces concepts. La notation utilisée dans les schémas S_Concepts est celle donnée dans la figure 4.5. Les schémas S_Spécifications utilisent le langage Z [Spi89] (cf. figure 4.3).

Le deuxième atelier utilisé au niveau méta-modélisation, l'atelier A2M' est construit automatiquement par l'atelier A2M. Cette génération automatique est basée sur une utilisation de l'outil GraphTalk : à partir de la spécification d'une méthode comme composition de `Modèles`, `Concepts`, `Relations` et `Propriétés` (cf. figure 4.5) ainsi que des relations définies entre ces composants, l'atelier A2M' est généré. On peut voir l'atelier A2M' comme une "extension" de l'atelier A2M. Cependant, les composants de A2M' ne sont pas ceux offerts par A2M (`Modèles`, `Concepts`, `Relations` et `Propriétés`) mais ceux offerts par une instance d'un atelier GraphTalk standard. Sans détailler GraphTalk (voir section 5.1.1), on peut dire que les `Modèles` de A2M sont transformés en graphes GraphTalk dans A2M', que les `Concepts` de A2M sont transformés en nœuds GraphTalk dans A2M', que les `Relations` de A2M sont transformées en liens GraphTalk dans A2M' avec la sémantique définie dans A2M et que les `Propriétés` de A2M sont transformées soit en nœuds, soit en propriétés GraphTalk dans A2M'. Dans la section 4.4.3 nous présentons un exemple qui illustre l'interaction entre les ateliers A2M et A2M'.

Pour pouvoir offrir un atelier de modélisation propre à une méthode, on doit compléter l'atelier généré par A2M, à partir des définitions réalisées dans celui-ci, en déclarant la notation (les formes) et en ajoutant des éléments supplémentaires propres à GraphTalk pour que celui-ci puisse générer l'atelier AM correspondant à la méthode choisie. On doit aussi attacher une propriété de type texte à chacun des nœuds pour lesquels on veut disposer, dans l'atelier final AM généré, de la possibilité de définir une spécification informelle. L'atelier

A2M génère aussi dans A2M', tous les composants nécessaires pour offrir la notion de vues dans l'atelier AM qui sera généré.

La définition des formes consiste à déclarer dans A2M' une représentation graphique pour chacun des concepts, relations ou propriétés utilisés dans l'atelier AM. L'inclusion des éléments supplémentaires est due à des besoins particuliers de l'outil GraphTalk : il faut par exemple, pour les structures du type héritage ou agrégation, introduire un nœud GraphTalk appelé "dispatcher" afin que l'atelier AM soit en mesure de présenter une telle structure. Comme cet ajout d'information n'apporte aucune définition sémantique nouvelle à la spécification d'une méthode, l'atelier A2M' ne gère pas de spécifications semi-formelles, informelles ou formelles (cf. figure 4.11). Par rapport aux schémas du méta-modèle proposé, le seul schéma existant est le schéma S_Concept qui utilise la notation propre à GraphTalk (voir section 5.1.1).

4.4.2 Niveau Multi-Modélisation

Ce niveau dispose de son propre atelier (AM) pour la création de modèles de systèmes d'information. Cet atelier est construit à partir des définitions produites dans A2M'. Les trois vues (Statique, Dynamique et Général) ne sont pas utilisées systématiquement, car chaque méthode, à travers ses propres modèles, définit ses propres vues ; cependant, les objets et les relations créés dans AM peuvent être classés selon les vues proposées par notre méta-modèle. Les schémas S_Global utilisent le langage Z [Spi89], les schémas S_Concepts utilisent la notation définie dans A2M' et les schémas S_Spécifications utilisent Object-Z [DKRS91] pour la définition des classes.

Le Modèle Formel, à travers le schéma S_Global, présente la déclaration des classes et des structures d'héritage conformément à la proposition de A. Hall [Hal90b, Hal94]. Nous avons réutilisé ce principe pour la représentation des structures d'agrégation. Les relations entre objets sont représentées à l'aide de relations Z (cf. table 4.1).

4.4.3 Interaction entre Niveaux

Dans cette section nous présentons, à travers un exemple de méta-modélisation et de multi-modélisation, comment les trois ateliers interagissent entre eux. La figure 4.12 présente cet exemple et illustre d'une manière concise les déclarations faites à propos des deux niveaux auxquels notre atelier peut être utilisé et aussi des relations entre les vues et les schémas. Dans cette figure nous utilisons comme méthode cible, la méthode OMT. La partie Remarques de type spécification informelle en langue naturelle n'a pas été illustrée.

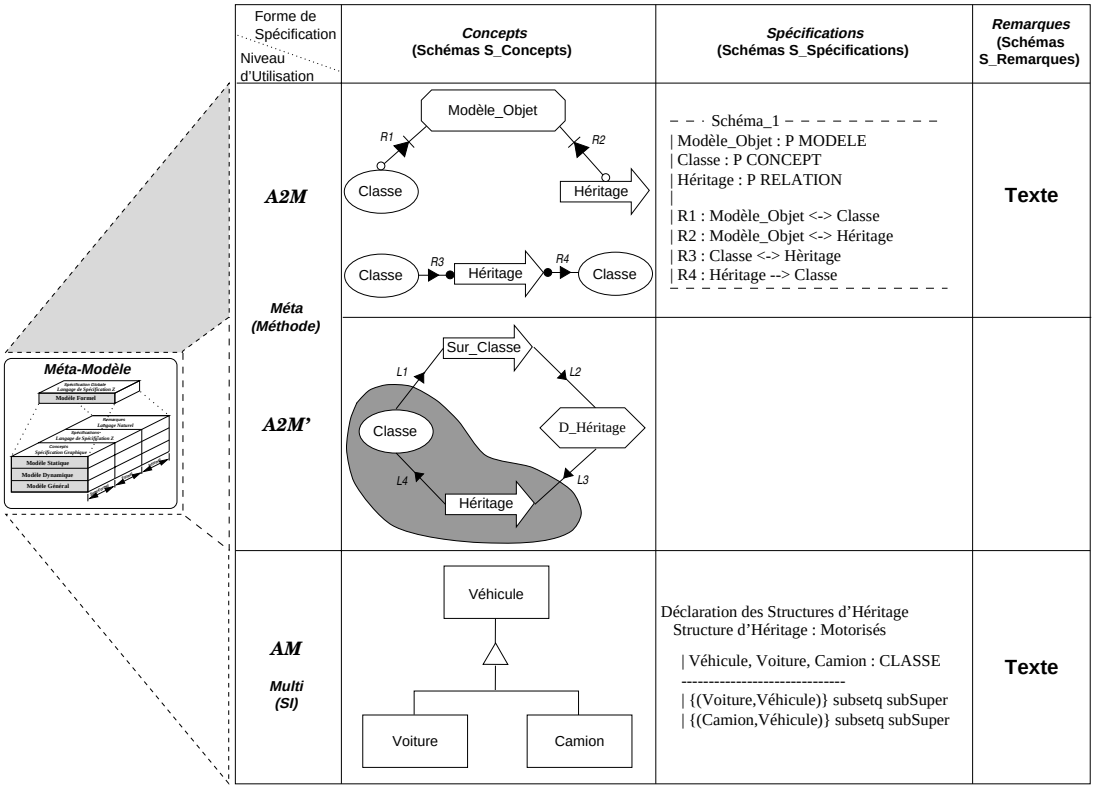


Figure 4.12 - Les Niveaux Proposés par le Prototype d'Atelier

La figure 4.12 met en évidence le fait déjà cité que les trois vues (Modèle Statique, Modèle Dynamique et Modèle Général) du méta-modèle proposé ne sont utilisées qu'au niveau méta-modélisation. Les ateliers A2M et A2M' fournissent un cadre qui conduit à la modélisation d'une méthode en respectant les vues proposées. La quatrième vue, Modèle Formel, ainsi que les trois parties (Concepts, Spécifications et Remarques) sont utilisées dans les trois niveaux.

La partie du niveau Méta-Modélisation relative à l'atelier A2M présente une modélisation très partielle du Modèle d'Objets d'OMT. Cette modélisation exprime semi-formellement et formellement le fait que les concepts de classe et d'héritage sont définis dans le Modèle d'Objets. La partie relative à A2M' présente les descriptions nécessaires pour l'utilisation effective du concept d'héritage entre classes. Dans A2M', Classe et Héritage sont générés automatiquement par A2M, car ils ont été définis au niveau supérieur. Les autres concepts (Sur_Classe et D_Héritage) sont introduits pour permettre au méta-outil GraphTalk de gérer le concept d'héritage, ses contraintes et sa représentation graphique dans l'atelier AM.

Dans cet exemple, le niveau Modélisation décrit l'utilisation de l'atelier de modélisation AM généré à travers la présentation du schéma S_Concept qui montre trois classes, Véhicule, Voiture et Camion, liées par une structure d'héritage. Le schéma S_Spécification montre une partie du schéma S_Global pour le schéma S_Concept. Pour simplifier, la spécification des classes n'est pas présentée.

Nous reviendrons sur ces différents éléments lors de la présentation de la réalisation de notre atelier (cf. chapitre 5).

4.5 Des Relations entre Éléments de Modélisation

Les deux modules principaux de l'atelier Gestion de Modèles et Gestion de Documents interagissent avec des spécifications de types différents : informelle, semi-formelle ou formelle. Les relations entre ces différents types de spécifications ainsi que les relations entre ces types de spécification et les modules sont développées dans les sections suivantes.

Ces relations sont essentielles dans la perspective d'une multi-modélisation introduite pour utiliser à un moment donné, pour un comportement donné d'un système donné, la forme de modélisation la mieux adaptée.

Dans un premier temps, les relations existantes entre les trois types de spécifications sont traitées dans l'optique du gestionnaire de modèles et ensuite ce sont les relations entre ces spécifications et les modules de l'atelier.

4.5.1 Relations entre Modèles

Cette section présente des types de relations pertinentes entre des fragments de spécification des trois approches de modélisation (cf. figure 4.13). Dans l'optique du méta-modèle proposé (section 4.2), ces relations existent entre différents schémas *S_Concept*, *S_Spécification* et *S_Remarque* d'un même modèle : ces relations maintiennent une "orientation horizontale" car S_Concept, S_Spécification et S_Remarque modélisent le même sujet.

Les flèches de la figure 4.13 expriment les types utiles de relations. Ces flèches partent de la *Modélisation Source* vers la *Modélisation Résultat*.

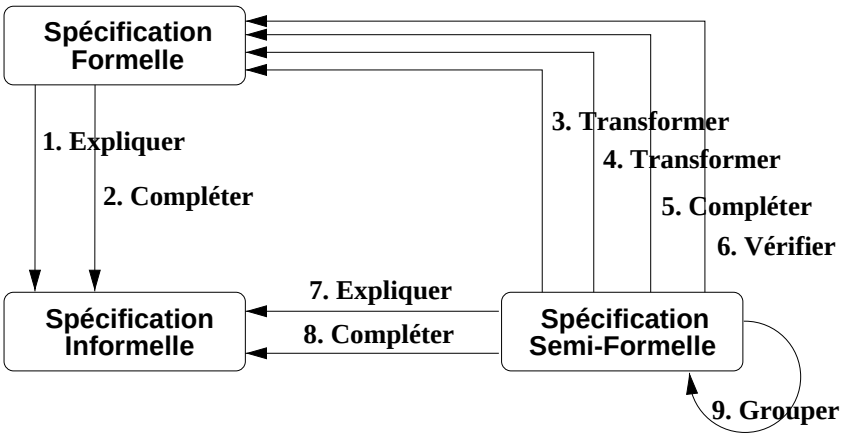


Figure 4.13 - Des Relations entre Spécifications

Par Modélisation Source, on entend la modélisation sur laquelle porte l'acte exécuté par la relation et, par Modélisation Résultat, on veut dire la modélisation qui présente le résultat de l'exécution de l'acte. Par exemple, la flèche 1 exprime le fait que la spécification informelle (la Modélisation Résultat) *explique* une spécification formelle (la Modélisation Source) et la flèche 4 exprime le fait qu'une spécification semi-formelle *est transformée* en une spécification formelle.

La table 4.2 présente pour chaque relation de la figure 4.13 différentes informations : (a) *pourquoi* ces relations existent, (b) *comment* elles sont implantées dans l'atelier, un *bilan* (c) qui présente s'il peut exister une perte "P", un ajout "A" ou si rien ne se passe "-" entre l'information de la Modélisation Résultat et la Modélisation Source lorsqu'on change de représentation ; (d) le *type* de la relation. La notion d'ajout ou perte d'information est fortement liée à la cohérence de la modélisation.

“Rel”	Pourquoi	Comment	“Bilan”	Type
1	expliquer les aspects d’une modélisation formelle.	génération de “notes textuelles” créées avec le gestionnaire de modèles.	-	expliquer
2	compléter les aspects d’une modélisation formelle qu’on n’arrive pas ou qu’on ne souhaite pas représenter formellement.	introduction de “notes textuelles” créées avec le gestionnaire de modèles.	A	compléter
3	fournir une “image” de la spécification semi-formelle, laquelle se traduit par une spécification formelle qui correspond à la spécification semi-formelle d’origine.	génération d’une spécification formelle à travers la passerelle GraphTalk/LEdit.	P	transformer
4	maintenir la cohérence entre la spécification semi-formelle et la spécification formelle en changeant cette dernière, à chaque fois qu’un changement s’opère sur la première.	mise à jour d’une spécification formelle déjà transformée à travers la passerelle GraphTalk/LEdit.	P	transformer
5	faire ressortir des aspects internes d’une spécification semi-formelle à travers la spécification formelle d’une manière à augmenter/assurer la cohérence.	création ou modification d’une spécification formelle composée d’éléments qui complètent une spécification semi-formelle.	A	compléter
6	générer une modélisation formelle qui correspond à la spécification semi-formelle et qui puisse être vérifiée par un “type-checker”, par rapport aux types de base aux structures définies.	génération d’une spécification formelle à travers la passerelle GraphTalk/LEdit suivie d’un appel à un “type-checker”.	P	vérifier
7	expliquer les aspects d’une modélisation semi-formelle qu’on n’arrive pas ou qu’on ne souhaite pas représenter graphiquement.	génération de “notes textuelles” créées avec le gestionnaire de modèles.	-	expliquer
8	compléter les aspects d’une modélisation semi-formelle qu’on n’arrive pas ou qu’on ne souhaite pas représenter graphiquement.	introduction de “notes textuelles” créées avec le gestionnaire de modèles.	A	compléter
9	fournir une vue sur une partie d’une modélisation dans le but d’améliorer la compréhension de celle-ci.	manipulation réalisée avec le gestionnaire de modèles sur les éléments internes d’une spécification.	-	grouper

Table 4.2 - Relations entre Fragments de Spécifications

Les types des relations exprimées par les flèches de la figure 4.13 sont décrits dans les sous-sections suivantes. Pour les exemples donnés, les Modélisations Sources sont présentées à gauche avec à droite les Modélisations Résultats. Ces exemples utilisent des niveaux différents de modélisation. Par la suite, le terme schéma est utilisé pour exprimer un élément modélisé ou un fragment de spécification.

4.5.1.1 Relation Expliquer

Un schéma peut expliquer un autre schéma (cf. figure 4.14). Ces deux schémas sont décrits selon deux paradigmes différents (par exemple : formel et semi-formel, ou semi-formel et informel). Il n'y a pas d'ajout d'information (cf. relations 1 et 7 de la figure 4.13).

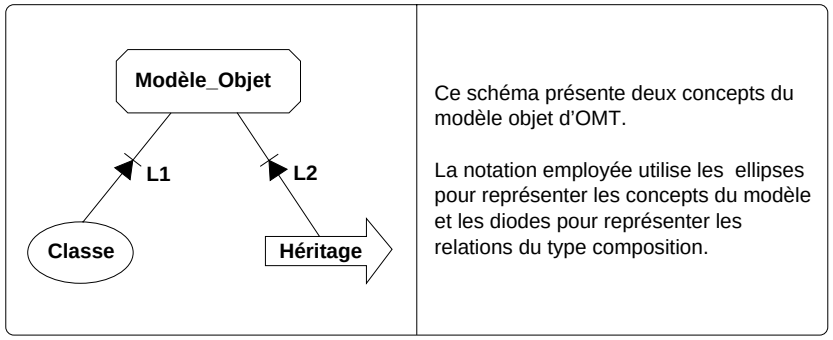


Figure 4.14 - Relation Expliquer

4.5.1.2 Relation Compléter

Un schéma peut compléter un autre schéma (ou une partie de celui-ci) par un ajout d'information qu'il serait difficile d'exprimer autrement (cf. figure 4.15) (cf. relations 2, 5 et 7 de la figure 4.13).

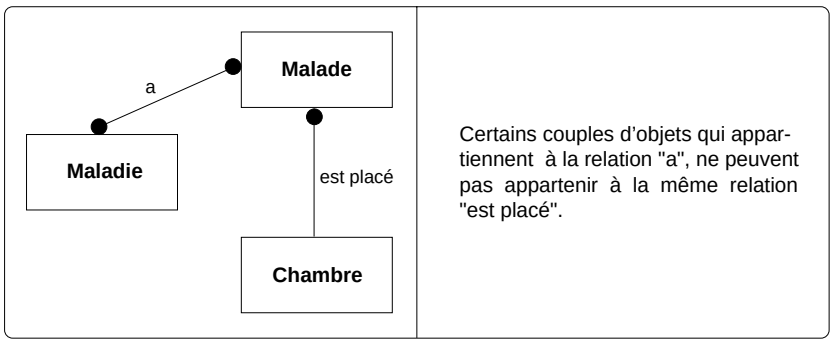


Figure 4.15 - Relation Compléter

4.5.1.3 Relation Transformer

Un schéma peut être le résultat d'une transformation appliquée sur un autre schéma (cf. figure 4.16). Il n'y a pas d'ajout d'information et il peut même exister une perte, selon le contenu du schéma source. La réversibilité entre les deux représentations n'est pas toujours assurée même s'il n'y a pas de perte d'information dans la première transformation (cf. relations 3 et 4 de la figure 4.13).

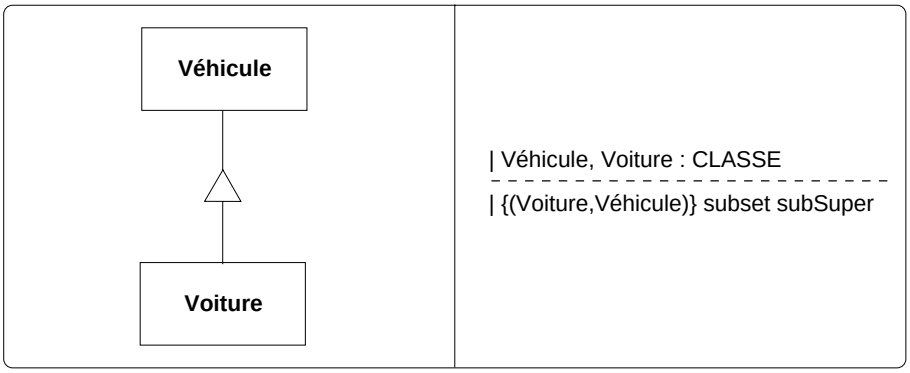


Figure 4.16 - Relation Transformer

4.5.1.4 Relation Vérifier

C'est un type de relation existant seulement entre des spécifications formelles et semi-formelles. Ce type de relation (cf. figure 4.17) s'appuie fortement sur un contrôleur de typage ("type-checker") qui exécute le processus de vérification. Le schéma S_Spécification sert à vérifier les composants du schéma S_Concept associé par rapport à la syntaxe du langage formel employé (Z) ainsi que par rapport aux structures formelles pré-définies dans le schéma de la vue Modèle Formel. Pour arriver au schéma présenté à droite de la figure 4.17, il faut faire une transformation de la représentation de gauche vers un schéma S_Spécification pour rendre possible la manipulation de celui-ci par le contrôleur de typage . De cette manière il peut exister une perte d'information, comme par exemple les cardinalités *MN* d'un lien, dans ce type de relation (cf. relation 6 de la figure 4.13).

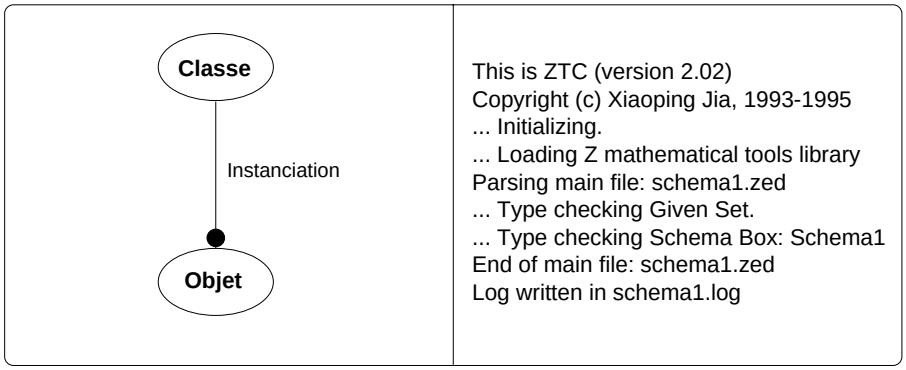


Figure 4.17 - Relation Vérifier

4.5.1.5 Relation Grouper

Certains éléments d'un schéma S_Concept sont groupés dans un autre schéma, de manière à fournir une vue sur le premier schéma (cf. figure 4.18). Il n'y a pas d'ajout d'information (cf. relation 9 de la figure 4.13).

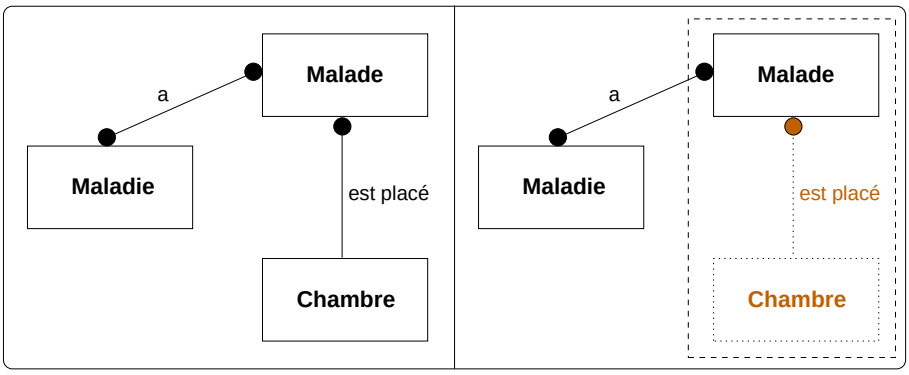


Figure 4.18 - Relation Grouper

4.5.2 Relations entre Modules

Dans cette section nous présentons les relations qui peuvent exister entre les deux modules principaux de l'atelier et les schémas d'une modélisation (cf. figure 4.19).

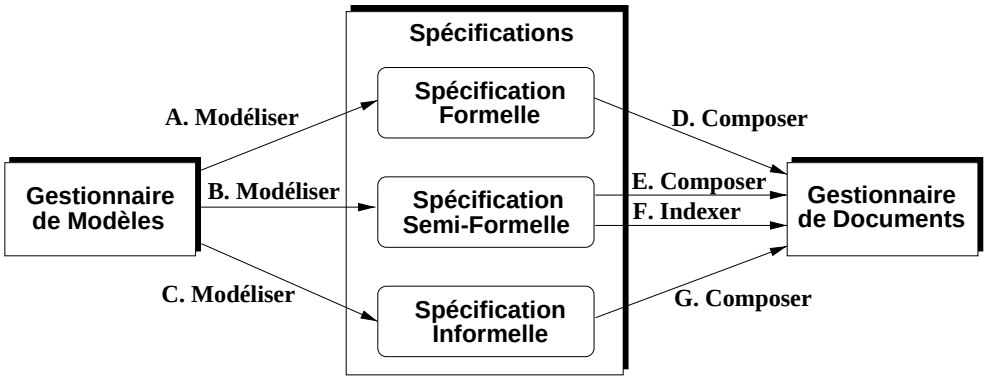


Figure 4.19 - Relations entre Modules et Modèles

La table 4.3 présentée ci-dessous utilise les mêmes notations que la table 4.2 pour résumer les relations de la figure 4.19.

"Rel"	Pourquoi	Comment	"Bilan"	Type
A	générer et maintenir la spécification d'un modèle d'un schéma S_Spécification.	l'outil GraphTalk est utilisé à travers le gestionnaire de modèles.	A	modéliser
B	générer et maintenir la spécification d'un modèle d'un schéma S_Concept.	l'outil GraphTalk est utilisé à travers le gestionnaire de modèles.	A	modéliser
C	générer et maintenir la spécification d'un modèle d'un schéma S_Remarque.	l'outil GraphTalk est utilisé à travers le gestionnaire de modèles.	A	modéliser
D	fournir les "notes formelles" qui seront des composants du texte structuré maintenu par le gestionnaire de documents.	importation des composants formels de la structure de stockage générée lors de l'utilisation du gestionnaire de modèles.	-	composer
E	fournir les "notes graphiques" qui seront des composants du texte structuré maintenu par le gestionnaire de documents.	utilisation de l'éditeur graphique qui génère une figure à partir de la structure de stockage.	-	composer
F	fournir les composants d'une modélisation qui guideront la construction du Modèle Formel par le gestionnaire de documents.	importation de la structure de stockage générée lors de l'utilisation du gestionnaire de modèles.	-	indexer
G	fournir les "notes textuelles" qui seront des composants du texte structuré qui est maintenu par le gestionnaire de documents.	importation des composants informels de la structure de stockage générée lors de l'utilisation du gestionnaire de modèles.	-	composer

Table 4.3 - Relations entre Modules et Modèles

Les relations exprimées par les flèches de la figure 4.19 sont décrites dans les sous-sections suivantes.

4.5.2.1 Relation Composer

Les schémas présents dans une modélisation sont utilisés comme des composants du document structuré manipulé par le gestionnaire de documents. Il n'y a pas d'ajout d'information (cf. figure 4.20) (cf. relations D, E et G de la figure 4.19).

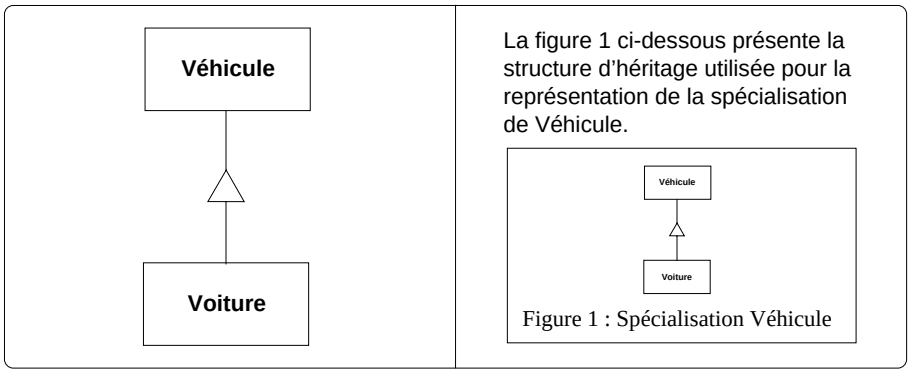


Figure 4.20 - Relation Composer

4.5.2.2 Relation Modéliser

Cette relation représente la création et la maintenance des différents types de schémas de modélisation par le gestionnaire de modèles. Il y a ajout d'information (cf. figure 4.21) (cf. relations A, B et C de la figure 4.19).

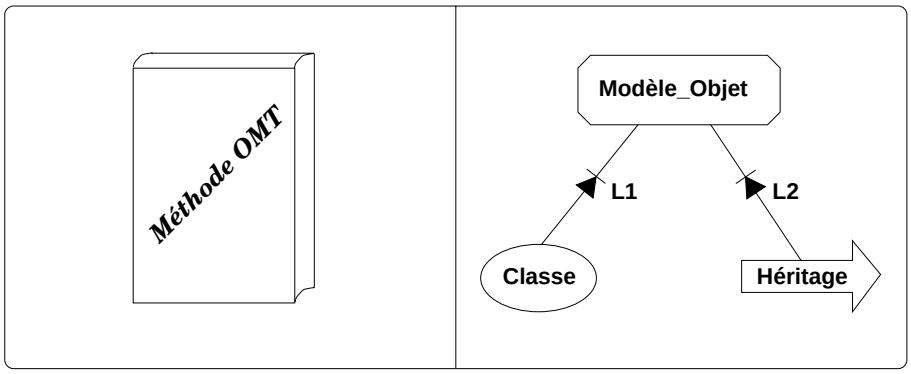


Figure 4.21 - Relation Modéliser

4.5.2.3 Relation Indexer

Les éléments présents dans les S_Concepts sont les “indices” à partir desquels le Gestionnaire de Modèles construit le *texte descriptif* afin d’assurer la cohérence entre les modèles et le Document de Spécification Globale - DSG (cf. figure 4.22) (cf. relation F de la figure 4.19).

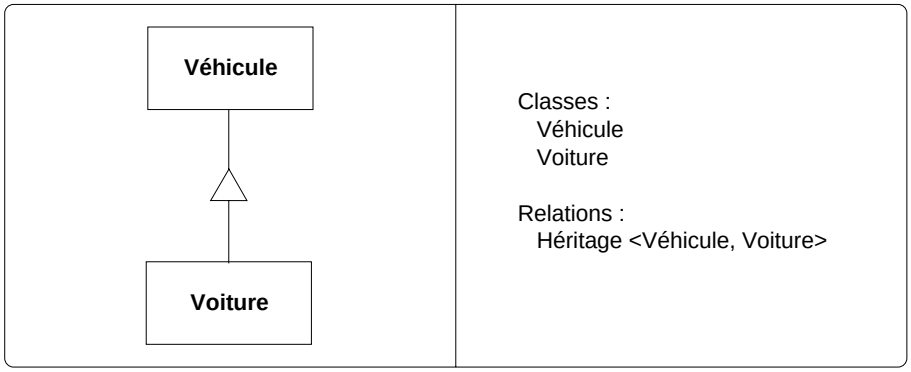


Figure 4.22 - Relation Indexer

4.6 Un Canevas de Modélisation

La section 4.3.2 a présenté le Gestionnaire de Documents qui, dans notre atelier, doit gérer le document de spécification d’un système. Cette gestion aura pour base un Canevas de Modélisation qui est présenté ci-dessous. Un tel canevas de modélisation constitue une solution partielle au guidage d’une démarche de modélisation d’un système d’information.

Dans la section 2.2.1, on a présenté le concept d’hypertexte afin de pouvoir l’utiliser comme base du Gestionnaire de Documents dans le but de construire un document structuré qui remplisse le rôle de document de spécification de systèmes.

L’un des buts de ce travail étant la présentation d’un prototype d’atelier qui intègre trois types différents de spécifications, le canevas adopté ici n’a pas pour objectif d’être un document qui traite toutes les étapes et détails d’une spécification de systèmes. Il s’agit simplement d’un cadre qui montre les possibilités et les bénéfices qu’une utilisation conjointe d’approches différentes peut offrir. Pour un document complet pour la spécification de systèmes, on peut voir par exemple la norme AFNOR NF 67-100-3 [AFN94] qui est une proposition d’un “canevas de documents opérationnels convenant aux différentes mailles d’approche

que la vie des projets informatiques peut susciter selon que l'on s'intéresse à l'ensemble de l'entreprise ou de l'organisation, à un domaine particulier ou à un sous-ensemble applicatif". D'une manière très résumée, on peut dire que cette norme AFNOR préconise la création d'un document composé des sous-documents suivants : Relations Contractuelles, Gestion de Projet, Assurance Qualité, Développement et finalement Utilisation et Soutien.

Le canevas qui nous proposons pour l'atelier est partiellement issu de cette norme. Il constitue ce qu'on appelle le *Document de Spécification Globale - DSG*. La figure 4.23 présente le squelette d'un tel canevas.

1. Introduction
2. Contexte
   - 2.1 Objectifs
   - 2.2 Hypothèses
3. Besoins Détaillés
   - 3.1 Modèle Formel
   - 3.2 Les Vues
     - 3.2.n.1 Les Concepts

Figure 4.23 - Canevas de Modélisation - Le Document de Spécification Globale

Dans les items 1 et 2 du DSG, une introduction textuelle descriptive du système à modéliser est produite. Ces items ne doivent pas entrer dans les détails ni de l'analyse ni de l'implantation, qui sont traités dans l'item 3.

Dans l'item 3, se concrétise l'interaction entre le Gestionnaire de Documents et le Gestionnaire de Modèles. Tous les composants présents dans cet item proviennent du Gestionnaire de Modèles. Ainsi les composants semi-formels sont la représentation des schémas S_Concepts, les composants informels la représentation des schémas S_Remarques et les composants formels la représentation des schémas S_Spécifications et du schéma S_Global. Nous pouvons aussi ajouter du texte informel "autour" de ces composants importés.

Dans le DSG, l'item 3.1 déclare la partie du schéma S_Global qui introduit la définition de types (dans le niveaux méta-modélisation et multi-modélisation) et les définitions axiomatiques (dans le niveau multi-modélisation).

L'item 3.2 du DSG est utilisé pour la spécification, selon les trois formalismes, des relations entre composants de la modélisation (dans le niveaux méta-modélisation et multi-modélisation) ainsi que des structures internes d'une modélisation (dans le niveau multi-modélisation). Pour cela nous devons définir dans le Gestionnaire de Modèles des vues qui correspondent aux relations et structures qui nous voulons inclure dans le DSG. Dans l'item

3.2.n.1 nous pouvons, pour chacune des vues, référencer les composants associés lorsqu'il est nécessaire de rajouter des détails de modélisation particuliers à un composant. À travers la définition de la vue présente dans la Structure de Stockage, nous pouvons accéder aux composants de celles-ci.

4.7 Conclusion

Afin de répondre à quelques uns des besoins des nouveaux ateliers de modélisation, un méta-modèle a été proposé. Celui-ci offre un cadre où des modélisations informelles, semi-formelles et formelles sont utilisées conjointement dans une modélisation.

L'utilisation de ce méta-modèle dans l'atelier permet la représentation et la gestion de modèles à deux niveaux d'abstraction hiérarchisés :

– modèles de systèmes d'information selon ces formalismes variés ;

– "modèles de modèles" de systèmes d'information, c'est-à-dire méta-modèles.

L'atelier atteint ainsi l'orientation préconisée par des experts des langages formels dans la mesure où il intègre des spécifications formelles à d'autres types de spécifications.

L'architecture adoptée par l'atelier montre une certaine indépendance entre les deux modules principaux. On peut dire que le Gestionnaire de Modèles est chargé de la création de la modélisation, et que le Gestionnaire de Documents gère la mise en forme de cette modélisation.

L'utilisation d'un même méta-modèle à deux niveaux différents permet, même si on peut dire que cette utilisation est secondaire au niveau multi-modélisation (car elle se restreint à l'utilisation des schémas), aux deux types de spécifications produites, d'avoir la même apparence dans la mesure où elles utilisent le même paradigme. Cela peut faciliter la compréhension des modélisations.

En résumé, les principales caractéristiques de l'atelier présenté dans ce chapitre sont :

1. une description de chaque modèle selon un même méta-modèle prenant en compte les trois dimensions, informelle, semi-formelle et formelle ;
2. une indépendance de toute représentation physique et graphique des modèles ;
3. une portabilité et une autonomie des trois composants (méta-outils) de l'architecture choisie ;

4. une combinaison de l'approche objet avec des aspects formels et une organisation en schémas ;
5. une généralisation de la multi-modélisation ;
6. une aide à la documentation ;
7. une possibilité de vérifier formellement une spécification semi-formelle.

L'étude sommaire réalisée sur les différents types de relations existantes entre les trois approches de modélisation permet d'imaginer des utilisations possibles de l'atelier en réponse, par exemple, au besoin de transformation présenté par C. Rolland (cf. section 4.1). Cette étude permet aussi une meilleure compréhension des mécanismes internes de l'atelier. Cette étude peut être étendue à d'autres types de relations avec les mêmes objectifs ; ainsi on peut soit étudier l'existence d'autres types de relations comme la simplification ou la réécriture, soit étudier l'existence de relations possibles entre tous les types différents de modélisation : informels vers formels, formels vers semi-formels, informels vers semi-formels ainsi que formels vers formels, semi-formels vers semi-formels et informels vers informels.

Chapitre 5

Implantation et Expérimentations

Dans le chapitre précédent, nous avons décrit les éléments essentiels d'un atelier de modélisation orienté multi-représentations et multi-méthodes. La section 4.3 a introduit l'architecture de l'atelier. Cet atelier est basé sur les modules Gestionnaire de Modèles et Gestionnaire de Documents. Dans ce chapitre, nous en présentons l'implantation ainsi que des exemples d'utilisation.

5.1 Les Outils Utilisés

Dans cette section, sont présentés les trois outils utilisés pour implanter les modules Gestionnaire de Modèles et Gestionnaire de Documents : le méta-outil GraphTalk et le méta-éditeur LEdit qui servent à implanter le module Gestionnaire de Modèles ainsi que l'éditeur structuré Thot utilisé pour le module Gestionnaire de Documents.

5.1.1 GraphTalk

GraphTalk [Par93a, Par93b] est un Environnement Objet de Développement et d'Utilisation d'Atelier de Génie Logiciel Graphique. Il permet une programmation graphique des modèles des méthodes. Une programmation GraphTalk produit un Atelier de Génie Logiciel dédié à une méthode spécifique qui permet la représentation d'un modèle de Systèmes d'Information.

La procédure de génération d'un AGL avec GraphTalk peut se faire soit à travers un modèle unique, soit en utilisant les quatre parties décrites ci-dessous.

1. **Spécification sémantique** : dans cette partie, sont déclarés les concepts utilisés pour décrire une méthode ainsi que les relations entre ces concepts.

2. **Affectation des propriétés** : cette partie sert à déclarer les propriétés qui peuvent être attachées à tous les concepts et relations définis dans la spécification sémantique. Les propriétés ne peuvent pas être attachées à n'importe quel type de concept ; chaque concept a des types de propriétés qui lui sont propres.

3. **Spécification des formes** : GraphTalk étant un méta-éditeur de formalismes graphiques, cette partie sert à spécifier les formes externes que doivent avoir chacun des composants d'une modélisation lors de l'utilisation d'une instance de l'atelier généré. On peut aussi attacher des propriétés à ces formes (ex : une propriété texte qui définit le nom d'un concept).

4. **Spécification des fenêtres** : la spécification de l'interface homme-machine particulière à l'AGL créé est déclarée. Ici, on peut définir les menus de l'atelier, attribuer des actions particulières aux composants de l'atelier, créer des sous-menus, etc.

La figure 5.1 présente le méta-outil GraphTalk avec deux de ses fenêtres. La fenêtre "GraphTalk - Essai" est la fenêtre principale de l'outil et la fenêtre "GraphTalk - Test" une instance du modèle unique GraphTalk de la fenêtre principale.

Le modèle unique GraphTalk fait en sorte de garder une compatibilité avec d'anciennes versions de l'outil ; il comporte la totalité des composants des quatre autres parties. La fenêtre "GraphTalk - Test" présente les composants de base (entité, graphe, lien, etc.) offerts par l'outil pour la modélisation d'une méthode. On peut voir aussi les types de propriétés (association, booléen, nombre, etc.) et les actions (menu, requête, etc.) qu'on peut attacher aux composants d'une modélisation.

L'utilisation de GraphTalk se fait à deux niveaux différents : le niveau méta-modélisation où est créé l'AGL propre à une méthode et le niveau modélisation où un modèle d'un SI est créé en utilisant l'AGL.

La méta-modélisation est réalisée avec un langage graphique de type méta-langage dans lequel sont pré-définies par GraphTalk quatre méta-classes : objet, lien, dispatcher et propriété. Autour des instances de ces méta-classes, est construit le méta-modèle qui décrit une méthode en offrant le cadre de l'AGL généré.

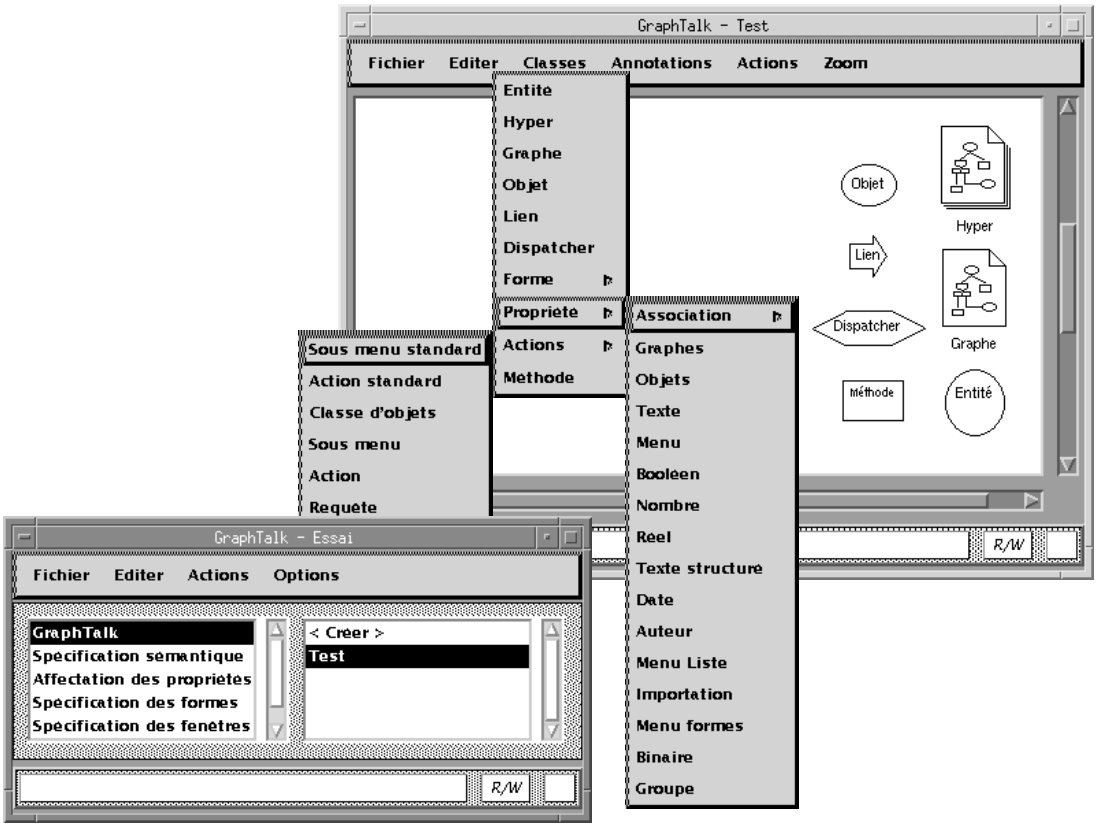


Figure 5.1 - Méta-Outil GraphTalk

Dans la figure 5.1, on peut voir, dans le menu `Classes`, des instances de la méta-classe objet : `Hyper` (qui représente un modèle d'une méthode dans sa totalité en tant que composé d'autres modèles), `Graphe` (qui représente les modèles d'une méthode), `Entité` (qui sert à factoriser des comportements communs à différents composants d'une modélisation), `Objet` (qui sert à déclarer les concepts qui font partie d'une méthode) et `Méthode` (équivalent des méthodes ou fonctions de la programmation orientée objet). La méta-classe lien, appelée `Lien`, permet la définition de la sémantique des relations entre instances de la méta-classe objet ; elle peut avoir, comme instances, différents types de liens : héritage, composition, connexion, etc. La méta-classe `Dispatcher` représente un type particulier de lien et permet la création d'arbres. Les composants du menu `Classes` sont appelés généralement, les nœuds d'une modélisation. Dans la fenêtre "GraphTalk - Test" de la figure 5.1, on peut voir aussi la notation utilisée par ces nœuds dans GraphTalk.

De plus, la figure 5.1 présente les différentes instances de la méta-classe propriété. Ces instances peuvent être attachées aux nœuds d'une modélisation. Nous présentons ci-dessous quelques unes de ces propriétés. La propriété `Texte` par exemple, lorsqu'elle est attachée

à un nœud, ajoute un attribut de type texte à ce nœud. C'est une propriété de ce type qu'on utilise lorsqu'on veut définir qu'un nœud a un composant de nature spécification informelle. La propriété `Objets` sert à déclarer qu'un nœud est composé d'autres nœuds ; c'est le cas par exemple d'une classe avec ses attributs et ses opérations. Pour finaliser, nous citons la propriété `Texte structuré` qui est utilisée pour faire le lien entre une spécification semi-formelle (un modèle GraphTalk) et une spécification formelle (un éditeur LEdit). Le fait d'attacher une telle propriété à un nœud d'un modèle GraphTalk permet simplement de déclarer le lien. Pour mettre en place la conversion GraphTalk-LEdit et vice versa, ainsi que pour assurer la cohérence entre les deux représentations, il est nécessaire de définir des démons (modules de programmes) qui exécuteront cette transformation.

L'option `Action` du sous-menu `Classes` est présentée aussi dans la figure avec ses composants. C'est avec ces composants que nous pouvons redéfinir l'interface homme-machine proposée dans un atelier généré par GraphTalk. Comme exemple, nous pouvons citer les `Sous menu standards` et les `Actions standards` qui permettent la redéfinition des composants standards de l'interface homme-machine et l'action `Action`, à travers laquelle nous pouvons faire appel à des "démons" écrits en C/C++. Ces démons peuvent utiliser l'API GraphTalk afin d'augmenter la puissance du méta-outil. Enfin, l'action `Requête` permet la réalisation de requêtes sur les composants d'une modélisation, en utilisant le langage GQL (similaire a SQL) qui est propre à GraphTalk.

5.1.2 LEdit

Le méta-outil LEdit [Par94] est un méta-éditeur qui offre une interface de programmation utilisable lors de la définition d'un éditeur syntaxique pour des langages définis par des grammaires en BNF (cf. figure 5.2). L'intégration de GraphTalk et LEdit dans un environnement de méta-outils permet au développeur d'AGL de spécifier respectivement les représentations graphiques et les représentations textuelles de l'outil qui est créé.

Dans la figure 5.2, la fenêtre "ledit - ledit.le" présente la définition syntaxique de l'éditeur affiché dans la fenêtre "ledit - Essai". La grammaire d'un langage en LEdit du type BNF ("Backus Naur Form"). Il existe aussi un autre éditeur appelé REdit qui sert à la définition de l'interface homme-machine. À travers REdit, on peut aussi attacher des actions à des nœuds particuliers de la grammaire de la même façon qu'on le fait sur les nœeuds des graphes décrits sous GraphTalk.

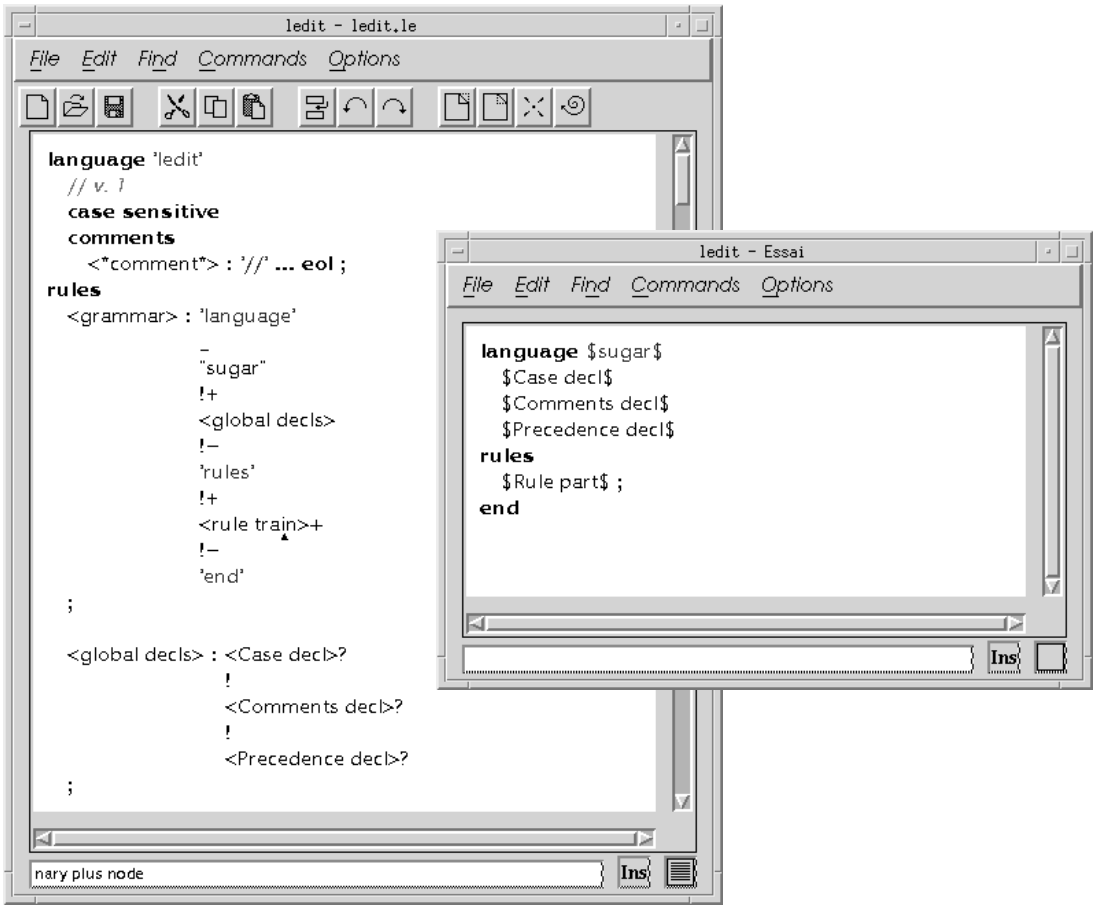


Figure 5.2 - Méta-Outil LEdit

Le méta-outil LEdit permet de décrire les langages cibles dans le cas d'une génération de code à partir d'un modèle de SI : comme par exemple, une génération de code de type squelettes de classes C++, ou langage de définition de données pour un environnement SGBD, ou enfin dans notre contexte squelettes de schémas Z ou Object-Z.

5.1.3 Thot

L'outil Thot est un éditeur de documents structurés. Dans le cadre de l'atelier présenté dans ce travail, il fonctionne comme éditeur hyper-texte à travers lequel un texte descriptif peut accéder aux différentes modélisations. Dans ce prototype d'atelier, il accomplit la tâche de regrouper les trois types de modélisation (informelle, semi-formelle et formelle). En tant qu'éditeur hyper-texte, on peut le classer comme étant du type rédaction ("authoring") et non navigation ("browsing"), selon l'une des dimensions de la classification proposée par F. G. Halasz [Hal88], dans la mesure où on va l'utiliser pour la composition d'un document de spécification.

L'utilisation de l'éditeur Thot passe par la définition de deux types de schémas différents, qui sont présentés ci-dessous.

Schéma de Structure

Ces schémas spécifient principalement les types des éléments utilisables et les relations qui peuvent les relier [QRRV95] ; ils définissent une structure hiérarchique dont les éléments terminaux sont du texte, des images, des graphiques, etc. Ces schémas sont définis avec le langage S offert par l'outil. En utilisant ce schéma de structure, l'utilisateur peut construire un document qui suit une structure logique pré-définie.

La figure 5.3 présente un exemple d'un schéma de structure. Un document créé selon la structure hiérarchique présentée dans la figure, est composé par un `En-tête` composé d'un `Titre`, d'une liste d'`Auteurs` et d'un `Résumé`. Après l'en-tête, le `Corps` du document est composé d'un `Préambule` et d'une `Suite_sections`. Il faut noter que cette définition n'est pas complète car il manque la définition de plusieurs composants (`Auteur, Paragraphe`, etc).

```
STRUCTURE Rapport ;
DEFPRES RapportP ;
ATTR
        Réservé = Integer ;
STRUCT
        Rapport (ATTR Numéro_page = Integer ;
            BEGIN
            En-tête =
              Titre = Contenu ;
              Auteurs = LIST OF (Auteur) ;
              Résumé = LIST OF (Paragraphe) ;
              END ;
            Corps =
              BEGIN
              Préambule = Suite_paragraphes ;
              Suite_sections ;
              END ;
            END ;
```

Figure 5.3 - Thot - Exemple de Schéma de Structure

Schéma de Présentation

Ces schémas spécifient comment chacun des composants d'un document doit être présenté. Les schémas de présentation sont décrits en langage P offert par l'outil.

Le schéma de présentation (cf. figure 5.4) définit la présentation des composants du schéma de structure ; il déclare aussi entre autres, quelles vues (sous-documents) seront présentées et quels documents seront proposés à l'impression. Les compteurs d'un document (numéro de page, numéro de section, etc.) sont aussi déclarés ici. C'est avec les `BOXES` que le schéma de présentation définit effectivement la forme des éléments d'un document. Dans la figure 5.4, on peut voir ainsi la forme que doit avoir le `Titre` d'un document.

```
PRESENTATION Rapport ;
VIEWS
        Texte_integral, Table_des_matières ;
PRINT
        Texte_integral, Table_des_matieres ;
COUNTERS
        CptSect1 : Rank of Section 1 Init Numéro_prem_section ;
BOXES
        BoiteTitre :
            BEGIN
            Content : Text 'Titre haut de page : ' ;
            Style : Bold ;
            Font : Creator = ;
            Size : Creator = ;
            END ;
```

Figure 5.4 - Thot - Exemple de Schéma de Présentation

Les deux exemples de schémas, des figures 5.3 et 5.3, sont des simplifications du schéma `Rapport` fourni avec l'outil. La figure 5.5 présente un document vierge qui utilise les définitions des schémas ci-dessus. On peut voir, dans cette figure, la fenêtre principale de l'éditeur ainsi que les trois vues différentes qui sont définies dans le schéma de présentation.

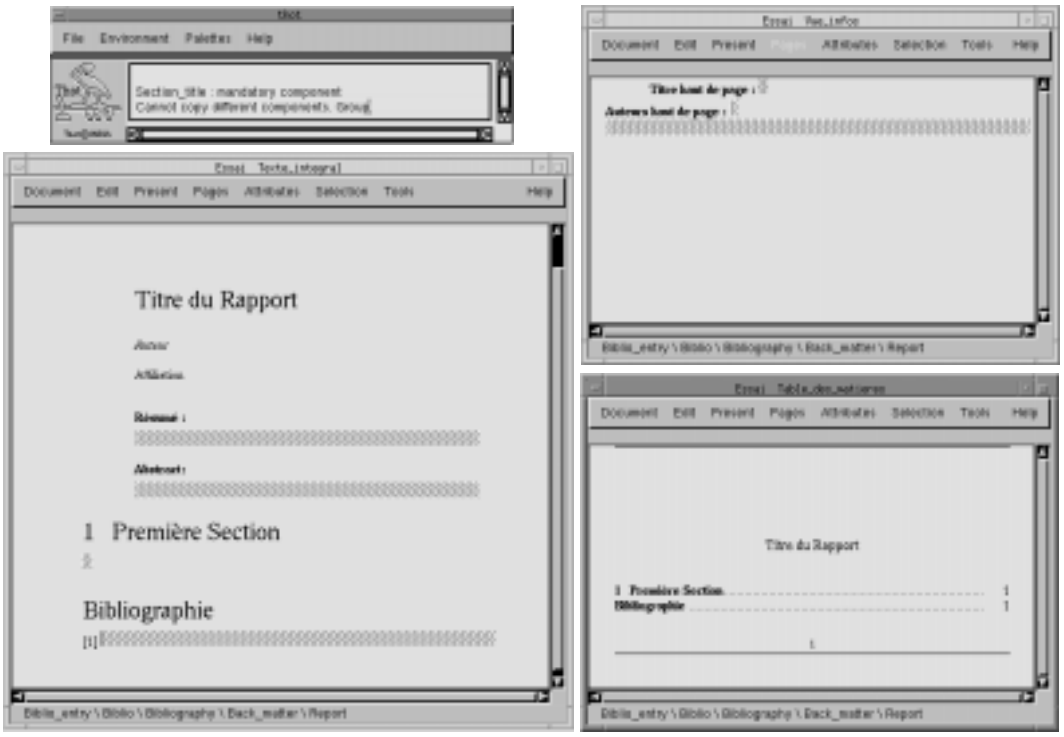


Figure 5.5 - Éditeur et Méta-Éditeur Thot

L'interface de programmation et son mécanisme d'appels externes sont conçus pour faciliter une intégration à d'autres applications ; c'est la raison principale du choix de l'utilisation de Thot comme composant de l'atelier.

5.2 La Réalisation d'un Prototype d'Atelier

Cette section décrit comment l'atelier a été développé grâce aux outils présentés à la section 5.1 et en respectant le méta-modèle proposé à la section 4.2.

5.2.1 Le Gestionnaire de Modèles

Le Gestionnaire de Modèles est adapté aux différents niveaux de modélisation (méta-modélisation et multi-modélisation) à travers trois ateliers : A2M, A2M' et AM. On ne présente dans cette section que des détails de l'implantation de l'atelier A2M, car les deux autres ateliers sont générés à partir de la modélisation initiale produite par A2M pour une méthode spécifique. Dans la section 5.3.1, l'atelier A2M' est décrit lors de la présentation

d'un exemple de méta-modélisation pour les méthodes OMT et OOA. L'atelier AM est illustré dans la section 5.3.3 avec un exemple de multi-modélisation de la méthode OMT.

5.2.1.1 L'Atelier A2M

L'atelier A2M est utilisé dans la modélisation de méthodes orientées objets. Il est construit avec six graphes différents que nous générons sur GraphTalk pour guider ce processus de modélisation en l'adaptant au méta-modèle proposé (cf. section 4.2). Ces graphes sont définis à partir des types de graphes offerts par l'outil GraphTalk : GraphTalk, Spécification sémantique, Affectation des propriétés, Spécification des formes et Spécification des fenêtres (cf. fenêtre "GraphTalk - Essai" de la figure 5.1).

Le premier graphe, de type GraphTalk est utilisé pour donner une vue de l'ensemble des modèles utilisés. Les graphes de type Spécification sémantique, sont utilisés pour déclarer les composants de l'atelier ainsi que les relations entre ces composants. Les graphes de type Affectation des propriétés, sont utilisés pour déclarer les propriétés des composants. Le cinquième, de type Spécification de formes, est utilisé pour définir la notation que les composants de l'atelier généré devront respecter. Finalement, le dernier graphe, de type Spécification de fenêtres, est utilisé dans la définition de l'interface homme-machine de l'atelier qui sera généré.

Nous présentons ci-dessous, chacun de ces six graphes. Ils ne sont pas décrits avec tous leurs composants. Seules les parties les plus importantes pour la compréhension y sont introduites.

1. Graphe GraphTalk : Graphes_A2M

Ce graphe définit l'atelier comme composé de trois modèles différents qui correspondent aux vues du méta-modèle proposé. C'est autour de ces modèles que les composants de l'atelier sont déclarés. Chacun des modèles est défini à travers un graphe différent (cf. figure 5.6).

2. Graphe Spécification Sémantique : S_CR_A2M

Dans ce graphe, sont introduits les concepts qui sont offerts dans l'atelier A2M qui sera généré. Ces concepts, qui sont la représentation des composants présentés dans la figure 4.5, sont utilisés dans la modélisation semi-formelle d'une méthode dans la création des schémas S_Concepts pour cette méthode. La figure 5.7 présente ces composants.

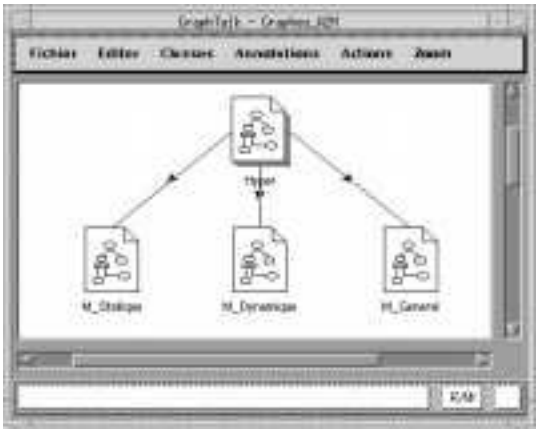


Figure 5.6 - Graphe GraphTalk - A2M

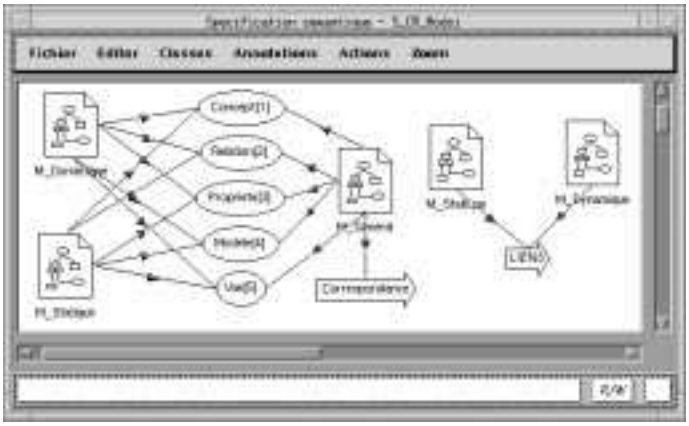


Figure 5.7 - Graphe Spécification Sémantique - Concepts A2M

On peut voir dans la figure 5.7, les concepts qui sont utilisés dans les schémas S_Concepts et qui ont été introduits à la section 4.2.5.2. Ainsi, on remarque les concepts `Concept, Relation, Propriété` et `Modèle`, le concept `Vue` utilisé dans la définition des vues, ainsi que les liens `Correspondance` et `LIEN`. Tous les autres liens de l'atelier (cf. figure 4.5) sont définis par des instanciations du lien `LIEN`, et ne sont pas présentés ici.

Dans GraphTalk, le fait qu'un nœud soit attaché à un graphe, veut dire que, dans l'atelier généré à partir de cette spécification, le concept représenté par ce nœud sera offert dans le graphe équivalent, comme un concept qui peut être utilisé dans ce graphe. Ainsi, les trois modèles, Statique, Dynamique et Général, peuvent proposer les concepts `Concept, Relation, Propriété` et `Modèle` ainsi que

des vues. Pour la même raison, le modèle Général utilise seulement le lien de type `Correspondance`.

3. Graphe Spécification Sémantique : S_SR_A2M

Dans ce graphe, sont déclarés les liens qui peuvent exister entre les concepts déclarés dans le graphe S_CR_A2M. La figure 5.8 présente une partie de la déclaration de la sémantique des liens définis entre ces concepts. On peut y voir la définition de la sémantique du lien `Héritage` (une instance du lien `LIEN` de la figure 5.7) et une partie de la définition de la sémantique du lien `Liaison` (une autre instance). L'instance `héritage` du lien `LIEN` présenté dans le graphe S_CR_A2M, qui représente la relation de type `Héritage` de la figure 4.5 est ainsi utilisée pour déclarer qu'un lien de ce type peut exister entre deux concepts de type `Concept`.

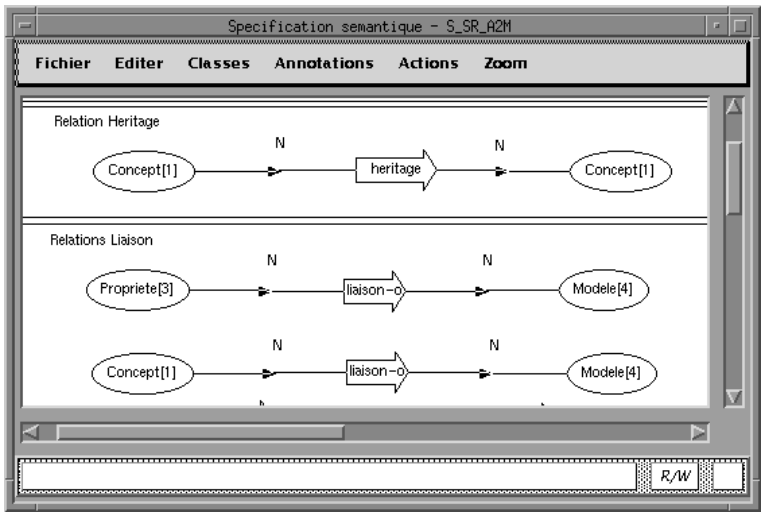


Figure 5.8 - Graphe Spécification Sémantique - Liens A2M

4. Graphe Affectation des Propriétés : P_A2M

Dans cette partie, sont déclarées toutes les propriétés des composants de l'atelier. Une propriété importante (cf. figure 5.9) est la propriété de type Texte Structuré attachée à l'hyper-graphe qui permet la définition de la vue Modèle Formel à travers la création du schéma S_Global. La propriété de type texte appelée `SpecInf` introduit le composant informel du méta-modèle. Ainsi, on peut ajouter une spécification textuelle informelle à chacun des graphes, à l'hyper-graphe et à chacun des concepts d'une modélisation.

Le nœud `Vue` implante la manière par laquelle le Gestionnaire de Modèles et le Gestionnaire de Documents communiquent entre eux. Le nœud `Bottom` est utilisé dans la définition de la notation d'une vue. À travers la propriété de type Objets appelée

Vues, on déclare qu'une Vue peut avoir plusieurs composants. Chacun de ces composants définit une vue. Ainsi au nœud CompVue, qui représente chacune de ces vues, sont attachées des propriétés de type texte, dans lesquelles, lors de la définition d'une vue, sont déclarés les identificateurs des nœuds qui font partie de la vue. Il existent cinq propriétés de type texte, une pour chacun des concepts utilisés pour modéliser une méthode, et une pour les liens entre ces concepts.

Les autres propriétés de la figure 5.9 représentent les noms des éléments et la cardinalité des liens.

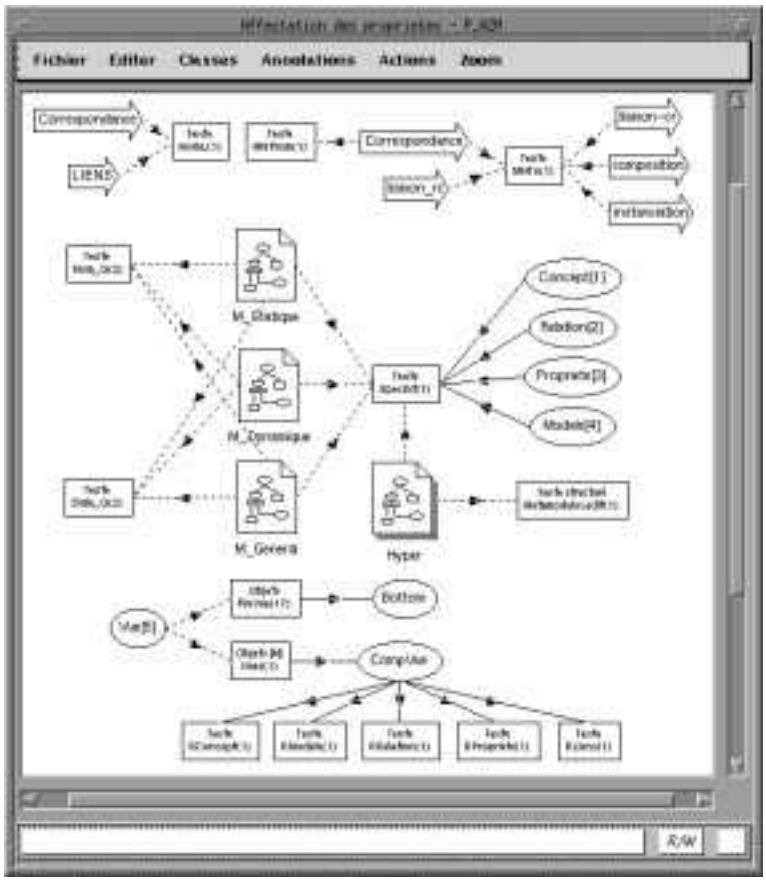


Figure 5.9 - Graphe Affectation des Propriétés - A2M

5. Graphe Spécification des Formes - Fo_A2M

Cette partie déclare les formes que doivent prendre tous les concepts et relations qui ont été introduits dans le graphe S_CR_A2M. La figure 5.10 présente une partie de cette déclaration des formes pour les types de base de la spécification formelle.

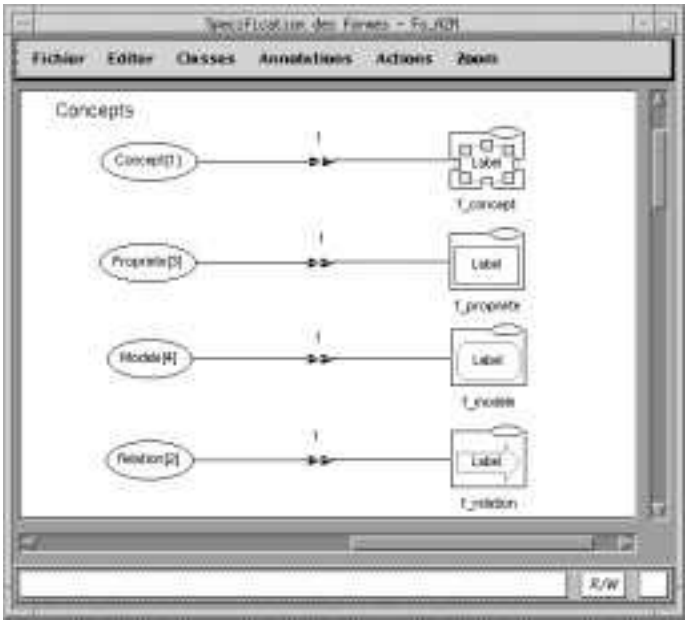


Figure 5.10 - Graphe Spécification des Formes - A2M

6. Graphe Spécification des Fenêtres : Fe_ A2M

Dans cette partie, on modifie l'interface homme-machine standard proposée par Graph-Talk. La figure 5.11 présente une partie de la définition de l'interface qui sera proposée par l'atelier A2M.

Par rapport à l'hyper-graphe, on a ajouté certains éléments et on en a supprimé d'autres. Les éléments supprimés sont les sous-menus standards `File` et `Actions` (ils sont barrés dans la figure). Après leur suppression, ils sont re-déclarés avec les sous-menus `Fichier` et `Actions`. Dans le nouveau sous-menu `Action` est déclarée l'action `Modéliser` qui génère l'atelier A2M' en utilisant le modèle créé avec A2M, à travers l'utilisation d'un démon. Le sous-menu `Spécifications` rend possible l'utilisation des spécifications informelles et formelles du méta-modèle dans l'atelier. Le composant `Spécification Z` de ce sous-menu représente l'appel à l'éditeur syntaxique LEdit pour la présentation de la spécification formelle de la modélisation et le composant `SpécInfor` fournit le moyen d'ajouter une spécification informelle à l'hyper-graphe.

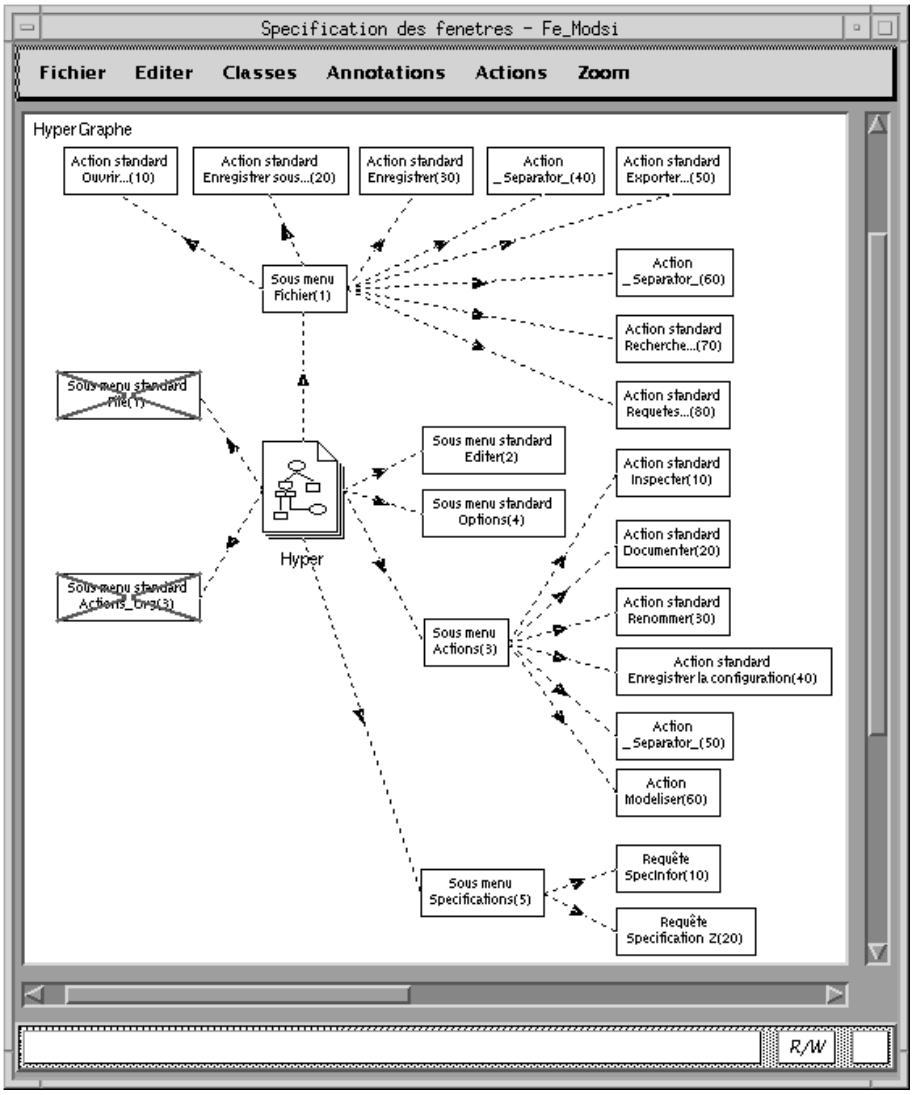


Figure 5.11 - Graphe Spécification des Fenêtres - A2M

Éditeur Syntaxique Z - Méta-Modélisation

Les sections 4.2 et 4.4 ont introduit les deux niveaux de spécification formelle proposés par l'atelier A2M. Le premier niveau représente la vue Modèle Formel, où les types de base ainsi qu'un squelette de la spécification formelle sont construits. Le deuxième niveau, qui représente les schémas S_Spécifications, est produit lors de la construction d'une modélisation. En réalité, comme nous l'avons déjà dit, le schéma S_Spécification correspond au schéma S_Global complété. Ce complément est exécuté d'une manière automatique par l'atelier A2M. À travers le lien existant entre GraphTalk et LEdit, déclaré par la propriété de type texte structuré

`MetamodeleLedit` présentée dans la figure 5.9, et de démons qui sont attachés à des actions standards GraphTalk (création d'un nœud, création d'un lien), au fur et à mesure qu'on construit la modélisation GraphTalk, l'atelier A2M, à travers l'utilisation des démons, complète le schéma S_Global.

La figure 5.12 présente tout simplement le squelette d'une modélisation, le schéma S_Global, avant son remplissage exécuté lors de la construction d'un modèle.

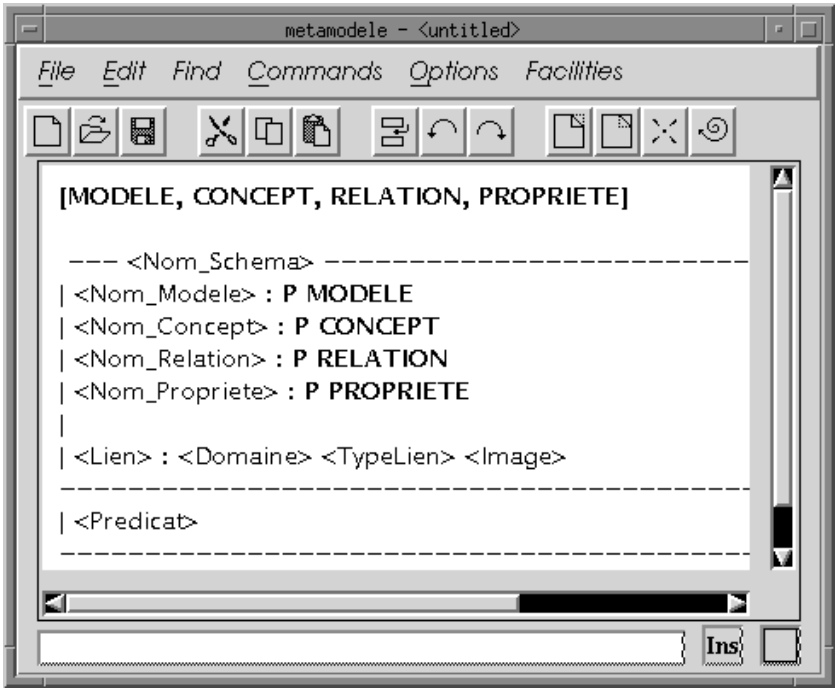


Figure 5.12 - Éditeur Syntaxique LEdit-A2M - S_Global/Méta-Modélisation

5.2.1.2 L'Atelier A2M'

L'atelier A2M' est utilisé dans la finalisation du processus de modélisation d'une méthode orientée objets, initialisé sur A2M, qui vise à générer un AGL pour cette méthode (l'atelier AM). Il est le résultat d'une modélisation exécutée avec l'atelier A2M.

Après avoir modélisé une méthode, avec A2M, celui-ci génère de manière automatique l'atelier A2M'. Cette génération automatique est réalisée à travers une action définie dans l'interface homme-machine. Cette action fait appel à un démon C pour réaliser la génération.

Comme l'atelier A2M' est le résultat de la modélisation d'une méthode spécifique, sa présentation doit se faire par rapport à une modélisation produite dans A2M. Dans la section 5.3.1, nous présentons et expliquons les modèles produits par les ateliers A2M et A2M' lors de la modélisation partielle des méthodes OMT et OOA.

5.2.1.3 L'Atelier AM

L'atelier AM est, quant à lui, généré automatiquement par l'atelier A2M', selon la modélisation d'une méthode réalisée avec A2M et complétée avec A2M'. La présentation de cet atelier est faite dans la section 5.3.3 où nous présentons une multi-modélisation partielle de la méthode OMT. Nous présentons cependant ici les éditeurs syntaxiques LEdit qui implantent les spécification formelles dans les ateliers AM.

Éditeur Syntaxique Z - Modélisation et Multi-Modélisation

Au niveau d'une multi-modélisation gérée par l'atelier AM, la vue Modèle Formel du méta-modèle proposé est implantée à travers l'éditeur syntaxique construit sur LEdit (cf. figure 5.13). On peut voir, dans cette figure, le squelette qui représente la vue Modèle Formel telle qu'elle a été proposée à la section 4.2.5.1. Ce squelette doit être complété par les schémas S_Spécifications pour la déclaration des relations et des structures d'une modélisation.

Figure 5.13 - Éditeur Syntaxique LEdit-AM - S_Global

Éditeur Syntaxique Object-Z - Modélisation et Multi-Modélisation

L'atelier AM est utilisé avec des méthodes orientées objets où le concept principal est le concept de classe. L'éditeur syntaxique, présenté dans la figure 5.14, est responsable de la déclaration de ces classes conforme au langage Object-Z. Dans cet éditeur, l'implantation des schémas S_Spécification pour les classes, correspond à celle présentée dans la figure 2.14.

Figure 5.14 - Éditeur Syntaxique LEdit-AM - S_Spécification pour les Classes

5.2.2 Le Gestionnaire de Documents

C'est avec le Gestionnaire de Documents que nous construisons et coordonnons les documents de spécification d'un système. Un document de spécification est basé sur le canevas de modélisation présenté à la section 4.6. Les deux parties décrites ci-dessous composent le Gestionnaire de Documents selon l'architecture de l'atelier (cf. figure 4.10).

5.2.2.1 Éditeur de Documents

Nous avons construit le schéma Thot qui permet la construction des documents de spécification globale en utilisant le schéma standard `Report` disponible dans Thot. Ce schéma est utilisé normalement pour la construction de documents de type rapport. Nous avons complété ce schéma standard par une partie qui rend possible l'utilisation du canevas de modélisation de la section 4.6. Ainsi la structure d'un document construit avec notre plateforme de modélisation est du type de celle de la figure 5.15.

Figure 5.15 - Gestionnaire de Documents - Document de Spécification Globale

Les sections 1 et 2 du document, sont des sections standards définies dans le schéma `Report` original. La section 3 a été redéfinie afin de rendre possible la présentation d'une modélisation en utilisant les approches informelle, semi-formelle et formelle.

Dans l'item 3.1, nous introduisons les définitions formelles exprimées par les définitions axiomatiques et par les définitions de types de base présents dans le schéma S_Global du système modélisé. Ces définitions formelles peuvent être paraphrasées par des commentaires textuels.

Dans l'item 3.2, nous introduisons effectivement les multi-modélisations qui utilisent les trois approches. La figure 5.16 présente un document vide, qui illustre la présentation externe de ces trois approches.

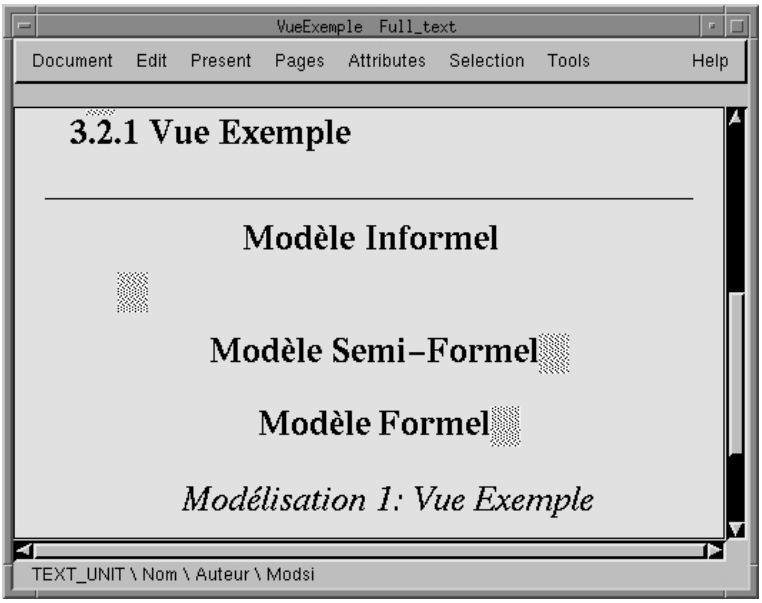


Figure 5.16 - Gestionnaire de Documents - Multi-Modélisation

La présence ou non des spécifications informelles et formelles à l'intérieur de la définition d'une vue (cf. figure 5.16) est déterminée par l'existence ou non de ce composant dans la modélisation de la vue exécutée dans le gestionnaire de modèles.

Les composants informels, lorsqu'ils existent, sont extraits directement par l'éditeur de documents, à partir de la modélisation de la vue. Chaque vue définie dans la structure de stockage maintient une liste avec tous ses composants. Si l'un des composants de la vue a une spécification informelle, nous pouvons y accéder, par l'interaction de démons Thot et GraphTalk, pour l'utiliser dans un document.

Les composants semi-formels sont extraits à travers l'éditeur graphique. Le mode d'extraction est présenté dans la section 5.2.2.2.

Les composants formels, lorsqu'ils existent, sont recherchés par l'éditeur de documents à partir de fichiers générés par l'outil LEdit, pendant la construction d'une modélisation sur le Gestionnaire de Modèles. En utilisant des démons Thot et des références aux composants des vues, l'éditeur de documents récupère dans ces fichiers les *fragments* qui correspondent à la spécification formelle de la vue.

5.2.2.2 Éditeur Graphique

L'éditeur graphique utilise un schéma standard de l'éditeur Thot, le schéma DRAW2, pour la construction des schémas S_Concepts qui représentent la partie semi-formelle d'une modélisation. Le schéma DRAW2 est utilisé normalement dans Thot pour la confection de documents qui ne sont composés que de dessins.

Chaque modèle doit avoir son propre éditeur graphique pour que le schéma S_Concept construit respecte la notation utilisée dans la méthode. Actuellement, nous avons développé deux éditeurs graphiques : un premier pour construire des modélisations semi-formelles pour des méta-modélisations de méthodes, en utilisant la notation présentée dans la figure 4.5 et un autre pour la construction de modélisations semi-formelles pour des modélisations de SI en utilisant la notation de la méthode OMT [RBP+91], plus précisément pour un sous-ensemble de cette méthode qui est défini dans la section 5.3.1.1.

L'éditeur graphique est appelé par l'éditeur de documents à chaque fois que, dans un document de spécification, on veut inclure une spécification semi-formelle. L'éditeur de documents envoie à l'éditeur graphique la vue pour laquelle il veut qu'une modélisation semi-formelle soit construite. L'éditeur graphique, à travers la structure de stockage, récupère une référence à la vue, puis à travers cette référence, "découvre" tous les composants de la vue. Ces composants sont définis dans la vue à travers un identificateur GraphTalk, référence unique gérée par GraphTalk pour chaque élément d'une modélisation.

Après avoir extrait l'identificateur de chaque élément, l'éditeur graphique utilise l'API de GraphTalk et récupère aussi les informations relatives à la position de chacun des éléments dans le graphe GraphTalk sur lequel la vue est définie. Puis, il utilise le schéma DRAW2 et des démons qui précisent la notation à utiliser pour chacun des composants de la vue. Il construit alors un sous-document qui représente le schéma S_Concept de la vue et le rend disponible à l'éditeur de documents pour son inclusion dans le document de spécification.

5.3 Des Études de Cas

Les sections qui suivent présentent quatre possibilités d'utilisation de l'atelier que nous avons expérimentées : (1) la méta-modélisation partielle des méthodes OOA et OMT ; (2) l'utilisation du cadre offert par l'atelier pour la comparaison des méthodes ; (3) la prise en compte de la méta-modélisation de la méthode OMT et de la multi-modélisation avec l'atelier OMT généré ; (4) l'utilisation des modèles créés par cet atelier dans la modélisation du modèle STORM.

5.3.1 Méta-Modélisation des Méthodes OMT et OOA

Nous présentons, dans cette section, une méta-modélisation partielle des méthodes OMT [RBP+91] et OOA [You94]. Le but de cette méta-modélisation étant la présentation des caractéristiques de l'atelier créé, nous ne prenons pas en compte tous les concepts proposés par chacune des méthodes. Ces "sous-ensembles" des méthodes sont appelés ci-après *OMT'* et *OOA'*.

Cette présentation introduit, dans un premier temps, les modèles créés avec l'atelier A2M, puis leur finalisation dans l'atelier A2M'. L'atelier A2M utilisé dans une méta-modélisation est une instance du modèle présenté dans la section 5.2.1.1. La figure 5.17 présente le résultat de cette instanciation.

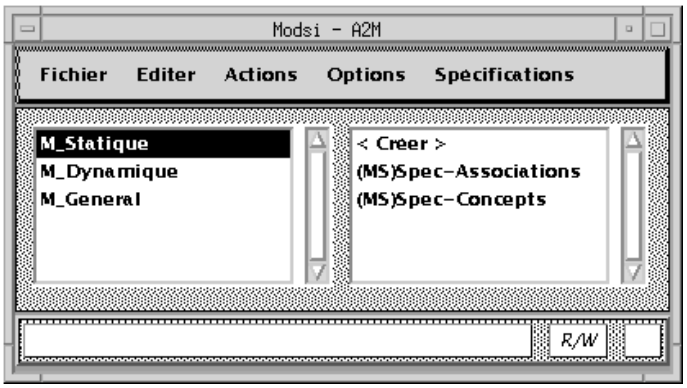


Figure 5.17 - Instance de l'Atelier A2M

Nous pouvons voir dans cette figure trois vues du méta-modèle qui sont représentées à travers les graphes, `M_Statique, M_Dynamique` et `M_Général`. Ces graphes sont présents car ils ont été attachés, dans la modélisation de l'atelier A2M, à un hyper-graphe (cf. figure 5.6). La quatrième vue, la vue Modèle Formel, est accessible à travers le sous-menu `Spécification`. À travers ce sous-menu, nous pouvons aussi attacher une spécification informelle à la totalité du modèle.

Pour chaque instance de l'atelier A2M, les graphes `M_Statique` et `M_Dynamique`, qui représentent les vues Modèle Statique et Modèle Dynamique du méta-modèle, sont définis par deux sous-graphes ; l'un pour la définition de leurs concepts et l'autre pour la définition de liens entre ces concepts. La vue Modèle Général définie par le graphe `M_Général` comporte un seul sous-graphe. Nous pouvons changer le nom de ces sous-graphes et en ajouter d'autres.

Dans la figure 5.17, nous pouvons voir deux de ces sous-graphes : `(MS)Spec-Associations` et `(MS)Spec-Concepts` qui sont des sous-graphes du graphe `M_Statique` qui représente la vue Modèle Statique du méta-modèle proposé.

Dans la présentation de la méta-modélisation des méthodes OMT et OOA les figures utilisées ne sont pas des copies d'écran afin de pouvoir réduire leur taille sans limiter leur lisibilité ; elles représentent des résultats de modélisations dans chacun des ateliers A2M respectifs.

Naturellement, ces méta-modélisations partielles des méthodes OMT et OOA ne sont pertinentes que pour montrer l'intérêt et la réalité des propositions de structuration du méta-modèle (cf. section 4.2) et la combinaison de différentes formes de modélisation.

5.3.1.1 Méta-Modélisation d'OMT'

Le sous-ensemble de la méthode OMT originale [RBP+91] méta-modélisée ici, comprend les concepts clefs du modèle objet (classe et objet) ainsi que les liens et les structures les plus importantes. La partie dynamique de la méthode présente seulement les diagrammes de transition d'états et les diagrammes de flux de données simples (pour simplifier, on n'utilise pas les diagrammes de flux de données orientés objets introduits en Annexe B). Dans la deuxième partie de la méta-modélisation produite avec l'atelier A2M', on ne présente que le modèle objet, bien que l'atelier A2M génère les deux autres modèles.

Atelier A2M

Dans la modélisation de la méthode OMT', nous avons utilisé deux sous-graphes pour le graphe `M_Statique`, quatre pour le graphe `M_Dynamique` et un pour le graphe `M_Général`. Le graphe `M_Statique` est composé des sous-graphes suivants : `(MO)Spéc-Concepts` et `(MO)SpécAssociations`. Le graphe `M_Dynamique` est composé des sous-graphes suivants : `(DE)Spéc-Concepts`, `(DE)Spéc-Associations`, `(MF)Spéc-Concepts` et `(MF)Spéc-Associations`. Enfin, le graphe `M_Général` est composé du sous-graphe `Général`. Chacun de ces sous-graphes est détaillé ci-dessous, ainsi qu'une partie de la spécification formelle générée par l'atelier. La notation utilisée sur les liens des figures présentées ci-dessous est celle introduite dans la figure 4.5.

1. Graphe M_Statique : (MO)Spéc-Concepts

Dans ce graphe, sont déclarés les composants qui, dans OMT', sont en rapport avec l'aspect statique d'un modèle. L'aspect statique dans OMT' est traité par le Modèle Objet. La figure 5.18 présente le modèle objet et ses composants. Les attributs et les opérations d'une classe sont déclarés comme des propriétés du concept `Classe`. Le nom, les rôles et les cardinalités d'une association sont déclarés comme des propriétés du concept `Association`.

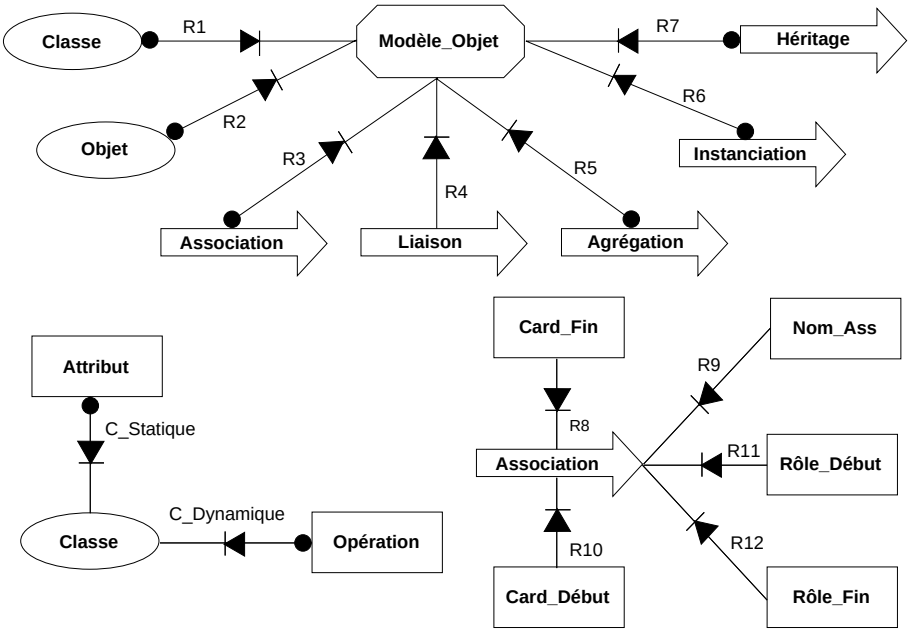


Figure 5.18 - A2M-OMT' - Modèle Statique - Graphe (MO)Spéc-Concepts

2. Graphe M_Statique : (MO)Spéc-Associations

Ce graphe déclare la sémantique des liens du modèle objet de la méthode OMT'. Par exemple, il existe une relation appelée `Instanciation` entre le concept `Classe` et le concept `Objet`.

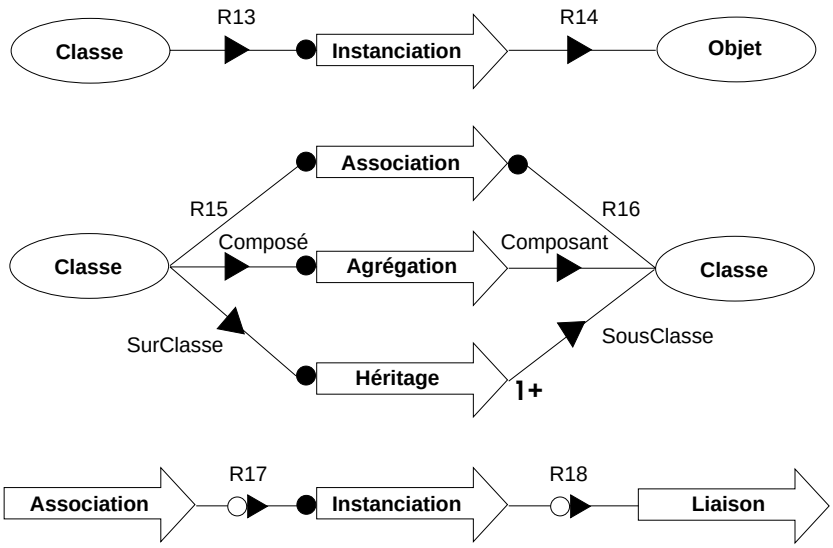


Figure 5.19 - A2M-OMT' - Modèle Statique - Graphe (MO)Spéc-Associations

3. Graphe M_Dynamique : (DE)Spéc-Concepts

Ce graphe (cf. figure 5.20) correspond à la description de Diagrammes de Transition d'États utilisés dans OMT' pour traiter l'aspect dynamique d'un système. Ce type de diagramme est représenté par les concepts de type `Modèle` appelé `Diagramme_Etat`. Un tel diagramme dans OMT' peut avoir comme composants trois concepts de type `Concept` (`Etat`, `Etat_Initial` et `Etat_Final`) et un concept de type `Relation` (`Transition_Etat`).

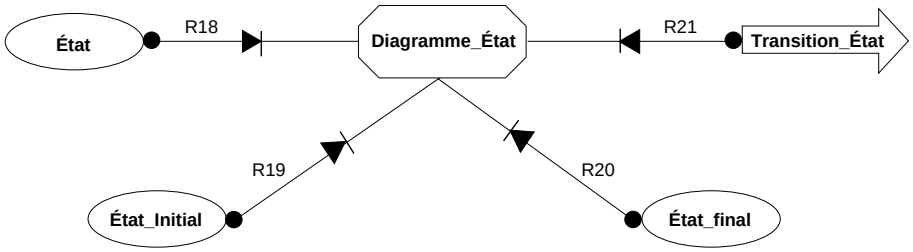


Figure 5.20 - A2M-OMT' - Modèle Dynamique - Graphe (DE)Spéc-Concepts

4. Graphe M_Dynamique : (DE)Spéc-Associations

Ce graphe (cf. figure 5.21) présente la sémantique associée à la création des diagrammes de transitions d'états dans la méthode OMT'. Des relations de type `Lien orienté` sont déclarées entre des concepts de types `Concept`, `État_Initial` et `État` ainsi qu'entre `État` et `État_Final`. Une relation de type `Lien non-orienté` est déclarée entre les concepts `État`.

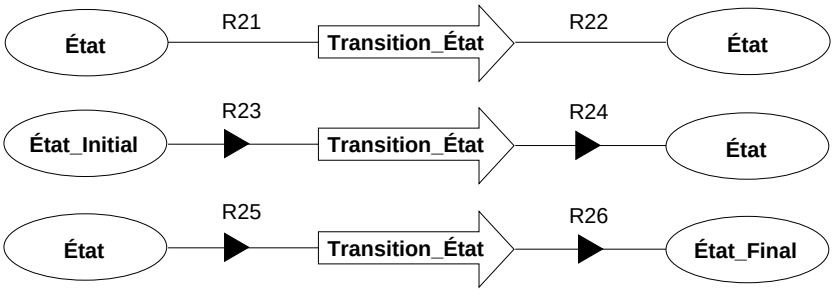


Figure 5.21 - A2M-OMT' - Modèle Dynamique - Graphe (DE)Spéc-Associations

5. Graphe M_Dynamique : (MF)Spéc-Concepts

Un dernier diagramme utilisé dans OMT' pour traiter l'aspect dynamique d'un système est le Modèle Fonctionnel. La figure 5.22 présente la méta-modélisation de ce diagramme avec ses composants. Un concept de type `Propriété` (`Nom_Flux`) est utilisé pour déclarer les noms des concepts de type `Relation` utilisés dans le Modèle Fonctionnel.

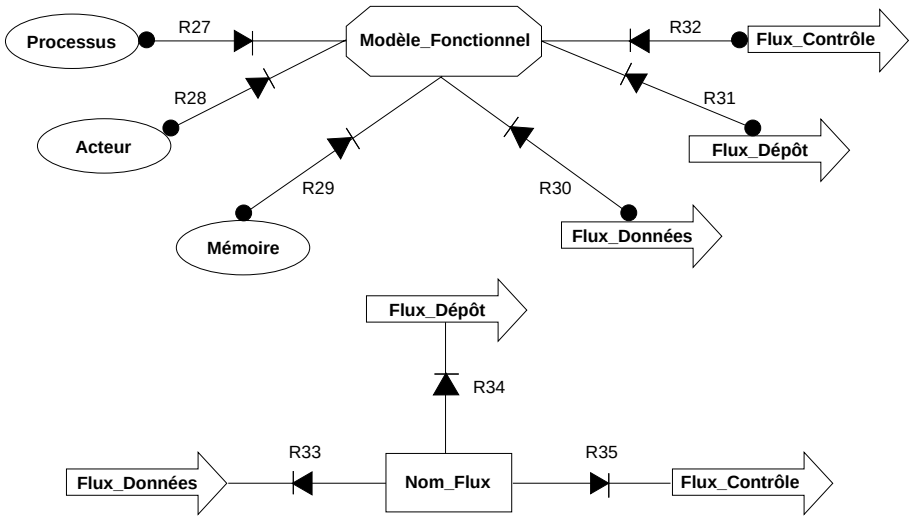


Figure 5.22 - A2M-OMT' - Modèle Dynamique - Graphe (MF)Spéc-Concepts

6. Graphes M_Dynamique : (MF)Spéc-Associations

Ce graphe est celui de la sémantique de création de liens dans des modèles fonctionnels. On peut voir, par exemple, qu'entre deux `Processus` il ne peut exister qu'une `Relation` appelée `Flux_Contrôle` et, qu'entre `Mémoire` et `Processus`, on peut avoir deux concepts de type `Relation` : `Flux_Données` et `Flux_Dépôt`.

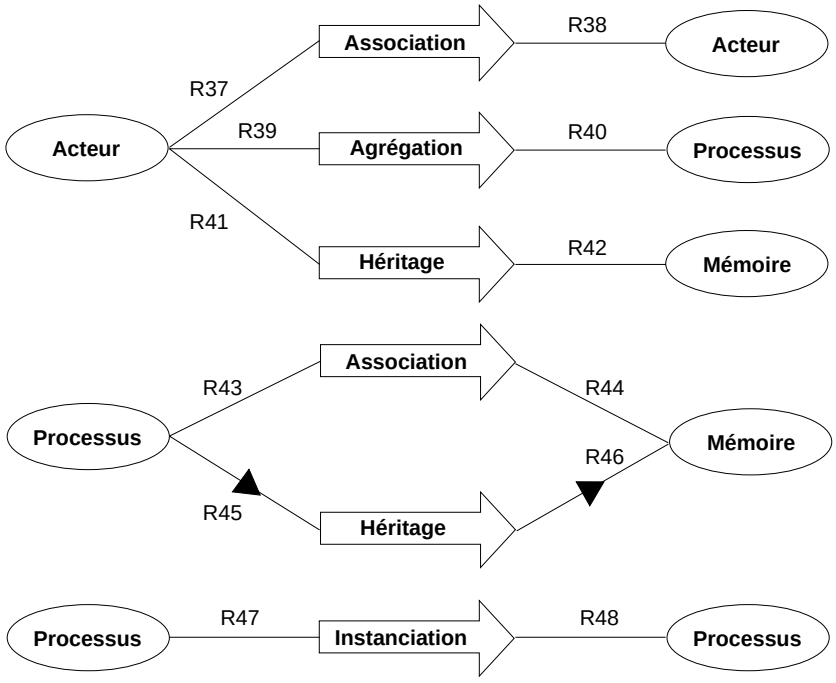


Figure 5.23 - A2M-OMT' - Modèle Dynamique - Graphe (MF)Spéc-Associations

7. Graphes M_Général : Général

Ce graphe représente les liens sémantiques entre des concepts différents. Ainsi, dans la méthode OMT', chaque classe a un diagramme de transition d'états ; on représente donc, dans le graphe Général (cf. figure 5.24), une relation de type `Correspondance` entre `Classe` et `Diagramme_Etat`. Pour la même raison, nous représentons l'autre relation entre `Opération` du modèle objet et `Modèle_Fonctionnel`.

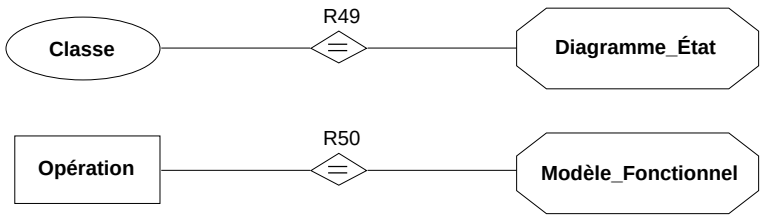


Figure 5.24 - A2M-OMT' - Modèle Général - Graphe Général

Spécification Formelle

Le processus de modélisation de la méthode OMT' exécuté avec l'atelier A2M crée une "image formelle" du modèle. Ainsi qu'il est dit en 5.2.1.1, à la partie Spécification Formelle, cette image est créée automatiquement par l'atelier au fur et à mesure de la construction de la modélisation semi-formelle. Une partie

de cette image est présentée dans la figure 5.25. On peut voir la spécification formelle du sous-graphe `(MO)Spéc-Concepts` présenté dans la figure 5.18. La figure 5.25 présente une copie d'écran de l'éditeur LEdit qui implante le schéma S_Global. Comme on peut le remarquer, le schéma S_Global est complété et dans ce cas il s'agit du schéma S_Spécification pour le schéma S_Concept qui représente le graphe `(MO)Spéc-Concepts`.

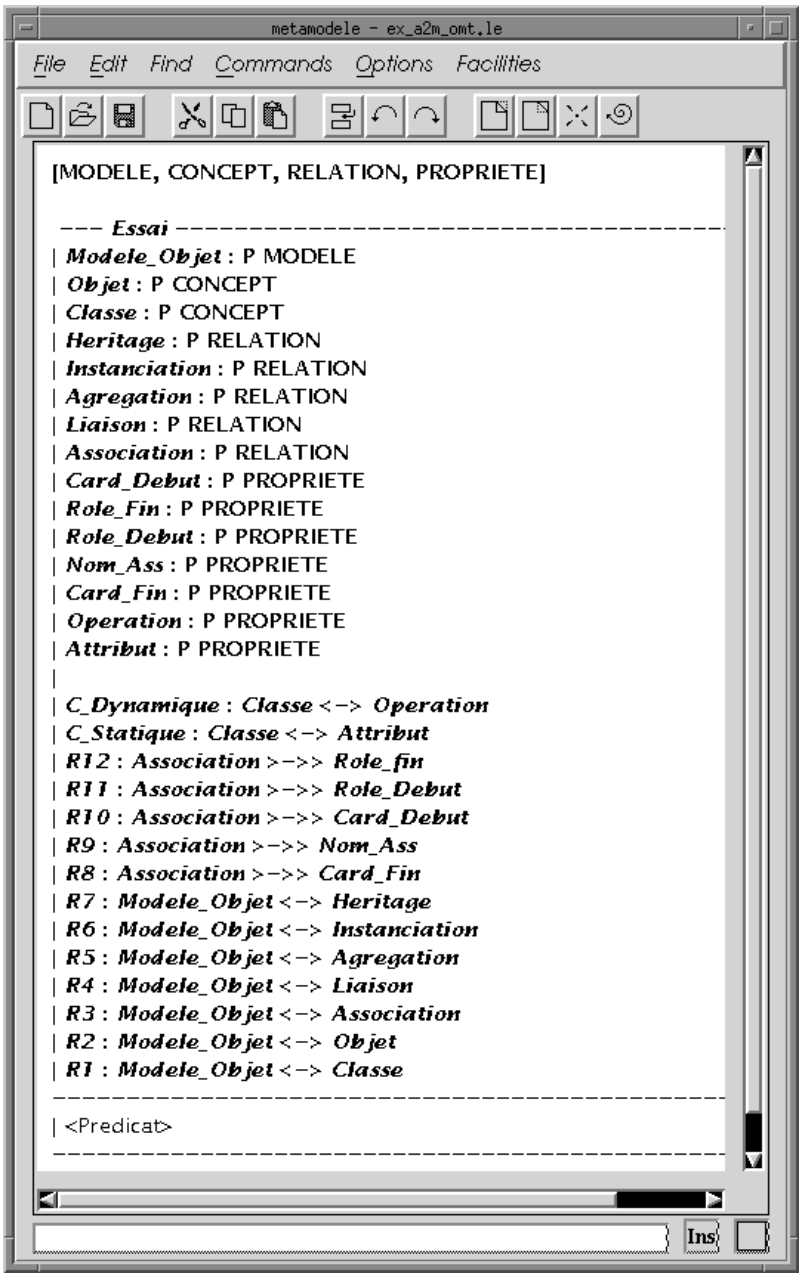


Figure 5.25 - A2M-OMT' - Spécification Formelle pour (MO)Spéc-Concepts

Spécification Informelle

Nous pouvons remarquer dans la figure 5.9 que chaque graphe qui représente une vue du méta-modèle proposé, est lié à une propriété GraphTalk de type Texte appelée `SpecInf`. Ceci permet de définir une spécification informelle pour le graphe `(MO)Spéc-Concepts`, sous-graphe du graphe `S_Statique`, donnée en exemple dans la figure 5.26.

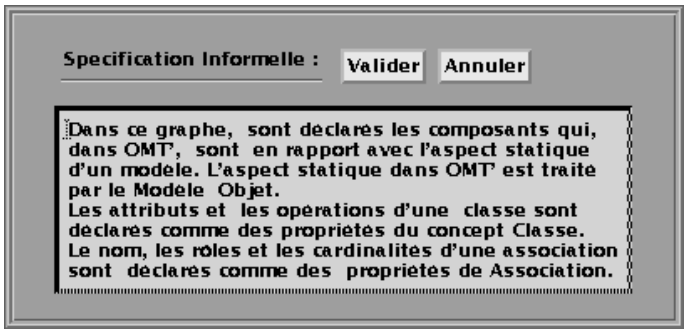


Figure 5.26 - A2M-OMT' - Spécification Informelle pour (MO)Spéc-Concepts

Atelier A2M'

L'atelier A2M' offre un cadre (généré par A2M) pour guider le processus de modélisation. Ainsi, pour chacun des concepts du type `Modèle` déclarés dans A2M, ce dernier génère un graphe GraphTalk équivalent dans l'atelier A2M' correspondant.

L'atelier A2M' est construit automatiquement par l'atelier A2M ; c'est un atelier standard GraphTalk avec les cinq graphes présentés dans la figure 5.1. L'atelier A2M crée un sous-graphe appelé `(Hyper)Atelier_OMT` sous le graphe `GraphTalk` de A2M'. Chacun des concepts du type `Modèle` déclarés dans A2M est attaché à l'hyper-graphe (cf. fenêtre "GraphTalk - (Hyper)Atelier_OMT" de la figure 5.27) afin que l'atelier AM qui sera généré par A2M' ait un graphe GraphTalk pour chacun de ces `Modèles`. Les quatre autres parties ont un sous-graphe pour chaque concept du type `Modèle` déclaré dans A2M. La figure 5.27 présente ces sous-graphes pour la partie spécification sémantique.

Chaque propriété déclarée dans A2M peut être redéfinie dans A2M' comme étant, soit un nœud, soit une propriété GraphTalk. Ainsi, on peut faire la différence entre une propriété qui représente, par exemple, le nom d'un concept et une autre qui est elle-même un concept (un attribut). On peut choisir le type (texte, menu, etc.) des propriétés redéfinies comme des propriétés GraphTalk.

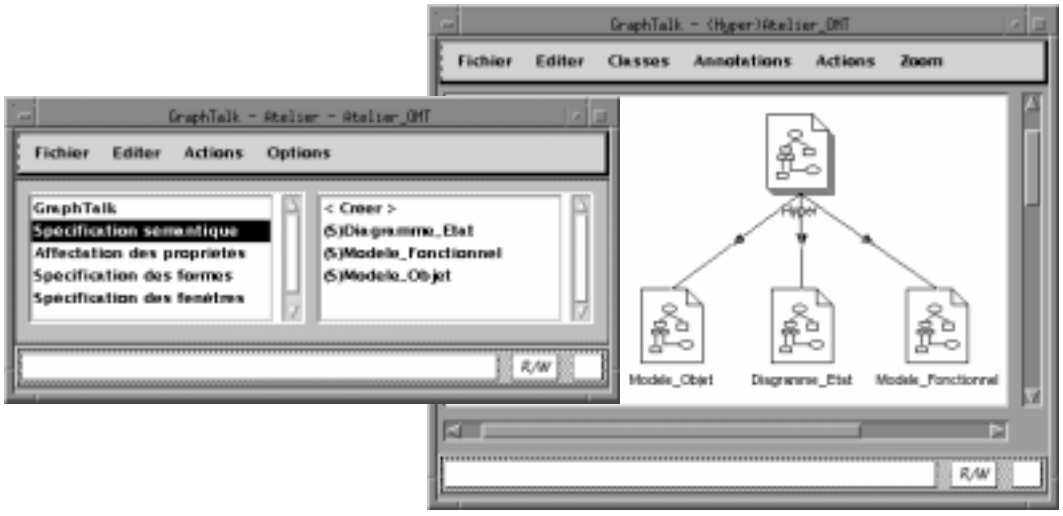


Figure 5.27 - A2M' - Atelier_OMT

L'atelier A2M' a pour fonctions principales la définition des formes des concepts déclarés dans A2M pour préparer leur utilisation dans l'atelier final AM et l'inclusion d'éléments GraphTalk (ex : un dispatcher) qui rendront l'atelier AM généré par A2M' compatible avec la méthode méta-modélisée. On présente donc, ci-dessous, les sous-graphes `(S)Modèle_Objet, (P)Modèle_Objet` et `(Fo)Modèle_Objet` (respectivement les parties Spécification Sémantique, Affectation des Propriétés et Spécification des Formes de GraphTalk) pour le Modèle Objet, afin de préciser ce qui est exécuté dans A2M'. On ne présente pas le sous-graphe (Fe)Modèle_Objet, car il s'agit simplement de la définition de l'interface homme-machine offerte dans l'atelier AM. On ne présente pas non plus, les graphes relatifs au Diagramme d'États et au Modèle Fonctionnel car ils sont semblables aux graphes présentés pour le Modèle Objet.

1. Graphe Spécification Sémantique : (S)Modèle_Objet

La figure 5.28 introduit une partie de la Spécification Sémantique réalisée dans l'atelier A2M'. Les éléments présentés sont soit générés directement par A2M (ceux entourés par un cadre), soit ajoutés dans A2M'. Le graphe `Modèle_Objet` provient du concept du type `Modèle` de l'atelier A2M. Les liens `Instanciation, Association, Agrégation, Héritage` et `Liaison` proviennent des concepts du type `Relation` de l'atelier A2M. Les nœuds `Classe` et `Objet` proviennent des concepts du type `Concept` de l'atelier A2M. Les nœuds `Attribut` et `Opération` proviennent des concepts du type `Propriété` de l'atelier A2M.

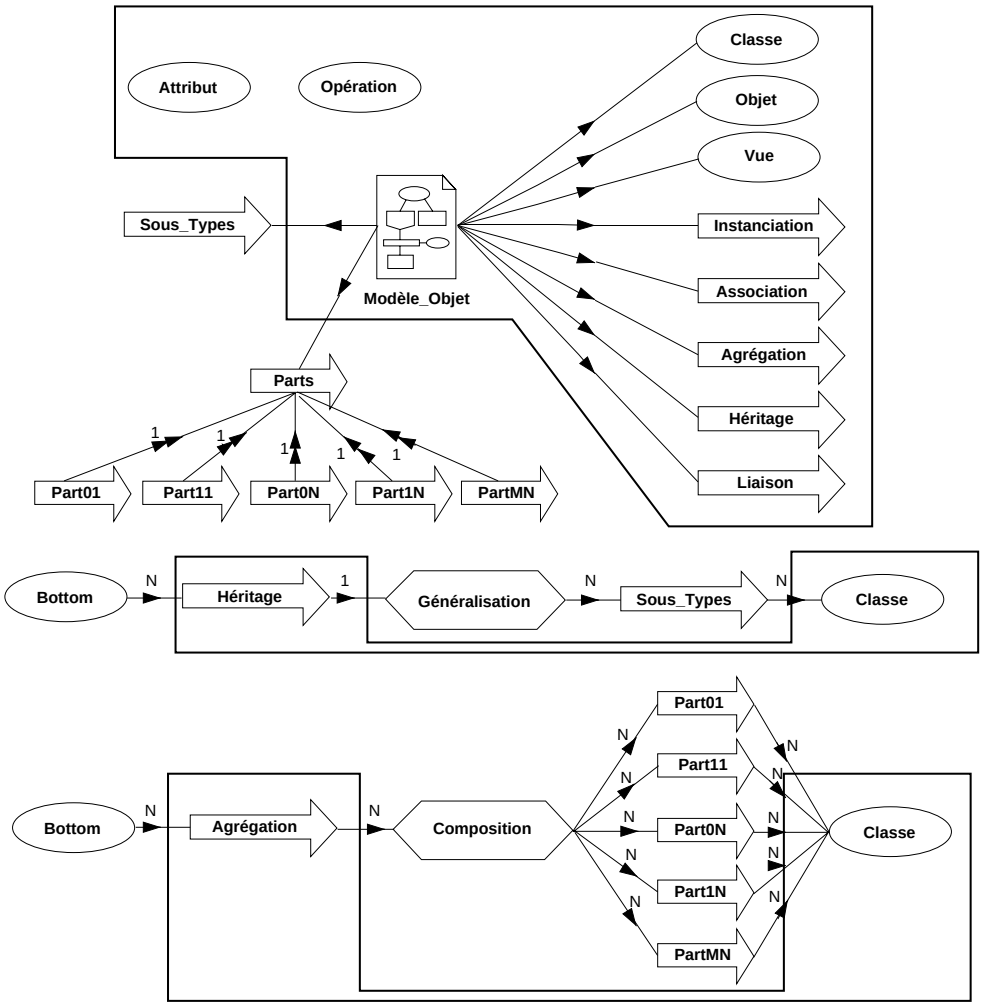


Figure 5.28 - A2M'- OMT' - Spécification Sémantique - (S)Modèle_Objet

L'élément `Vue` attaché au graphe `Modèle_Objet`, rend possible le contrôle de vues dans l'atelier AM qui sera généré. Tout atelier A2M' généré par A2M a un élément `Vue` attaché à chacun des graphes qui représente un modèle dans une modélisation A2M.

Les éléments qui ne sont pas encadrés ont été ajoutés manuellement afin d'apporter plus de cohésion sémantique et graphique à la modélisation générée par A2M. Par exemple, le dispatcher `Généralisation` et le lien `SousType` sont introduits pour qu'ils existent sémantiquement et graphiquement dans l'atelier AM ; ils correspondent à la possibilité de création d'arbres d'héritage. Le dispatcher `Composition` et les liens `PartXX` sont utilisés de manière analogue pour les structures d'agrégation. Le nœud `Bottom` est utilisé simplement dans

la description de la forme d'une classe (voir la partie Spécification des Formes - A2M').

2. Graphe Affectation des Propriétés : (P)Modèle_Objet

La figure 5.29 présente une partie de l'Affectation des Propriétés dans l'atelier A2M' ; comme dans la partie Spécification Sémantique, les éléments encadrés sont générés automatiquement par A2M et les autres sont ajoutés manuellement dans A2M'.

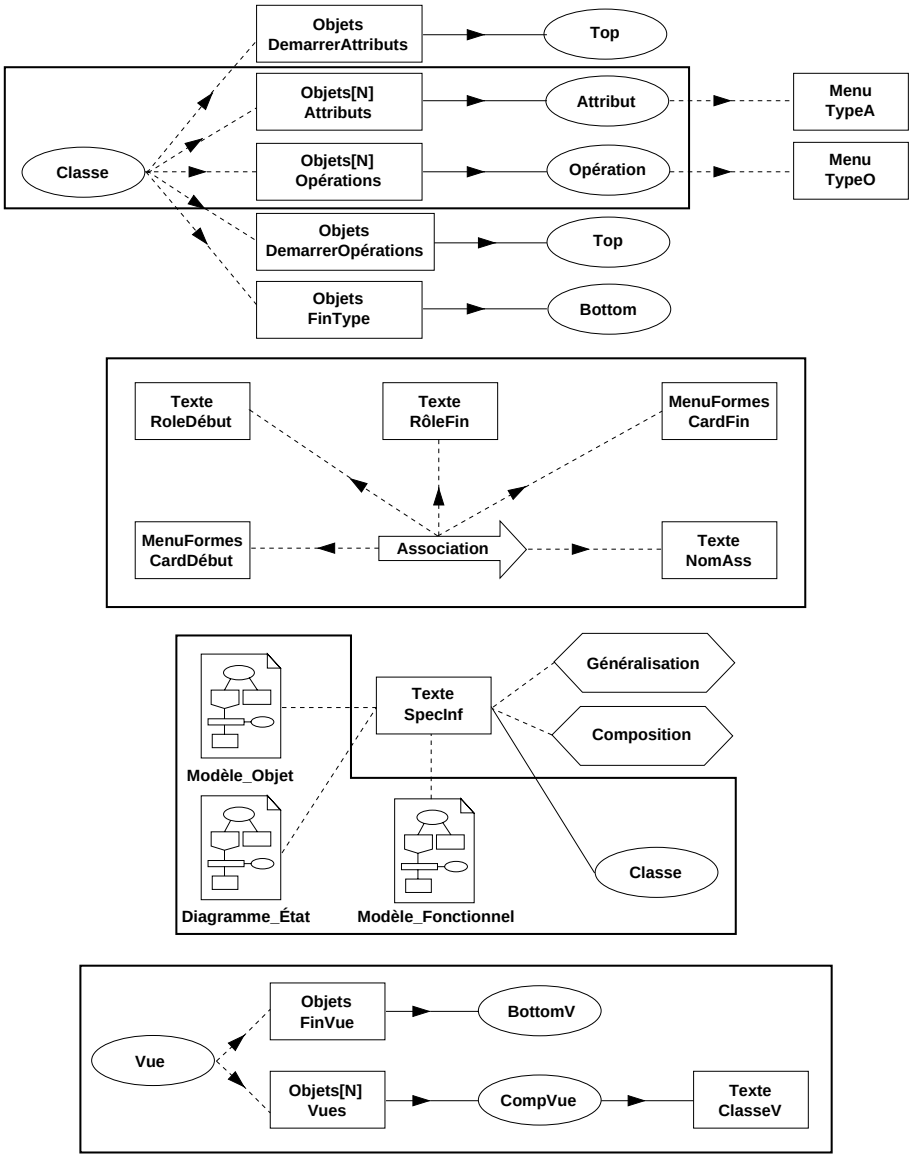


Figure 5.29 - A2M'-OMT' - Affectation des Propriétés - (P)Modèle_Objet

Lors de la génération de l'atelier A2M' par A2M, nous avons choisi que les propriétés `Attribut` et `Opération` de l'atelier A2M soient des nœuds dans A2M'. Grâce à cela et parce que ces propriétés sont attachées à un concept du type `Concept`, A2M crée une propriété GraphTalk de type `Objets` appelée `Attributs` entre `Classe` et `Attribut` et une autre entre `Classe` et `Opération` appelée `Opérations` ; cela veut dire dans GraphTalk que ces deux nœuds dépendent du premier. Les autres éléments attachés à `Classe` le sont pour permettre la gestion de sa notation. Les propriétés du lien `Association` d'A2M (cf. figure 5.18) sont aussi présentées ; celles-ci ont été choisies comme étant des propriétés GraphTalk de type `Texte`. On peut voir aussi, comment à travers l'ajout manuel de la propriété `SpecInf` dans A2M', on rend possible la spécification informelle pour chacun des modèles, pour le concept `Classe` ainsi que pour les structures d'héritage et d'agrégation dans l'atelier AM.

On peut voir de la même façon, dans la figure 5.29, la déclaration des éléments qui rendent utilisable le concept de vues. Ces éléments sont générés automatiquement par A2M. Ainsi, l'élément `BottomV` sert à définir la notation d'une `Vue` et la propriété de type `Objets` appelée `Vues` sert à déclarer explicitement les vues. Un concept `Vue`, attaché à un graphe qui représente un modèle (cf. figure 5.28), est composé d'une ou plusieurs vues : l'élément `CompVue`. Chacune des vues définies par `CompVue` a une propriété de type `Texte` où est déclaré l'identificateur des classes qui composent une vue dans l'atelier AM.

3. Graphe Spécification des Formes - (Fo)Modèle_Objet

Tous les concepts définis dans A2M ont une forme dans la partie Spécification des Formes de A2M'. Cela permet à l'atelier AM d'utiliser la notation propre à la méthode spécifique qui est l'objet de la méta-modélisation. La figure 5.30 présente quelques nœuds et leurs formes. Les éléments encadrés sont aussi générés par A2M ; cependant la définition explicite de la forme est faite dans A2M'. Les éléments en dehors du cadre sont utilisés pour créer la "boîte" qui représente une classe. On peut voir aussi que les propriétés `TypeA` et `TypeO` présentes dans la partie affectation des propriétés de A2M' (cf. figure 5.29) sont incluses respectivement dans la définition de la forme des attributs et des opérations pour les rendre utilisables.

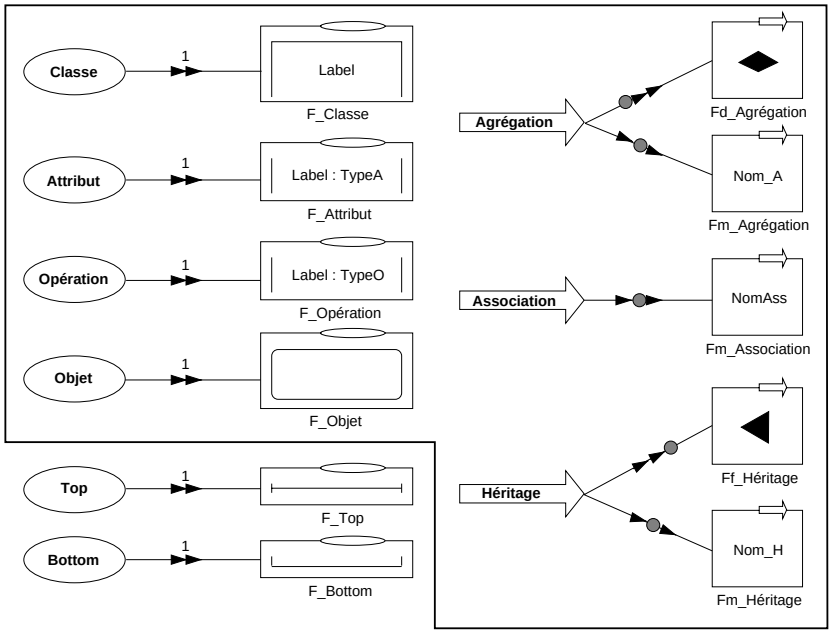


Figure 5.30 - A2M'-OMT' - Spécification des Formes - (Fo)Modèle_Objet

5.3.1.2 Méta-Modélisation d'OOA'

La méthode OOA' que nous définissons ci-dessous comprend, elle aussi, les principaux concepts de la méthode OOA [You94]. Dans sa méta-modélisation partielle présentée ci-après, on utilise les classes et les objets avec les attributs et services, les liens ainsi que les structures de généralisation et de composition pour la définition d'un modèle objet qui définit la partie statique. La partie dynamique est décrite à l'aide des diagrammes de l'histoire des objets (équivalents à des diagrammes de transitions d'états) et des connexions de messages.

Atelier A2M

Chacun des graphes qui composent la modélisation d'OOA' réalisée avec l'atelier A2M est présenté ci-après. La vue Modèle Statique du méta-modèle est donnée par les sous-graphes `(MO)Spéc-Concepts` et `(MO)Spéc-Associations`. La vue Modèle Dynamique du méta-modèle est représentée par les sous-graphes `(MD)Spéc-Concepts` et `(MD)Spéc-Associations`. Enfin, le sous-graphe `Général` représente la vue Modèle Général. Chacun de ces sous-graphes est détaillé ci-dessous ainsi qu'une partie de la spécification formelle générée par l'atelier.

1. Graphe M_Statique : (MO)Spéc-Concepts

Une partie de l'aspect statique de la méthode OOA' est déclarée dans la figure 5.31. Un modèle objet, qui n'existe pas explicitement dans OOA, est introduit et

regroupe les concepts utilisés pour déclarer les aspects statiques d'un système. Les propriétés de `Classe` et d'`Association` sont aussi déclarées.

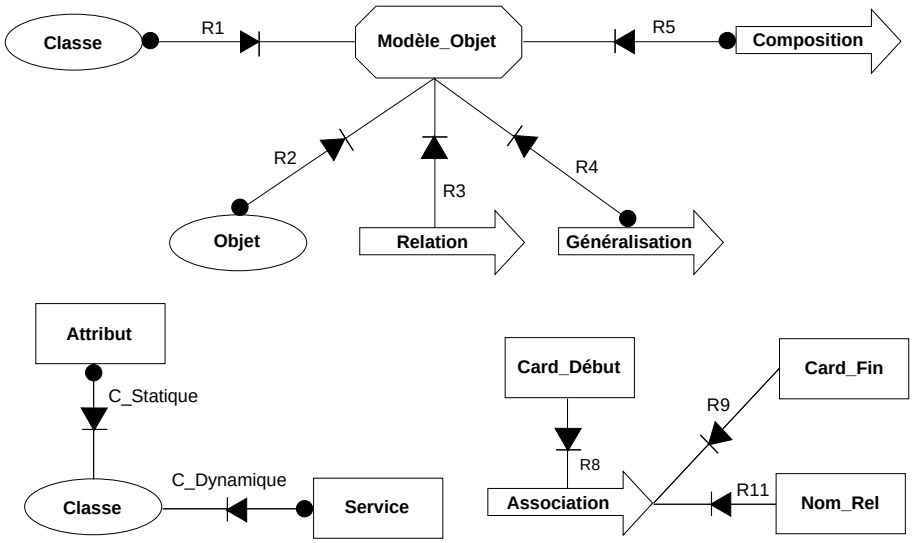


Figure 5.31 - A2M-OOA' - Modèle Statique - Graphe (MO)Spéc-Concepts

2. Graphe M_Statique : (MO)Spéc-Associations

La description des liens qui peuvent exister entre les éléments du modèle objet introduits dans la figure 5.31 est détaillée dans la figure 5.32.

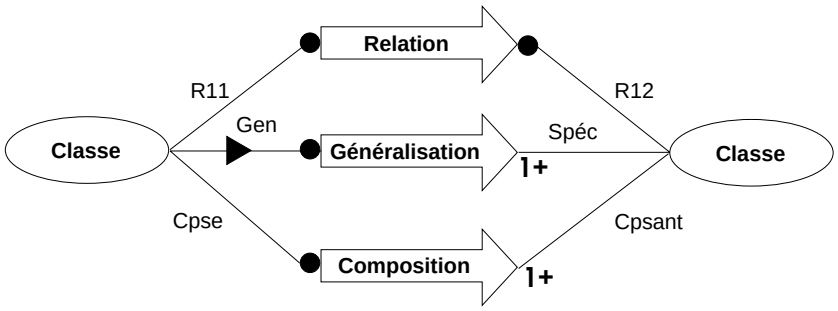


Figure 5.32 - A2M-OOA' - Modèle Statique - Graphe (MO)Spéc-Associations

3. Graphe M_Dynamique : (MD)Spéc-Concepts

La figure 5.33 montre les éléments utilisés dans OOA' pour représenter la dynamique d'un système : les diagrammes d'états sont composés des `Etats` et des `Transition_Etats` avec les propriétés `Condition` et `Action`. Le concept de type `Relation` appelé `Conn_Message` est aussi utilisé pour décrire la dynamique d'un système, mais il n'appartient pas aux diagrammes d'états.

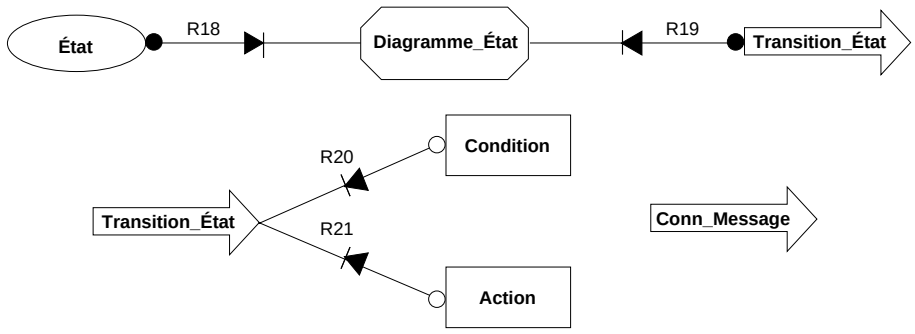


Figure 5.33 - A2M-OOA' - Modèle Dynamique - Graphe (MD)Spéc-Concepts

4. Graphe M_Dynamique : (MD)Spéc-Associations

La figure 5.34 présente la sémantique des relations utilisées pour décrire la dynamique dans OOA'. Le lien `Conn_Message` entre `Classes` représente la demande d'un service d'une classe à une autre. Entre une `Classe` et un `Objet`, il représente l'instanciation d'un objet.

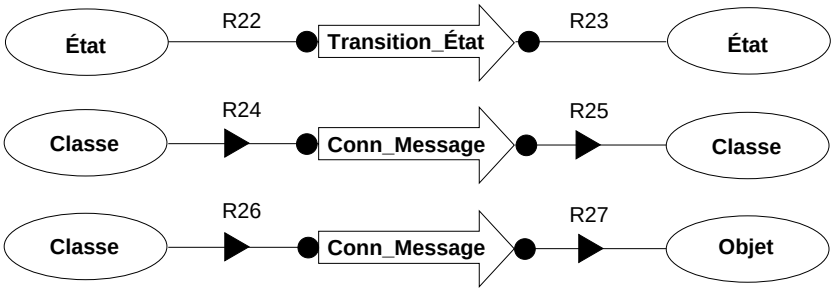


Figure 5.34 - A2M-OOA' - Modèle Dynamique - Graphe (MD)Spéc-Associations

5. Graphe M_Général : Général

Le seul lien sémantique entre concepts différents existants dans OOA' est présenté dans la figure 5.35 et montre qu'un `Objet` est lié à un `Diagramme_Etat`.

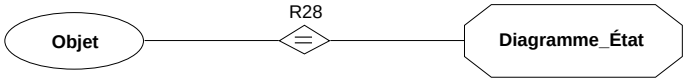


Figure 5.35 - A2M-OOA' - Modèle Général - Graphe Général

Spécification Formelle

Une partie de l'"image formelle" du modèle créé lors du processus de méta-modélisation exécuté avec l'atelier A2M est présentée dans la figure 5.36. Dans cette figure nous pouvons voir le schéma S_Spécification pour le graphe `(MO)Spéc-Concepts` (cf. figure 5.31). Tous les concepts et liens existants entre ces concepts pour ce graphe sont représentés.

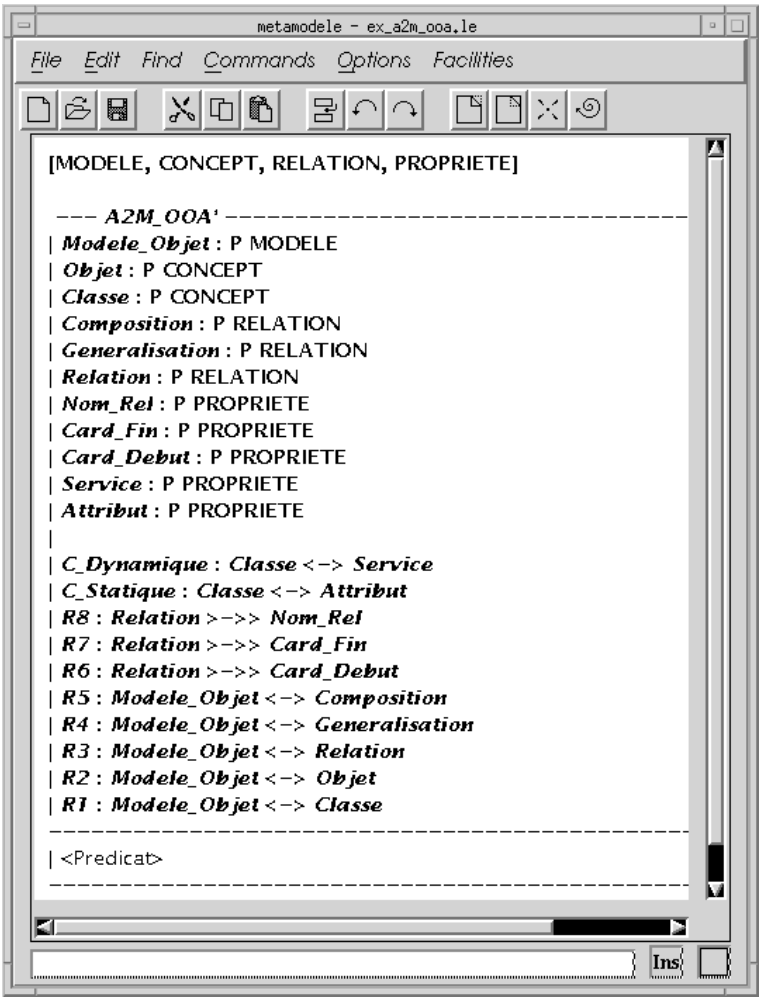


Figure 5.36 - A2M-OOA' - Spécification Formelle pour (MO)Spéc-Concepts

Spécification Informelle

Dans la figure 5.37 nous pouvons voir la spécification informelle attachée au sous-graphe `(MO)Spéc-Concepts` du graphe `S_Statique`.

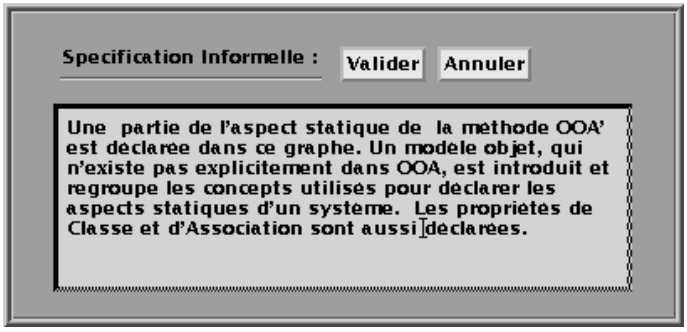


Figure 5.37 - A2M-OOA' - Spécification Informelle pour (MO)Spéc-Concepts

Atelier A2M'

La finalisation de la méta-modélisation qui doit être exécutée avec l'atelier A2M' ne sera pas présentée ici car elle est du même genre que celle exécutée pour la méthode OMT' (cf. page 146). Comme pour la méthode OMT', il faut ajouter la forme des concepts déclarés dans A2M ainsi que quelques concepts GraphTalk nécessaires à la génération, par l'atelier A2M', d'un atelier AM compatible avec la méthode OOA'.

5.3.2 Cadre pour la Comparaison des Méthodes

Dans la section 3.2.5, nous avons comparé quatre méthodes orientées objets à l'aide de tables. La réflexion sur cette comparaison (cf. section 3.2.5.5) présente quelques travaux qui utilisent un méta-modèle pour la comparaison des méthodes dans la mesure où cette comparaison est rendue plus facile lorsqu'une approche basée sur les méta-modèles est utilisée. Dans cette section, nous utilisons des concepts de notre méta-modèle afin d'illustrer une comparaison des méthodes. Cette illustration porte sur les méthodes OMT' et OOA' méta-modélisées dans la section 5.3.1.

L'utilisation d'un méta-modèle pour la comparaison de méthodes a été proposée par divers auteurs. Nous pouvons citer par exemple le travail de G. Eckert et P. Golder [EG93]. Dans ce travail, après une analyse textuelle mettant en évidence les aspects de l'analyse et de la spécification d'un système, chaque méthode est représentée et étudiée graphiquement par un diagramme entité-relation des concepts employés. Une table est utilisée ensuite pour comparer les méthodes. Pour finir G. Eckert et P. Golder proposent une "méthode globale" (le méta-modèle) qui utilise ou perfectionne les concepts des méthodes orientées objet étudiées.

Un autre travail qui utilise l'approche des méta-modèles est celui de S. Hong, G. Goor et S. Brinkkemper [HvdGB93]. Après avoir introduit chacune des méthodes par un texte bref, celles-ci sont présentées à travers d'une part un modèle graphique de données (un diagramme entité-relation), d'autre part un modèle graphique de la procédure de développement (un diagramme de flux). Les auteurs suggèrent que toutes les méthodes peuvent être considérées comme des sous-modèles d'un méta-modèle général et comparent, à l'aide de tables, chaque méthode étudiée par rapport à ce méta-modèle.

Nous proposons ici une approche semblable basée sur la création automatique d'un méta-modèle général (le méta-méta-modèle) pour les modèles de méthodes orientées objets. Il est possible de créer, à l'aide de l'atelier A2M, un tel méta-méta-modèle afin de décrire la méthode globale dans laquelle tous les concepts proposés par les modèles des méthodes à comparer sont présents.

Même si on ne présente pas ici un exemple complet pour illustrer cette approche (un tel méta-méta-modèle complet peut être inféré du processus de méta-modélisation des méthodes OMT' et OOA'), on peut cependant démontrer l'utilité d'une telle approche en analysant les concepts d'une partie du modèle dynamique (les diagrammes d'états) de deux méthodes.

Cette approche de création automatique d'un méta-méta-modèle général est basée sur la compréhension de la sémantique des modèles de chaque méthode : les diagrammes d'états d'une méthode peuvent aussi s'appeler diagrammes de transitions d'états dans une autre méthode. La création du méta-méta-modèle doit donc être exécutée sous contrôle humain. Un processus de guidage de la construction du méta-méta-modèle peut être créé dans notre atelier à l'aide de démons : après une première génération semi-automatique des concepts présents dans les deux méta-modèles et possédant la même sémantique et le même nom (ex : le concept Classe), le processus de construction du méta-méta-modèle est achevé manuellement sous contrôle humain.

La figure 5.38 présente un méta-méta-modèle des méta-modèles des diagrammes d'états de deux méthodes issus de l'atelier A2M lors de la méta-modélisation d'OMT' et d'OOA'. Les éléments encadrés en ligne pleine sont communs aux deux méthodes ; les éléments encadrés en pointillé sont propres à la méthode OOA' ; les autres sont propres à la méthode OMT'.

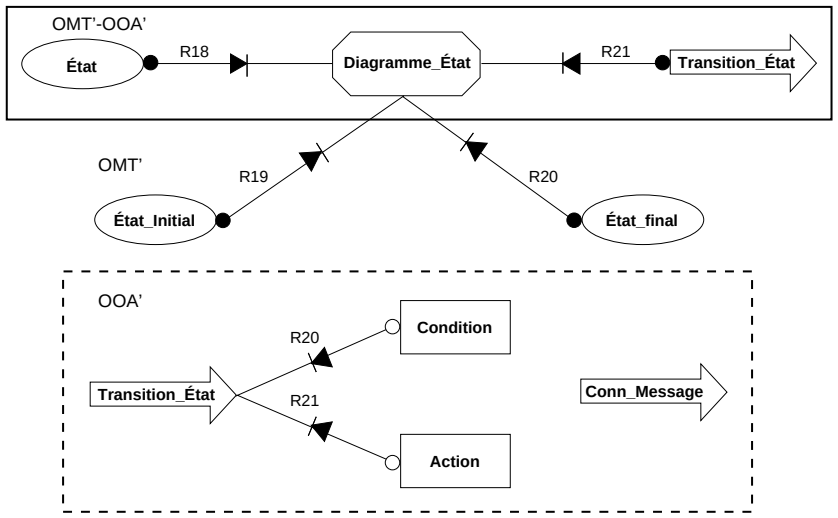


Figure 5.38 - Comparaison OMT'/OOA' - Diagramme d'États

Ci-dessous, en analysant la figure 5.38, nous faisons quelques remarques sur la comparaison de méthodes.

– quelques éléments utilisés par les deux méthodes sont communs et d'autres complémentaires. Les éléments communs aux deux méthodes, et qui peuvent alors être générés automatiquement dans un méta-méta-modèle commun sont `État, Transition_État` et `Diagramme_État` ;

– la méthode OOA' offre un cadre plus étendu que la méthode OMT' pour utiliser les conditions/actions dans les diagrammes d'états. En regardant la figure on peut remarquer que seulement la méthode OOA' attache des conditions et des actions aux transitions d'états ;

– même si la figure fait croire que la représentation de la dynamique dans la méthode OOA est plus complète que dans OMT, ce n'est pas le cas. En réalité, les diagrammes de transition d'états de la méthode OMT complète sont plus complexes que ceux présentés pour OMT', et en plus, la méthode OMT utilise un autre diagramme pour la dynamique, le modèle fonctionnel, qui n'existe pas dans OOA. Cette vue fonctionnelle dans OOA est représentée uniquement par les connexions de messages.

Nous devons prendre en compte le fait que l'analyse présentée ici est basée sur des modèles restreints des méthodes OMT' et OOA', et que le méta-méta-modèle cité en exemple dans la figure 5.38 est un reflet de ces modèles. L'analyse de la figure 5.38 permet seulement

une “comparaison visuelle” des deux méthodes. En utilisant la totalité du méta-modèle proposé (les trois approches : semi-formelle, informelle et formelle), on peut avoir un cadre de comparaison plus complet et plus rigoureux.

Il est clair qu’aujourd’hui une bonne définition d’une méthode passe par la description du méta-modèle de chacun de ses modèles. C’était l’un des handicaps de la première version d’OMT : le premier ouvrage était très déclaratif et ne contenait aucun méta-modèle.

5.3.3 Multi-Modélisation avec la Méthode OMT

Le processus de méta-modélisation conduit sous les ateliers A2M et A2M’ a pour but la génération d’un atelier AM particulier à une méthode. On présente ici le résultat du processus de méta-modélisation de la méthode OMT’ de la section 5.3.1.

La définition de l’hyper-graphe composé des trois graphes `Modèle_Objet`, `Diagramme_Etat` et `Modèle_Fonctionnel` (cf. figure 5.27) génère un atelier AM où ces trois modèles sont les graphes de base autour desquels une modélisation est construite (cf. fenêtre “Atelier_OMT - Essai” de la figure 5.39).

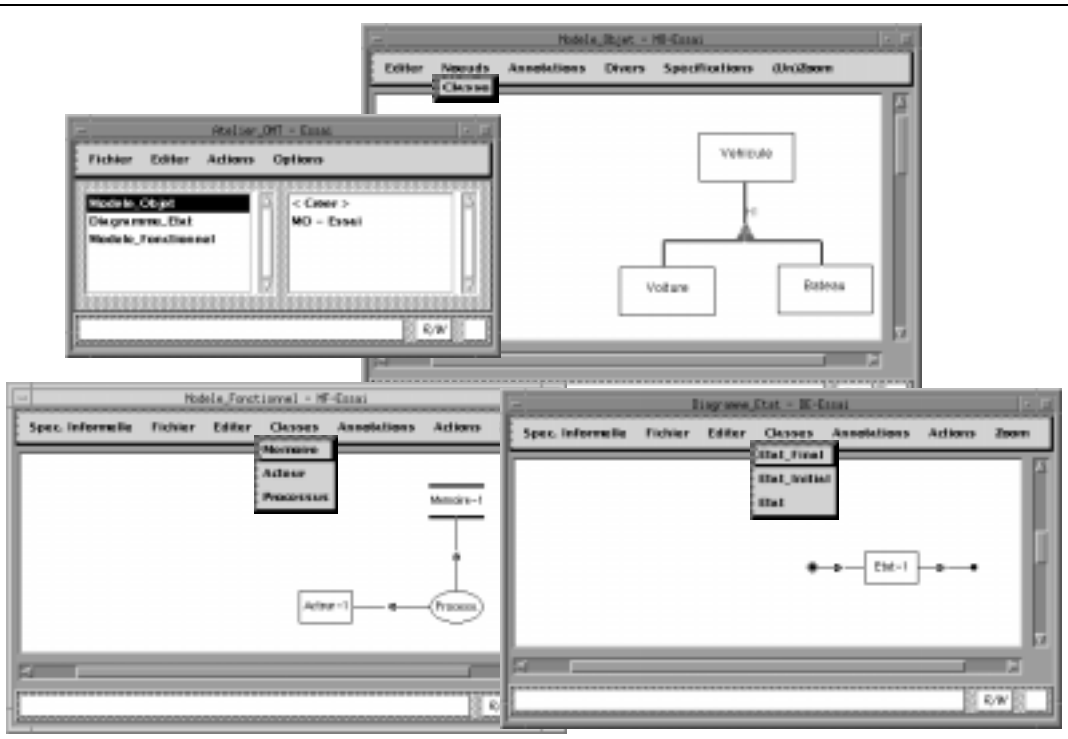


Figure 5.39 - Atelier AM-OMT - Partie Semi-Formelle

Les éléments offerts par chacun de ces graphes pour la modélisation d'un système sont ceux qui ont été déclarés dans l'atelier A2M' pour chacun des modèles, soit dans la partie spécification sémantique, soit dans la partie spécification des fenêtres (à travers une action). Ainsi, on a pour le modèle objet le nœud `Classe`, pour le diagramme d'états les nœuds `État`, `État_Initial` et `État_Final` et pour le modèle fonctionnel les nœuds `Processus`, `Acteur` et `Mémoire` (cf. fenêtres "Modèle_Objet - MO Essai", "Diagramme_État - DE Essai" et "Modèle_Fonctionnel - MF Essai" de la figure 5.39).

Les formes déclarées dans l'atelier A2M' pour les composants de la méthode OMT' sont celles présentées dans les fenêtres de la figure 5.39. La sémantique des liens est déclarée dans A2M et adaptée dans A2M'.

Dans la fenêtre "Modèle_Objet - MO Essai", on peut voir aussi la raison pour laquelle il est nécessaire de rajouter le dispatcher à la définition de la sémantique du lien entre les concepts `Classe` et `Héritage` (cf. figure 5.28) : cela permet la construction de structures arborescentes comme la structure d'héritage présentée dans la fenêtre.

Ces aspects (modèle objet, diagramme d'états, modèle fonctionnel) constituent la forme la plus courante de multi-modélisation : modélisation d'un SI à l'aide de plusieurs modèles à coordonner.

Le processus de modélisation sous l'atelier AM génère aussi une spécification formelle de cette modélisation à travers les éditeurs syntaxiques qui ont été présentés à la section 5.2.1.3. Dans ce processus de modélisation nous pouvons aussi gérer des spécifications informelles. Dans la figure 5.40 nous présentons ces deux types de spécifications correspondant aux exemples de la figure 5.39. Nous pouvons voir la spécification formelle, donnée par la partie du schéma S_Global qui modélise l'arbre d'héritage `H1` ainsi que la spécification informelle, donnée par la remarque textuelle à propos de cet arbre.

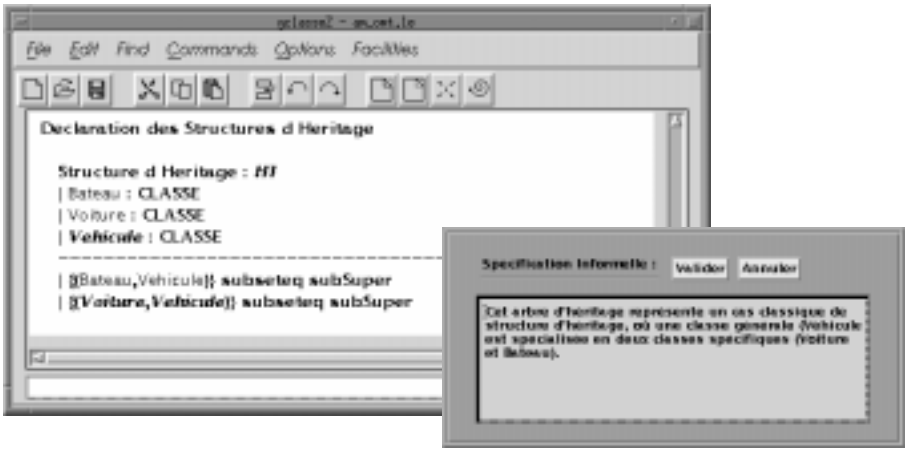


Figure 5.40 - Atelier AM-OMT - Parties Formelle et Informelle

La deuxième forme de multi-modélisation, celle qui combine et coordonne des modélisations informelles, semi-formelles et formelles n'est pas présente dans la plupart des ateliers commerciaux. Elle est offerte par notre atelier à travers les interactions possibles entre les quatre types de schémas S_Global, S_Concepts, S_Spécifications et S_Remarques.

La prochaine section présente une application de l'atelier OMT' générée à travers son utilisation par le Gestionnaire de Modèles pour la modélisation de STORM. Cette utilisation introduit une illustration de spécifications formelles et informelles coordonnées avec des spécifications semi-formelles.

5.3.4 Modélisation de STORM

Le processus qui, partant de la méta-modélisation d'une méthode avec l'atelier A2M, arrive à la génération d'un atelier AM pour cette méthode, a pour but d'offrir l'outil de base du Gestionnaire de Modèles de l'atelier de modélisation proposé. En fait, c'est à partir des modèles semi-formels, informels et formels générés par l'atelier AM et gérés par le Gestionnaire de Modèles que l'atelier de modélisation peut construire un document de spécification global.

Le modèle STORM est un modèle basé sur une approche objet [Adi95] qui propose une solution aux problèmes de modélisation et de gestion des données multimédias (image, texte, audio et vidéo) dans le cadre d'un SGBD orienté objet. Ce modèle est présenté en détails dans l'Annexe E.

Nous présentons, dans cette section, un document de spécification globale pour une partie du modèle STORM ; cette partie comprend la "Structure Algébrique" du modèle STORM telle qu'elle est présentée dans l'Annexe E (cf. figure E.5).

La construction d'un document de spécification globale pour un système passe toujours par un processus de multi-modélisation de ce système exécuté avec l'atelier AM. Pour arriver à la présentation d'un document de spécification globale pour la structure algébrique du modèle STORM, on doit donc utiliser l'atelier AM-OMT' afin de générer les composants de cette spécification.

Le processus de modélisation semi-formelle d'un système conduit sous AM-OMT génère en même temps sa spécification formelle. La fenêtre "Modèle_Objet - Struc_ Algébrique" de la figure 5.41 présente une reproduction de la structure algébrique présentée dans la figure E.5. La construction de cette modélisation produit la spécification formelle représentée dans la fenêtre "gclasseZ - STORM.le". Cette fenêtre correspond au schéma S_Global qui représente la vue Modèle Formel du méta-modèle proposé. On ne présente pas

la partie introductive où les types de bases et les relations utilisés dans la spécification sont déclarés. On peut voir dans cette fenêtre la spécification formelle des éléments modélisés de manière semi-formelle dans la fenêtre “Modèle_Objet - Struc_Algébrique”. Ainsi, on a la déclaration formelle des classes, des liens, des deux structures d’héritage et de la structure d’agrégation. On peut voir aussi une spécification informelle donnée par la remarque textuelle qui est déclarée pour la structure d’héritage H1.

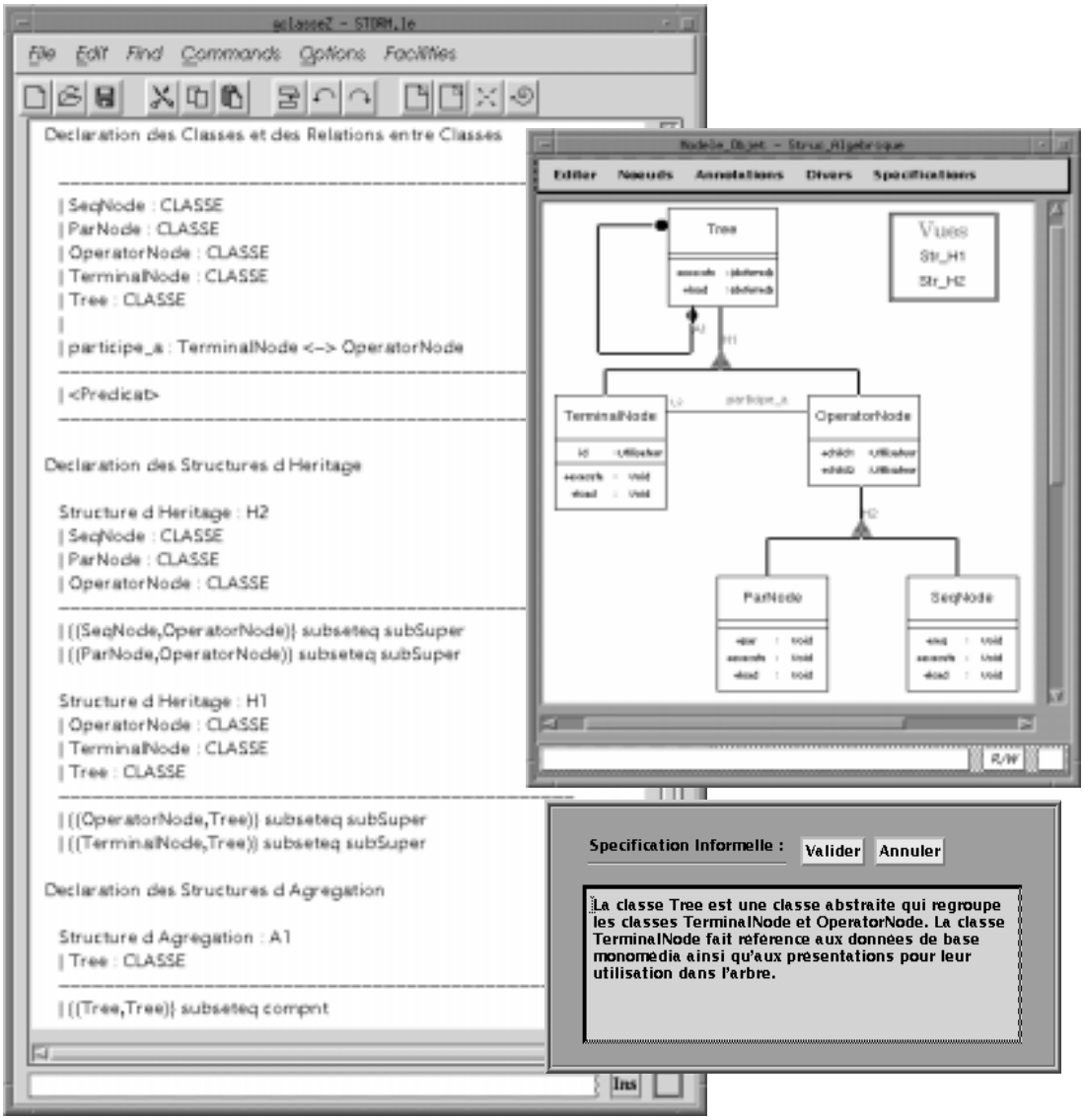


Figure 5.41 - Structure Algébrique de STORM - Modélisation avec AM-OMT

La fenêtre “Modèle_Objet - Struc_Algébrique” montre aussi le contrôle de vues dans l’atelier. Cela est réalisé avec le composant Vues, élément à travers lequel est réalisé le

lien entre le Gestionnaire de Modèles et le Gestionnaire de Documents. Dans cette fenêtre deux vues sont définies, l'une qui englobe les classes `Tree, TerminalNode` et `OperatorNode` (la vue `Str_H1`) et l'autre qui englobe les classes `OperatorNode, ParNode` et `SeqNode`. Lorsqu'on définit une vue dans l'atelier, on doit choisir les classes qui font partie de cette vue à travers leur sélection avec la souris. Ainsi tous les liens ainsi que toutes les structures auxquels ces classes participent sont rattachés aussi à la vue. Grâce à cela, et au choix des classes, la vue `Str_H1` est composée des classes `Tree, TerminalNode` et `OperatorNode` du lien `participe_a` et des structures `H1` et `A1`. La vue `Str_H2` est composée des classes `OperatorNode, ParNode` et `SeqNode` et de la structure `H2`. Le lien `participe_a` n'en fait pas partie car la classe `TerminalNode` n'est pas définie dans la vue.

Figure 5.42 - Structure Algébrique de STORM - S_Spécification de la Classe TerminalNode

À chaque fois qu'une classe est déclarée dans le modèle semi-formel, l'atelier AM-OMT' lui attache automatiquement une spécification formelle basée sur le langage Object-Z, grâce à un démon déclenché systématiquement lorsqu'une classes est créée. Cette spécifi-

cation formelle est gérée avec l'éditeur syntaxique LEdit et constitue un simple squelette de schéma Object-Z qui doit ensuite être complété. Elle présente cependant tous les éléments déclarés dans le langage. La figure 5.42 présente une telle spécification pour la classe `TerminalNode` de la structure algébrique (cf. figure 5.42).

Après avoir créé les modélisations présentées ci-dessus avec l'atelier AM-OMT, on est prêt pour la production du document de spécification globale qui est construit par l'atelier de modélisation. La figure 5.43 présente une fenêtre qui montre le document de spécification globale pour une partie de la Structure Algébrique du modèle STORM qui a été modélisée (cf. figure 5.41). Dans ce document de spécification globale, la partie modélisée correspond à la vue `Str_H1` qui a été définie dans le modèle semi-formel.

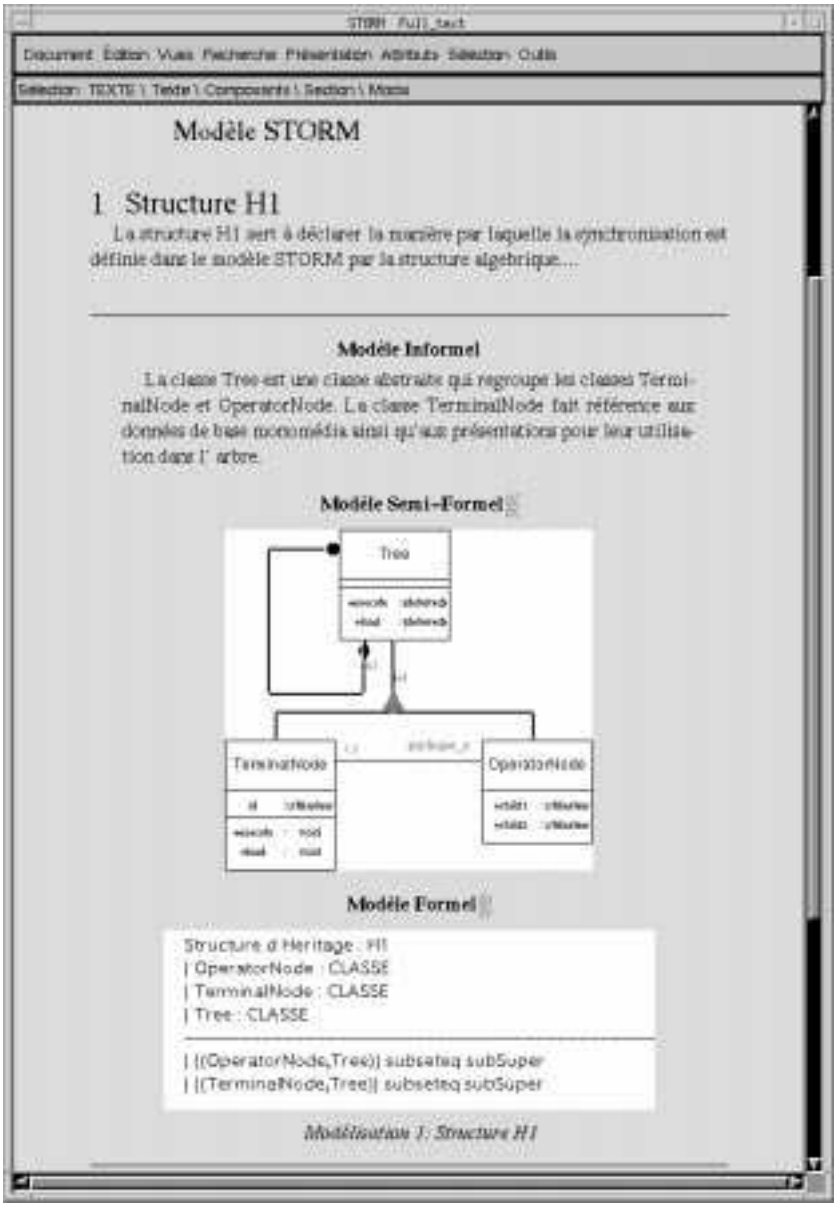


Figure 5.43 - Document de Spécification Globale - Structure d'Héritage H1

Dans cet exemple, deux relations entre fragments de modélisation ont été utilisées : expliquer une modélisation semi-formelle par un texte informel et transformer une modélisation semi-formelle en spécification formelle.

5.4 Conclusion

De nombreux facteurs freinent l'utilisation des ateliers de modélisation, en particulier, leur prix et leur spécialisation vis-à-vis d'une méthode. Ce dernier facteur a beaucoup évolué depuis le début des années 90 avec le développement d'ateliers intégrant un méta-référentiel pour proposer des interfaces pour différentes méthodes. Mais ces ateliers ont cependant deux limites fortes. D'une part, ils permettent rarement de combiner, pour un même SI, des modélisations selon des méthodes différentes, et d'autre part, ils ne sont pas adaptés à la coordination de spécifications semi-formelles, formelles et informelles.

Un atelier construit avec des outils existants peut d'une certaine manière contourner le problème de coût. L'atelier de modélisation, dont l'implantation et l'utilisation ont été traitées dans ce chapitre, utilise d'autres outils dans la construction d'un nouvel atelier de modélisation. Ces outils ont été présentés au début du chapitre afin de démontrer leurs capacités.

L'aspect multi-méthodes a été pris en compte par l'implantation de l'atelier de modélisation dont l'architecture a été présentée à la section 4.3. Cette architecture est basée sur deux gestionnaires (un de modèles et un autre de documents). Cette implantation est ensuite détaillée selon chacun des outils présentés antérieurement en offrant deux niveaux d'utilisation : méta-modélisation et multi-modélisation. Ces deux niveaux permettent d'intégrer des spécifications formelles aux spécifications semi-formelles et informelles.

P. Facon et R. Laleau [FL95] présentent une classification de la manière dont les spécifications semi-formelles sont "transformées" en spécifications formelles. Selon les auteurs deux processus existent :

- *la compilation* : à travers un guide méthodologique, une traduction entre les deux types de spécifications est produite, soit manuellement, soit avec un assistant ("un outil qui génère un squelette de spécification formelle à compléter) ;
- *l'interprétation* : la traduction s'exécute en passant par la définition d'un méta-modèle formel du système d'information, où les concepts de celui-ci sont définis avec des types abstraits, des ensembles, etc ; après la définition de ce méta-modèle, les concepts du système d'information sont modélisés par rapport à ce méta-modèle.

L'approche proposée dans ce travail utilise, d'une part, la traduction automatique pour la génération d'un squelette conformément à un méta-modèle (les types de base Concept, Relation, Propriété et Modèle ainsi que la définition des structures et des relations) et, d'autre part, une interprétation de ces spécifications toujours autour d'un méta-modèle.

L'utilité d'un tel atelier est ensuite présentée à travers d'études de cas. On présente quatre exemples d'utilisation de l'atelier à deux niveaux différents. Un AGL dédié à la méthode OMT est utilisé pour la production partielle du document de spécification globale pour le modèle STORM. Cet exemple permet d'illustrer les capacités de notre prototype d'atelier à combiner simplement des modèles graphiques, textuels et formels, d'une part dans la modélisation d'une méthode (OMT) et d'autre part dans la modélisation d'un système d'information (STORM).

Chapitre 6

Bilan et Perspectives

Dans notre travail, nous avons souhaité faciliter l'utilisation conjointe de trois approches (informelle, semi-formelle et formelle) dans le cadre de la spécification de systèmes d'information. Une telle combinaison d'approches doit être possible à deux niveaux différents : la méta-modélisation de modèles de méthodes orientées objets et la multi-modélisation de systèmes d'information. Cette utilisation coordonnée de formalismes a été matérialisée par le développement d'un prototype de générateur d'ateliers de construction de documents de spécifications de systèmes.

6.1 Bilan et Contributions

Dans un premier temps, nous avons abordé dans cette thèse, une étude générale de la modélisation. Le processus de modélisation d'un système passe obligatoirement par l'utilisation de méthodes. Les méthodes utilisées d'une manière conjointe avec les outils qui les supportent, ainsi qu'avec un processus qui guide leur utilisation, incluent le concept de cycle de vie du Génie Logiciel.

Des ateliers de génie logiciel ont été créés, d'une part pour assurer l'utilisation correcte des méthodes selon un cycle de vie et, d'autre part pour faciliter cette utilisation ; nous avons décrit quelques exemples de tels ateliers de génie logiciel. Cette description partielle d'ateliers a été faite en respectant une double classification : commerciale/domaine public, méta-modélisation/modélisation.

Les méthodes intégrées aux ateliers de génie logiciel nécessitent un langage pour la production d'une modélisation. Nous avons introduit une classification des langages selon

l'approche utilisée et nous avons résumé leur processus d'utilisation.

Notre étude de modélisations informelles a mis en évidence l'intérêt de spécifications informelles standardisées. Nous avons concrètement expérimenté de telles spécifications par l'utilisation d'un éditeur hyper-texte intégré dans notre plate-forme de modélisation de systèmes.

En ce qui concerne les modélisations semi-formelles, nous pouvons dire que leur large utilisation dans les ateliers de génie logiciel actuels, alliée aux résultats significatifs obtenus au fil des ans en matière d'augmentation de la qualité des systèmes, prouvent leur utilité. Nous les avons étudiées attentivement pour les intégrer d'une manière homogène dans notre prototype d'atelier.

Dans le cadre de la modélisation formelle, nous avons évalué plusieurs méthodes. Après cette étude, nous avons opté par la méthode Z car elle nous semble la plus appropriée pour la spécification des systèmes d'information. Ce choix est dû essentiellement à l'accessibilité du langage Z aux lecteurs non spécialistes de spécifications formelles mais qui sont néanmoins familiers avec la théorie des ensembles et l'approche base de données. Cette étude sur les méthodes formelles nous permet de dire aussi que leur utilisation, qui se fait chaque jour plus présente et nécessaire, peut encore améliorer la qualité des spécifications de systèmes lorsqu'elle se fait d'une manière conjuguée avec d'autres approches semi-formelles ou informelles.

Aujourd'hui, le mot phare de l'informatique est bien sûr l'"objet". Nous nous sommes alors intéressés particulièrement à ce type de modélisation semi-formelle. Les plus grands bénéfices de son utilisation reposent d'une part sur l'adéquation des objets à modéliser le monde réel à travers des abstractions et, d'autre part sur les facilités offertes au niveau de la réutilisation d'objets de travaux antérieurs.

Après une étude sur des méthodes orientées objets et des méthodes formelles, nous pouvons corroborer l'affirmation de K. Lano [Lan95] selon laquelle les méthodes formelles et les méthodes orientées objets sont complémentaires, car si l'un (l'objet) encourage la création d'abstractions, l'autre (formelle) introduit les moyens pour les décrire, si l'un (l'objet) offre des mécanismes de structuration et des disciplines de développement, l'autre (formelle) peut s'en servir dans la modélisation de grands systèmes et, en plus, les méthodes formelles fournissent une manière précise pour représenter des concepts des méthodes objets. De nos jours, cette utilisation conjointe commence à être une réalité à travers l'extension des langages formels afin d'incorporer des concepts du paradigme objet (Objet-Z, VDM++, PLUSS, etc).

Afin de concrétiser cette utilisation conjointe, formel/semi-formel, dans le cadre d'un

l'atelier de génie logiciel qui intègre aussi l'approche informelle, nous avons proposé l'utilisation d'un méta-modèle. Ce méta-modèle rend possible la modélisation de systèmes selon deux axes : l'un qui intègre les trois formalismes, informel, semi-formel et formel et l'autre, qui permet une visualisation d'un système par rapport à ses caractéristiques statiques ou dynamiques. Ce méta-modèle est au cœur du générateur expérimental d'ateliers de modélisation que nous avons développé.

L'architecture et les fonctionnalités de ce générateur d'ateliers permettent un double usage : méta-modélisation de modèles de méthodes et multi-modélisation de systèmes d'information selon des méthodes particulières. Le méta-modèle permet l'interaction de différentes approches de modélisation. Nous avons proposé une étude de cette interaction en décrivant des relations qui peuvent exister entre les différents types de modélisations lorsqu'elles sont utilisées conjointement dans la modélisation de systèmes d'information.

Nous pouvons dire que les principales caractéristiques de notre atelier (générateur d'ateliers) portent surtout sur les points suivants :

- une description de chaque modèle selon un même méta-modèle qui prend en compte trois dimensions, informelles, semi-formelles et formelles ;
- l'extensibilité, la portabilité et l'autonomie des trois composants (méta-outils) de l'architecture choisie ;
- une combinaison des approches objets et formelles avec une organisation en schémas ;
- une généralisation de la multi-modélisation de systèmes d'information ;
- une aide à la documentation des systèmes organisée par un éditeur hyper-texte ;
- une possibilité de vérifier formellement une spécification semi-formelle.

Nous avons expérimenté notre atelier dans des situations variées : méta-modélisation de méthodes orientées objets (OMT et OOA), comparaison de méthodes orientées objets, multi-modélisation d'un système d'information et production d'un exemple de document de spécification d'un système en utilisant la totalité des fonctions de notre plate-forme de modélisation.

Ces résultats font ressortir, nous semble-t-il, un travail original dans le cadre de la modélisation de systèmes d'information et plus particulièrement, notre contribution est significative sur deux points :

– notre générateur d'ateliers permet, contrairement aux ateliers commercialisés, de combiner et de coordonner pour un même système d'information des modélisations selon des approches différentes : informelles, semi-formelles et formelles ;

– notre étude préliminaire sur les relations à gérer entre les différents types de modélisation et les différents fragments de modèles est certainement le point de départ d'une nouvelle démarche de modélisation non proposée, à notre connaissance, dans les ateliers du marché.

De plus, notre atelier facilite la gestion de "fragments" de modèles utilisables dans un éditeur hyper-texte dédié à la documentation de spécifications de systèmes d'information. Si cette caractéristique est présente dans l'atelier Rose à travers l'outil SODA, il s'agit dans ce cas d'importation de modèles semi-formels complets. Avec la structure hyper-texte et le concept de vues présents dans notre atelier, nous permettons l'intégration de parties spécifiques d'une modélisation, qu'elles soient semi-formelles, formelles ou informelles.

6.2 Perspectives

Une utilisation effective des propositions de notre travail passe sûrement d'une part par des améliorations et des expérimentations du prototype développé et d'autre part par des approfondissements d'aspects trop brièvement décrits et évalués.

L'amélioration du prototype doit se faire en particulier par rapport aux points suivants :

– définition d'un canevas de modélisation réel complet qui prendrait en compte une norme de documents de spécification de systèmes d'information, comme par exemple la norme AFNOR NF67-100-3 [AFN94] ;

– intégration plus souple des modélisations produites par le module gestionnaire de modèles et le module gestionnaire de documents ;

– dans notre travail la phrase "traduction automatique du semi- formel vers le formel" cache un travail important de développent (environ 12000 lignes de code C). Nous pensons que grâce à la généralité du code développé nous pouvons constituer une bibliothèque d'applicatifs, pour rendre réutilisable une partie du code C développé pour construire des ateliers pour d'autres méthodes.

D'une manière pratique, nous souhaitons poursuivre la validation de notre travail dans le cadre du projet STORM. Une première expérimentation de notre atelier a été réalisée pour

la construction de présentations multimédias qui ont été modélisées selon des approches semi-formelles et formelles. Dans cette expérimentation, nous avons généré un atelier à partir de notre plate-forme. Cet atelier dédié permet de représenter graphiquement une présentation multimédia, qui peut être composée d'objets monomédias stockés dans un SGBD objet et de parties formelles qui sont des requêtes exprimées selon le langage OQL. Cette première expérimentation nous permet, aujourd'hui, d'envisager l'utilisation de cet atelier dans un contexte de spécifications exécutables. Une étude plus approfondie à propos de diverses relations existantes entre les représentations utilisées pour modéliser une présentation multimédia est aussi envisagée. Cette étude doit sûrement faire ressortir de nouvelles caractéristiques envisageables dans de tels ateliers.

Nous voulons aussi poursuivre l'étude sur la méta-modélisation complète de méthodes orientées objets (et non de sous-ensembles de ces méthodes comme nous l'avons fait) afin de mieux valider notre méta-modèle et de construire un méta-méta-modèle général qui rende possible des transformations entre les modèles semi-formels de méthodes différentes. Ainsi, par exemple, une modélisation exprimée selon le formalisme OOA pourrait être transformée, d'une manière semi-automatique, vers une autre modélisation exprimée selon OMT ou UML.

Des approfondissements de notre travail sont nécessaires sur des aspects importants et liés mais encore peu étudiés. En particulier, nous souhaitons :

- compléter et valider notre étude initiale sur les relations (expliquer, compléter, modéliser, etc.) inter-fragments de modèles afin de prendre en compte par exemple les relations entre les modélisations informelles et formelles, ou informelles et semi-formelles ;
- étudier les problèmes fondamentaux et les problèmes de cohérence entre les représentations selon les langages différents (formels et semi-formels par exemple).

De tels approfondissements permettraient de mettre en évidence par exemple la réversibilité ou les pertes d'informations possibles dans une opération de transformation de modèles.

Nous avons présenté dans ce travail des propositions permettant une intégration de trois types de spécifications. Nous pensons que notre travail peut être utile dans le cadre d'une étude plus approfondie sur le couplage entre des spécifications semi-formelles moins proches de la modélisation de données et des spécifications formelles. L'approfondissement de cette étude sur le couplage entre des diagrammes traitant la dynamique d'un système et des méthodes formelles a déjà donné lieu à un travail de DEA [Dup97]. Ce travail se

poursuit et se concrétise dans le cadre d'un travail de magistère où nos propositions et notre atelier seront utilisées.

Il faut noter aussi que notre recherche a porté essentiellement sur l'ingénierie des modèles, sans traiter de l'ingénierie des méthodes. Ce nouveau domaine de recherche est devenu très actif [Rol94] dans les équipes systèmes d'information et génie logiciel ; il aborde en particulier la description des démarches de modélisation. Ce type de description n'est pas explicitement intégré aujourd'hui dans notre méta-modèle.

Malgré ces limites de notre travail, les premières expérimentations réalisées nous permettent d'envisager la prise en compte des différentes prolongations de cette recherche en nous appuyant directement sur le cadre de méta-modélisation que nous avons défini ainsi que sur le générateur expérimental d'ateliers de modélisation que nous avons développé.

Annexe A

La Méthode OOA

Les concepts présentés dans cette annexe sont basés sur le dernier livre de E. Yourdon [You94] où une approche multi-modèles est utilisée, contrairement à l'approche adoptée dans les premiers livres [CY91a, CY91b] qui préfèrent un modèle unique multi-couches. On peut dire que cette nouvelle approche reprend le contenu des couches pour les transformer en modèles. Ci-dessous, on présente les concepts et notations proposés par E. Yourdon pour la méthode OOA présentés dans [You94].

Le terme modèle n'est pas utilisé dans [You94] comme élément de modélisation, mais pour garder une cohérence avec les autres méthodes présentées on l'utilisera dans ce texte.

A.1 Classes et Objets

Objet : c'est une abstraction de quelque chose du domaine d'un problème ou de son accomplissement qui reflète la capacité d'un système à maintenir les informations et/ou à interagir sur cette abstraction ; c'est une encapsulation de la valeur des attributs et de ses services exclusifs.

Classe : c'est une description d'un ou plusieurs objets, à travers un ensemble uniforme d'attributs et de services.

Classe-et-Objets : ce terme signifie : "une classe et ses objets" ; dans la figure A.1.a on trouve la notation d'une classe-et-objets et, dans la figure A.1.b, celle d'une classe toute seule, laquelle ne peut pas avoir d'instances.

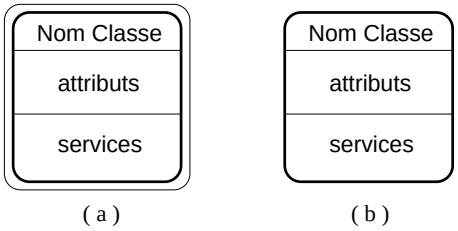


Figure A.1 - OOA - Classes et Objets

Sujet : c'est une mécanisme proposé par la méthode pour contrôler la partie du système qui peut être vue, en guidant le lecteur (analyste, spécialiste dans le domaine du problème, gérant, client) à travers un modèle vaste et complexe.

Un sujet est une abstraction du plus haut niveau qui peut être représentée de deux manières différentes : soit à travers une boîte rectangulaire qui peut avoir à son intérieur une liste des classes que le sujet englobe, soit par une boîte qui englobe la notations des classes avec ses relations. Dans les deux cas, le sujet doit être nommé à l'intérieur de la boîte.

A.2 Structures

Les structures sont une représentation de la complexité du domaine du problème. Cette représentation a un rapport avec les responsabilités du système. Le terme structure est utilisé comme un terme général qui peut être employé pour décrire une Structure Généralisation-Spécialisation ou une Structure Composé-Composant.

Structure Généralisation-Spécialisation : elle aide à la représentation de la hiérarchie classe/membres dans le domaine du problème, en présentant la généralisation et la spécialisation des entités du monde réel à travers les classes. Dans la figure A.2, on trouve la notation d'une structure généralisation-spécialisation. Dans la nouvelle version de la méthode, cette structure ne lie pas explicitement des classes, comme c'était les cas dans la proposition originale.

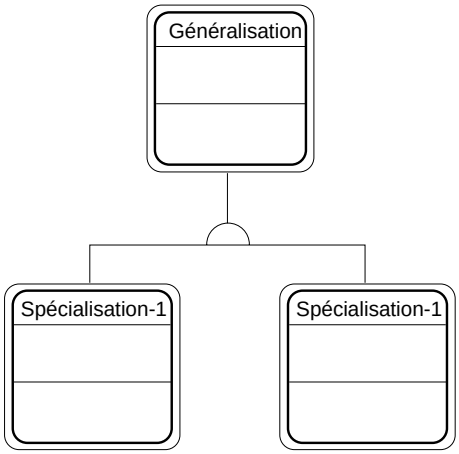


Figure A.2 - OOA - Structure Généralisation-Spécialisation

Structure Composé-Composant : elle représente le modèle et ses parties composantes ; si, avec la structure généralisation-spécialisation, l'analyste identifie les services et les attributs communs, avec la structure composé-composant, il peut définir le domaine du problème, par composition du composé à partir de ses parties (cf. figure A.3). Les chiffres à côté de chaque liaison expriment le nombre de composants qu'un composé peut avoir et vice-versa : dans la figure A.3, la classe-et-objets `Composé` peut avoir `n3` à `n4` composants de la classe-et-objets `Composant-1`, et une classe-et-objets `Composant-1` peut être le composant de `n1` à `n2` classe-et-objets `Composé`.

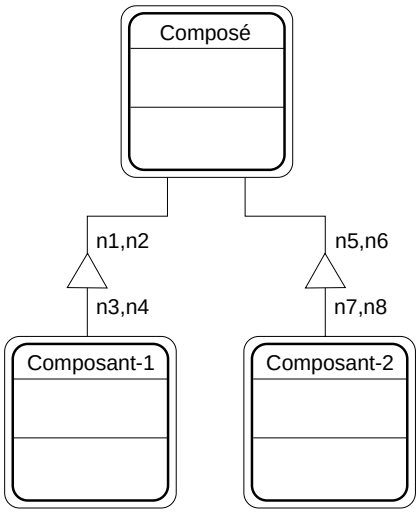


Figure A.3 - OOA - Structure Composé-Composant

A.3 Relations

Relations Binaires : c'est un type de relation qui, comme son nom l'indique lie deux classe-et-objets. Dans la méthode ces relations sont appelées aussi de `Connexion d'Instances`. Ces relations sont statiques et sont utilisées pour représenter les applications ("mappings") entre les objets du domaine du problème afin que ces objets puissent accomplir leurs responsabilités. L'information d'état des objets modélisée par les attributs est complétée par les connexions d'instances.

La cardinalité de la relations est représentée par des intervalles `max,min` qui sont placés à coté de chaque objet qui fait partie de la relation. La relation est explicitement entre objets, c'est-à-dire, entre des instances d'une classe et non entre des classes. Dans la figure A.4, cela veut dire que une instance de la classe `Classe-1` est associée à un nombre de instances de la classe `Classe-2` qui est compris entre `n1,n2` ; des noms pour les deux sens de la relations peuvent être ajoutes à la représentations donnée à la figure A.4. Dans cette figure, la classe-et-objets `Classe-1` peut être liée à `n3` à `n4` classe-et-objets `Classe-2`, et la classe-et-objets `Classe-2` peut être liée à `n1` à `n2` classe-et-objets `Classe-1`.

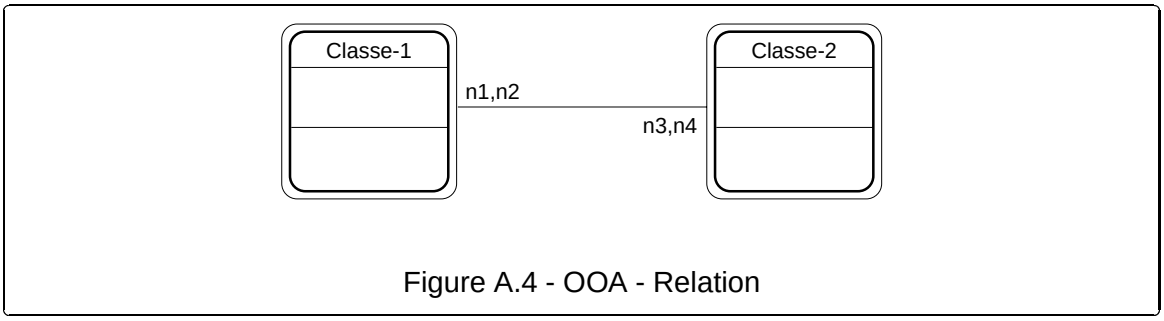


Figure A.4 - OOA - Relation

Relation Étendue : c'est le terme utilisé par la méthode pour représenter des modélisations où plus que deux classes sont liées entre elles (cf. figure A.5). Des modélisations de ce type apparaissent plutôt lorsque une relation est caractérisée par des informations qui doivent être stockées ; les cardinalités dans ce type de relation ne sont pas représentées car elle peuvent introduire des ambiguïtés au modèle.

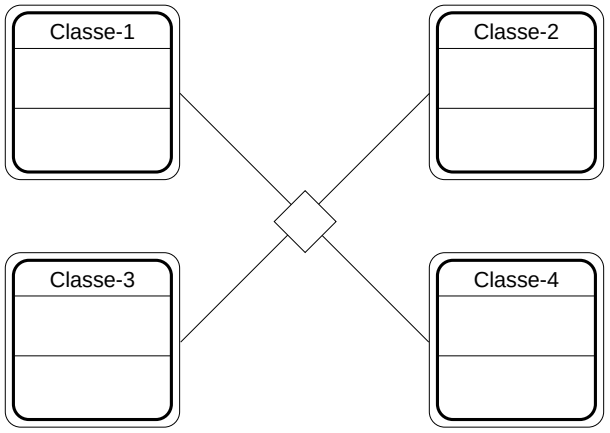


Figure A.5 - OOA - Relation Étendue

A.4 Attributs

Attribut : c'est une information d'état qui a une valeur propre pour chaque objet d'une classe ; cette valeur ne peut être manipulée que par les services de cette classe. A travers la description de chaque classe-et-objets avec attributs, le modèle d'analyse devient plus spécifique et détaillé : l'ajout des attributs contribue à augmenter la complexité des autres modèles. Les attributs sont notés dans la section médiane du symbole graphique d'une Classe et Objets (cf. figure A.1).

A.5 Comportement de l'Objet

Diagramme de l'Histoire des Objets - DHO : c'est la manière utilisée par la méthode pour représenter le comportement dynamique intrinsèque des objets. Ces diagrammes sont une version des Diagrammes de Transitions d'États présents dans presque toutes les méthodes, quoique la proposition de Yourdon soit plus pauvre que celles qui utilisent les Diagrammes de Transitions d'États basés sur le travail d'Harel [Har87].

Un DHO est composé d'**États des Objets**, de **Transitions d'États**. Les Transitions d'États peuvent avoir des **Conditions**, qui génèrent une transition d'état, ainsi que des **Actions** exécutées par les objets lorsqu'ils changent d'état (cf. figure A.6). Un état d'un diagramme peut être décomposé dans un nouveau diagramme, ce qui introduit une notion de plusieurs niveaux d'abstraction dans les DHO.

Une caractéristique particulière introduite par la méthode est la présentation conjointe des DHO avec les diagrammes qui ont comme composants des classe-et-objets. Dans ce cas, les classe-et-objets peuvent être liés à des états par des messages.

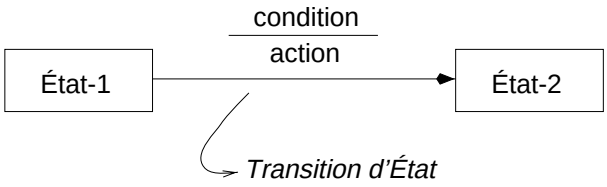


Figure A.6 - OOA - Diagramme de l'Histoire des Objets

A.6 Services

Service : c'est un comportement spécifique exhibé sous la responsabilité d'un objet. Il correspond à une procédure qui va être exécutée après la réception d'un message envoyé par une autre classe-et-objets. Les services peuvent être notés, selon les figures A.7.a ou A.7.b, dans la section inférieure de la notation d'une classe-et-objets.

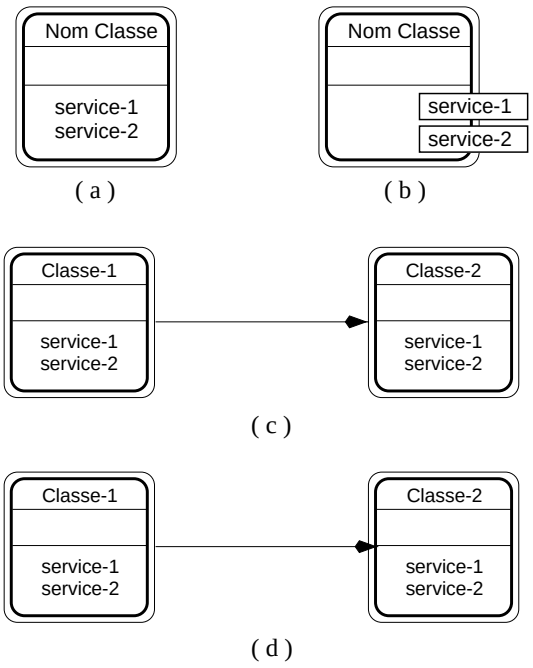


Figure A.7 - OOA - Services et Connexions de Messages

Connexion de Message : elle sert à modéliser la dépendance d'exécution de services des objets, en montrant à quels services externes un objet doit faire appel pour accomplir ses responsabilités. Dans la figure A.7, on trouve la notation employée pour représenter une connexion de messages, soit entre instances de classes (cf. figure A.7.c), soit entre une instance et une classe (cf. figure A.7.d).

A.7 La Procédure de Développement

Les six "modèles" présentés ci-dessus ne sont pas vraiment des modèles comme pour la plupart de méthodes ; ils sont plutôt des directives qui conduisent la procédure de développement. Ci-dessus nous avons présenté les concepts et les notations qui sont introduites dans chacun de ces modèles ; ci-dessous, nous présentons quelques pas de la procédure de développement proposée par la méthode.

1. *Identifier les Classes et Objets* : une première approche de la modélisation est faite en regardant, dans le domaine d'application, les classe-et-objets qui doivent être identifiés et celles qui formeront la base de l'application à développer. Cette identification peut être faite avec trois points de vues différents : l'un basé sur les données, l'autre basé sur les fonctionnalités et le dernier basé sur le comportement des objets.

   Lors de cette identification, il faut analyser les responsabilités du système dans ce domaine ; lors de cette analyse, d'autres éléments qui doivent être connus du système peuvent être identifiés. Après l'identification, il faut vérifier et évaluer les objets pour être sur qu'ils sont "utiles" au système ; pour cela, on peut vérifier, par exemple, s'ils représentent des concepts dont le système doit se souvenir, s'ils ont plus qu'un attribut (dans le cas contraire, ils peuvent être simplement un attribut d'un autre objet) et s'ils ont besoin d'avoir des services. La dernière tâche de cette étape est le groupement des classe-et-objets dans des sujets.

2. *Identifier les Structures* : les structures du domaine d'application doivent être explicitées à travers l'identification des deux types de structures proposées par la méthode : les Structures généralisation-spécialisation et les structures composé-composant. Avec les structures généralisation-spécialisation, il faut capturer la hiérarchie d'héritage parmi les classe-et-objets déjà définies. Avec les structures composé-composant, il faut capturer le concept du tout et de ses parties. Il faut identifier aussi les *Structures Multiples* possibles, c'est-à-dire, les structures qui sont composées de structures généralisation-spécialisation et de structures composé-composant.

3. *Identifier les Relations* : dans cette étape, les relations binaires ou non, entre classe-et-objets doivent être identifiées, à travers l'observation des processus d'analyse antérieurs ou à travers un représentation des relations présentes dans le domaine d'application. Les cas particuliers de relations telles que la relations `N :N`, les connexions d'instances récursives, les connexions d'instances multiples entre classe, les connexions un-aires entre classes et les résultats d'ajout ou suppression d'une instance par rapport à la connexion d'instances, doivent être étudiés en détails.

4. *Définir les Attributs* : cette étape doit être faite en identifiant les informations qui peuvent être associées à chaque instance du modèle. Pour chaque objet du modèle, il faut identifier les attributs qui sont nécessaires pour le caractériser, c'est-à-dire, il faut vérifier quelles sont les informations du domaine d'application dont un objet d'une classe est responsable. Après leur identification, les attributs doivent être mis au niveau correct de la hiérarchie d'héritage. Chaque Attribut trouvé doit être identifié par un nom qui doit être dans le domaine d'application et il peut aussi être spécifié avec des contraintes.

5. Représenter le Comportement des Objets : dans cette étape il faut définir le comportement des objets avec l'utilisation des Diagrammes de l'Histoire des Objets. Ces diagramme peuvent être divisés en sous-diagrammes pour faciliter leur visualisation et peuvent aussi être associés aux autres diagrammes de la méthode.

6. *Définir les Services* : dans cette étape, il faut définir les services des classe-et-objets du modèle en identifiant les *Services Implicites* (création, modification, recherche et suppression), les `Services Associés avec les Messages`, les `Services Associés avec les Connexion d'Instances`, les `Services Associés avec les Attributs` et les `Services Introduits par le DHO` [You94]. Les besoins d'opérations sur les objets doivent être identifiés avec les messages. La dernière étape consiste à spécifier en détails ces services soit avec du français structuré, soit avec des diagrammes de flux de donnés ou avec d'autres diagrammes qui s'appliquent à ce fin.

Selon E. Yourdon, il faut que la notation utilisé dans la conception soit le plus proche de celle utilisé lors de l'analyse ; cette proposition est adoptée dans la méthode OOA. Dans la conception orienté objet, il faut établir des stratégies pour, par exemple, indiquer le point de départ de la phase de conception, il faut aussi donner une heuristique au processus et il faut aussi choisir en cadre de travail ainsi qu'une architecture pour le système. Les critères selon lesquels la procédure de développement peut être conduite sont nombreux [You94] ;

la méthode suggère, par exemple, que les interactions entre classe-et-objets qui ne font pas partie d'une même structure doivent être évités et que les classe-et-objets doivent être le plus simples possible. Pour une liste complète, voir [You94]. Une stratégie non complexe pour les tests est aussi proposée.

Annexe B

La Méthode OMT

Dans cette annexe, on présente les composants principaux de trois modèles de la méthodes. Quoique dans [RBP+91] ces trois modèles soient présentés à deux niveaux : les *concepts essentiels* et les *concepts avancés*, dans ce texte, on ne fait pas cette distinction.

On présente les concepts qu'on juge les plus importants, prenant en compte surtout l'évolution proposée par J. Rumbaugh [Rum95c, Rum95a, Rum95b] en dépit de la méthode originale. De cette manière, lorsque dans les papiers qui présentent l'évolution de la méthode, un concept est redéfini, on prend cette nouvelle définition. Par rapport à la première version, le Modèle Fonctionnel est celui qui présente la plus grande différence ; les autres sont une évolution de la première version, tandis que ce dernier introduit des concepts nouveaux.

B.1 Modèle des Objets

Classe : c'est une abréviation pour le terme `Classe d'Objets`. Une classe décrit un groupe d'objets ayant des propriétés similaires (attributs), des comportements similaires (opérations) et des relations communes avec d'autres objets qui partagent une sémantique commune. La figure B.1 présente la notation employée pour représenter les classes ; dans la partie (a) de cette figure les composants représentées entre les signes "<" et ">" représentent des parties optionnelles.

Les classes sont organisées en hiérarchies de classes qui ont des structures et des comportements communs. Elles sont aussi reliées entre elles par des associations. Les classes définissent les attributs que posséderont chacune de leurs instances ou objets, ainsi que les opérations exécutables par chaque objet.

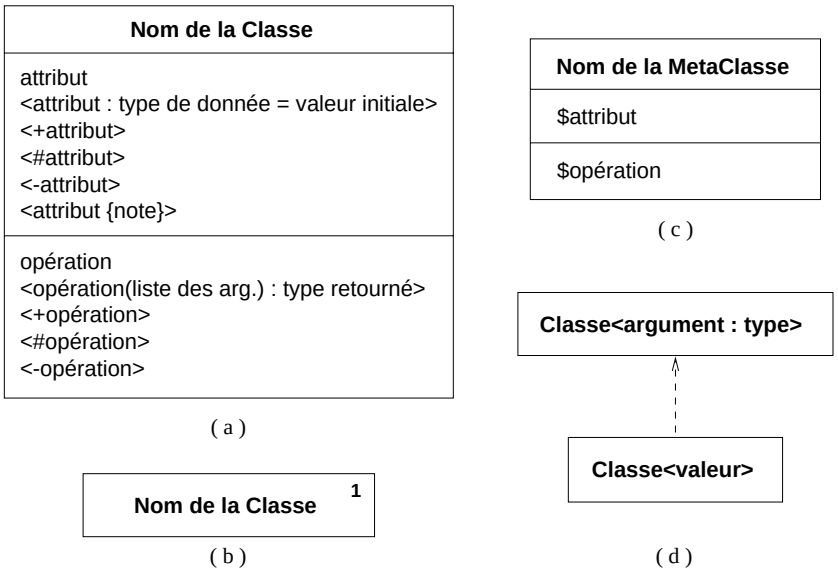


Figure B.1 - OMT - Classes

Attribut : c'est une donnée dont la valeur est maintenue par un objet dans une classe. Les attributs d'une instance d'une classe peuvent avoir des valeurs différentes de ceux d'une autre instance. Les noms des attributs doivent être uniques dans une classe. Un attribut peut être un objet ou une valeur de donnée pure.

Opération : c'est une fonction ou une transformation qui peut être appliquée par ou sur des objets d'une classe. Tous les objets d'une classe partagent les mêmes opérations. Chaque opération a comme argument un objet cible. L'action d'une opération dépend donc de l'objet cible.

Une même opération peut être appliquée à différentes classes. Les différentes formes d'une même opération dans différentes classes expriment le concept de polymorphisme.

La visibilité des attributs et opérations peut être représentée avec les signes "+" (élément publique), "#" (élément protégé) et "-" (élément privé) lesquels doivent être placés avant le nom de l'élément (cf figure B.1.a). Une note peut être aussi utilisée pour représenter une caractéristique particulière d'un élément, comme, par exemple, `{deferred}` qui indique qu'une opération sera implantée dans une sous-classe.

Une classe peut être représentée avec une expression qui montre le nombre d'instances qu'elle peut avoir. Par défaut il existe plusieurs instances et le cas le plus important est celui de la classe singleton (cf. le 1, en haut à droite - figure B.1.b).

La méthode introduit la notion de `Méta-Classe`, classe qui décrit des classes et dont les instances sont des classes. Les méta-classes sont décrites avec des attributs de classe, lesquels décrivent une valeur commune à toutes les classes d'objets, plutôt qu'à chaque instance et avec des opérations de classe, qui sont des opérations de la méta-classe elle-même, comme, par exemple, une opération pour créer des instances de classes (cf. figure B.1.c). La notion de `Classe Paramétrée` est aussi introduite (cf. figure B.1.d). Ces classes sont des classes qui servent de patron ("template") pour d'autres classes. Dans la figure B.1.d la classe paramétrée est celle d'en haut et la classe instanciée est celle d'en bas. Le processus d'instanciation est représenté par une flèche pointillé.

Objet : c'est un concept, une abstraction de quelque chose du monde réel. Les objets sont utilisés pour favoriser la compréhension du monde réel et pour établir une base pratique pour l'implémentation.

Les objets ont une identité, c'est-à-dire qu'ils sont distincts par essence et non par leurs propriétés. La figure B.2 présente la notation employée pour représenter les objets. La notation utilisée, avec le nom de la classe de l'objet représenté dans la "boîte objet", est assez explicite, mais on peut représenter l'instanciation par une flèche pointillé de l'objet vers la classe (cf. figure B.2).

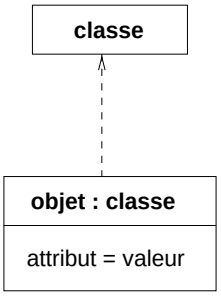


Figure B.2 - OMT - Objets

Association : c'est un concept qui est utilisé pour représenter des relations structurelles existantes entre des objets de différentes classes. Les instances d'une association sont les **Liaisons** ; une liaison est une connexion conceptuelle ou physique entre deux objets. Une association décrit un ensemble de liaisons qui ont une structure et une sémantique communes.

La notation que la Méthode utilise pour représenter les associations ou liaisons est présentée dans la figure B.3. Le nom de l'association ou liaison peut être présent ou

non ; s'il est présent, il est écrit en italique et peut avoir une flèche qui montre dans quel sens l'association doit être lue (cf. figure B.3.a).

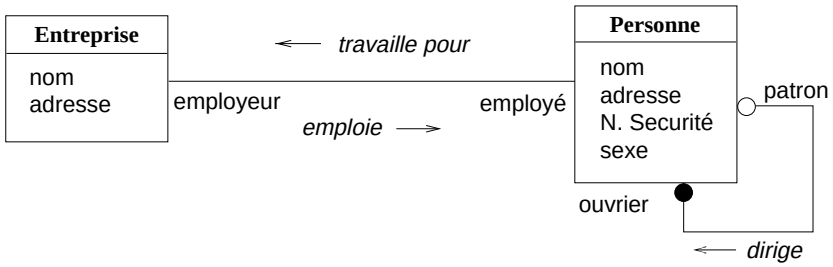


Figure B.3 - OMT - Associations

De chaque côté de l'association, il peut exister un `rôle` (qui montre comme la classe est vue par les autres classes) ainsi que la cardinalité de l'association. Les associations d'une classe vers elle même sont permises. La notation utilisée pour les cardinalités est présenté à la figure B.4.

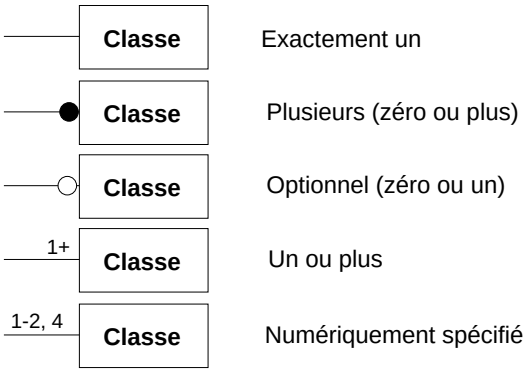


Figure B.4 - OMT - Cardinalités

Lorsqu'une association a des caractéristiques propres, elle peut être vue comme une classe (cf. figure B.5.a). Une dégénération de ce concept est celui d'**Attribut d'une Liaison** où seulement l'attribut de la classe pour la liaison est représenté et, dans ce cas, seulement les noms des attributs sont représentés dans la notation de la classe. Une variante de l'attribut d'une liaison est l'**Association Qualifiée** (cf. figure B.5.b) où un qualificatif est une valeur unique qui réduit la cardinalité du rôle opposé dans une association entre deux classes.

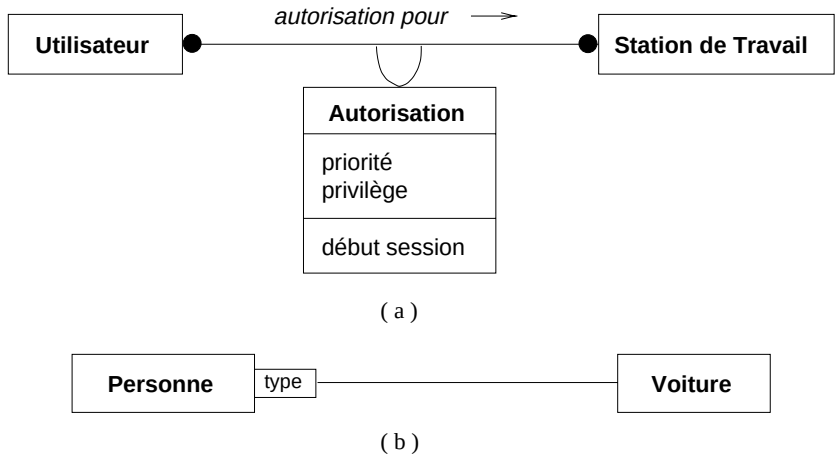


Figure B.5 - OMT - Associations et Attributs

Les associations et les liaisons peuvent être d'ordre binaire, ternaire ou d'ordre supérieur et, dans ce cas, un losange est utilisé pour grouper ces associations ou liaisons (cf. figure B.6).

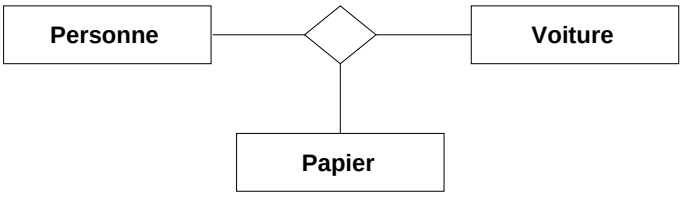


Figure B.6 - OMT - Associations N-Aires

Agrégation : c'est une relation du type Composé-Composant (cf. figure B.7). La propriété la plus importante de l'agrégation est la transitivité : si A est un composant de B et B est un composant de C, alors A est un composant de C. L'agrégation est aussi antisymétrique, c'est-à-dire que si A est un composant de B, alors B n'est pas un composant de A.

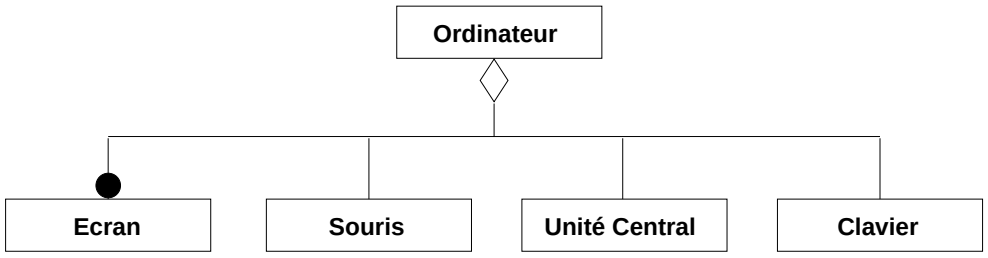


Figure B.7 - OMT - Agrégation

Généralisation : c'est une puissante abstraction pour partager les similarités entre classes en préservant leurs différences. La généralisation est une relation (transitive sur un nombre arbitraire de niveaux) entre une classe et une ou plusieurs autres classes qui sont des raffinements de la première (cf. figure B.8). La classe qui a été raffinée est une `sur-classe` et les classes raffinées sont des `sous-classes`. Chaque sous-classe hérite des caractéristiques de la sur-classe associée. Les attributs et les opérations de la sur-classe sont partagés par leurs sous-classes. Chaque sous-classe peut aussi modifier ou ajouter des attributs et des opérations par rapport à ceux hérités de sa sur-classe.

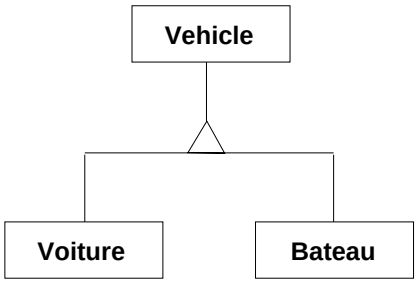


Figure B.8 - OMT - Généralisation

Une instance d'une sous-classe est simultanément une instance de toutes ses classes ancêtres. L'état d'une instance inclut une valeur pour chaque attribut de chaque classe ancêtre ; toute opération de toute classe ancêtre peut être appliquée à toute instance de la sous-classe.

Les mots généralisation, héritage et spécialisation font partie d'une idée commune : la généralisation fait référence à une relation ascendante entre classes tandis que la spécialisation traduit une relation descendante.

Expression de Navigation : c'est une manière pour représenter l'accès à la valeur d'un attribut ou la "traversée" d'une association (**ex** : `Mâle := Personne [Personne.sexe = ''mâle''], Personne.employeur` - cf. figure B.3).

Contraintes : c'est une restriction sur une valeur qui est représentée comme un expression attachée à une classe ou une association (cf. figure B.9).

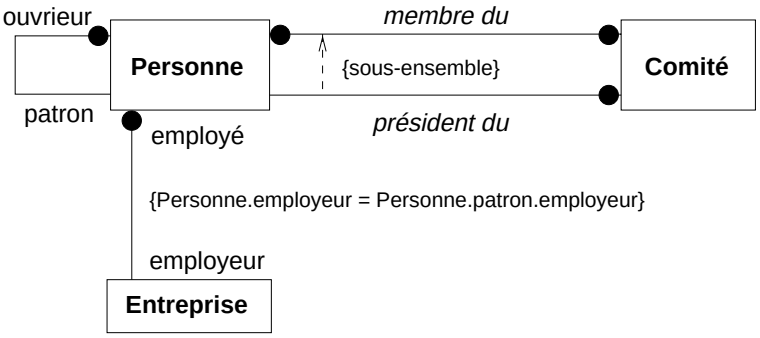


Figure B.9 - OMT - Contraintes

Bien que que les contraintes soient utilisées pour empêcher des redondances dans les diagrammes, elles peuvent être utilisés aussi pour le contraire : c'est le cas des éléments dérivés qui sont représentés avec un "/" devant eux (cf. figure B.10).

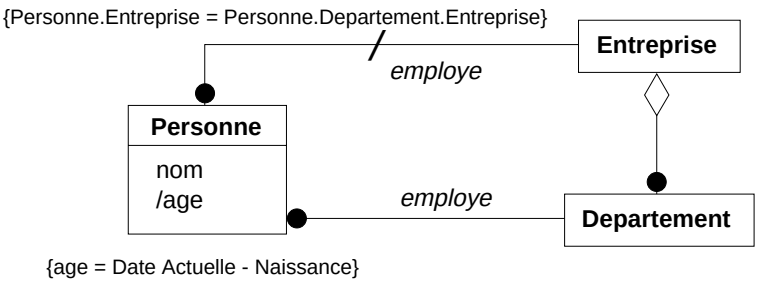


Figure B.10 - OMT - Éléments Dérivés

Sous-Système : c'est un sous-ensemble, une partie d'un modèle entier ; il existe comme outil d'organisation qui n'a aucun sens sémantique. Les sous-systèmes peuvent être emboîtés et les sous-systèmes de plus bas niveaux sont appelés **Modules**. Chaque classe a son module "maison" où ses détails sont déclarés ; une classe peut apparaître dans différents modules, mais doit présenter des détails seulement dans son module maison.

Deux représentations sont utilisés pour les `Diagrammes de Sous-Systèmes` (diagramme où il n'y a que des sous-systèmes) : comme des sous-systèmes emboîtés (cf. figure B.11.a) ou comme d'arbres d'agrégation (cf. figure B.11.b). Un sous-systèmes est représenté comme une boîte pointillée et les dépendances entre modules

sont représentées avec des flèches pointillées qui vont du sous-système qui dépend vers celui qui est indépendant.

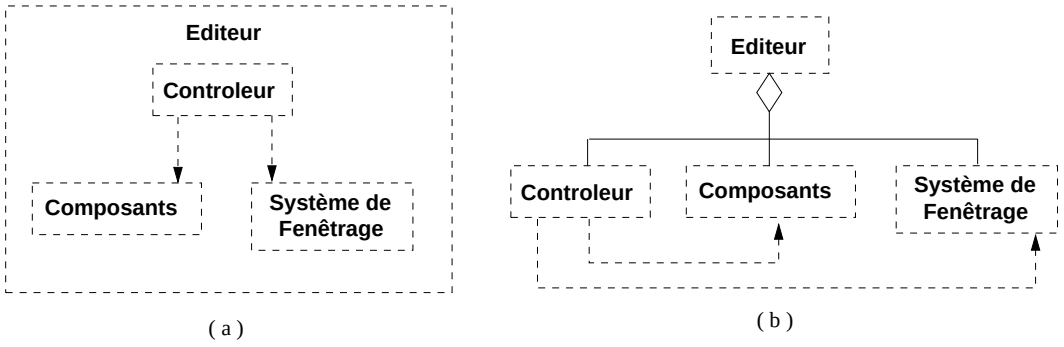


Figure B.11 - OMT - Sous-Systèmes

Un autre diagramme, utilisé avec les sous-systèmes, est le `Diagramme d'Interface de Sous-Systèmes` qui présente les sous-système avec leurs classes et relations publiques (cf. figure B.12). La classe `Fenêtre` suivie de la mention “Importé” indique qu'elle est détaillé dans un autre sous-système.

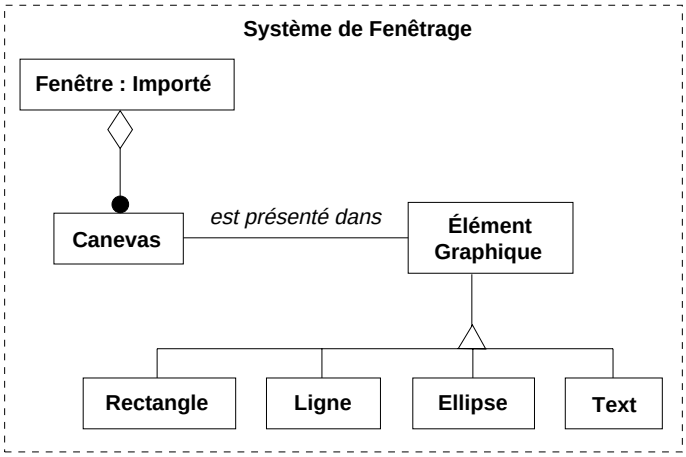


Figure B.12 - OMT - Diagramme d'Interface de Sous-Systèmes

Objet Composé : c'est un type étendu d'agrégation où le composé est vu comme étant une abstraction de niveau plus élevé que les composants ; les objets composés n'ont pas de sens sémantique mais servent à organiser la compréhension d'un modèle. Le composé est représenté comme une boîte englobant les composants (cf. figure B.13). Une

association qui croise la limite de la boîte lie deux objets composés différents ; si l'association ne croise pas les limites, elle lie des éléments internes de l'objet composé.

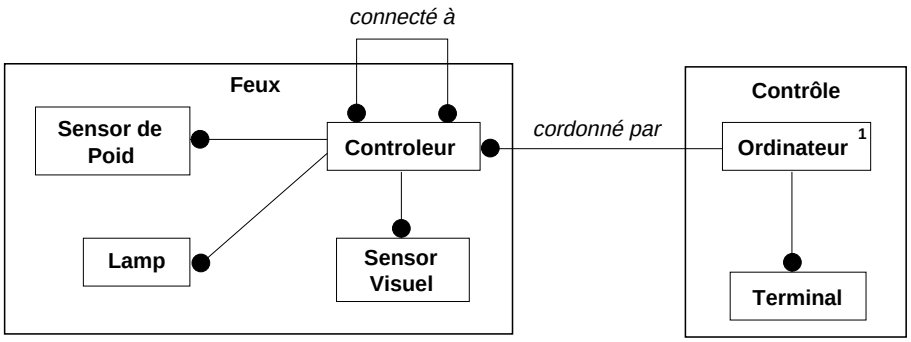


Figure B.13 - OMT - Objet Composé

B.2 Modèle Dynamique

Scénario : c'est un concept utilisé pour représenter une série particulière d'interactions entre objets dans l'exécution d'un système ; il est une instance d'un ensemble de Diagrammes d'États - DE, et décrit une histoire unique, sans conditions sur un déclenchement. Les scénarios représentent des interactions naturelles entre DE sous-jacents qui ne sont pas visibles dans un seul DE. Ils sont utilisés pour la compréhension d'un système lorsque les DE sont utilisés pour la spécification. Les scénarios peuvent être représentés de deux manières : comme un `Traceur d'Événements`, ou comme une boite avec du texte dedans(cf. figure B.14.b).

Dans le Traceur d'Événements les objets sont représentés par des lignes verticales et les événements (entre l'objet qui envoie l'événement et celui que reçoive) sont représentés par une flèche (cf. figure B.14.a) ; une échelle temporelle est implicitement présente : les événements qui sont en bas du traceur arrivent après ceux qui sont en haut ; une `Marque de Temporisation` pour indiquer le temps entre un événement et le suivant peut être ajoute à la ligne qui représente un objet. Une variation pour le Traceur d'Événements ajoute des boîtes sur la ligne qui représente l'objet pour montrer le temps pendant lequel, celui-ci garde le contrôle sur le processus.

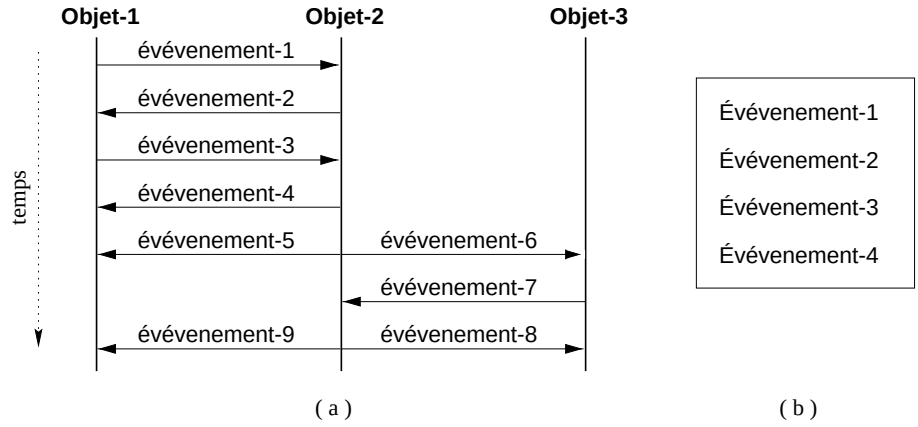


Figure B.14 - OMT - Traceur d'Événements

Cas d'Utilisation : c'est une description générique d'une transaction complète qui englobe plusieurs objets. Un cas d'utilisation peut être présenté comme une description textuelle informelle des acteurs externes et de la séquence d'événements entre les objets qui on produit une transaction ; il peut aussi décrire le comportement d'un ensemble d'objets.

Un scénario est une instance d'un cas d'utilisation et tous les deux sont des modèles d'observation d'un système.

Événement : c'est une transmission d'information asynchronique en sens unique d'un objet vers un autre. Les événements sont le concept les plus fondamental du Modèle Dynamique ; les états peuvent être considérés comme des concepts dérivés.

La spécification complète d'un événement peut optionellement inclure : un nom, une liste de paramètres, les objets émetteur et récepteur de l'événement, la description de la signification de l'événement, le mécanisme d'implantation et une marque de temporisation.

Les événements peuvent être modélisés comme une structure de généralisation similaire à celles d'un diagramme de classes ; dans cette structure, les attributs sont des paramètres de l'événement.

État : c'est la période de temps pendant laquelle un objet attend qu'un événement se produise. Les états peuvent avoir des `variables d'états` qui sont des attributs d'objets décrits par un DE et que sont valides lorsque l'objet est dans l'état ou dans un sous-état de cet état ; plus particulièrement ce sont des attributs qui ont un rapport avec le flux de contrôle. Les variables d'états peuvent être accédées et modifiées par

les `opérations d'état`, inclus les deux types particuliers qui sont les opération d'entrées et les opération de sortie.

Un état peut avoir une `opération interne` exécutée après la réception d'un événement, sans que celui-ci déclenche un changement d'état, ainsi que des `activités`. Les activités sont des opérations qui ont besoin de temps pour être réalisées. Il peut aussi avoir une `opération d'entrée` et une `opération de sortie` attachée ; celles-ci sont des opérations qui se déroulent à l'entrée ou à la sortie d'un état et ne sont pas citées sur l'arc de transition mais dans la boîte de l'état. Si un état a plusieurs opérations associées, elles seront réalisées dans l'ordre suivant : l'opération de la transition d'entrée, l'opération d'entrée, les activités, l'opération de sortie et l'opération de la transition de sortie. La figure B.15.a présente la notation utilisée pour représenter les états. Nous pouvons remarquer, que la notation utilisée, est celle proposée par D. Harel [Har87].

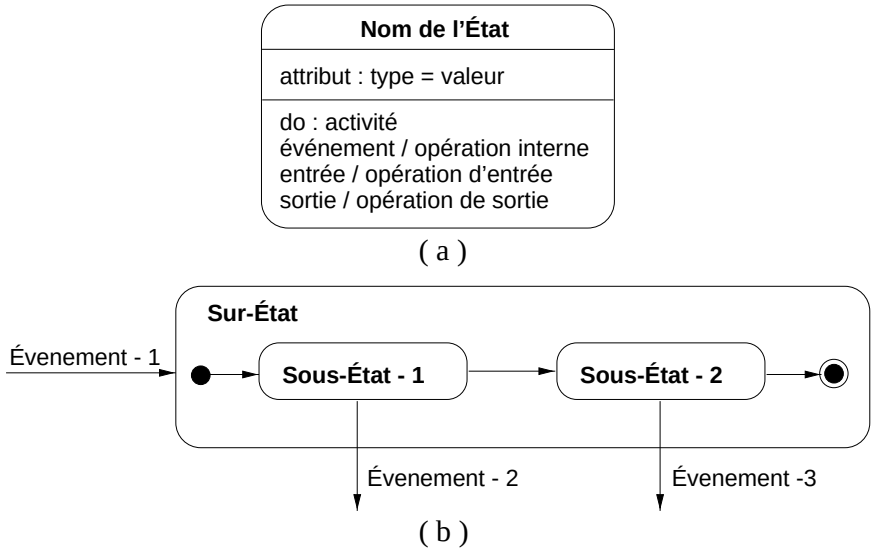


Figure B.15 - OMT - États

Un état peut être généralisé comme un `sur-état` pour lequel des `sous-états` disjoints sont définis (cf. figure B.15.b) ; les sous-états sont le résultat du raffinement du sur-état ; c'est l'équivalent des sous-classes disjointes. Un sous-état hérite les propriétés de son sur-état : les variables d'état et les transitions (internes et externes). Dans la boîte d'un sur-état comme dans un DE général, on peut représenter un `état de départ` (le cercle à gauche dans le sur-état) et un `état d'arrivée` (le cercle double à droite du sur-état).

Les sous-états d'un sur-état peuvent être concurrents et, dans ce cas, lorsque le sur-état est atteint, un "flux de contrôle" ("thread") est crée pour chaque état concurrent (cf. figure B.16). Une transition d'un sous-état vers un état autre que les sous-état du sur-état provoque l'achèvement de tous les états par une sortie forcée ; une transition sans nom qui part d'un sur-état indique que tous les sous-états doivent être atteints pour que cette transition se produise.

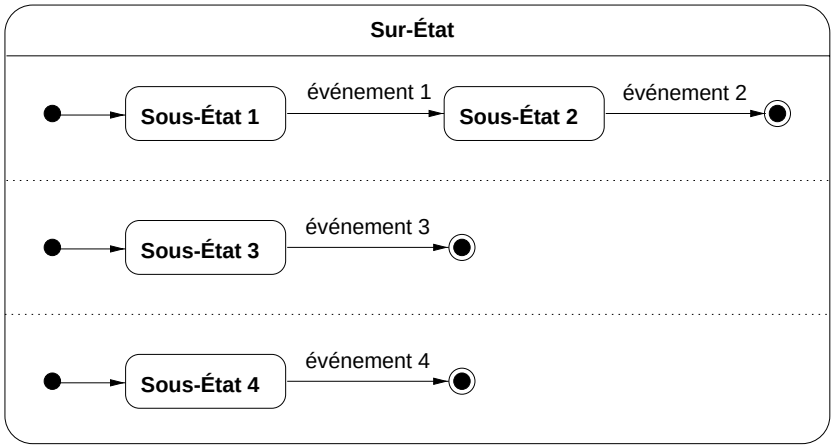


Figure B.16 - OMT - États Concurrents

État Composé : c'est une vue de haut niveau d'un état qui peut être étendue comme une vue détaillée de celui-ci ; c'est l'équivalent d'un objet composé. Dans le haut niveau, un état composé est comme un état commun. Dans les niveaux plus bas, le nombre de détails est plus important et comprend plusieurs états et transitions d'état ; les événements des niveaux plus bas ne sont pas visibles dans les niveaux plus hauts.

Les états composés ont un état de départ par défaut : les transitions qui arrivent au état composé représentent une transition vers cet état de départ. La figure B.16 présente la notation utilisé pour représenter les états composés.

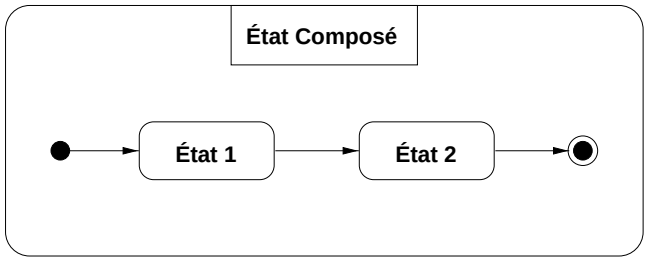


Figure B.17 - OMT - États Composés

Transition d'État : c'est la conséquence de la réception d'un événement. Une transition qui arrive à un état peut invoquer une opération ainsi que le changement d'état d'un objet. Les transition d'états peuvent être externes, cas où elles conduisent à un nouvel état et peuvent invoquer une opération, ou interne cas où elles invoquent une opération sans provoquer un changement d'état.

Une transition peut être caractérisée par les éléments suivants (optionnelles) : `événement(arguments)`, `[condition]`,^ `cible.événement(arguments)` et `opération(argument)` ; les valeurs des paramètres des événements sont disponibles pour les opérations déclenchées par celui-ci ; pour que la transition se déclenche il faut que `[condition]` soit satisfaite.

Opération : elle est invoquée par les transitions pour les objets sous contrôle et est appelée `action` qui est une opération instantanée, c'est-à-dire, non interruptible. Une action peut être implantée comme un méthode de l'objet sous contrôle.

Une opération comme une méthode, a accès aux paramètres de l'événement déclencheur ainsi qu'aux variables d'état et aux autres attributs de l'objet sous contrôle. Elle peut aussi invoquer d'autres opérations accessibles par l'objet sous contrôle.

Envoi d'Événement : c'est une action qui peut être exécutée par un objet et qui, étant donné sa grande influence sur le flux de contrôle, a une syntaxe particulière. L'envoi d'un événement, spécifié comme un élément d'une transition (`cible.événement(arguments)`), peut être aussi représenté par une flèche pointillée avec le nom de l'événement envoyé. Cette flèche doit partir de la transition que l'envoie vers l'objet cible. La figure B.18 présente les deux représentations. Un diagramme qui ne présente que des objets avec les événements qu'ils envoient est appelé `Diagramme de Flux d'Événements`.

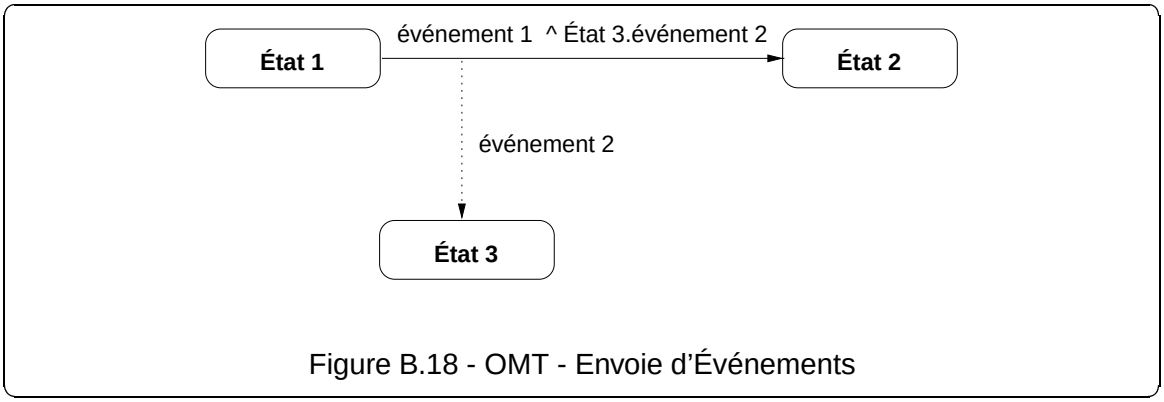


Figure B.18 - OMT - Envoie d'Événements

Un cas particulier d'envoi d'événement est la `Création d'Objets`. Dans ce cas, une classe envoie un événement à elle même pour la création des nouveaux objets de

la classe. Les arguments de l'événement envoyé sont utilisés pour instancier le nouvel objet.

La `Destruction d'Objets`, lorsque l'objet arrive à son état terminal, peut entraîner un envoi d'événements, quoique la destruction, au contraire de la création, ne soit pas un envoi d'événement.

B.3 Modèle Fonctionnel

Description des Opérations : c'est la spécification d'une opération en utilisant du texte informel avec des pré et post conditions. Quoique la manière la plus sûre pour spécifier une opération soit une spécification formelle, dans [Rum95d] J. Rumbaugh suggère qu'une spécification informelle est plus claire et dans la plus part des cas satisfaisante (cf. figure B.19).

Opération :	Nom de l' opération (signature)
Responsabilité :	Déscription de l' objectif pour lequel l' opération sera créee
Entrées :	Déscription des entrées de l'opération
Objets Modifiés :	Noms des objets qui sont modifiés par l'opération
Pré Conditions :	Déscription de l'état d'un objet avant l'exécution d'une opération
Post Conditions :	Déscription de l'état d'un objet après l'exécution d'une opération

Figure B.19 - OMT - Spécification des Opérations

Dans les pré conditions, on déclare l'état du système au début de l'exécution d'une opération et, dans la post condition, on décrit l'état du système à la fin de l'exécution d'une opération par rapport à l'état du système avant l'exécution de l'opération.

Le Modèle Fonctionnel doit spécifier uniquement les opérations de haut niveau du système, lesquelles sont invoquées par des interactions avec des acteurs externes.

Diagramme de Flux de Données Orienté Objet - DFDOO : il présente les effets d'une opération sur l'objet et leurs valeurs dans un système, c'est-à-dire les effets de l'exécution d'une opération particulière sur un état particulier d'un système. Les DFDOO (cf. figure B.20) sont utiles pour la compréhension des effets d'une opération à travers plusieurs objets.

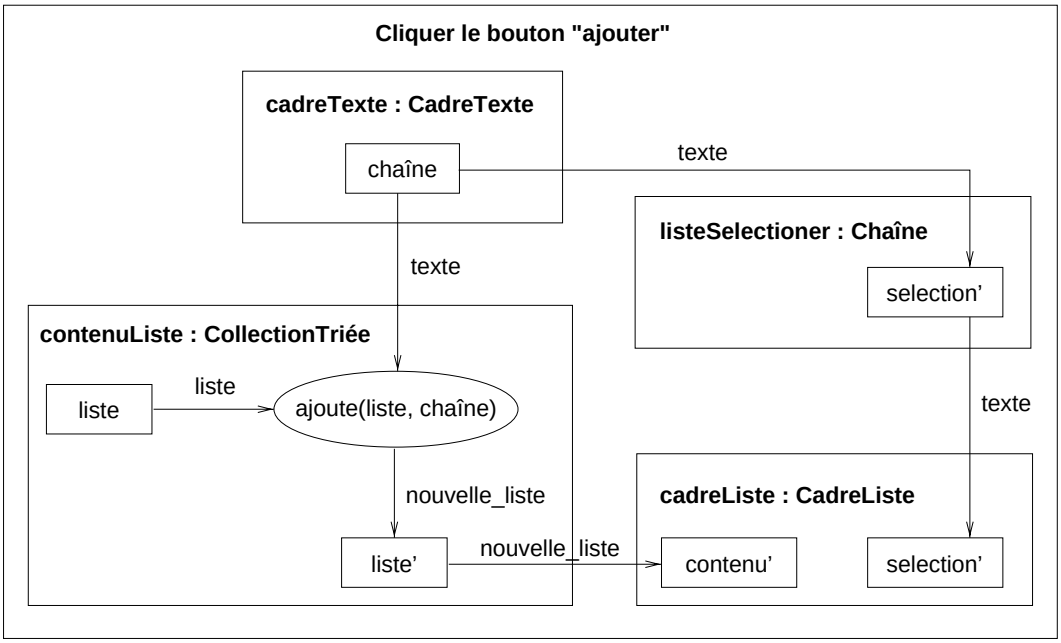


Figure B.20 - OMT - Diagramme de Flux de Données Orienté Objet

La figure B.20 présente un DFDOO pour une opération particulière (“Cliquer le bouton Ajouter”) avec des composants standard des DFDOO qui sont :

- `objets` : ils sont les objets du Modèle des Objets ; ils sont représentés dans une boîte avec le nom de l’objet suivi de deux points et du nom de leur classe ;
- `valeurs des objets` : ce sont les valeurs des attributs de la classe pour l’objet ; les valeurs sont décorées avec un apostrophe pour exprimer un changement. Une même valeur peut être utilisée comme entrée pour plusieurs fonctions, mais elle ne peut être utilisée qu’une seule fois comme valeur de sortie d’une fonction ; elles sont représentées avec le nom de l’attribut dans une boîte dans la boîte de l’objet ;
- `fonctions` : ce sont des applications (“mappings”) entre une ou plusieurs valeurs d’entrée vers une ou plusieurs valeurs de sortie ; elles sont représentées comme une ellipse avec le nom de la fonction dedans ;
- `flux de données` : ils font la connexion entre la sortie d’une fonction et une valeur d’attribut générée, ou entre la valeur d’un attribut et l’entrée d’une fonction, ou entre la sortie d’une fonction et l’entrée d’une autre fonction si la valeur de sortie est temporaire ; un flux de données peut aussi connecter deux valeurs et dans ce cas la deuxième valeur est une copie de la première ; ils sont représentés avec des flèches avec le nom du flux de donnée.

Quoique les DFDOO puissent être composés d'opérations complexes qui peuvent être décomposées, la méthode conseille la présence uniquement de fonctions pures (des fonctions qui n'ont pas d'état interne ni des effets de bord qui ne soient pas des valeurs de sortie) dans les DFDOO, car presque tous les DFDOO peuvent être décomposés en un autre, avec uniquement des fonctions pures.

Les DFDOO doivent être utilisés pour la compréhension de la manière par laquelle une opération affecte un système et non pour réaliser une spécification formelle de celui-ci.

Diagramme d'Interaction des Objets - DIO : c'est un diagramme d'objets qui aide à la compréhension d'une opération. Il présente la séquence de messages qui implantent une opération ayant comme composants les objets (et leurs liaisons) qui ont rapport avec cette opération. Le DIO présente les objets et liaisons qui existent avant l'exécution d'une opération et aussi ceux qui sont crées pendant cette exécution. Dans le diagrammes, les objets et liaisons existant sont représentés avec des lignes noires et ceux qui sont crées avec des lignes grises (cf. figure B.21).

Les changements d'un objets sont étroitement liés aux flux de contrôle. Ces flux de contrôle, dans les systèmes orientés objets, suivent les `liaisons de données` (associations et autres liens transitoires comme par exemple les paramètres de procédures et les variables locales) et, pour cela, le DIO contient tous les chemins ("paths") qu'un flux de contrôle peut suivre. Les liens transitoires sont représentés avec des lignes pointillées avec leur nom entre parenthèses (cf. figure B.21).

Un `message` d'un objet vers un autre est représenté par une étiquette composée d'une chaîne de caractères avec une flèche qui présente la direction du message ; plusieurs messages peuvent être attachés à la même liaison. L'étiquette est composée des éléments suivants (quelques uns optionnels) :

- `numéro de séquence` : il présente les séquences d'appels emboîtés. Un même numéro utilisé par deux messages indique que les deux sont concurrents (ils sont exécutés en parallèle) ; une lettre peut être attachée pour distinguer des sous arbres concurrents ;
- `indicateur d'interaction` : c'est une étoile qui peut être suivie optionnellement d'une expression d'itération entre parenthèses ; une itération indique qu'un message est envoyé d'une manière séquentielle plusieurs fois à la même cible, ou simultanément à des éléments d'un ensemble ;

– `valeur de retour` : s'il y a un, cet élément indique que l'opération retourne une valeur qui est désignée par un nom ;

– `nom du message` : c'est un élément optionnel car le nom de la classe du message est explicite dans l'objet cible ;

– `liste d'arguments` : c'est une expression construite des valeurs d'entrée des transactions, des valeurs de retour de sous-opération ou des valeurs d'attributs de l'objet cible.

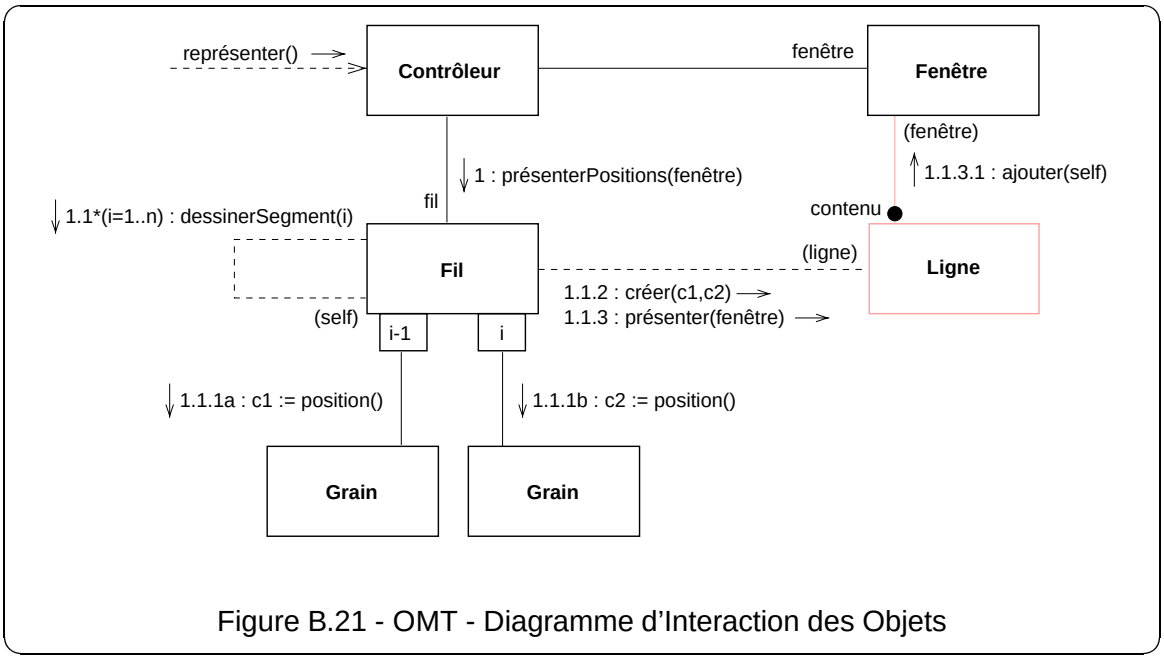


Figure B.21 - OMT - Diagramme d'Interaction des Objets

Diagramme d'Interactions Concurrents - DIC : c'est un diagramme qui présente les flux de contrôle d'un système lorsqu'il y a de la concurrence. Avec ces diagrammes, il existe plus qu'une zone de contrôle ("locus of control") à un instant du temps pour un `objet actif`. Un objet actif est un objet qui a son propre flux de contrôle et que échange des événements asynchroniquement avec d'autres objets ; il est un objet composé qui peut avoir à son intérieur d'autres objets, eux-mêmes pouvant être actifs, et qui est représenté par une boîte avec des lignes épaisses (cf. figure B.22).

Les messages entre les objets actifs suivent les `liaisons de données` de la même façon que les messages entre les objets passifs ; une représentation particulière avec une "demi-flèche" (cf. figure B.22) peut être utilisé pour représenter les messages entre les objets actifs. L'objet actif qu'envoie un message asynchrone n'attend pas le message de retour.

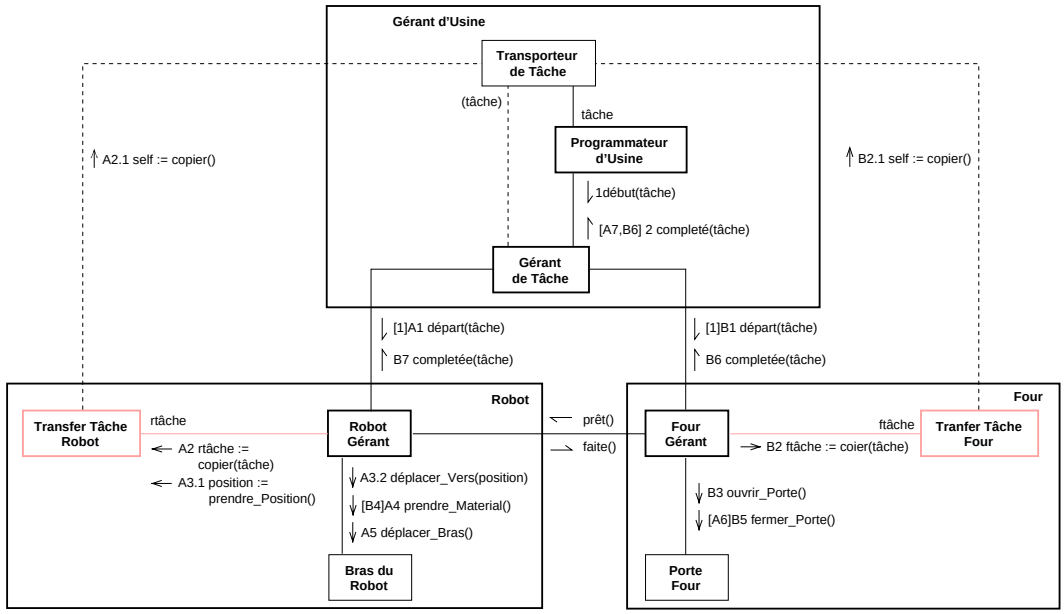


Figure B.22 - OMT - Diagramme d'Interactions Concurrents

Le flux de contrôle dans les DIC peut se diviser et se fondre et, à cause de cela, il faut une notation particulière pour le représenter. Les différentes sous-séquences sont représentées par une lettre suivie d'un numéro (comme "B3"), où chaque lettre définit une sous-séquence ; la sous-séquence principale ne présente pas de lettres. La dépendance entre les sous-séquences concurrentes est représentée de la manière suivante : "[B2]C4" montre que C4 suit le message B2 (explicitement) et C3 (implicitement). Les messages entre les objets passifs sont représentés comme dans les DIO.

Code : c'est, selon la méthode, la meilleur manière pour concevoir le comportement d'une opération. Pour arriver au code, on peut passer par une étape intermédiaire qui est le `Pseudo-code`.

Les Diagramme d'Interaction des Objets et les Diagramme d'Interactions Concurrents présentent l'effet de l'exécution d'une opération de haut niveau unique : ils sont des instances et non des descriptions génériques.

Pour la description de la Procédure de Développement adopté par la méthode ainsi que pour la présentation d'un bilan sur celle-ci, en plus des directives données par J. Rumbaugh [Rum95c], on prend en compte le travaux de M.E. Fayad et all. [FTAF94] et de D. D'Souza [D'S93]. Dans le premier papier, les auteurs font une analyse sur l'utilisation de la méthode OMT pour la construction d'un système d'information pour un "Système de Planification

d'une Mission pour un Missile Auto Guidé". Dans le deuxième, l'auteur présente une analyse sur l'utilisation de la méthode OMT dans des projets commerciaux et dans l'enseignement.

B.4 Procédure de Développement

La procédure de développement proposée par la méthode est composée de quatre étages. Ces étages ainsi que leurs étapes et les produits finals sont présentés ci-dessous.

– *Analyse* : cette étape a comme rôle la conception d'un modèle précis, concis, intelligible et correct du monde réel. L'analyse sert à faciliter la compréhension du domaine du problème d'un système avant qu'on puisse bâtir un système logiciel/matériel complexe. L'analyse commence par l'écriture d'une description initiale du domaine du problème sur laquelle un *Modèle d'Analyse* est construit. Cette description montre la structure statique (Modèle des Objets), l'ordonnancement des interactions (Modèle Dynamique) et la transformation des données (Modèle Fonctionnel).

  Le Modèle des Objets favorise la communication entre les personnes du domaine informatique et les experts du domaine du problème. La construction du Modèle d'Analyse commence par le Modèle des Objets car la structure statique d'un système est normalement la partie la mieux définie. Bien que l'analyse ne soit pas un processus linéaire, on peut dire que pour construire le Modèle des Objets les étapes suivantes sont conseillées :

  1. écrire une description initiale du problème ;
  2. construire un Modèle d'Objets en utilisant les diagrammes proposés par la méthode ainsi qu'un dictionnaire de données ;
  3. construire un Modèle Dynamique en utilisant les diagrammes proposés par la méthode ;
  4. construire un Modèle Fonctionnel en utilisant les diagrammes proposés par la méthode ;
  5. faire interagir les trois modèles et les raffiner ;

  L'étape d'analyse a comme produits finals une Description du Problème, le Modèle d'Objets, le Modèle Dynamique et le Modèle Fonctionnel.

– *Projet du Système* : cette étape a comme rôle la construction de l'architecture globale du système ; c'est une étape qu'on peut dire de "projet préliminaire", dans laquelle les décisions de haut niveau sur l'architecture du système sont prises.

  Le Projet de Système est réalisé avec un *Modèle de Projet*. Pour construire ce modèle on peut suivre les étapes suivantes :

  1. organiser le Système en Sous-Systèmes ;
  2. identifier les concurrences inhérentes au problème ;
  3. soumettre les sous-systèmes aux processeurs et aux tâches ;
  4. choisir la stratégie de base pour l'implémentation des dépôts de données en considérant les structures de données, les fichiers et les bases de données ;
  5. identifier les ressources globales et déterminer les mécanismes pour contrôler leurs accès ;
  6. choisir une approche pour accomplir le contrôle du logiciel ;
  7. considérer les conditions frontières ;
  8. établir des compromis de projet.

– *Projet des Objets* : cette étape a comme rôle l'élaboration et le raffinement des modèles de la phase d'analyse afin de produire un projet pratique de l'implémentation du système. Cette étape est centralisée sur la définition des structures de données et algorithmes qui seront utilisés pour implanter les classes ; l'étape doit optimiser, raffiner et développer les Modèles des Objets, dynamique et fonctionnel jusqu'à ce qu'ils soient prêts pour l'implantation. Les étapes suggérées sont :

  1. obtenir les opérations pour le modèle d'objets ;
  2. développer des algorithmes pour ces opérations ;
  3. optimiser les chemins d'accès aux données ;
  4. implanter le contrôle du logiciel ;
  5. raffiner les structure des classes pour augmenter l'héritage :
  6. déterminer la représentation exacte des attributs des objets ;

– *Implantation* : dans cette étape, les classes et les relations développées dans les autres étapes sont traduites dans une langage de programmation spécifique. Jusqu'à ce qu'on arrive à cette représentation finale pour les classes, elles peuvent évoluer et d'autres classes peuvent être ajoutées aux modèles.

Annexe C

La Méthode OOD

Dans cette annexe on présente les composants principaux des quatre vues de la méthode OOD proposé par G. Booch, à travers les diagrammes que les composent. Cette présentation est basée sur la description de la méthode faite par G. Booch [Boo94].

C.1 Diagramme de Classes

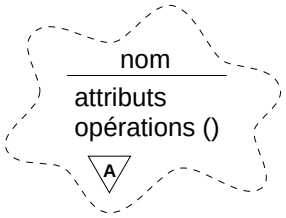
Les Diagrammes de Classes sont utilisés pour montrer l'existence des Classes et leurs Relations. Un Diagramme de Classes représente une abstraction du domaine du problème. Les deux éléments essentiels d'un Diagramme de Classes sont les Classes et leurs Relations.

Classe : c'est un ensemble d'objets qui partagent une structure et un comportement commun. Les classes qui ne peuvent pas être instanciées portent le nom de `Classes Abstraites` et sont notées par un petit triangle avec un "A" dedans, lequel est mis dans le symbole des classes (cf. figure C.1.a).

Attribut : il exprime une propriété particulière d'une classe ; il faut que les attributs aient un `Nom` et appartiennent à une classe. Les attributs peuvent être représentés de différents manières (cf. figure C.1.b).

Opération : elle exprime les services accomplis par une classe ; la notation pour différencier les attributs des opérations consiste à utiliser des parenthèses après le nom de l'opération ; les opérations peuvent aussi avoir une signature complète, comme présenté ci-dessous.

- `N()` : seulement le nom de l'opération ;
- `R N(Arguments)` : la classe retournée, le nom de l'opération et les arguments s'ils existent.



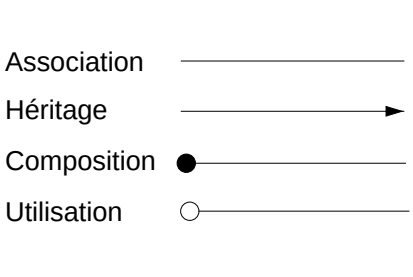
(a)

A	seulement le nom de l'attribut.
: C	seulement la classe de l' attribut
A : C	le nom de la classe et de l'attribut.
A : C = E	le nom de l'attribut, de la classe et une expression par défaut.

(b)

Figure C.1 - OOD - Classes, Attributs et Opérations

Relations : elles expriment quelques types de partages ou de connexions sémantiques. Les notations associées aux types de relations proposées par la méthode sont présentées dans la figure C.2.a.



(a)

1	exactement un.
N	nombre quelconque (zéro ou plusieurs).
0 .. N	zéro ou plus.
1 .. N	un ou plus.
0 .. 1	zéro ou un.
3 .. 7	intervalle spécifié.
1 .. 3, 7	intervale spécifié ou nombre exact.

(b)

Figure C.2 - OOD - Relations et Cardinalités

Les relations `Association` représentent une connexion sémantique entre deux classes. Les relations `Héritage` représentent des relations du type Généralisation-Spécialisation. Les relations `Avoir` représentent des relations du type Composé/ Composant et les relations `Utiliser` représentent des relations du type Client/ Fournisseur, où le fournisseur fournit des services au client. Les associations doivent être nommées avec des noms qui expriment leur nature et elles peuvent avoir des cardinalités qui sont représentées selon la figure C.2.b.

Catégorie de Classes : elle est utilisée pour fractionner le Modèle Logique d'un système. Une catégorie de classes est composée de classes et d'autres catégories de classes (cf. figure C.3).

Une catégorie de classes peut utiliser d'autres catégories de classes ou des classes ; une classe peut utiliser une catégorie de classes. Ces relations sont du type "Utiliser" et emploient la même notation que celle de la relation "utiliser" entre classes (cf. figure C.2.a).

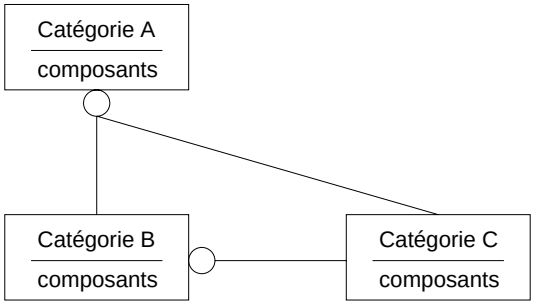


Figure C.3 - OOD - Catégories de Classes

La Méthode présente les notions de concepts essentiels et de concepts avancés. Jusqu'à maintenant, on n'a présenté que des concepts essentiels. Ci-dessous, on présente les concepts avancés des Diagrammes de Classes.

Classe Paramétrée : elle représente des types de classes introduits par certains langages de programmation orientés objets, comme C++, Eiffel et Ada ; une classe paramétrée sert de patron ("template") pour d'autres classes, objets et/ou opérations et elles ne peuvent être utilisées qu'après avoir été instanciées. La relation entre la classe paramétrée et la classe instanciée est représentée par une flèche pointillée de la classe instanciée vers la classe paramétrée (cf. figure C.4).

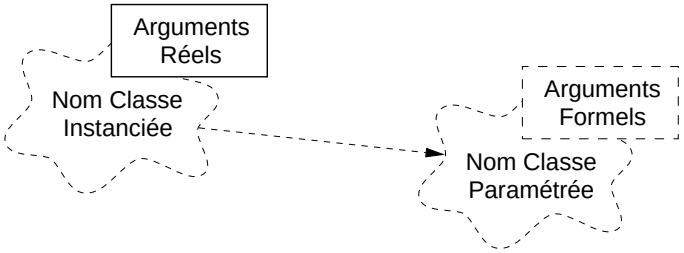


Figure C.4 - OOD - Classes Paramétrées

Méta-Classe : c'est un concept introduit par les langages comme CLOS et Smalltalk pour représenter une classe de classes. Le lien entre une méta-classe et une classe instanciée est donné par une `Méta-Relation` (représentée par une flèche en gris) qui représentent l'instanciation d'une classe depuis une méta-classe (cf. figure C.5.a).

Classe Utilitaire : c'est un concept utilisé pour modéliser des sous-programmes qui ne font partie d'aucune classe, mais qui ont une existence justifiée pour exécuter des algorithmes utiles au système (cf. figure C.5.b).

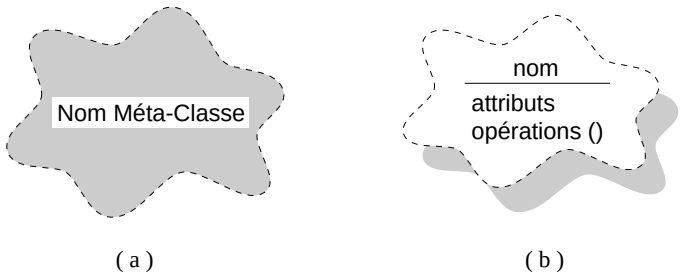


Figure C.5 - OOD - Méta-Classes et Classes Utilitaires

Contrôle d'Exportation : c'est un concept introduit pour modéliser la caractéristique de séparation entre l'interface et l'implantation d'une classe, c'est-à-dire, un moyen pour contrôler l'accès à leurs opérations que la plupart des langages de programmation orientés objet offrent. La notation graphique de ce concept est présentée à la figure C.6.a.

Propriété : c'est un concept utilisé pour modéliser quelques aspects sémantiques des relations introduites par certains langages de programmation orientés objets. Pour exprimer ces propriétés, on utilise de petits triangles ; pour C++, par exemple, on peut utiliser la notation présentée à la figure C.6.b.

Appartenance Physique : les relations "Avoir" n'ont pas le sens de possession d'un composé sur ses composants. Cette possession est modélisée dans la méthode grâce au concept d'`Appartenance par Valeur`, pour montrer que le composé n'existe qu'avec ses composants et avec le concept d'`Appartenance par Référence`, pour montrer que le composant peut avoir une existence indépendante de son composé (cf. figure C.6.c).

Rôle : il représente la fonction ou la capacité avec laquelle une classe ou un objet fait partie d'une relation. Les rôles sont représentés comme un complément textuel, qui doit être placé à côté de la classe qui offre le rôle.

Clé : c'est un attribut d'une classe dont la valeur unique identifie un objet cible. Les clés sont représentées dans la méthode entre crochets ([ ]).

Contrainte : une contrainte est l'expression de conditions sémantiques qu'il faut préserver ; c'est un "invariant" (une expression booléenne de conditions dont la vérité doit être préservée) d'une classe ou d'une relation ; cet invariant doit être préservé quand le système est dans un état stable. Les contraintes sont représentées entre accolades ({ }).

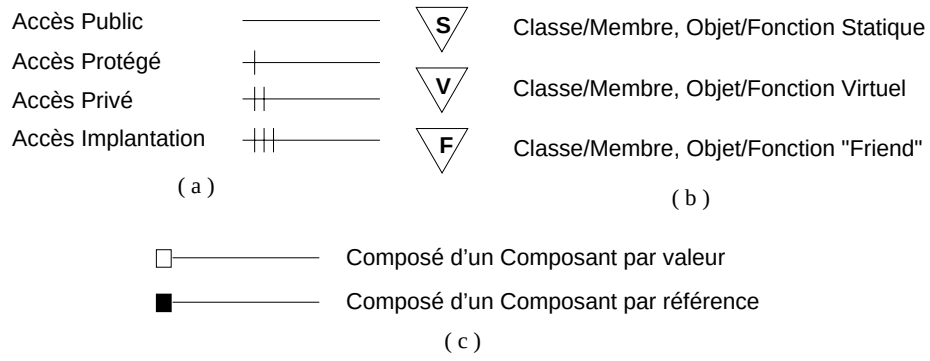


Figure C.6 - OOD - Contrôle d'Exportation, Propriétés et Appartenance Physique

Association Attribuée : c'est un concept qui aide à la modélisation des relations N :N. Une Association Attribuée est une classe qui contient les propriétés de la Relation qu'elle caractérise. Cette classe est reliée à l'association par une ligne pointillée comme on peut le voir à la figure C.7.

Note : c'est une généralisation des associations attribuées. Les notes capturent les suppositions et les décisions qui ont été prises pendant la modélisation. La notation utilisée pour représenter les notes peut être vue à la figure C.7.

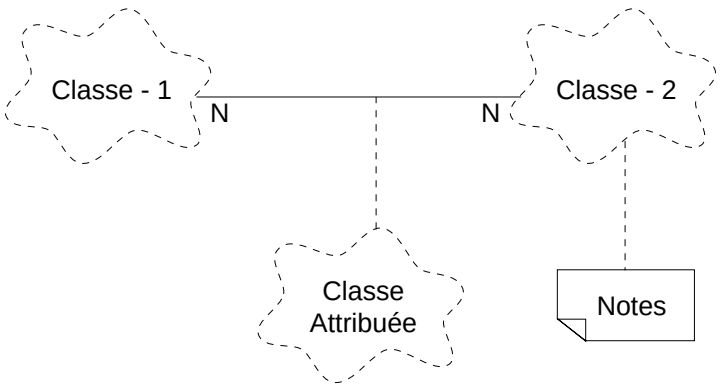


Figure C.7 - OOD - Associations Attribuées et Notes

Spécification : c'est un complément textuel de la modélisation graphique du système, introduit pour donner une définition complète des entités de la notation. Tous les éléments du modèle ont au moins les entrées suivantes : un `Nom` défini par un identificateur et une `Définition` textuelle. La spécification textuelle pour les classes est présentée dans la figure C.8.

1.	Responsabilités :	Texte
2.	Attributs :	Liste des attributs
3.	Opérations :	Liste des opérations
4.	Contraintes :	Liste des Contraintes
5.	Machine d'État :	Référence à une Machine d'État
6.	Contrôle d'Exportation :	public | implantation
7.	Cardinalité :	Expression
8.	Paramètres :	Liste des Paramètres formels ou actuels
9.	Persistance :	transitoire | persistant
10.	Concurrence :	séquentiel | gardé | synchrone | actif
11.	Complexité d'Espace :	Expression

Figure C.8 - OOD - Spécification Textuelle des Classes

Dans la figure C.8, les items 1, 2, 3, 4 montrent la spécification textuelle qui complète la notation graphique des aspects essentiels pour chaque classe du modèle. Les items 5, 6, 7 montrent la spécification textuelle qui complète la notation graphique des aspects avancés pour chaque classe. L'item 8 montre la spécification textuelle qui complète la notation graphique d'une classe paramétrée. Les items 9, 10, 11 montrent quelques aspects fonctionnels des classes non représentés dans la notation graphique.

La Spécification textuelle pour les opérations est présentée dans la figure C.9 ; les items 1 et 2 montrent la spécification textuelle qui complète la notation graphique pour chaque opération de chaque classe et chaque "sous-programme libre" dans le modèle. L'item 3 montre des aspects qui ont un rapport avec certains langages. Les items 4 et 5 montrent la spécification textuelle qui complète la notation graphique des aspects avancés. Les items 6, 7, 8 et 9 montrent certains aspects sémantiques des opérations non représentés sur la notation graphique. Les items 10, 11 et 12 montrent certains aspects fonctionnels des opérations non représentés sur leur notation graphique.

1.	Classe Retournée :	Référence à une classe
2.	Arguments :	Liste des arguments formels
3.	Qualification :	Texte
4.	Contrôle d'Exportation :	public | protégé | privé | implantation
5.	Protocole :	Texte
6.	Pré-conditions :	texte | référence au code source | référence au diagramme d'objet
7.	Sémantiques :	texte | référence au code source | référence au diagramme d'objet
8.	Post-conditions :	texte | référence au code source | référence au diagramme d'objet
9.	Exceptions :	Liste des exceptions
10.	Concurrence :	séquentiel | gardé | synchrone
11.	Complexité d'Espace :	Expression
12.	Complexité du Temps :	Expression

Figure C.9 - OOD - Spécification Textuelle des Opérations

C.2 Diagramme de Transitions d'États

Les Diagrammes de Transitions d'États sont utilisés pour montrer l'espace des états (une énumération des états possibles) d'une classe, les événements qui produisent une transition d'un état vers un autre et les actions qui résultent d'un changement d'état ; un Diagramme de Transitions d'États représente une vue du Modèle Dynamique d'une classe unique ou du système complet. Les constituants essentiels d'un Diagramme de Transitions d'États sont les états et les transitions d'états. La méthode utilise pour ces Diagrammes de Transitions d'États, la notation proposée par Harel [Har88].

État : il représente un résultat du comportement d'un objet. Il faut noter qu'à n'importe quel instant, l'état d'un objet englobe ses propriétés (la totalité des attributs de l'objet et des associations auxquelles il participe) avec les valeurs courantes de ces propriétés.

Comme toutes les instances d'une classe sont dans un même espace d'états, on peut généraliser le concept d'état d'un objet au concept d'état d'une classe. La figure C.10.a montre la notation employée pour les états. Chaque Diagramme de Transitions d'États doit avoir un `état de départ` et un `état d'arrêt` (cf. figure C.10.b).

Transition d'État : c'est la représentation d'un changement d'état d'un objet. Une transition d'état est déclenchée par un événement qui peut déclencher lui-même une action. La notation utilisée est présentée à la figure C.10.c.

Événement : c'est une occurrence de quelque chose qui peut conduire à un changement d'état d'un système et au déclenchement d'une action. Un événement peut avoir une `garde` qui conditionne son déclenchement.

Action : c'est une opération que l'on considère comme instantanée (durée égale à zéro) et qui dénote l'invocation d'une méthode, le déclenchement d'un autre événement, ou le départ ou l'arrêt d'une **Activité** (opération de durée différente de zéro). Les actions sont notées selon la figure C.10.d.

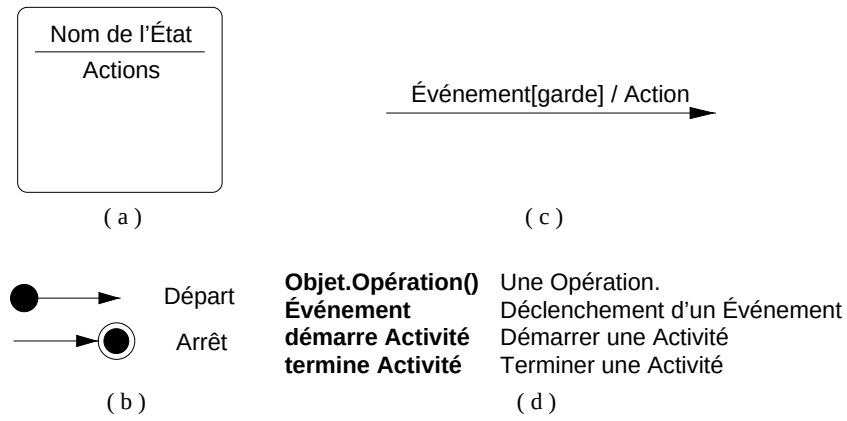


Figure C.10 - OOD - États et Transitions d'États

On présente maintenant des concepts avancés pour les Diagrammes de Transition d'États.

Actions d'Entrée et de Sortie : ce sont des actions exécutées lorsqu'un objet entre dans un état, ou avant qu'il quitte un état. La méthode propose une notation textuelle placée dans la notation de l'état de la façon suivante :

- **entrée** Action : pour l'Action d'Entrée ;
- **sortie** Action : pour l'Action de Sortie.

Activité d'État : on peut associer une activité à un état ; cela signifie que quand un objet est dans un état, il exécute une activité. La notation est composée, à l'intérieur d'un état, du nom de l'activité précédé du mot `do`.

Transition Conditionnelle : elle sert à modéliser le fait qu'une transition d'état peut déclencher ou non une action après l'évaluation d'une condition. La notation textuelle proposée est l'écriture de la condition entre crochets ([ ]) après le nom de l'événement.

État Emboîté : il permet de considérer des sur-états et des sous-états (le sur-état est un état composé de quelques sous-états). Les états emboîtés introduisent une certaine profondeur dans les Diagrammes de Transitions d'États.

Historique : il permet de connaître le dernier sous-état atteint lors de la dernière évaluation d'un sur-état en vue du prochain passage. La notation textuelle proposée consiste dans l'écriture d'un `H` dans un petit cercle dans le sur-état pour lequel on veut préserver la connaissance du dernier sous-état atteint.

C.3 Diagramme d'Objets

Les Diagrammes d'Objets sont utilisés pour montrer l'existence des objets et de leurs relations dans le Modèle Logique du système. Ils représentent une photo instantanée dans le temps d'une séquence transitoire d'événements sur une certaine configuration d'objets ; c'est un scénario qui trace le comportement d'un système. On peut dire aussi qu'un Diagramme d'Objets représente les interactions ou les relations structurelles qui peuvent exister entre un ensemble donné d'objets : ils représentent une vue de la structure des objets du système. Les éléments essentiels d'un Diagramme d'Objets sont des objets et des relations.

Objet : c'est une chose dont on peut faire quelque chose. Les objets ont un état, un comportement et une identité. La structure et le comportement des objets similaires doivent être définis dans leur classe commune. Les termes instance et objet sont équivalents. La figure C.11.a présente la notation employée pour représenter les objets dans la méthode. La représentations des nom des objets suit la syntaxe des attributs (cf. figure C.11.b).

Liaisons : c'est la représentations des relations entre objets ; elles sont des instances des associations de la même façon que les objets sont des instances de classes. Les liaisons sont permises seulement entre objets dont les classes sont liées par une association. La notation employée pour représenter les liaisons est présentée dans la figure C.11.c.

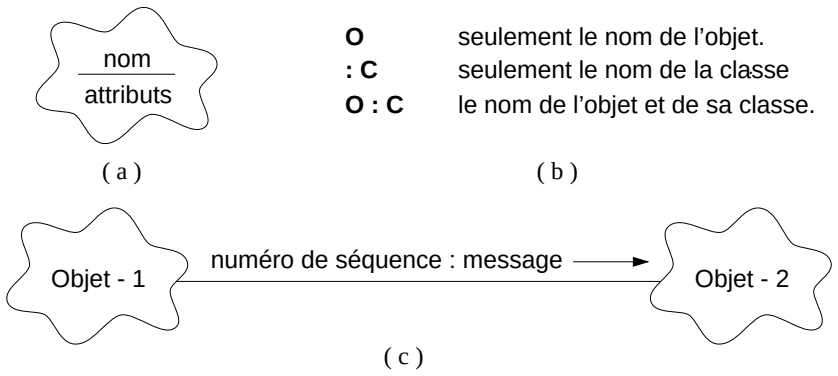


Figure C.11 - OOD - Objets et Liaisons

Message : il montre une opération qu'un objet réalise sur un autre ; c'est la manière de représenter la demande d'exécution d'opération qu'un objet fait à un autre. L'objet qui appelle une opération d'un autre objet est l'`Objet Client` et l'objet qui exécute l'opération est l'`Objet Fournisseur`. Généralement le client connaît le fournisseur mais le fournisseur ne connaît pas le client. Les messages sont des décorations de liaisons et comportent les éléments suivants :

- `D` : un symbole de synchronisation qui montre la direction de l'invocation ;
- `M` : l'invocation d'une opération ou d'un événement envoyé ;
- `S` : un numéro de séquence optionnel.

Une flèche est utilisée comme symbole de synchronisation pour montrer la direction du type le plus simple de message : les `messages séquentiels`. Les formes plus avancées de synchronisation sont abordées dans la description des concepts avancés.

Les appels d'opérations sont le type de messages le plus commun ; la représentation des opérations comme messages suit la syntaxe des opérations, il faut donc les écrire de la façon suivante :

- `N()` : seulement le nom de l'opération ;
- `R N (Arguments)` : la classe retournée, le nom de l'opération et ses arguments actuels.

Les concepts avancés des Diagrammes d'objets sont listés ci-dessous.

Rôle, Clé et Contrainte : ils ont, pour les liaisons, la même fonction que pour les associations et sont représentés de la même façon (cf. figure C.12).

Flux de Données : il montre le flux des données entre les objets, car les données peuvent aller en sens contraire de la direction d'un message. Un `Message de Retour` est représenté avec le symbole de synchronisation avec un cercle au début, comme on peut voir à la figure C.12.

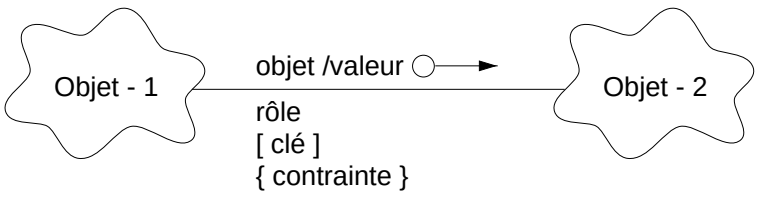


Figure C.12 - OOD - Rôle, Clé et Contrainte

Visibilité : c'est un concept qui sert à montrer comment un objet est vu par les autres dans un Diagramme d'objets. La Méthode présente la visibilité avec des lettres dans des carrées qui doivent être placées à côté de l'objet fournisseur, et qui servent à montrer si l'identité d'objet est partagée (boîte vide) ou non (boîte pleine). Les lettres qui sont placées dans le carré ainsi que leur signification sont données ci-dessous.

- *G* : l'objet fournisseur est global pour le client : il est complètement visible par le client ;
- *P* : l'objet fournisseur est un paramètre pour certaines opérations du Client ;
- *F* : l'objet fournisseur est une partie du Client : il est un champ (attribut) du client ;
- *L* : l'objet fournisseur est un objet déclaré localement dans la portée du Diagramme d'Objets.

Objet Actif : c'est un objet autonome, qui peut avoir un comportement qui ne dépend pas des messages - il a sa propre séquence de contrôle ce que lui donne le rôle de `racine de contrôle` dans une séquence de contrôle. Par contre, un **Objet Passif** change d'état seulement quand il reçoit un message.

L'aspect concurrence entre objets est introduit par les objets actifs et passifs ; les autres aspects sont exprimés avec les spécifications textuelles des classes. Les objets peuvent être dans un état qui peut être séquentiel, gardé ou synchrone par rapport à la concurrence. On note ces quatre aspects en écrivant les mots actif, séquentiel, gardé ou synchrone dans le symbole d'objet en bas, du côté gauche. Par défaut les objets où rien n'est écrit sont des "objets séquentiels".

Synchronisation : elle montre les aspects temporels des objets. On a déjà présenté le message séquentiel. Cependant il existe d'autres formes de synchronisation. Ces autres formes de Messages sont :

- *Message Synchrone* : l'objet client doit attendre que l'objet fournisseur accepte le message ;
- *Message "Balking"* : l'objet client doit abandonner le message si l'objet fournisseur n'y répond pas immédiatement ;
- *Message "Timeout"* : l'objet client doit abandonner le message si l'objet fournisseur n'y répond pas dans un intervalle de temps déterminé ;
- *Message Asynchrone* : l'objet client envoie le message et n'attend pas la réponse de l'objet fournisseur pour continuer.

Les notations employées pour représenter les divers types de synchronisation sont données dans la figure C.13.

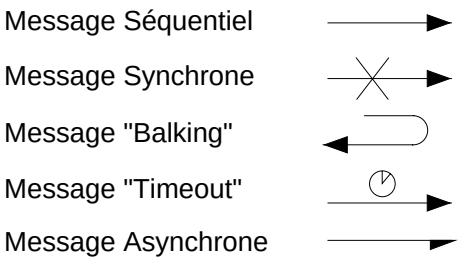


Figure C.13 - OOD - Synchronisation

Budget Temporel : il est utilisé pour montrer la séquence temporelle des messages, c'est-à-dire, combien de temps après l'activation d'un objet actif chaque message sera envoyé. On note cela en utilisant un nombre (le nombre de secondes par rapport à l'activation de l'objet actif, après lequel chaque message sera envoyé) précédé du signe (+), suivi de deux-points (:) et du nom du message. Un exemple peut être : +5 : message - qui veut dire que le message sera envoyé 5 secondes après l'activation de l'objet actif.

Spécification : c'est la spécification du contexte d'un Diagramme d'Objets ("catégorie" signifie Catégorie de Classes) :

Contexte : global | catégorie | classe | opération

C.4 Diagramme d'Interaction

Le Diagramme d'Interaction est utilisé de la même façon que les Diagrammes d'Objets pour tracer l'exécution d'un scénario. Un Diagramme d'Interaction est une autre façon de représenter un Diagramme d'objets.

L'avantage des Diagrammes d'Interaction réside dans leur capacité à visualiser le passage de messages dans un ordre donné, alors que celui des Diagrammes d'Objets réside dans la richesse de ses détails.

Les Diagrammes d'Interaction de Booch sont des généralisations du Traceur d'Événements de Rumbaugh [RBP+91] et du Diagramme d'Interaction de Jacobson [JCJO92].

Les Diagrammes d'Interaction n'introduisent aucun symbole ou concept nouveau ; ils font simplement une restructuration des Diagrammes d'Objets et utilisent la même syntaxe pour la représentation des objets et des messages ainsi que pour les symboles de synchronisation.

La figure C.14 montre un Diagramme d'Interaction. Les objets sont placés au-dessus des lignes pointillées verticales. Les messages, qui peuvent dénoter des événements ou l'invocation d'une opération, sont représentés par les flèches entre ces lignes lesquels vont de l'objet client vers l'objet fournisseur. L'ordonnancement est représenté par la position verticale de la flèche et l'écriture de messages par ordre chronologique de haut en bas, c'est-à-dire, le premier message est celui qui est au plus haut et le dernier celui qui est au plus bas.

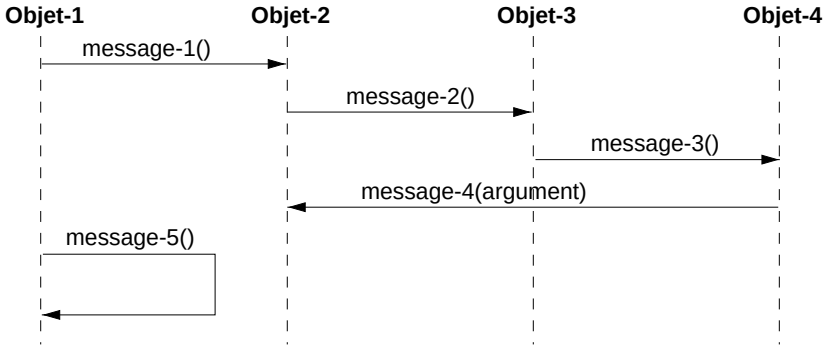


Figure C.14 - OOD - Diagramme d'Interaction

Bien que les Diagrammes d'Interaction soient très simples, deux concepts avancés peuvent aider à représenter quelques caractéristiques de modes d'interactions complexes. Ces concepts, présentés dans les Diagrammes d'Interaction Avancés, sont décrits ci-dessous.

Routeur : il est utilisé pour montrer la raison par laquelle le message a été invoque. Chaque pas du routeur doit être écrit comme du texte en français structuré ou libre, aligné sur le message correspondant et du côté gauche du Diagramme d'Interaction (cf. figure C.15).

Centralisation du Contrôle : elle est utilisée pour exprimer l'intervalle de temps pendant lequel l'objet est soumis à un flux du contrôle. Cet intervalle est représenté par un rectangle sur la ligne pointillée de l'objet (cf. figure C.15).

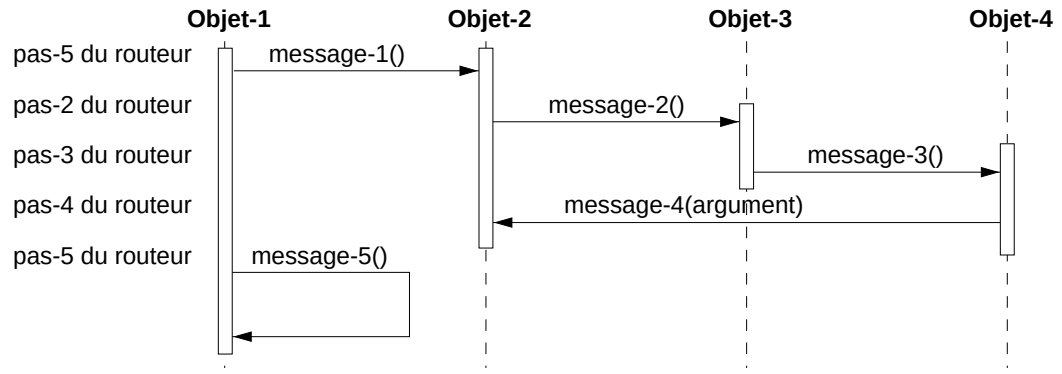


Figure C.15 - OOD - Diagramme Avancé d'Interaction

C.5 Diagramme de Modules

Les Diagrammes de Modules sont utilisés pour montrer l'allocation des classes et des objets dans des modules du Projet Physique du Système. Un Diagramme de Modules simples exprime une vue de la structure modulaire d'un système. Les éléments essentiels d'un Diagramme de Modules sont les modules et les dépendances.

Module : c'est une unité de code qui sert de bloc de construction pour le Diagramme de Modules. Les modules peuvent ne pas être utiles si le langage de programmation choisi n'inclut pas ce concept, comme par exemple le langage ADA. Il existe trois types de modules :

– `Module Programme Principal` : il dénote un fichier qui contient la racine d'un programme ; en C++, par exemple, c'est un fichier "*.cpp" qui contient la fonction `main` ; il y a exactement un module de ce type par programme (figure C.16.a) ;

– `Module Spécification` : il dénote des fichiers qui contiennent la déclaration des entités d'un système ; en C++, par exemple, ce sont des fichiers "*.h" (figure C.16.b) ;

– `Module Corps` : il dénote des fichiers qui contiennent les définitions des entités d'un système ; en C++, par exemple, ce sont des fichiers "*.cpp" (figure C.16.c).

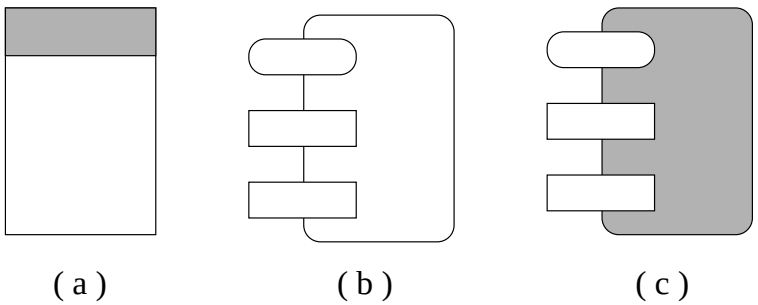


Figure C.16 - OOD - Modules

Le nom du module est généralement le nom du fichier qui contient la déclaration ou la définition des classes, des objets et d'autres détails du langage.

Dépendance : c'est une relation entre modules qui dénote une dépendance de compilation ; en C++, par exemple, les dépendances de compilation sont créées par les directives "include". Généralement il n'y a pas de boucle dans un ensemble de dépendances de compilation. La notation utilisée est une flèche qui part du module "qui dépend" de l'autre.

Sous-Système : c'est un concept utilisé pour partager le Modèle Physique d'un système. Un sous-système est un agrégat qui contient des modules et d'autres sous-systèmes. La figure C.17 présente la notation employée pour représenter les sous-systèmes dans la méthode.

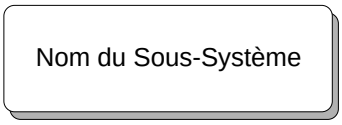


Figure C.17 - OOD - Sous-Systèmes

Dans les sous-systèmes, le contrôle d'accès à chaque module (la présentation d'un module avec un accès public, privé, etc.) est réalisé avec les mêmes concepts et les mêmes notations que pour les classes.

Il peut exister des dépendances entre sous-systèmes et entre modules et sous-systèmes, présentées avec la même flèche que celle utilisée par les dépendances entre modules.

Pour les Diagrammes d'Interaction, les concepts avancés sont utilisés pour montrer quelques caractéristiques particulières de certains langages ; par exemple, pour montrer d'autres types de modules propres à certains langages et pour montrer les aspects de segmentation de modules quand existent des problèmes de mémoire. Les Diagrammes Avancés d'Interaction sont donc créés par la personne qui implémente le système pour représenter des caractéristiques particulières de son implantation par rapport au langage utilisé.

C.6 Diagramme de Processus

Les Diagrammes de Processus sont utilisés pour montrer l'allocation d'un processus à un processeur dans le Projet Physique du Système. Un Diagramme de Processus simple représente une vue de la structure des processus d'un système. Les éléments essentiels d'un Diagramme de Processus sont les processeurs, les dispositifs et les connexions.

Processeur : c'est une partie du matériel capable d'exécuter des programmes. On peut ajouter à un processeur les processus qu'il réalise. Ces processus dénotent la racine d'un programme principal (d'un Diagramme de Modules) ou le nom d'un objet actif (d'un Diagramme d'Objets). La figure C.18.a donne la notation employée pour représenter les processeurs.

Dispositif : c'est une partie du matériel qui n'a pas la capacité d'exécuter des programmes (figure C.18.b).

Connexion : elle représente une liaison physique entre un processeur et un dispositif et est représentée par une ligne entre le processeur et le dispositif avec le nom de la connexion au dessus de la ligne.

Les concepts avancés des Diagrammes de Processus sont les suivants :

Emboîtement : il est utilisé pour représenter les configurations complexes de certains matériels ; il est obtenu en emboîtant des Diagrammes de Processus ; il peut exister des connexions entre groupes emboîtés.

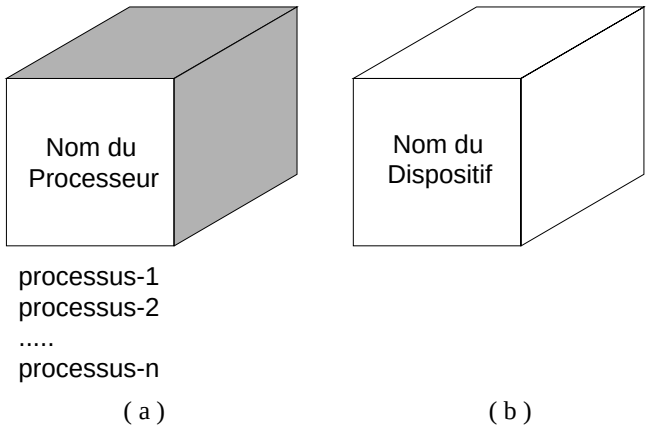


Figure C.18 - OOD - Diagramme de Processus

Plan de Processus : il est utilisé pour planifier l'exécution des processus par un processeur. Les cinq types d'approches de base de la planification sont : `préemptive` (système de priorités), `nonpréemptive`, `cyclique`, `exécutif` (réglé par un algorithme) et `manuel`. On utilise ces noms pour représenter quel type de Plan de Processus un processeur utilise.

C.7 La Procédure de Développement

La Méthode propose de modéliser le développement d'un système à travers des Micro et des Macro Procédures.

Micro Procédure : c'est une procédure guidée par le flux de scénarios et les produits architecturaux qui émergent de la Macro Procédure. Elle représente les activités journalières de l'équipe de développement et est composée de quatre étapes :

1. *Identifier les classes et les objets* : il faut chercher les "abstractions clés" dans l'espace du problème et dans le comportement des objets. Ces "abstractions clés" sont trouvées en étudiant la terminologie du domaine du problème avec les experts du domaine. Elles doivent être placées dans un dictionnaire de données ;
2. *Identifier les Sémantiques* : il faut établir la signification des classes et des objets trouvés antérieurement, avec leurs attributs et leur comportement ; il faut aussi rechercher comment un objet en utilise un autre et enregistrer ces informations dans le dictionnaire de données ;

3. *Identifier les relations* : il faut étudier les relations entre les classes et les objets et déterminer le processus d'interaction à travers les différents types d'associations ; la sémantique statique et dynamique entre les objets est aussi définie, de la même façon que la visibilité entre les objets ;
4. *Implanter les classes et les objets* : il faut étudier les classes et les objets pour trouver une manière de les implémenter ; cette décision est prise par rapport au langage choisi pour réaliser le système. Les classes et les objets doivent être structurés en modules.

Macro Procédure : c'est une procédure qui sert à contrôler le cadre de la Micro Procédure et comporte des produits et des activités qui permettent une diminution des problèmes de développement de manière à ce que les corrections dans la Micro Procédure soient faites le plus tôt possible. La Macro Procédure comporte cinq étapes :

1. *Conceptualisation* : on cherche à déterminer les besoins centraux pour le système et à produire des prototypes par rapport à ces besoins ;
2. *Analyse* : sa finalité est la création d'un modèle du comportement du système avec des scénarios qui soient fondamentaux pour l'application ; ces scénarios doivent être validés par les experts du domaine du problème, par les utilisateurs, par les analystes et par les architectes ;
3. *Projet* : on cherche à créer une architecture pour le développement de l'implantation, à établir un plan des révisions futures ainsi que l'établissement des règles de gestion qui puissent être utilisées par différents éléments du système (gestion de mémoire, gestion de stockage, etc.). L'architecture créée doit être validée par rapport au prototype et par rapport au plan de révisions futures ;
4. *Évolution* : on augmente et on transforme l'implantation par des raffinements successifs, jusqu'à la sortie du produit ;
5. *Maintenance* : on réalise la maintenance du produit ; cette étape est une continuation de la phase antérieure sauf s'il faut créer quelques caractéristiques nouvelles pour le produit.

Annexe D

La Méthode OOAD

Dans cette annexe, on présente les concepts les plus significatifs des quatre niveaux de la méthode OOAD proposée par J. Martin et J. J. Odell [MO95].

D.1 Niveau Structurel de Base - Structure de l'Objet

Objet Type (Concept) : c'est une notion applicable aux choses ou aux objets que l'on connaît. Les concepts peuvent être très variés ; ils s'appliquent à des choses qui peuvent être touchées, dégustées, senties, entendues ou vues.

L'emploi d'un concept consiste à définir des critères pour déterminer si ce concept s'applique ou non aux "choses" considérées. Si un objet répond à un critère relatif à un concept, il est une instance de ce concept. Il existe des concepts qui n'ont pas d'instances, car aucun objet ne satisfait les critères définis.

Dans la méthode, un concept peut être représenté de trois façons différentes : par un Symbole (cf. figure D.1.a), par son Intention, définition du concept et des critères d'appartenance d'un objet à ce concept (cf. figure D.1.b) ou par son extension, ensemble de tous les objets qui répondent aux critères du concept (cf. figure D.1.c). Dans la figure D.1, comme dans la majeure partie de la méthode, le terme objet type est privilégié par rapport au terme concept. Les concepts peuvent ne pas avoir un nom, ni une définition ; ils peuvent avoir des synonymes et des homonymes.

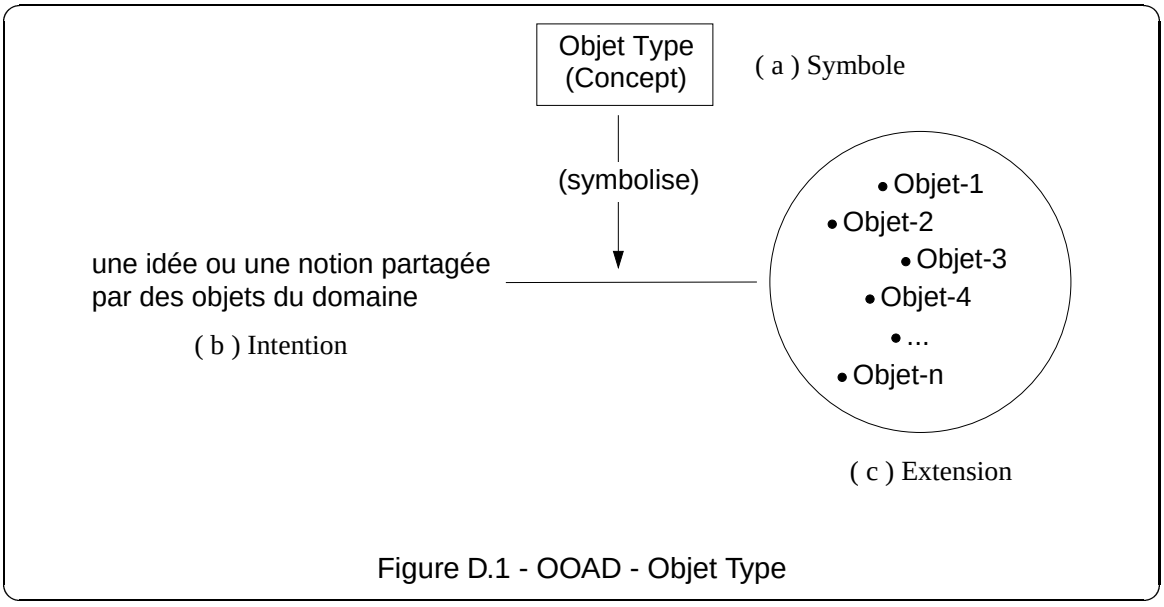


Figure D.1 - OOAD - Objet Type

Domaine : c'est une collection pertinente des instances de la Spécification du Domaine (collection des concepts applicables au Domaine). Le Domaine d'un système est le monde réel que l'analyste analyse. La Spécification du Domaine est le produit de cette analyse.

La méthode présente plusieurs techniques pour représenter la Spécification du Domaine, telles que les Diagrammes d'Objets, les Diagrammes d'événements, les Diagrammes de Transitions d'États et les Diagrammes de Flux de Données. La Spécification du Domaine définit l'ensemble des concepts à travers une spécification structurelle et une spécification comportementale pour un Domaine particulier.

Objet (Instance) : c'est quelque chose sur quoi on peut appliquer un concept : c'est une instance d'un concept. Les termes objet et instance sont interchangeables dans la méthode.

L'Instanciation est la création d'un objet à partir d'un objet type. La méthode note l'instanciation avec une ligne pointillée fléchée selon la figure D.2.

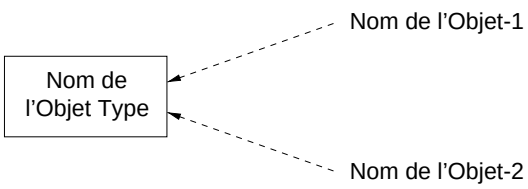


Figure D.2 - OOAD - Instanciation

Tous les concepts de la méthode sont, par définition, des instances d'un concept appelé *Concept*. Un concept peut donc être instancié par des objets qui sont eux-mêmes des concepts ; tous les concepts sont donc des objets, mais tous les objets ne sont pas des concepts.

Les objets ont un cycle de vie : ils existent depuis la première fois où ils ont été acceptés pour leur appartenance à un concept et ils cessent d'exister quand ils n'appartiennent plus à aucun concept. Entre sa " vie" et sa "mort", un objet passe par plusieurs états ; cet ensemble d'états correspond dans la méthode au cycle de vie d'un objet.

Dans la phase d'analyse, où les contraintes imposées par les langages de programmation n'existent pas, la méthode introduit l'appartenance multiple d'un objet à des ensembles : un objet peut, à tout instant, faire partie de plusieurs ensembles d'objets. La *Classification* définit l'appartenance d'un objet à un ensemble ; elle définit une liaison entre un objet type et ses instances.

Schéma Relationnel (Relation) [1] : c'est un objet type dont l'extension est un ensemble de tuples (un objet composé invariable), c'est-à-dire un ensemble d'objets obtenu par produit cartésien d'ensembles d'objets. La figure D.3 présente la notation la plus simple utilisée pour représenter les schémas relationnels. La notation utilisée pour représenter le schéma relationnel comme un objet type est donnée à la figure D.5.

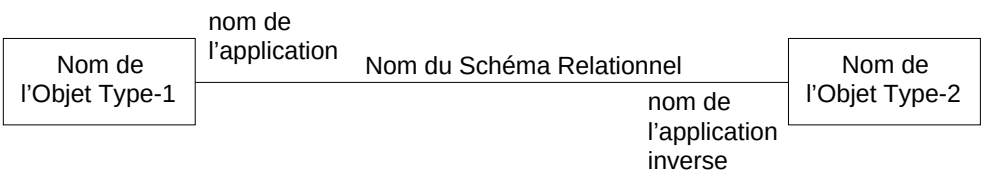


Figure D.3 - OOAD - Schéma Relationnel et Application

I-Relation [2] : c'est une instance d'un schéma relationnel, une configuration d'objets, lesquels sont les composants invariables d'un objet composé : le schéma relationnel. Chaque objet composant de la i-relation est une instance d'un schéma relationnel spécifique.

[1] "Relationship Type" - "Relation" dans la méthode
[2] "Relationship" dans la méthode

Application [3] : c'est la manière utilisée par la méthode pour traiter les i-relations en regardant leurs parties composantes : en partant "d'un côté" d'une i-relation, on doit arriver à "l'autre côté" si on utilise une application. On dit aussi qu'une application affecte des objets d'un type à des objets d'un autre type.

Dans le monde des objets, une application est un processus pour lequel, étant donné un objet, on obtient un autre objet (application monovaluée) ou un ensemble d'objets (application multivaluée). Mathématiquement, une application peut être considérée comme une fonction entre deux ensembles. À la figure D.3 on peut voir la notation employée pour représenter les applications.

La méthode permet aussi d'ajouter des contraintes à une application. Pour restreindre le nombre d'objets "obtenus" à partir d'une application, on utilise les *Contraintes de Cardinalité*. Les symboles utilisés pour représenter ces contraintes sont présentés à la figure D.4 et doivent être placés du côté de l'objet sur lequel pointe une application (objet cible de l'application).

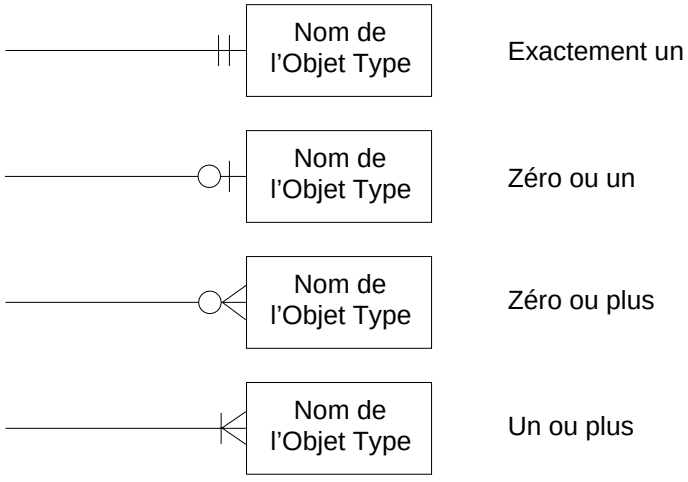


Figure D.4 - OOAD - Contraintes de Cardinalité

Une contrainte de cardinalité peut ne pas être représentée comme une ligne simple, mais explicitement comme un objet type. Pour utiliser cette représentation, il faut faire figurer les contraintes de cardinalité initiales dans la nouvelle représentation. La figure D.5 présente la notation utilisée pour représenter un schéma relationnel comme un objet type. Dans cette figure, le schéma relationnel et les contraintes de cardinalité initiales sont représentés par des lignes pointillées ; "`[p]`" sert à indiquer les

[3] "Mappings" dans la méthode - un mapping assigne un objet d'un type à un objet ou à un ensemble d'objets d'un autre type.

i-relations originales du schéma relationnel et Application-1 fait référence aux nouvelles applications entre les objets type originaux et l'objet type qui représente le schéma relationnel (représenté avec un losange).

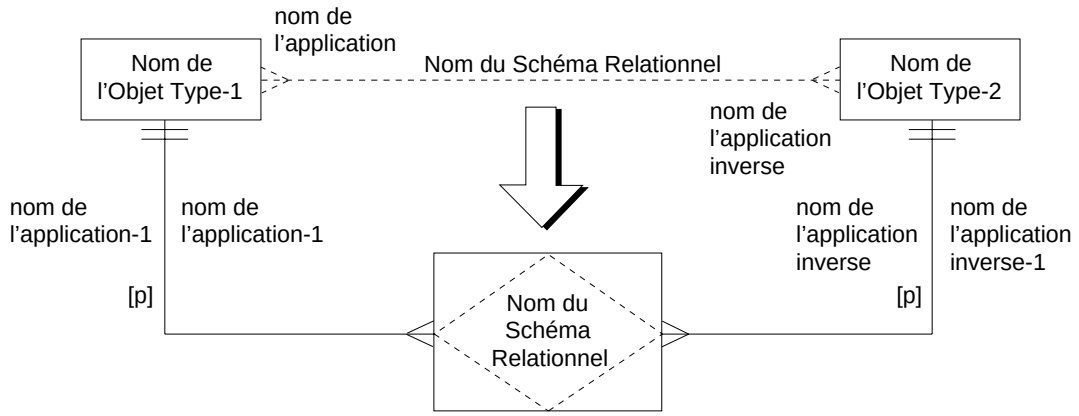


Figure D.5 - OOAD - Schéma Relationnel représenté comme un Objet Type

La méthode induit la représentation d'un Schéma Relationnel par un objet type lorsque ce schéma relationnel a des propriétés qui peuvent le caractériser. Ces propriétés sont représentées comme un objet type attaché à la représentation du Schéma Relationnel. Cette représentation est celle de la figure D.6 sans ajout, ni des applications, ni des contraintes de cardinalité.

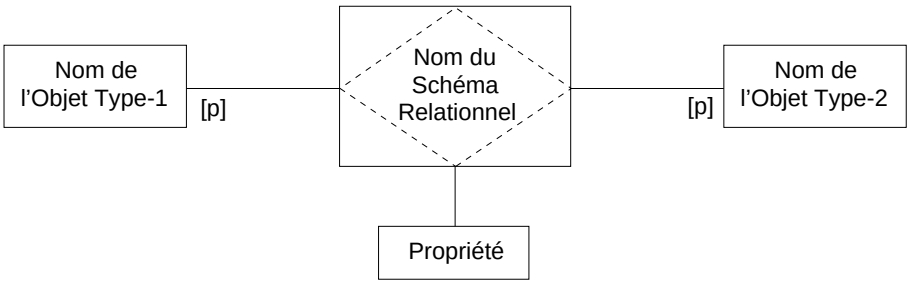


Figure D.6 - OOAD - Propriété d'un Schéma Relationnel

La méthode différencie des types d'applications. Les *Applications de Base* sont le résultat d'une assertion (expression qui peut être ou ne pas être satisfaite) alors que les *Applications Dérivées* sont le résultat d'un calcul ou d'une inférence. Il y a aussi les *Applications Hybrides* qui peuvent être à la fois de base et dérivées. La figure D.7 présente les notations employées pour représenter ces applications.

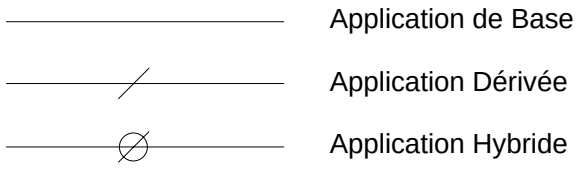


Figure D.7 - OOAD - Applications

Propriété Type : c'est un terme utilisé pour faire référence à toutes les applications d'un objet type sur d'autres objets types ; c'est un type d'association qui lie un ensemble d'objets avec un autre ensemble. Dans les langages orientés objet, une propriété type associe des objets d'une Classe avec des valeurs ou avec des pointeurs sur des objets d'une autre classe.

Les instances d'une propriété type sont les propriétés définies comme association identifiable qui lie un objet à un autre objet ou à un ensemble d'objets.

Association : c'est une manière de lier des objets types d'une façon significative. Dans la méthode, une association est représentée soit par une i-relation, soit par une application. Les i-relations peuvent être utilisées lorsque l'analyste a besoin de considérer une association comme un objet. Les applications au contraire peuvent être utilisées lorsque l'analyste connaît un objet et veut le lier à d'autres objets inconnus.

Généralisation : c'est la mise en évidence d'un objet type plus général qu'un autre, c'est-à-dire l'identification d'un objet type qui inclut ou englobe un autre objet type. La généralisation spécifie un sur-type.

Un sur-type est un objet type généralisé, c'est-à-dire un objet type dont l'ensemble inclut tous les membres d'un ou de plusieurs autres ensembles (définition en extension) ; c'est un objet type dont la définition est plus générale que celle d'un autre (définition en Intention).

La *Spécialisation* est la mise en évidence d'un objet type totalement inclus dans un autre. La spécialisation produit des sous-types.

Un sous-type est un objet type spécialisé, c'est-à-dire un objet type dont les membres sont tous inclus dans un ensemble plus englobant (définition en extension) ; c'est un objet type dont la définition est plus spécialisée qu'une autre (définition en Intention).

Toutes les propriétés applicables à un sur-type sont applicables aussi aux sous-types associés, bien que les sous-types aient des caractéristiques additionnelles.

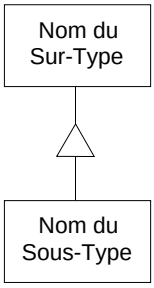


Figure D.8 - OOAD - Hiérarchies "Simples" d'Objets Types

Généralisation et spécialisation sont des concepts utiles pour comprendre le placement d'un objet type dans une hiérarchie d'objets type. La méthode présente deux manières pour représenter les hiérarchies d'objets type. Si la hiérarchie est simple, la méthode suggère d'utiliser la représentation donnée à la figure D.8.

Lorsqu'une hiérarchie d'objets type se présente de façon complexe, avec la nécessité d'une distinction précise entre les sous-types d'un objet type, la méthode introduit le concept de **Partition Type** pour représenter cette hiérarchie. Une partition type est une division, ou partition, d'un objet type en sous-types disjoints.

Entre partitions type, les sous-types peuvent se recouvrir, mais dans une partition type les sous-types doivent être exclusifs et non recouverts. De plus, on peut ajouter une contrainte pour indiquer qu'un sous-type est invariable : lorsqu'un objet est classifié comme étant d'un sous-type invariable, il ne peut plus en changer pendant toute sa vie. Cette contrainte est représentée avec la notation "`[i]`" présente dans la boîte du sous-type (cf. sous-type-1, figure D.9). Les partitions type peuvent être divisées en *Partition Type Complète*, où tous les sous-types sont spécifiés et en *Partition Type Incomplète*, comprenant une liste partielle de sous-types (cf figure D.9).

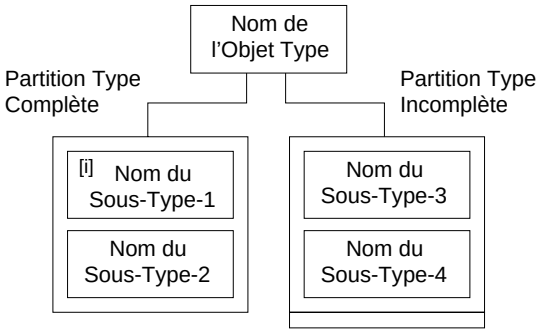


Figure D.9 - OOAD - Partitions Types

La définition d'un sous-type en extension exprime le fait qu'un sous-type est un ensemble dont les membres sont tous inclus dans un ensemble plus englobant. Les i-relations sont aussi des ensembles d'objets, les tuples. Donc elles peuvent aussi être sous-typées. La figure D.10.a présente la notation employée pour représenter une i-relation sous-type (celle du dessous).

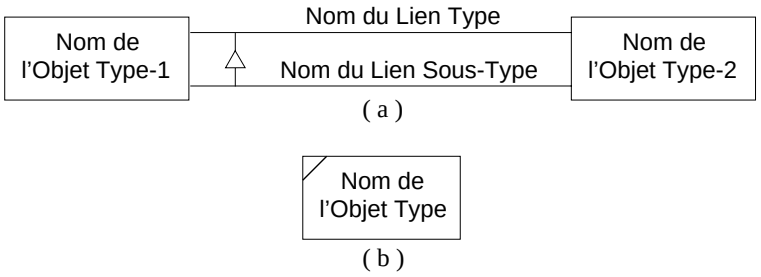


Figure D.10 - OOAD - Sous-type d'une I-Relation et Objet Type Dérivé

Après avoir introduit la généralisation/spécialisation, la méthode introduit le concept d'objet type de base ou dérivé, dans le même sens que celui employé pour les applications. Un objet type de base est un objet type dont les instances sont déclarées. Un objet type dérivé est un objet type dont les instances peuvent être calculées ou inférées. La notation employée pour représenter les objets types dérivés est présentée à la figure D.10.b.

Composition (Agrégation) : c'est la représentation d'un objet composé d'autres objets. La composition doit être utilisée lorsqu'il faut s'adresser à l'objet complet et non à ses composants.

La méthode introduit le concept de *Composition Invariable,* où les composants ne peuvent être ni changés, ni enlevés sans détruire le composé, et de *Composition Modifiable,* où un composant peut être changé ou enlevé sans la destruction du composé. La figure D.11 présente la notation employée pour représenter la Composition. La composition invariable est notée avec "`[i]`" à côté du symbole de la composition.

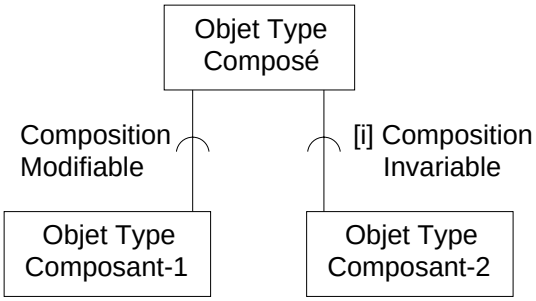


Figure D.11 - OOAD - Composition

La classification, la généralisation et la composition sont des concepts utilisés par la méthode pour gérer la complexité d'un système. La figure D.12 montre les différences entre ces trois concepts. La classification est une liaison entre des objets types et des objets (cf. figure D.12.a), la généralisation est une liaison entre des objets types (cf. figure D.12.b) et la composition est une liaison entre des objets (cf. figure D.12.c).

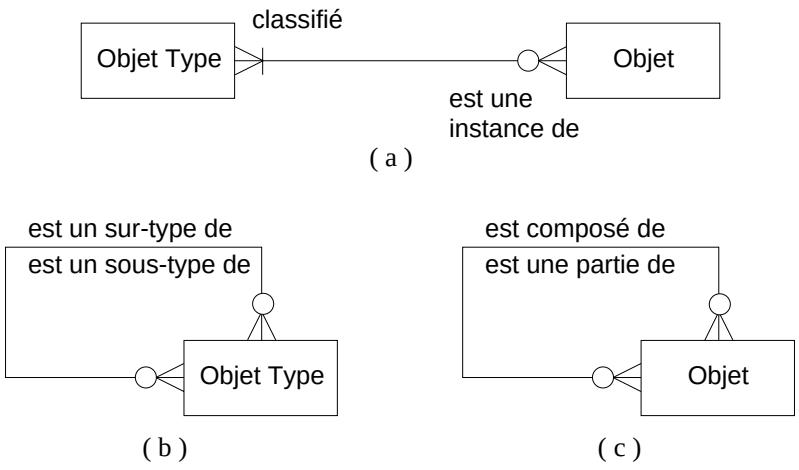


Figure D.12 - OOAD - Gestion de la Complexité

État : c'est la collection des associations qu'un objet a avec d'autres objets et avec des objets type.

L'état complet d'un objet est composé de toutes les associations dont l'objet a fait partie pendant son existence. Il peut être vu comme un historique (une trace). Cependant, en pratique, les seuls types d'états réellement intéressants sont des états Ponctuels, survenant à un instant particulier du temps et des états Périodiques, dont la durée est mesurable.

D.2 Niveau Structurel de Base - Comportement de l'Objet

Changement d'État : c'est pour un objet une transition remarquable d'un état vers un autre. Les changements d'états dans la méthode sont représentés avec un Diagramme à Lignes de Transitions d'états (cf. figure D.13). Dans cette figure, les lignes horizontales représentent les états, alors que les lignes verticales fléchées représentent les changements d'états permis à un objet.

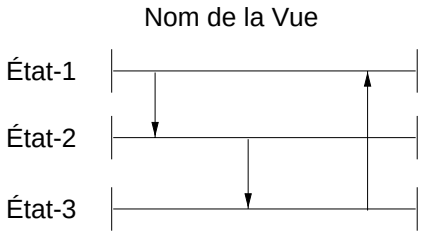


Figure D.13 - OOAD - Changements d'États

Dans un Diagramme à Lignes de Transitions d'États, un objet se trouve, par ligne, dans un seul état, qui le représente à un instant quelconque du temps. Cette caractéristique permet une étude concentrée sur une vue (un aspect particulier) du cycle de vie de l'objet. Un Diagramme à Lignes de Transitions d'États peut présenter aussi un ensemble de vues, mais peut dans ce cas, être très complexe.

En fait, un changement d'état montre un ou plusieurs changements d'associations dont l'objet fait partie. On peut donc dire qu'un changement d'état est un changement des associations d'un objet.

Événement : c'est un changement dans l'état d'un objet. La méthode présente six types d'événements simples, tous basés sur l'addition ou la suppression d'un objet. Ces événements sont :

- *événement création* : c'est un événement associé à l'apparition d'un objet nouveau, considéré comme une instance d'un objet type quelconque ;
- *événement terminaison* : c'est un événement associé à la disparition d'un objet existant. Cette disparition peut être relative à un objet type particulier ou à tous les objets type dont l'objet est une instance ;

- *événement classification* : c'est un événement associé à la classification d'un objet existant. Un objet devient membre d'un ensemble auquel il n'appartenait pas, il devient instance d'un objet type ;
- *événement déclassification* : c'est un événement associé à la déclassification d'un objet existant. Un objet cesse d'être membre d'un ensemble particulier ;
- *événement connexion* : c'est un événement associé à l'addition d'une i-relation nouvelle entre deux objets existants. On peut dire que les événements du type connexion sont des événements du type création pour les i-relations, car la i-relation est elle-même un objet - un tuple ;
- *événement déconnexion* : c'est un événement associé à la suppression d'une i-relation existante entre deux objets. De la même façon que pour l'événement Connexion, l'événement déconnexion est un événement Terminaison associé à une i-relation.

Il existe aussi des événements décrits par plus d'une addition ou d'une suppression ; ce sont les événements composés :

- *événement reclassification* : c'est un événement associé à une déclassification d'un objet par rapport à un objet type, suivie d'une classification du même objet par rapport à un autre objet type ;
- *événement reconnexion* : c'est un événement associé à une suppression d'un objet et à la création d'un nouvel objet. L'objet supprimé et l'objet créé doivent être des instances du même objet type. Ce type d'événement est très utile pour traiter des applications avec des contraintes de cardinalité minimale et maximale égale à un ;
- *événement terminaison de composé* : c'est un événement associé à la suppression d'un objet composé et de tous ses objets composants en même temps ;
- *événement union* : c'est un événement associé au fait que des objets distincts deviennent un objet unique. Le contraire est un *événement séparation*.

A chacun des événements cités ci-dessus, est associé un pré-état de l'événement, état où l'objet doit être avant que l'événement se produise, et un post-état de l'événement, état où l'objet doit être après l'application de l'événement.

Les événements ont pour cause la terminaison d'une opération quelconque. Si l'opération appartient au domaine de l'analyste, c'est un événement interne ou événement.

Si l'opération est hors du domaine de l'analyste, c'est un événement externe. Si l'événement est le résultat d'une opération en relation avec le temps, c'est un *Événement Temporel* (il peut être interne ou externe au système).

Événement Type : c'est un type ou une classification d'événements. Un événement type peut avoir plusieurs instances car c'est un objet type.

De la même façon que pour les objets types, on peut parler de Partition pour les événements types. Un événement type peut avoir des sous-types et des sur-types ; il peut être dérivé. La figure D.14 présente la notation employée pour représenter la Partition des événements types (cf. figure D.14.a) et les événements types dérivés (cf. figure D.14.b).

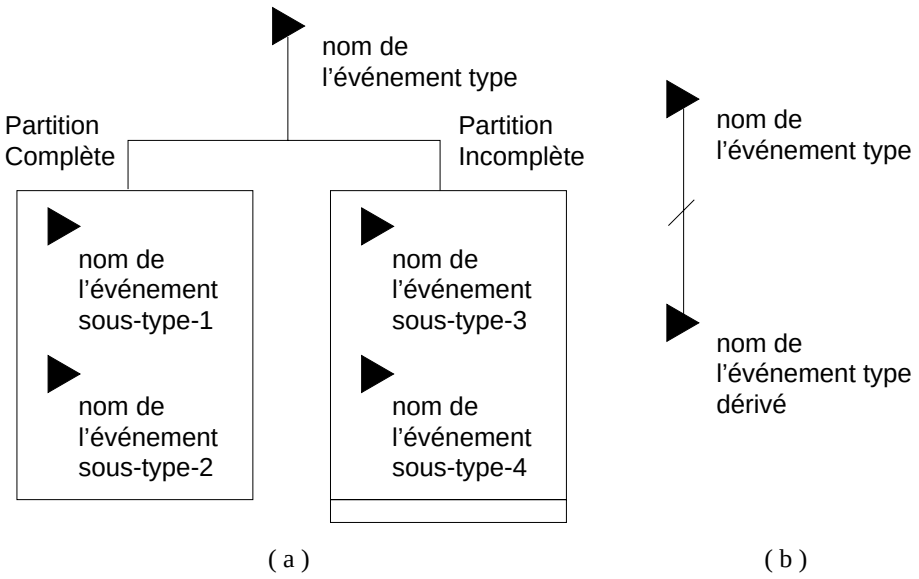


Figure D.14 - OOAD - Événement Type

Opération : c'est un processus qui peut être requis comme une unité et qui exécute, pas à pas , deux fonctions de base : l'interrogation d'un objet ou le changement de l'état d'un objet.

Les opérations sont utilisées lorsque l'analyste veut analyser un système du point de vue des mécanismes de changement, contrairement aux événements types utilisés pour analyser un système du point de vue des effets des changements d'états. Les changements d'états et les opérations doivent donc être définis ensembles.

Les opérations sont des objets types ayant pour instances des opérations invoquées. Chaque opération est définie dans le contexte d'un ou plusieurs objets, et toutes les opérations ont besoin des objets comme variables.

Les *Variables d'Entrée* d'une opération servent d'index pour organiser la connaissance que l'analyste a sur les opérations, pour savoir à quel objet type une opération est attachée. Les *Variables de Sortie* servent à spécifier quels objets types peuvent être produits comme résultat d'une opération.

De la même façon que pour les événements, une opération peut être interne, externe ou temporelle. La figure D.15 présente la notation employée pour représenter les opérations internes (cf. figure D.15.a), les opérations externes (cf. figure D.15.b), les opérations temporelles internes (cf. figure D.15.c) et les opérations temporelles externes (cf. figure D.15.d).

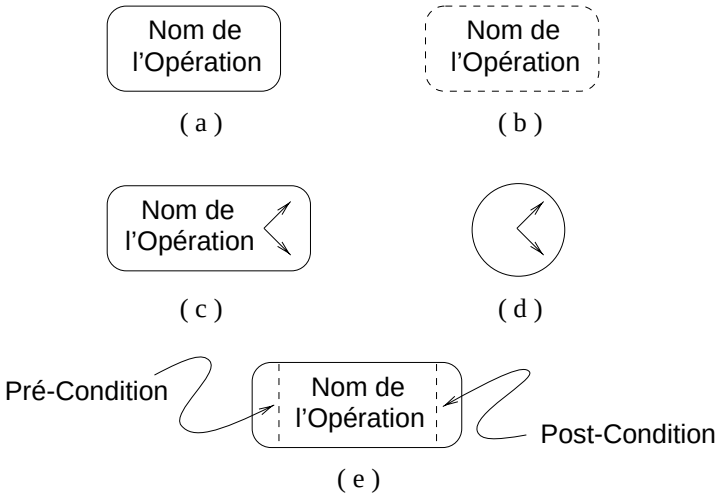


Figure D.15 - OOAD - Opérations

Pour assurer qu'une opération exécute ce qu'on lui demande, on peut lui spécifier des pré et des post conditions. Une pré condition spécifie les contraintes sous lesquelles l'opération s'exécutera correctement. Une post condition spécifie ce qui doit arriver lorsqu'une opération est accomplie. Les pré et post conditions assurent les pré-état et post-état d'un objet pour l'application d'une opération. La figure D.15.e présente la notation employée pour représenter les pré et post conditions.

Méthode : c'est une spécification de la manière dont une opération doit être réalisée. Cette spécification peut être implémentée, par exemple, avec du pseudo-code, ou par un Diagramme d'événements composé d'une série d'événements, de déclencheurs (définition ci-dessous) et de conditions de contrôle (définition ci-dessous). Une opération peut avoir plusieurs méthodes. Ce polymorphisme lui permet de supporter différents objets types, chacun avec sa propre méthode.

Pour réaliser sa tâche, une méthode d'une opération peut invoquer plusieurs opérations. Cela veut dire que pour qu'un événement se produise, d'autres événements peuvent aussi être nécessaires : le changement d'état d'un objet peut impliquer des changements d'états d'autres objets.

Les méthodes peuvent être spécifiées comme des modules contenant plusieurs composants. Les composants de plus bas niveau sont des opérations élémentaires pour lesquelles le traitement est limité simplement à la création, terminaison, classification, déclassification, reclassification et reconnexion d'objets.

Règle de Déclenchement : c'est une règle qui s'applique à un type d'événement et qui sert à spécifier le fait que, lorsque ce type d'événement se produit, une opération particulière doit être invoquée. Un *Déclencheur* définit le rapport entre un événement et le processus réactif invoqué ; il invoque une opération particulière lorsqu'un événement se produit.

Les règles de déclenchement ont trois composants : un événement type (la cause), une opération (l'effet) et une application. L'application prend les variables de sortie de l'opération antérieure à la règle et les applique comme des variables d'entrée (des objets requis) à l'opération qu'elle invoque. La figure D.16 présente ces trois composants.

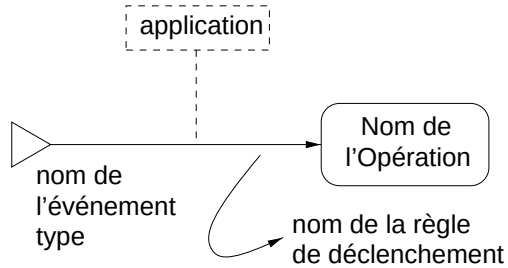


Figure D.16 - OOAD - Règle de Déclenchement

Les applications peuvent être divisées en *Applications Triviales* qui prennent l'objet qui sort comme variable de sortie d'une opération, et le transmettent comme variable d'entrée pour l'opération invoquée, ou comme *Applications Nommées* présentes dans des déclencheurs et qui doivent résoudre les différences entre l'objet qui sort d'une opération et l'objet requis par l'opération invoquée. La résolution de ce problème est faite par l'utilisation d'une application entre l'objet sortant et l'objet requis, définie dans les diagrammes de structure de l'objet. L'application nommée aura le nom de cette application définie dans la structure des objets.

Condition de Contrôle : c'est un mécanisme associé à une opération lui permettant de commencer seulement si certaines contraintes sont vérifiées.

Le but primaire d'une condition de contrôle est la synchronisation d'une série d'opérations. Une condition de contrôle peut être exprimée sous la forme "`IF ... THEN ...`" pour déterminer la véracité de la condition. Les arguments de la condition peuvent être simples, ou complexes sous la forme d'un ensemble de "`AND`" et "`OR`". Une opération peut avoir plusieurs conditions de contrôle.

Une condition de contrôle est représentée dans la méthode par un losange vide. Si une condition de contrôle est définie simplement par un "`AND`" de plusieurs déclencheurs, on peut employer la *Condition de Contrôle Et-Simple* dont la notation est un losange avec un "&" à l'intérieur ; si une condition de contrôle est employée pour indiquer que l'opération doit démarrer seulement lorsque tout le traitement propre à la condition est fini, on peut employer la *Condition de Contrôle Lorsque-Tout-Est-Prêt* dont la notation est un losange avec un "z" à l'intérieur.

D.3 Niveau Structurel Étendu

Dans cette section, on reprend les concepts de composition, contrainte et règle déjà présentés dans les niveaux antérieurs en les exploitant avec une nouvelle optique, celle du Niveau Structurel Étendu ; en plus de concepts propres à ce niveau sont introduits.

Composition (Agrégation) : c'est un mécanisme utilisé pour définir un "objet entier" en utilisant d'autres objets comme ses composants. Cela réduit la complexité d'un système en traitant plusieurs objets comme un objet unique.

Il y a plusieurs types de relations de composition déterminées par la combinaison de trois propriétés de base :

- *configuration* : quand un composant a une relation fonctionnelle particulière ou structurelle avec un autre composant ou avec l'objet qu'il constitue ;
- *homéomérique* : quand les composants sont de même type que le composé ;
- *invariance* : quand les composants ne peuvent pas être séparés du composé sans destruction de celui-ci.

Nous présentons ci-dessous, six types de composition prenant en compte ces trois propriétés.

- *composition objet composant objet composé* : c'est la composition la plus commune. C'est une relation de composition qui définit la configuration des composants par rapport au composé ; ex : les *scènes* composent un *film* ;
- *composition objet-matière* : c'est un type de relation de composition qui définit une configuration invariable des composants avec le composé où les composants ne peuvent pas être enlevés sans affecter le composé ; ce type de relation de composition définit la matière d'un objet ; ex : le *pain* est composé partiellement de *farine* ;
- *composition objet-partie* : c'est un type de relation de composition qui définit une configuration homéomérique des composants avec le composé où les composants sont de même type que le composé ; ex : une *tranche de pain* est une partie d'un *pain* (chaque tranche est similaire aux autres tranches et aussi au pain entier) ;
- *composition zone-endroit* : c'est un type de relation de composition qui définit une configuration homéomérique invariable, c'est-à-dire, une configuration homéomérique où un composant ne peut pas être enlevé sans détruire le composé ; ex : *San Francisco* est une partie de la *Californie* ;
- *composition membre-collection* : c'est un type de relation de composition qui implique qu'un composant n'a pas besoin d'avoir une relation fonctionnelle ou structurelle particulière avec d'autres composants ou avec le composé ; c'est une relation de composition qui définit une collection de parties d'un entier ; ex : un *arbre* est une partie d'une *forêt* ;
- *composition membre-association* : c'est un type de relation où les membres d'une association définissent une forme invariable d'une composition membre-collection, c'est-à-dire, une relation de composition qui définit une collection invariable de composants d'un composé ; ex : *Ginger* et *Fred* forment un couple de danse.

Contrainte : c'est une propriété qui doit toujours être vraie [Gra92] ; elle peut être classifiée soit comme structurelle, soit comme comportementale. Les contraintes structurelles limitent la manière avec laquelle un objet s'associe à un autre : elles restreignent l'état de l'objet. Ci-dessous sont présentés plusieurs types de contraintes structurelles (crochets présents après le nom de l'application) :

- *cardinalité* : on a déjà présenté les contraintes de cardinalité, zéro, un et plusieurs ; on peut aussi représenter d'autres contraintes différentes ; ex : une *assemblée* doit avoir entre 2 et 20 *personnes* : contrainte de cardinalité `[2,20]` ;

- *contrainte dans l'application sur des listes* : on contraint une application sans emploi de la cardinalité pour indiquer qu'un objet a une application sur une collection ordonnée d'objets ; ex : un *polygone* doit avoir au minimum 3 et au maximum n *points* et ces points doivent être ordonnés : contrainte `Connecté Via [3,M;liste];`
- *contrainte dans l'application dupliquée sur des objets* : elle indique qu'un objet s'applique plusieurs fois sur un autre objet ; ex : un *patient* peut avoir le même problème de santé diagnostiqué plusieurs fois : contrainte `Plaindre de [sac];`
- *contrainte dans l'application unique sur des objets* : elle sert à indiquer qu'un objet a une application sur un ensemble d'objets qui porte seulement une fois sur un même objet du même ensemble ; ex : une *personne* ne peut être employée qu'une seule fois dans la même *entreprise* : contrainte du type `Employer [ensemble];`
- *contrainte sur l'ordre des objets-treillis* : c'est un cas particulier de la contrainte liste. La contrainte treillis est appliquée à un seul *objet* qui doit avoir un ordre interne ; ex : un objet n'est pas à la fois sur-type et sous-type : contrainte `Sur-Type [treillis];`
- *contrainte sur l'ordre des objets-arbre* : elle est utilisée pour représenter des applications portant sur des hiérarchies sous forme de structure d'arbre ; ex : un *arbre* a plusieurs branches mais seulement une racine : contrainte `Dirige [arbre];`
- *contrainte i-relation* : les i-relations ont presque les mêmes contraintes qu'une relation mathématique : *réflexive*, *irréflexive*, *symétrique*, *asymétrique*, *antisymétrique*, *transitive* et *intransitive*. Une contrainte de ce type est représentée avec son nom entre crochets ;
- *contrainte d'applications invariantes* : elle est employée pour représenter des applications qui doivent être invariables ; cette contrainte est représentée avec le mot "invariant" ou la lettre "i" dans des crochets ("[*invariant*]" ou "[*i*]") ;
- *contrainte d'unicité* : elle est employée pour représenter un objet qui doit être identifié de façon unique dans une application ; cette contrainte est représentée avec le mot "unicité" suivi de l'expression d'unicité entre crochets.

Les contraintes comportementales limitent l'exécution de processus ; une manière de restreindre un processus est d'imposer des limites à ses changements d'états, ce qui est

fait avec les événements types qui représentent les changements d'états permis comme résultat d'une opération. Les contraintes comportementales du type stimulus/réponse et les contraintes sur les opérations sont présentées ci-dessous.

Règle : c'est une déclaration, un principe ou une condition qui doit être satisfait [Obj92]. On peut classifier les règles en *Règles de Contraintes* et *Règles de Dérivation*.

Les règles de contraintes spécifient des principes ou des conditions qui restreignent la structure et le comportement d'un objet, elles peuvent être des types suivants : *Stimulus/Réponse*, *Contrainte d'Opération* et *Contrainte Structurelle*. Les règles de dérivation spécifient des principes ou des conditions pour inférer ou calculer des faits basés sur d'autres faits, elles peuvent être des types suivants : *Règle d'Inférence* ou *Règle de Calcul*. La table D.1 présente des exemples de règles.

Type de Règle	Exemple
Stimulus/Réponse	**Quand** le niveau du stock d'un produit devient plus petit que le niveau minimum ; **Alors** commander le produit.
Contrainte d'Opération	Marier un homme et une femme **Seulement Si** cette femme n'est pas mariée et si cet homme n'est pas marié.
Contrainte Structurelle	**Il Faut Toujours Que** le salaire d'un employé ne soit pas plus grand que le salaire de son chef.
Règle d'Inférence	**Si** un polygone a un périmètre ; **Alors** un triangle a un périmètre.
Règle de Calcul	Le prix net d'un produit **Est Calculé de la Manière Suivante** : `prix du produit*(1 + taux/100)`.

Table D.1 - OOAD - Exemples de Règles

La méthode introduit la possibilité d'utilisation d'une combinaison des règles avec les diagrammes déjà présentés. Les règles peuvent remplacer, par exemple, les contraintes de cardinalité d'une application ou même remplacer tout un diagramme avec des événements (règle stimulus/réponse par exemple).

Le choix est laissé au développeur du système et dans ce cas un problème se pose : s'il y a beaucoup de règles, on aura certainement divers types de règles ce qui peut produire un diagramme illisible, ou qui peut rendre difficile la réutilisation du système. La solution suggérée par la méthode est autant que possible l'attachement des règles aux diagrammes à des endroits appropriés.

La table D.2 présente des possibilités pour l'attachement des règles présentées à la table D.1 aux diagrammes proposés par la méthode.

Type de Règle	Attachement	Particularité
Stimulus/Réponse	Condition de Contrôle	du genre WHEN/IF/THEN
	Déclencheur	du genre WHEN/THEN
Contrainte d'Opération	Pré Condition/ Post Condition	-
Contrainte Structurelle	I-Relation	rapport avec une I-Relation.
	Objet-Type	
	Nom de l'Attribut	rapport avec un attribut type.
Règle de Calcul	\ de la I-Relation	rapport avec une application dérivée.
	\ de l'Objet Type Dérivé	rapport avec un objet type dérivé.

Table D.2 - OOAD - Attachement de Règles aux Diagrammes

Méta-Modèle : c'est un modèle des modèles ; il contient des objets types dont les instances sont aussi des objets types.

D'une manière générale, un Méta-Modèle est un Modèle utilisé pour exprimer plusieurs types de modèles. Le Méta-Modèle doit définir des objets types, qui imposent la façon d'exprimer des niveaux avec un plus petit niveau d'abstraction.

Pour illustrer ceci avec une approche traditionnelle pour la modélisation, le niveau Modèle est situé au-dessous du niveau Méta-Modèle et au-dessus du niveau de données et processus (cf. figure D.17).

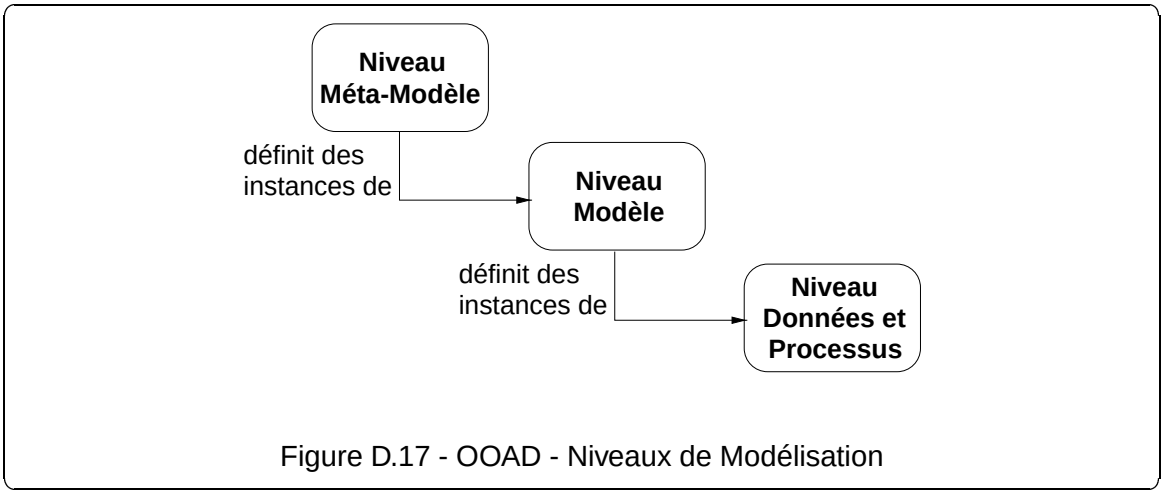


Figure D.17 - OOAD - Niveaux de Modélisation

Cette approche conventionnelle en trois niveaux pose des problèmes pour définir des modèles complexes pour lesquels existe le besoin d'une représentation en sous niveaux. Un autre problème que pose l'utilisation de cette structure est en rapport avec le changement des types d'objets qui peuvent être définis au Niveau Modèle, car cela entraîne une modification au Niveau Méta-Modèle.

La méthode propose une autre représentation qui doit résoudre ces problèmes. Cette représentation est faite à l'aide d'un Cadre de Travail Unique, où existe un "Noyau Méta-Modèle", qui sera employé pour décrire le modèle même, et qui doit avoir quelques objets types primitifs, tels que objet type, objet, schéma relationnel et application. Pour supporter les processus, il doit aussi exister les objets types opération, règle de déclenchement, condition de contrôle et événement type. Cependant, pour qu'un Cadre de Travail Unique marche bien, le Noyau Méta-Modèle doit être défini d'une façon précise et consistente.

Supra Type : c'est un objet type dont les instances sont des sous-types d'un autre objet type. La figure D.18 présente la représentation des supra types. Ce concept permet de travailler avec des collections de collections d'objets.

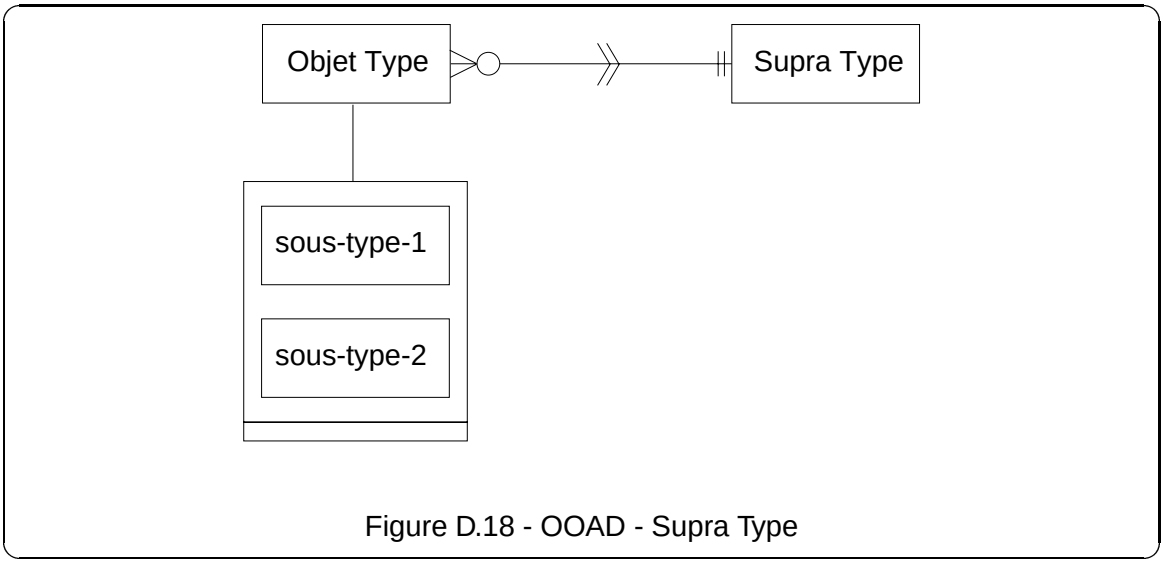


Figure D.18 - OOAD - Supra Type

Le diagramme de la figure D.18 peut être interprété comme : "chaque instance de l'objet type s'applique à exactement une instance du supra type" ou comme, "les instances du supra type sont des sous-types de l'objet type".

Dans le cas où un objet type s'applique à deux supra types différents, la méthode présente deux manières d'attacher les sous-types de l'objet type au supra type. La première consiste à utiliser une flèche en trait pointillé, depuis la boîte de sous-types vers celle du supra type (indiquant que les sous-types sont des instances du supra type). La deuxième consiste à écrire le nom du supra type au-dessous de la boîte de sous-types.

D.4 Niveau Application

Dans le Niveau Application, on doit utiliser les concepts présentés aux deux autres niveaux pour spécifier ou implémenter le système.

Martin et Odell présentent dans [MO95] plusieurs façons de spécifier la phase d'Analyse en utilisant les concepts présents dans les deux autres niveaux. Cette spécification est implémentée avec une Spécification Structurelle et une Spécification Comportementale.

La méthode n'introduit pas une représentation unique pour cette spécification, elle ouvre même la possibilité d'utiliser un type de diagramme spécifique, à condition que ce diagramme soit défini en fonction des concepts présentés dans les deux premiers niveaux de la méthode ; ex : pour modéliser un Diagramme de Flux de Données, il faut définir flux de données, opération et mémoire comme étant des instances d'objet type, ainsi que les contraintes pour les associations entre ces instances.

Spécification Structurelle : pour cette étape de spécification d'un système, l'ISO ("International Organization for Standardization") a défini trois façons de représenter conceptuellement les aspects structurels d'un système : le Modèle de Logique Prédicatif Interprété, le Modèle Binaire de Relations et le Modèle Entité-Relation-Attribut.

- *Modèle de Logique Prédicatif Interprété ("Interpreted Predicate Logic" - IPL)* : c'est un modèle où les données sont exprimées comme des phrases codées dans un langage formel quelconque.
- *Modèle de Relations Binaire* : c'est un modèle employé pour exprimer des types de choses et des associations entre ces types de choses ; il a ses origines dans les domaines de l'intelligence artificielle et de la linguistique.

  Avec une orientation à objet, une propriété est définie comme étant une application identifiable d'un type vers un autre type ou vers un ensemble d'objets. La figure D.19.a montre un exemple d'un Modèle de Relations Binaires.
- *Modèle Entité-Relation-Attribut (ERA)* : c'est une autre manière pour représenter les relations. Les Attributs types, qui sont des relations "attachées" à une entité type, sont représentés graphiquement à l'aide d'un autre formalisme (cf. figure D.19.b équivalente à la figure D.19.a au formalisme près). On peut dans le Modèle ERA mélanger les deux représentations, c'est-à-dire, représenter quelques relations comme des applications et d'autres comme des Attributs types.

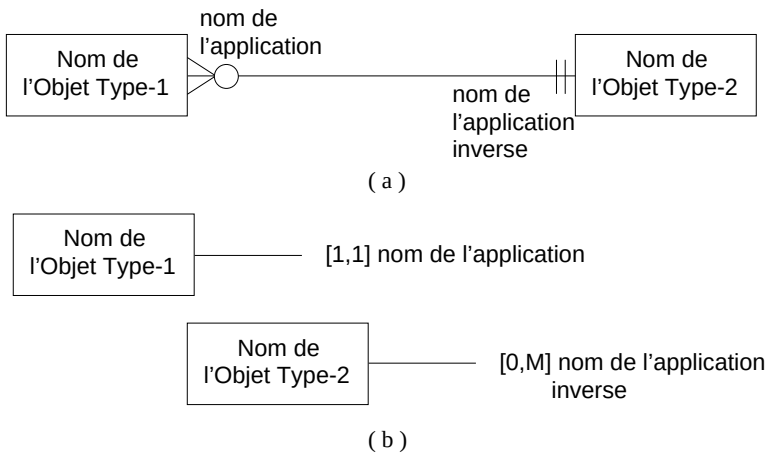


Figure D.19 - OOAD - Modèles de Relations

Spécification Comportementale : cette étape de spécification d'un système comprend la spécification pour le système ou pour une de ses parties, des entrées requises, des sorties qui doivent être produites et des relations qui existent entre les entrées et les sorties. La représentation de la Spécification Comportementale peut être réalisée selon deux approches : avec des états ou sans Relation avec des états.

– *Approche avec des États* : c'est une approche où le comportement d'un système est spécifié avec des états et des changements d'états. Les deux exemples principaux sont :

1. *Machines d'États Finis* : c'est une machine hypothétique qui peut exister, dans un instant donné du temps, seulement dans un seul état parmi plusieurs états dénombrables ; c'est une spécification des changements des états et des opérations pour un objet type particulier.
   Chaque machine est associée à un objet type particulier et, de cette façon, les opérations de la machine sont elles aussi associées au même objet type, ce qui permet un "mapping" vers des langages de programmation orientés objet, où chaque machine est implémentée comme une classe.
   La méthode OOAD n'introduit pas une représentation spécifique pour les Machines d'États Finis, mais propose d'utiliser les notations habituelles.
2. *Spécification Basée sur des Scénarios* : c'est la représentation du comportement comme un script ; c'est une spécification d'une série de changements d'états et d'opérations qui peut toucher plusieurs objets à la fois. Un *Scénario* est une spécification comportementale qui exprime un processus particulier comme une séquence d'événements et d'opérations.

La représentation des Spécifications Basées sur des Scénarios peut être faite de deux façons : avec les Scénarios Spécifiques et avec les Scénarios Généraux.

Les *Scénarios Spécifiques* décrivent une interaction simple avec des objets individuels pour lesquels la Spécification Comportementale est faite en utilisant la terminologie employée par les utilisateurs. Une des approches les plus connues est l'Analyse du Comportement des Objets ("Object Behavior Analysis" - OBA) proposée par Rubim [RG92], où le scénario est représenté comme une table appelée script. Chaque ligne de la table a les champs suivants : instigateur, événement, participant et opération. L'événement est initialisé par l'instigateur, l'opération est le processus déclenché par l'événement avec la participation de participants. Un autre type de Scénario Spécifique est la Description en Deux Colonnes, où dans une colonne est décrit le processus avec la terminologie de l'utilisateur, alors que dans l'autre, la description doit être appliquée à un modèle sous-jacent. Une autre technique est celle du Traceur d'événements proposé par Rumbaugh [RBP+91].

L'autre manière pour représenter les Spécifications Basées sur des Scénarios est faite avec des *Scénarios Généraux* pour lesquels les interactions sont décrites en termes des objets types. La méthode utilise comme Scénario Général les Diagrammes d'événements déjà présentés.

– *Spécification avec des Machines d'États Finis* : c'est une approche très utile pour modéliser un système avec une vue orientée vers les types. La Spécification Basée sur des Scénarios par contre est très utile pour modéliser un système avec un regard vers le processus. L'analyste peut donc combiner les deux types de spécifications pour représenter les situations où le concept d'état est important.

Approche sans Relation avec des États : c'est une approche où l'axe principal est l'expression du comportement en termes de décisions ou en utilisant un langage. C'est un type d'approche centralisé plus sur la spécification des opérations que sur l'état des objets. Les deux exemples principaux sont les Spécifications Basées sur des Décisions et les Spécifications Basées sur des Langages. Ces deux Approches sans Relations avec des États ne sont pas fondamentalement orientées objets car elles peuvent être utilisées sans être associées à un objet type. Ci-dessous, on présente ces deux approches.

– *Spécification Basée sur des Décisions* : c'est une spécification qui utilise des règles pour spécifier le comportement ; ces règles fournissent un ensemble de

décisions qui doivent être prises en compte par le système. Ce genre de spécification est utile pour décrire des systèmes de diagnostic, des systèmes de contrôle de processus et d'autres applications où la prise de décisions est le point central.

La représentation des spécifications Basées sur des Décisions est faite en utilisant une Table de Décision avec les règles applicables (Représentation textuelle) associées à un Arbre de Décision (Représentation graphique - "flowchart")

– *Spécification Basée sur des Langages* : c'est une spécification qui utilise le langage naturel pour spécifier le comportement, en français structuré ou en pseudo-code.

La division de la spécification en deux parties, Structurelle et Comportementale, est un fait commun car une grande partie de l'analyse est concernée par la spécification des aspects structurels et comportementaux d'un système. Cependant, certains aspects ne sont liés ni à la structure ni au comportement. On présente donc deux autres catégories de spécifications proposées par la méthode : la Spécification Contextuelle et la Spécification Fonctionnelle.

Spécification Contextuelle : c'est une spécification qui exprime les relations entre un système quelconque et l'environnement. Le système est un domaine composé des objets qui forment le tout et qui sont en accord avec un plan ou un but. L'environnement d'un système comprend les objets hors du système qui ont une relation représentative avec lui. Le domaine externe fait partie de l'environnement du système.

La méthode propose la représentation de la Spécification Contextuelle de deux manières : en utilisant le Diagramme Initial d'un Système proposé par Jacobson [JCJO92], ou le Diagramme de Contexte proposé par McMenamin et Palmer [MSP84].

Spécification Fonctionnelle : c'est une spécification qui définit les dépendances des processus prenant en compte ce qui est produit par un processus et consommé par un autre. La Spécification Comportementale définit les processus avec une précision algorithmique, en explicitant les procédures pas à pas. La Spécification Fonctionnelle ne descend pas à ce niveau là.

Une des représentations proposées pour la Spécification Fonctionnelle est l'utilisation de Diagrammes de Flux de Données - DFD. Comme les DFD ne sont pas fondamentalement orientés objets, ils sont modifiés pour prendre en compte cette philosophie, comme Shlaer et Mellor [SM92] le proposent.

L'utilisation de Diagrammes de Flux d'Objets est la représentation proposée par la méthode. Ce diagramme a deux éléments primaires : les Activités et les Produits.

Une Activité est un processus dont la production et la consommation sont spécifiées. Un Produit est le résultat final qui accomplit le but d'une Activité. Une Activité produira un Produit qui sera consommé par une ou plusieurs autres Activités. Le troisième élément est l'entité externe qui représente les entités hors du système. La figure D.20 présente la notation employée pour les Diagrammes de Flux d'Objets.

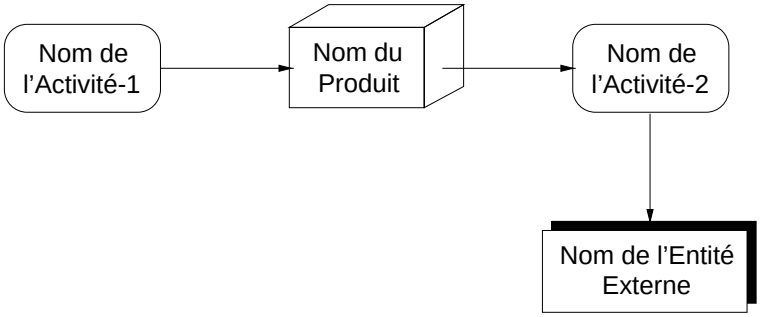


Figure D.20 - OOAD - Diagramme de Flux d'Objets

Ces diagrammes ne sont pas orientés objets car ils n'expriment pas des propriétés comportementales d'un objet type particulier. Ils induisent cependant un modèle orienté objet car chaque processus est organisé autour d'un groupe d'objets types appelés produit/ressource. Dans les Diagrammes de Flux d'Objets, chaque produit/ressource est produit par une seule opération (Activité). Le diagramme est orienté produit, ou orienté groupe d'objets, plutôt qu'orienté objet.

Un Diagramme de Flux d'Objets n'est pas vraiment orienté objet, car il a comme but principal la représentation du traitement à un niveau très haut, stratégique pour une entreprise. Cependant il peut être décomposé en d'autres Diagrammes de Flux d'Objets d'un niveau plus bas. Les Produits présents dans les Diagrammes peuvent être décrits avec plus de détails par un Diagramme d'Objets, de la même façon que les Activités peuvent être décrites avec plus de détails par un Diagramme d'événements et/ou avec du pseudo-code.

D.5 La Procédure de Développement

La méthode identifie trois phases pour le développement d'un système : l'Analyse, la Conception et la Construction.

L'Analyse est la représentation du monde réel à travers un Modèle Conceptuel. La Conception est la procédure qui spécifie l'implémentation d'un système en utilisant un modèle conceptuel du monde réel. La Construction est la procédure de construction d'un système à partir des spécifications de l'implémentation.

La figure D.21 présente les phases du développement par rapport aux modèles produits et aux modèles consommés.

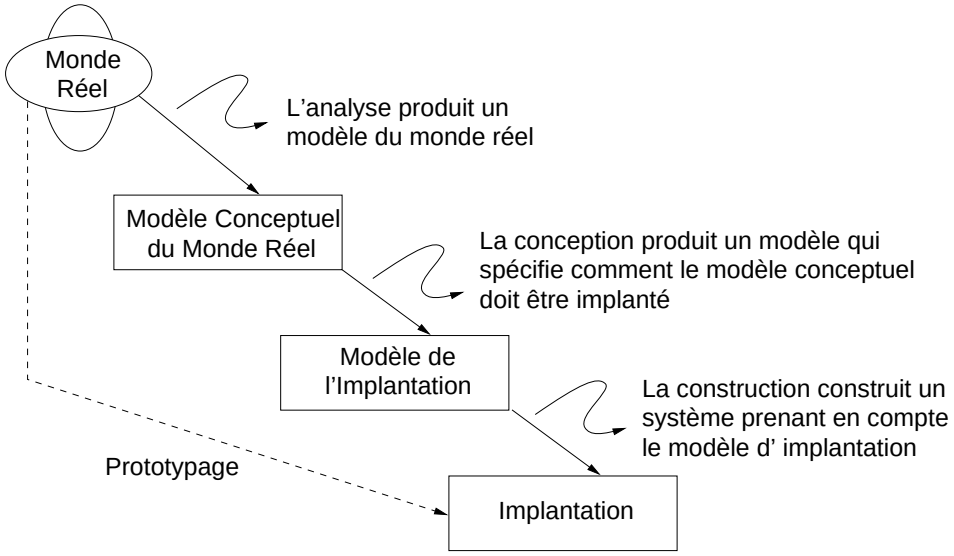


Figure D.21 - OOAD - Phases et Modèles de Développement

Dans Martin et Odell [MO95], on ne trouve pas une description complète de la démarche de Développement d'un système. Cette démarche, appelée COE "Corporate Object Engineering", est introduite dans [MO96] mais elle n'est pas présentée dans ce texte pour que celui-ci ne deviennent trop volumineux. Selon les auteurs, une stratégie de "traduction" de termes des diagrammes proposés vers les langages de programmation, en est un résumé des techniques proposés avec la démarche COE dans [MO95]. Cette stratégie de traduction est présenté, ci-dessous.

Les tables D.3 et D.4 présentent la conversion des termes de l'analyse orientée objet vers les langages orientés objets.

Termes des Diagrammes d' Objets	Équivalent dans les Langages de Programmation Orientés Objets
Objet Type	une classe avec les méthodes de création et terminaison.
Objet	objet.
Application	champ avec des méthodes pour la récupération et la modification des informations.
Hiérarchie de Généralisation	hiérarchie de classes/sous-classes avec héritage.

Table D.3 - OOAD - Conversion : Diagrammes d'Objets vers Langages OO

Termes des Diagrammes d'Événements	Équivalent dans les Langages de Programmation Orientés Objets
Opération	Opération.
Règle de Déclenchement	Un message ou une requête ; peut être supporté par un "event-scheduler".
Événement Type	Sans équivalent ; doit être implicite dans une méthode ou supporté par un "event-scheduler".
Schéma d'Événements	Méthode ou "event-scheduler".

Table D.4 - OOAD - Conversion : Diagrammes d'Événements vers Langages OO

Avec cette "traduction" des modèles de l'analyse vers des langages orientés objets, la technique de prototypage devient une technique très utile. La figure D.21 présente cette technique avec la flèche pointillée. Cependant, un développement avec cette technique peut produire du code qui n'est pas efficace, et donc, après avoir validé le modèle conceptuel, il faudra réimplanter le système.

Annexe E

Le Modèle STORM

Dans cette section on présente une modélisation d'un cas d'étude. Cette modélisation porte sur le modèle STORM [Adi95] et est faite en utilisant la méthode OMT [RBP+91].

E.1 Modèle STORM

Le modèle STORM [Adi95] propose une solution aux problèmes de modélisation et de gestion des données multimédias basées sur le temps ainsi que des possibilités, au niveau des langages et des interfaces pour la construction, la recherche et la mise à jour de données multimédias. Aujourd'hui quelques extensions ont été rajoutés au modèle original. Elles portent surtout sur la construction et l'interrogation de présentations [Adi96, AM97] ainsi que sur la modélisation des données vidéos [Loz96]. Un prototype au dessus du SGBD O_2 [O293] a été développé [MMA96].

Cette proposition est basée sur une approche objet, laquelle permet de décrire des présentations séquentielles ou parallèles, de les construire, de les mettre à jour et de les interroger. La présentation peut être considérée comme un objet sur lequel des opérations spécifiques peuvent être définies (interrogation, mise à jour et exécution) ; elle inclut les objets à présenter, les contraintes temporelles à leur appliquer (délai, durée) et les contraintes de synchronisation entre objets (l'un après l'autre, en parallèle ...). En effet, le modèle temporel utilisé est basé sur les intervalles et les opérateurs associés proposés par J. F. Allen [All83]. Le fait de stocker les présentations sous forme d'objets permet de les modifier et de les retrouver facilement et assure une homogénéité à l'approche.

Il est possible d'associer diverses présentations à un même objet : par exemple une image peut être présentée seule pendant deux minutes, ou bien associée (et synchronisée) avec un commentaire audio pour une autre présentation. Les présentations se réfèrent aux objets multimédias, mais ne les "contiennent" pas. Ceci est important car ces objets ont, en général, une grande taille et ne doivent pas être dupliqués. Avec cette approche, différentes présentations peuvent se partager le même objet.

Avec la modélisation de données vidéos, le modèle permet la création d'un modèle de structure hiérarchique pour une donnée vidéo, la construction des vidéos virtuelles en utilisant une algèbre ainsi que l'annotation des vidéos physiques ou virtuelles afin de permettre l'extraction de séquences vidéos selon certains critères qui sont ensuite utilisés pour l'extraction de segments vidéos à partir de requêtes.

E.1.1 Ombre Temporelle

Chaque objet multimédia doit pouvoir être présenté à l'utilisateur, mais pour un même objet, on peut en vouloir des présentations différentes. Par exemple, si X est une image, elle peut être présentée 3 minutes dans une présentation et 30 secondes dans une autre. Cet intervalle de temps (en secondes) durant lequel l'objet est perçu par l'utilisateur, constitue sa durée (notée $duration(X)$). Une *durée* a soit une valeur illimitée ou indéfinie (qualifiée de `free` dans le modèle), soit limitée (qualifiée de `bound`). Pour les objets statiques, la durée est illimitée par défaut. Ceci signifie qu'une fois une image affichée, l'utilisateur a la responsabilité de l'effacer. Naturellement, il est possible d'allouer un temps fixe (5 minutes par exemple) durant lequel l'image est affichée et ensuite automatiquement effacée. Par contre pour les objets dynamiques ou éphémères comme l'audio ou la vidéo, la durée est fixée par la nature même de l'information à présenter, si nous ne voulons pas de distorsion.

Dans une présentation, nous associons aussi à chaque donnée un *délai*. Pour chaque objet X, $delay(X)$ est le temps (en secondes) avant d'observer X. En d'autres termes, il y a un temps d'attente avant de présenter (jouer) l'objet au niveau de l'interface ; par exemple, attendre dix secondes avant de jouer un commentaire audio. Ici aussi, nous considérons un délai illimité (`free`) ou limité (`bound`). Un délai limité a une signification évidente et vient juste d'être décrit. La notion de délai illimité nécessite cependant certaines explications. Par définition, un délai illimité signifie que l'on est en attente "pour toujours" ! Le modèle propose d'utiliser ce concept pour exprimer le fait que le système est prêt à présenter un objet, mais attend qu'un événement se produise, typiquement une action de l'utilisateur. Par exemple, le système affiche une icône montrant qu'un objet vidéo est prêt à être joué (une petite TV) avec des boutons spécifiques tels que "play", "stop", "backward", "forward" et attend jusqu'à ce que l'utilisateur choisisse, par exemple le bouton "play".

Ceci nous conduit à la notion d'*Ombre Temporelle* associée à chaque présentation d'objet, elle est composée de deux intervalles : un délai et une durée. Ainsi, selon l'éclairage temporel que l'on applique à un objet à présenter, l'ombre temporelle est différente.

E.1.2 Objet STORM

Un *Objet STORM* (ou `SO` pour Storm Object) est une présentation, définie par un *quadruplet* (i, δ, d, c) où `i` est un identificateur. La durée `d` et le délai δ constitue l'*ombre temporelle*. L'information à présenter est le *contenu* `c` qui peut être soit monomédia (vidéo, audio, texte et image), soit composé de différents objets en utilisant des opérateurs séquentiels et/ou parallèles.

Certaines contraintes peuvent être appliquées à la synchronisation entre objets à l'aide de relations temporelles, basés sur les relations proposées par J. F. Allen [All83]. Ces contraintes sont les suivantes : `before, meet, equal, start, finish, overlap` et `during`. Les contraintes `before` et `meet` sont utilisées pour une présentation séquentielle ; `before` sert à introduire un délai entre les objets de la présentation et `meet` indique un délai zéro. Les autres contraintes sont utilisées avec une présentation parallèle ; la contrainte `equal` oblige les présentations de deux objets à commencer et à se finir en même temps ; la contrainte `start` implique que les présentations de deux objets commencent au même temps et la contrainte `finish` que les présentations se finissent au même temps ; la contrainte `overlap` impose que la présentation d'un objet commence après l'autre et la contrainte `during` impose que la présentation d'un objet commence après et se finisse avant la présentation de l'autre objet.

E.2 Modélisation OMT de STORM

Dans cette section, on présente la modélisation OMT pour le modèle STORM. Pour cela, on présente d'abord le Modèle des Objets suivi d'une partie du Modèle Dynamique ainsi que d'une partie du Modèle Fonctionnel.

E.2.1 Modèle des Objets de STORM

Le modèle des objets de STORM peut être divisé en *Sous-Systèmes* selon la figure E.1. Le sous-système *Structure Algébrique* contient les classes qui font la structuration des données multimédias ; le sous-système *STORM* contient les classes qui composent le méta-

schéma du modèle ; le sous-système *Vidéo* contient les classes qui font l'implantation d'une vidéo dans le système ; le sous-système *Structure Vidéo* contient les classes qui implantent la structuration des données vidéo dans le système ; enfin le sous-système *Classes O2* contient les classes qui sont dérivées directement de la classe *Objet* du SGBD O_2.

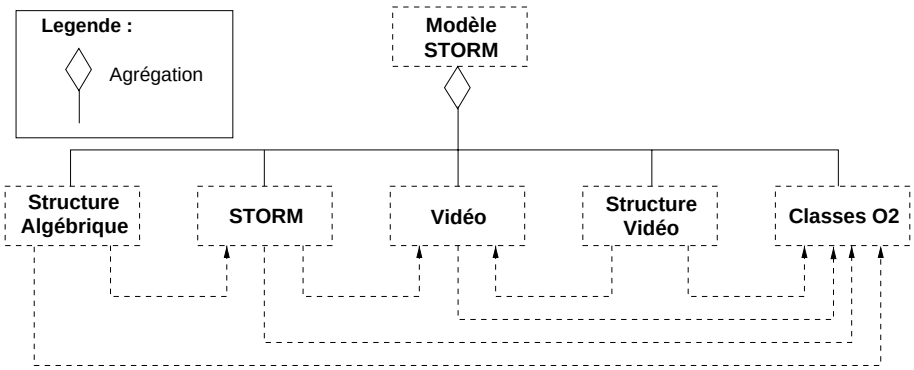


Figure E.1 - STORM - Diagramme de Sous-Systèmes

Le modèle STORM est développé au dessus du SGBD O_2, ce qui impose que toutes les classes utilisées dans le modèle héritent directement ou indirectement de la classe `Objet` définie dans le SGBD. Le Diagramme d'Interface de Sous-Système Classes O2 (cf. figure E.2) présente ces classes.

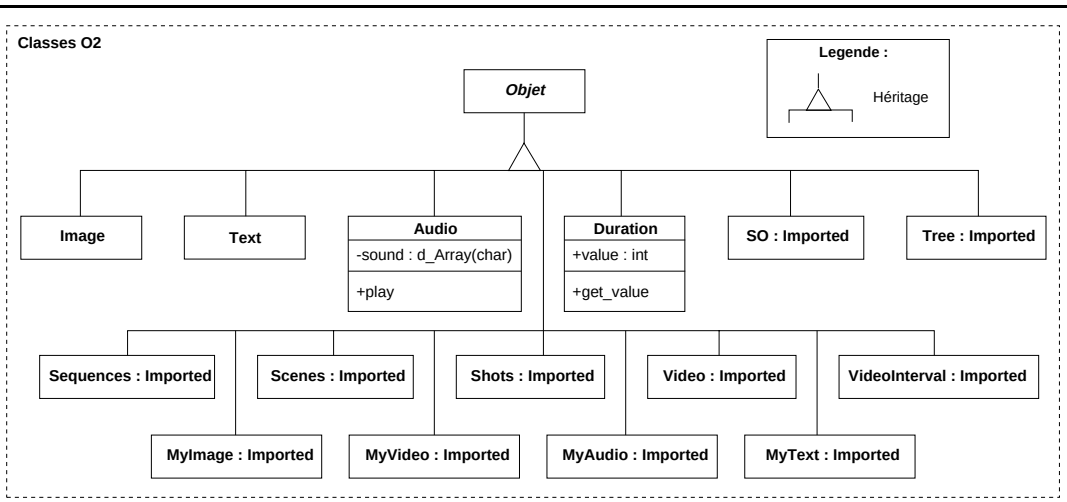


Figure E.2 - STORM - Diagramme d'Interface du Sous-Système Classes O2

La classe `Objet` ainsi que les classes `Image` et `Text`, qui sont des classes monomédias du SGBD, n'ont pas leur définition présentée. La classe `Audio` (elle aussi monomédia) a

été créée pour manipuler le son dans le SGBD. La classe `Duration` sert a contrôler le temps dans le modèle. Les classes `SO`, `Tree`, `Sequences`, `Scenes`, `Shots`, `Video`, `VideoInterval`, `MyImage`, `MyVideo`, `MyAudio` et `MyText` sont des classes définies dans d'autres sous-systèmes ; cependant à cause des caractéristiques du SGBD O_2 elles sont implantées comme des sous-classes de la classe `Objet`.

Le Diagramme d'Interface de Sous-Système STORM (cf. figure E.3) présente les classes qui composent le méta-schéma du modèle STORM.

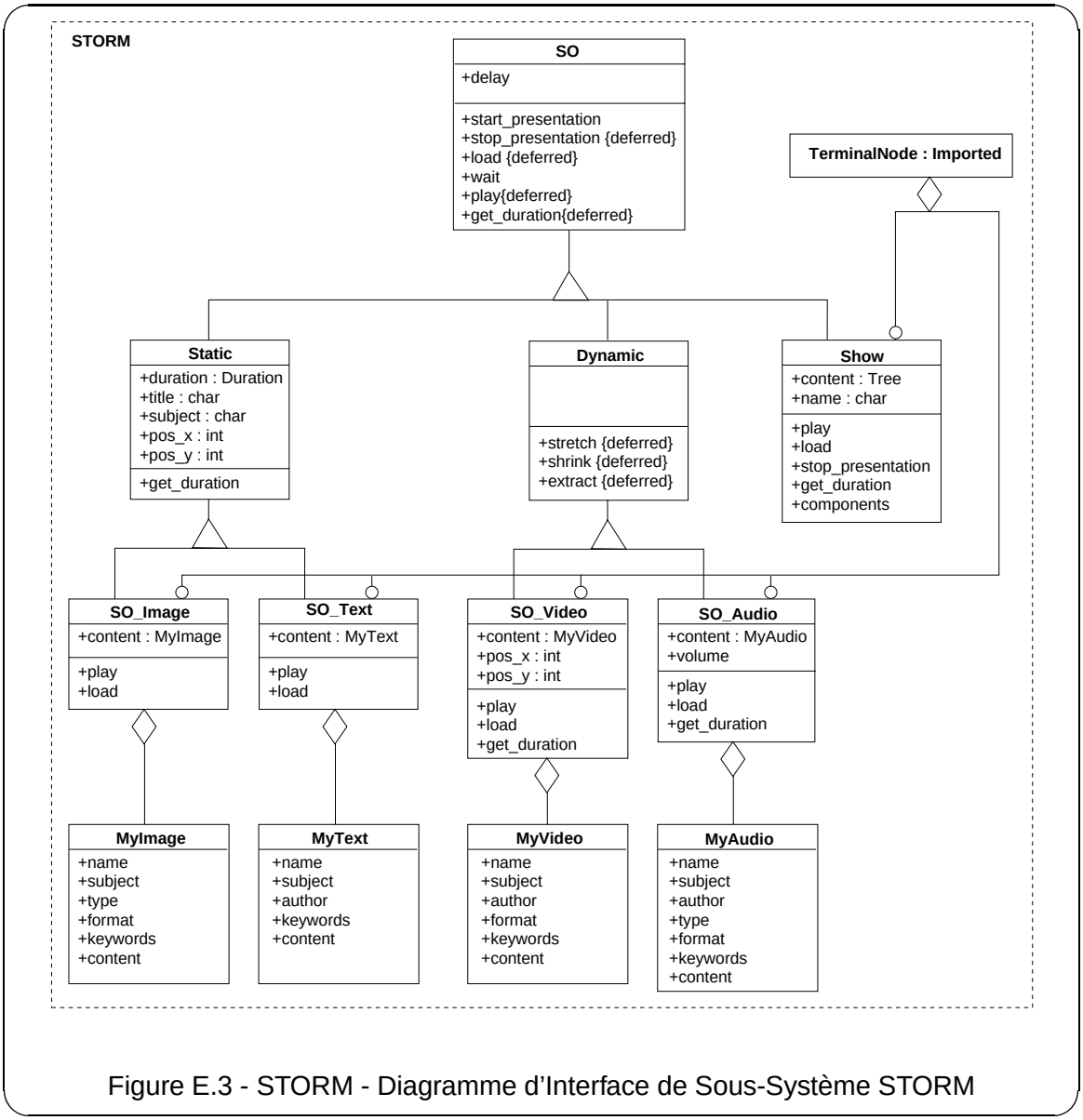


Figure E.3 - STORM - Diagramme d'Interface de Sous-Système STORM

La classe `SO` est la classe générique des objets STORM et elle est associée à des méthodes spécifiques applicables à tout objet STORM. Tout objet héritant de cette classe est

un objet STORM et possède une *Ombre Temporelle* constituée d'un délai et d'une durée. Il pourra être joué seul ou synchronisé avec d'autres objets dans une présentation (elle-même considérée comme un objet STORM). Les objets STORM sont ensuite divisés en objets statiques (classe `Static`), qui ont une durée non spécifique (on peut présenter un objet statique pendant une durée déterminée, en donnant une valeur entière à l'attribut `duration`) et dynamiques (classe `Dynamic`) qui ont une durée inhérente.

Les classes `SO_Image`, `SO_Text`, `SO_Video` et `SO_Audio` correspondent respectivement à des présentations d'objets monomédias image, texte, vidéo et audio. Cette approche établit une distinction claire entre les objets de la base, les objets des classes `MyImage`, `MyText`, `MyVideo` et `MyAudio` et leurs présentations (les classes `SO_Image`, `SO_Text`, `SO_Video` et `SO_Audio`). Ces classes `MyX` (X == Image | Text | Video | Audio) ont un contenu faisant référence à un objet de la classe `X` permettant de stocker dans la base des images, textes, sons ou vidéos. La classe `MyImage`, par exemple, permet de décrire son image (référence dans l'attribut `content`) ou une liste de mots clés (`keywords`). L'utilisation des identificateurs d'objets permet différentes présentations avec différentes *Ombres Temporelles* qui partagent le même objet.

On peut aussi vouloir une présentation de plusieurs objets monomédias en même temps. On utilise alors des objets des classes de base `SO_Image`, `SO_Text`, `SO_Video` et `SO_Audio` que l'on compose avec des contraintes de synchronisation. Ces objets possèdent une *Ombre Temporelle* et ils peuvent être synchronisés suivant leurs attributs temporels (`delay` et `duration`) ; ils peuvent être présentés d'une manière séquentielle ou parallèle ou une combinaison des deux. Les présentations de plusieurs objets sont stockées sous forme d'objets de la classe `Show` dont l'attribut `content` décrit la synchronisation des objets monomédias. L'attribut `content` fait référence à une classe "`Tree`" (non présentée dans le sous-système), un arbre étant une manière assez naturelle pour représenter une séquence d'objets ou un parallélisme entre eux ; les nœuds de cet arbre représentent la synchronisation et les feuilles les objets monomédias à présenter.

Le Diagramme d'Interface de Sous-Système Vidéo (cf. figure E.4) présente les classes qui implantent la donnée monomédia vidéo dans le système.

La racine de la donnée monomédia vidéo est la classe abstraite `Video`, qui regroupe les trois classes `DocumentVideo`, `RawVideo` et `VirtualVideo` en leur donnant les opérations de base "*play*", "*pause*", "*rewind*", "*forward*" et "*stop*".

La classe `RawVideo` sert à intégrer des vidéos physiques de différents formats au système à travers l'utilisation de fichiers qui correspondent à des vidéos physiques. La classe `VirtualVideo` sert à spécifier la composition de nouvelles vidéos, lesquelles sont com-

posées des instances de la classe `VideoInterval`. La classe `VirtualVideo` surcharge les opérateurs `+`, `-`, `%` et $\wedge$, respectivement concaténation, différence, union et intersection pour leurs utilisation soit avec des instances de la classe `VirtualVideo` soit avec des instances de la classe `VideoInterval`.

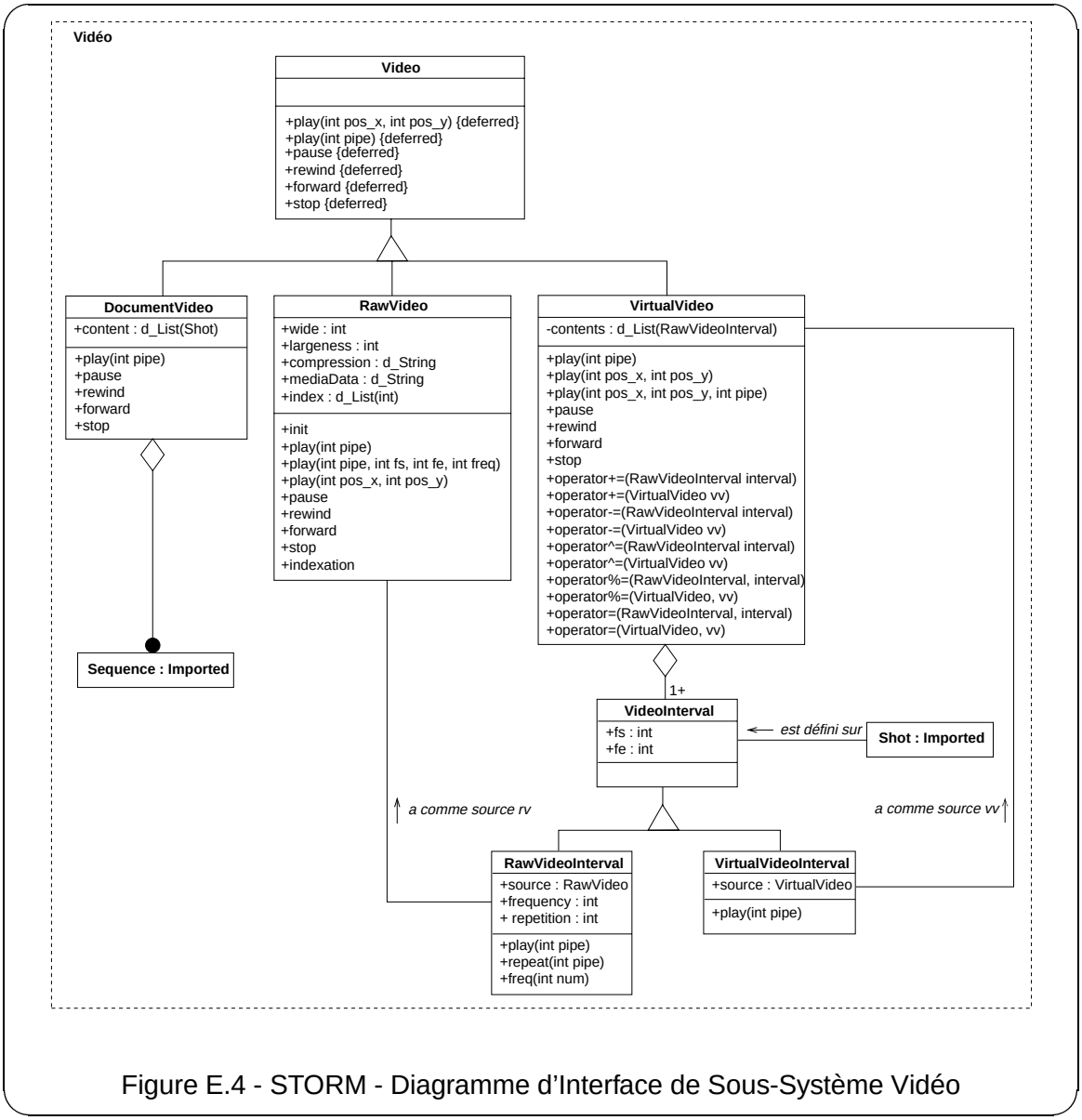


Figure E.4 - STORM - Diagramme d'Interface de Sous-Système Vidéo

La classe `VideoInterval` définit des intervalles (avec les attributs `fs` et `fe`) soit sur des instances de la classe `RawVideo` à travers la relation "*a comme source rv*" soit sur des instances de la classe `VirtualVideo` à travers la relation "*a comme source vv*". La classe `DocumentVideo` est utilisée pour spécifier la structure logique hiérarchisée d'une vidéo du type *RawVideo* ou *VirtualVideo* à travers des classes définies dans le sous-système Structure Vidéo.

Le Diagramme d'Interface de Sous-Système Structure Algébrique (cf. figure E.5.a) présente les classes qui implantent la structure sous forme d'arbres des présentations. Ces arbres ont pour feuilles des instances de classes STORM (`SO_Image`, `SO_Text`, `SO_Video`, `SO_Audio` ou `Show`) et pour nœuds des opérateurs de synchronisation séquentielle ou parallèle. À ces opérateurs de synchronisation, on peut attacher des contraintes : pour le séquence, les contraintes `meet` et `before` et pour le parallélisme, les contraintes `equal`, `start`, `finish`, `overlap` et `during`.

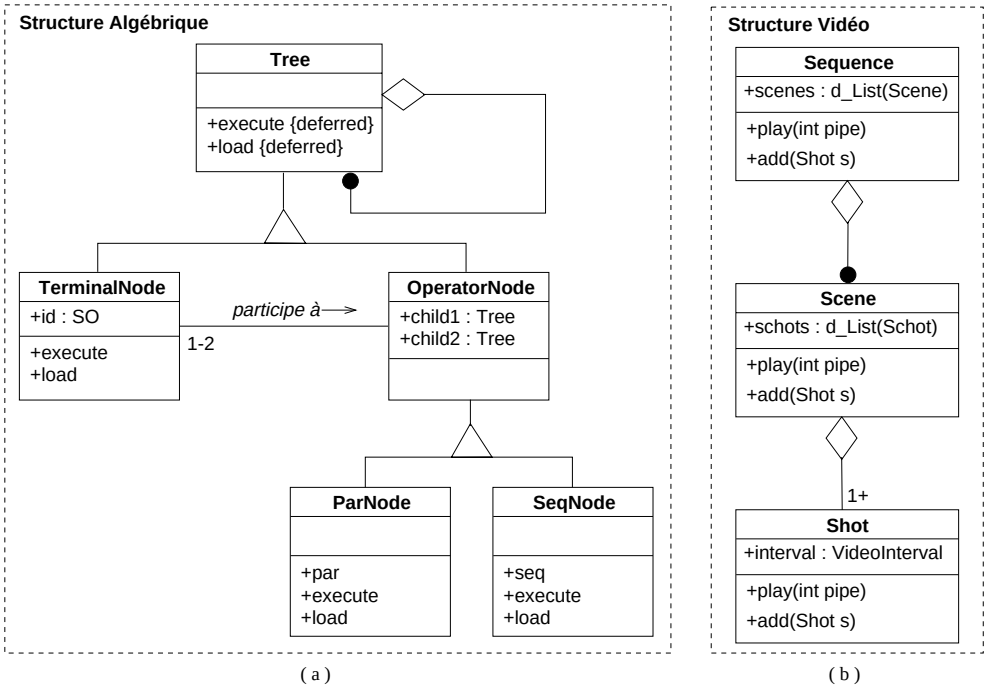


Figure E.5 - STORM - Diagrammes d'Interface de Sous-Système Structure Algébrique et Vidéo

La classe `Tree` est une classe abstraite qui regroupe les classes `TerminalNode` et `OperatorNode`. La classe `TerminalNode` fait référence aux données de base monomédia ainsi qu'aux présentations pour leur utilisation dans l'arbre. La classe `OperatorNode` est une classe abstraite qui regroupe les classes `ParNode` et `SeqNode`. Ces classes définissent la synchronisation entre les opérandes (des données monomédias ou des présentations) : soit en parallèle (méthode `execute` de la classe `ParNode`) soit d'une manière séquentielle (méthode `execute` de la classe `SeqNode`).

Le Diagramme d'Interface de Sous-Système Structure Vidéo (cf. figure E.5.b) décrit les classes qui implantent la structure selon laquelle les `DocumentVideo` (cf. figure E.4) sont présentés. L'approche utilisée est ascendante. De cette manière, une classe de bas niveau de structuration, la classe `Shots`, est définie. Cette classe introduit un intervalle de

frames sur un objet vidéo (physique ou virtuelle) dont l'identificateur est stocké dans l'attribut `source`. Les classes plus structurées `Scene` et `Sequence` sont composées soit des instances de la classe `Shot` soit des instances de la classe `Scene`.

Le Modèle des Objets complet pour le modèle STORM avec les sous-systèmes décrits ci-dessus est présenté à la figure E.6. On peut voir toutes les relations entre les différents sous-modèles. De cette manière, les relations entre les objets de la base (les classes `Image`, `Text`, `Video` et `Audio`), leurs descriptions (les classes `MyImage`, `MyText`, `MyVideo` et `MyAudio`) et leurs présentations (les classes `SO_Image`, `SO_Text`, `SO_Video` et `SO_Audio`) y sont présents, ainsi que la relation entre ces données monomédias, leurs présentations (la classe `Show`) et la Structure Algébrique. Le Modèle des Objets présente aussi la relation de composition entre les `DocumentVideo` et la Structure Vidéo. Un `DocumentVideo` est donc composé des `Sequence` qui sont elles-mêmes composées de `Scene`, elles-mêmes découpées en `Shot` défini par un `VideoInterval`.

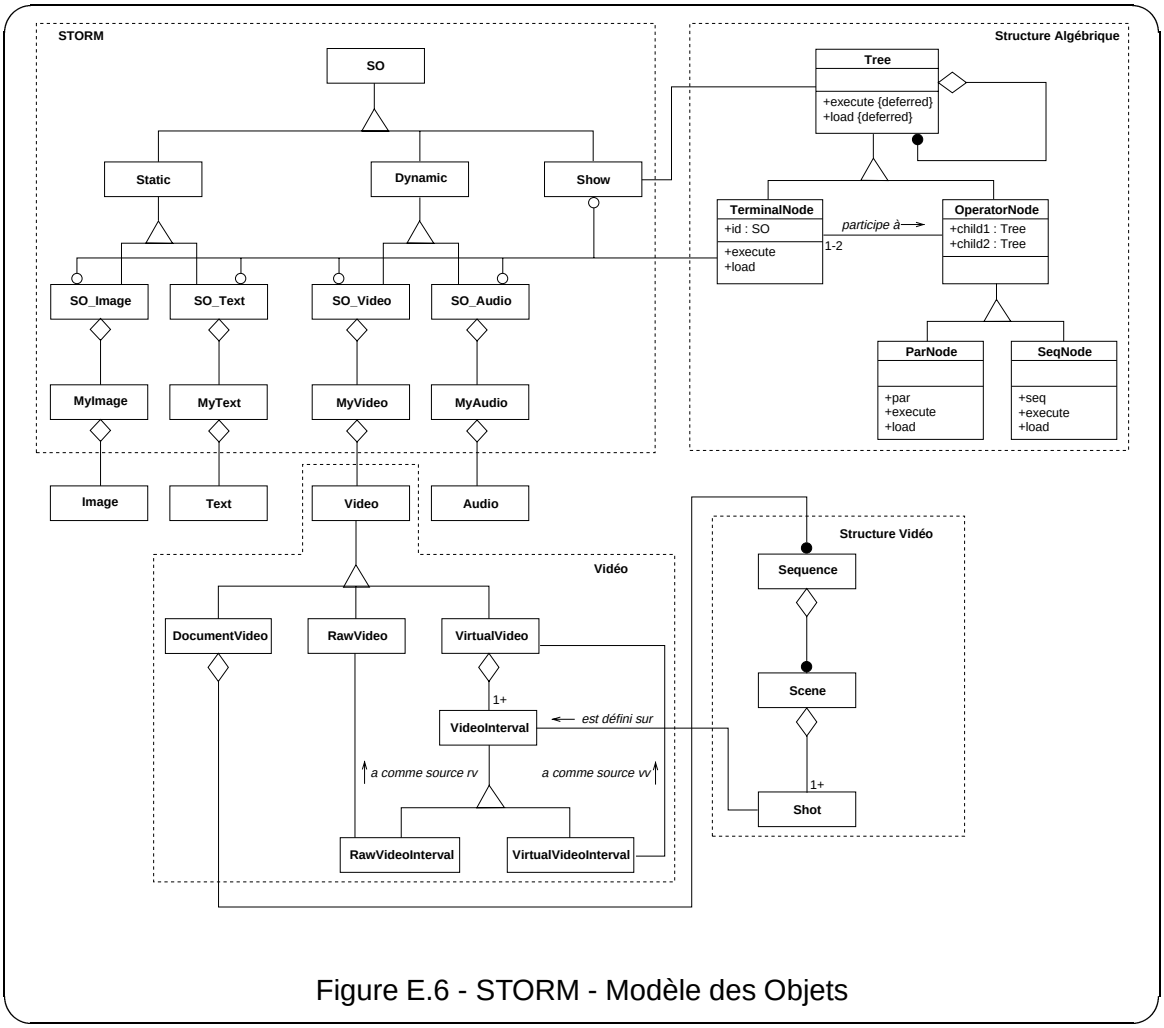


Figure E.6 - STORM - Modèle des Objets

Le Modèle des Objets STORM (cf. figure E.6) montre qu'un Show est décrit par une structure d'arbres binaires dont la classe `Tree` est la représentation. Chaque arbre binaire a pour nœud un `OperatorNode` lequel est composé de deux fils, `child1` et `child2`. Le fils `child1` est un `TerminalNode` et le fils `child2` est soit un `TerminalNode` soit un `OperatorNode` et dans ce cas il existe un sous arbre binaire construit de la même façon. Le `TerminalNode` est composé soit d'un autre Show soit d'un objet monomédia du type `SO_Image`, `SO_Text`, `SO_Video` ou `SO_Audio`. Le nœud `OperatorNode` peut être soit un `ParNode` soit un `SeqNode` et il détermine la synchronisation entre ses deux fils.

Le Diagramme d'Interface de Sous-Système Classe O2 n'est pas décrit dans sa totalité car il n'ajoute pas d'information sémantique à la présentation.

E.2.2 Modèle Dynamique de STORM

Cette section, par ses caractéristiques de présentation d'un cas d'étude, ne présente pas le modèle dynamique complet pour le système STORM. La classe `Show`, étant la classe qui définit les présentations dans le modèle, a été choisie comme source pour le Diagramme d'État présenté (cf. figure E.7).

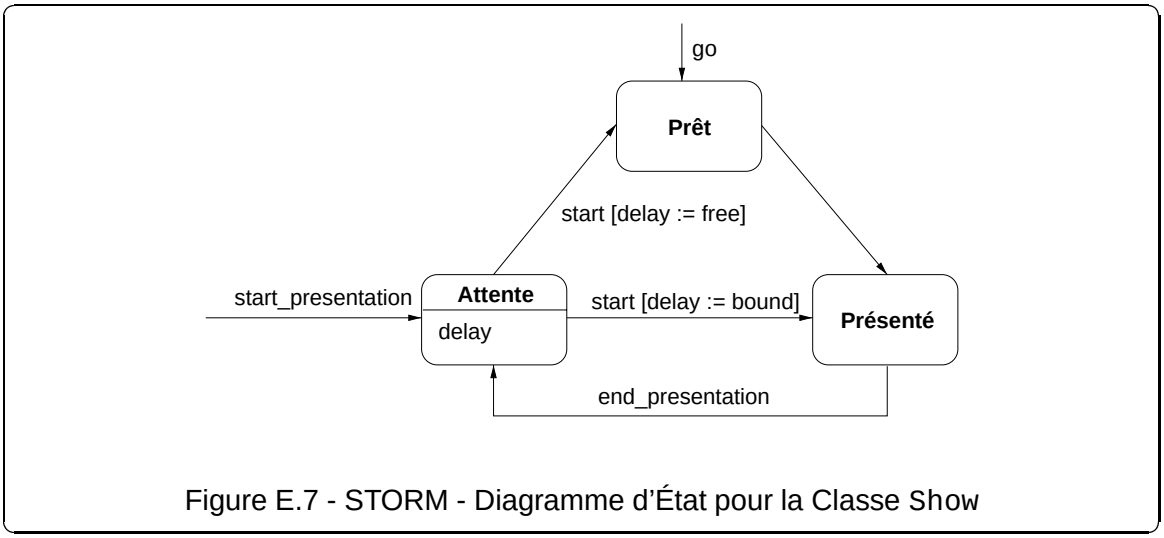


Figure E.7 - STORM - Diagramme d'État pour la Classe `Show`

Un Show est présenté selon une structure d'arbre (cf. figure E.6) dont les feuilles définissent les objets à présenter (dont un Show peut faire partie) et dont les nœuds représentent la synchronisation entre les composants (en parallèle ou en séquence). La présentation d'un Show démarre avec un message `start_presentation`; après avoir reçu ce message l'attribut `delay` indique si la présentation est exécutée après un délai défini (`delay :=`

bound) ou s'il faut attendre l'intervention de l'utilisateur pour la démarrer (delay := free). Dans le premier cas, le Show reste dans l'état Attente pendant la durée de delay puis il passe dans l'état Présenté. Dans le deuxième cas, le Show passe directement dans l'état Prêt. Lorsqu'il reçoit l'événement go qui est une action d'un utilisateur, il passe dans l'état Présenté. Cet état définit le déroulement de la présentation. C'est un sur-état constitué de plusieurs sous-états lesquels ne sont pas présentés. Après sa présentation, un Show retourne dans l'état Attente.

E.2.3 Modèle Fonctionnel de STORM

Le modèle fonctionnel n'est pas présenté dans sa totalité. Dans cette section, on présente la Description de l'Opération start_presentation de l'objet Show (cf. figure E.8) ainsi qu'un Diagramme d'Interaction d'Objets (cf. figure E.9). On ne présente pas un Diagramme de Flux de Données Orienté Objet parce qu'il n'y a pas de données échangées entre les objets lors d'une présentation.

Opération :	start_presentation
Responsabilités :	jouer une présentation décrite par une structure d'arbres binaires dont les feuilles sont des objets monomédias.
Entrées :	aucune
Sorties :	aucune
Objets Modifiés :	aucun
Pré Conditions :	toutes les feuilles de l'arbre doivent être des objets monomédias de type SO et les contraintes de synchronisation doivent être vérifiées.
Post Conditions :	les feuilles correspondant aux objets monomédias doivent être présentées selon la structure construite pour la présentation en respectant la synchronisation imposée.

Figure E.8 - STORM - Description de l'Opération start_presentation

La structure d'arbre, à laquelle un Show est lié par ses caractéristiques, rend difficile l'utilisation de la notation proposée par J. Rumbaugh dans [Rum95b] : par exemple, un OperatorNode peut être soit un ParNode soit un SeqNode et la séquence d'exécution en dépend. Les diagrammes originaux proposés par J. Rumbaugh ne prévoient pas la conditionalité ni le cheminement qui en résulte. Pour cette raison, le Diagramme d'Interaction d'Objets de la figure E.9 utilise la notation proposée par le Langage de Modélisation Unifié [BJR96]. Avec cette notation, lorsqu'une séquence dépend d'une condition, celle-ci est représentée entre crochets ("[ ]"), avant le nom du message ; une lettre majuscule au milieu d'un nombre de séquence indique qui celle-ci dépend de la condition qui lui est attaché.

Le diagramme d'Interaction d'Objets de la figure E.9 explicite l'interaction entre les objets nécessaires à l'exécution d'une présentation.

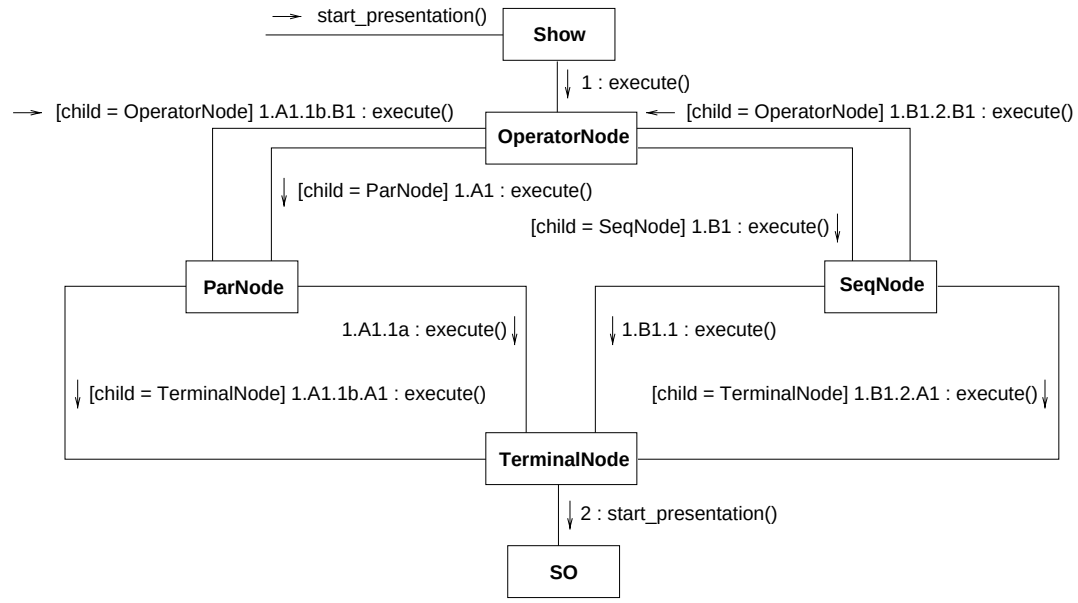


Figure E.9 - STORM - Diagramme d'Interaction d'Objets

La présentation étant définie par une structure d'arbres binaires, son déroulement en dépend. La présentation démarre avec l'exécution d'un `OperatorNode`, c'est-à-dire le lancement de son opération `execute`. Celle-ci déclenche deux autres opérations `execute`. Celle-ci déclenche deux autres opérations `execute` sur ses fils. L'`OperatorNode` est soit un `ParNode` soit un `SeqNode` (cf. figure E.5). Dans le cas d'un nœud `ParNode`, les deux opérations `execute` sur ses fils sont lancées en parallèle et dans le cas contraire en séquence. Les conditions sur les fils `child1` ou `child2` (`child` dans la figure) définissent s'il s'agit d'un `OperatorNode` ou d'un `TerminalNode` et suivant le cas, on lance l'opération `execute` appropriée. Lorsque l'arbre conduit vers un `TerminalNode` son exécution lance l'opération `start_presentation` sur un objet `SO`.

Annexe F

Z - Notations et Utilisations

F.1 Notation Z

Le texte présenté ci-dessous, est une traduction du document de présentation [Bow96c] d'un glossaire des composants du langage Z. Nous remercions son auteur, J. P. Bowen, de nous avoir permis de le reproduire en réalisant cette traduction.

Notations

a,b	identificateurs
d,e	déclarations (e.g., $a : A ; b, \ldots : B \ldots$)
f,g	fonctions
m,n	nombres
p,q	prédicats
s,t	séquences
x,y	expressions
A,B	ensembles
C,D	sacs
Q,R	relations
S,T	schémas
X	texte de schéma (e.g., d, $d \mid p$ or S)

Définitions

$a == x$	Définition d'abréviation
$a ::= b \mid \ldots$	Déf. de type libre (ou $a ::= b\langle\langle x \rangle\rangle \mid \ldots$)
$[a]$	Introduction d'ensemble de base (ou $[a, \ldots]$)

$a\_$	Opérateur préfixé
$\_a$	Opérateur postfixé
$\_a\_$	Opérateur infixé

Logiques

true	Constante logique vrai
false	Constante logique faux
$\neg\, p$	Négation logique
$p \wedge q$	Conjonction logique
$p \vee q$	Disjonction logique
$p \Rightarrow q$	Implication logique ($\neg\, p \vee q$)
$p - q$	Equivalence logique ($p \Rightarrow q \ \wedge \ q \Rightarrow p$)
$\forall X \bullet q$	Quantificateur universel
$\exists X \bullet q$	Quantificateur existentiel
$\exists_1 X \bullet q$	Quantificateur existentiel unique
let $a == x; \ldots \bullet p$	Définition locale

Ensembles et expressions

$x = y$	Égalité entre expressions
$x \neq y$	Inégalité entre expressions($\neg\, (x = y)$)
$x \in A$	Appartenance ensembliste
$x \notin A$	Non appartenance ensembliste ($\neg\, (x \in A)$)
$\varnothing$	Ensemble vide
$A \subseteq B$	Inclusion d'ensemble
$A \subset B$	Stricte inclusion d'ensemble ($A \subseteq B \wedge A \neq B$)
$\{x, y, \ldots\}$	Ensemble d'éléments (en extension)
$\{X \bullet x\}$	Ensemble d'éléments (en intension)
$\lambda X \bullet x$	Lambda-expression – fonction
$\mu X \bullet x$	Mu-expression – valeur unique
let $a == x; \ldots \bullet y$	Définition locale
if p **then** x **else** y	Expression conditionelle
$(x, y, \ldots)$	Tuple ordonné
$A \times B \times \ldots$	Produit cartésien
$\mathbb{P} A$	Ensemble des sous-ensembles
$\mathbb{P}_1 A$	Ensemble des sous-ensembles non vides
$\mathbb{F} A$	Ensemble des sous-ensembles finis
$\mathbb{F}_1 A$	Ensemble des sous-ensembles finis non vides

$A \cap B$	Intersection d'ensembles
$A \cup B$	Union d'ensembles
$A \setminus B$	Différence d'ensembles
$\bigcup A$	Union généralisée d'un ensemble d'ensembles
$\bigcap A$	Intersection généralisée d'un ensemble d'ensembles
first x	Premier élément d'une paire ordonnée
second x	Second élément d'une paire ordonnée
$\#A$	Cardinalité d'un ensemble fini

Relations

$A \leftrightarrow B$	Relation ($\mathbb{P}(A \times B)$)
$a \mapsto b$	Application ((a, b))
$\mathrm{dom}\, R$	Domaine d'une relation
$\mathrm{ran}\, R$	Image d'une relation
$\mathrm{id}\, A$	Identité d'une relation
$Q \fatsemi R$	Composition relationnelle
$Q \circ R$	Composition relationnelle inverse ($R \fatsemi Q$)
$A \lhd R$	Restriction du domaine
$A \lhd\!\!\!\!- R$	Anti-restriction du domaine
$R \rhd A$	Restriction de l'image
$R \rhd\!\!\!\!- A$	Anti-restriction de l'image
$R(\!\mid A \mid\!)$	Image relationnelle
iter n R	Relation composée *n* fois
R^n	idem *iter n R*
$R^{\sim}$	Relation inverse (R^{-1})
R^*	Fermeture transitive réflexive
R^+	Fermeture transitive irréflexive
$Q \oplus R$	Sucharge relationnelle ($(\mathrm{dom}\, R \lhd\!\!\!\!- Q) \cup R$)
$a \underline{R} b$	Relation infixée

Fonctions

$A \pfun B$	Fonction partielle
$A \fun B$	Fonction totalle
$A \pinj B$	Injection partielle
$A \inj B$	Injection totale
$A \psurj B$	Surjection partielle
$A \surj B$	Surjection totale

$A \rightarrowtail\!\!\!\!\!\rightarrow B$	Fonction bijective
$A \twoheadrightarrow B$	Fonction finie partielle
$A \rightarrowtail\!\!\!\!\!\twoheadrightarrow B$	Injection finie partielle
$f\ x$	Application de fonction (ou $f(x)$)

Nombres

$\mathbb{Z}$	Ensemble des entiers
$\mathbb{N}$	Ensemble des naturels $\{0, 1, 2, \dots\}$
$\mathbb{N}_1$	Ensemble des naturels non-nuls ($\mathbb{N} \setminus \{0\}$)
$m + n$	Addition
$m - n$	Soustraction
$m * n$	Multiplication
$m \operatorname{div} n$	Division
$m \operatorname{mod} n$	Modulo arithmétique
$m \leq n$	Inférieur ou égal
$m < n$	Inférieur
$m \geq n$	Supérieur ou égal
$m > n$	Supérieur
succ n	Fonction successeur $\{0 \mapsto 1, 1 \mapsto 2, \dots\}$
$m \mathinner{..} n$	Intervalle énuméré
min A	Minimum d'un ensemble de nombres
max A	Maximum d'un ensemble de nombres

Séquences

$\operatorname{seq} A$	Ensemble de séquences finies
$\operatorname{seq}_1 A$	Ensemble de séquences finies non-vides
$\operatorname{iseq} A$	Ensemble de séquences finies injectives
$\langle\rangle$	Séquence vide
$\langle x, y, \dots \rangle$	Séquence $\{1 \mapsto x, 2 \mapsto y, \dots\}$
$s \frown t$	Concaténation de séquences
$\frown / s$	Concaténation distribuée
head s	Premier élément de la séquence ($s(1)$)
tail s	Tous les éléments de la séquence sauf le premier
last s	Dernier élément de la séquence ($s(\#s)$)
front s	Tous les éléments de la séquence sauf le dernier
rev s	Image miroir d'une séquence
squash f	Compactage

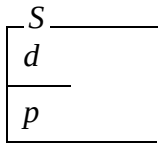
$A \upharpoonright s$	Extraction ($squash(A \lhd s)$)
$s \upharpoonright A$	Filtrage ($squash(s \rhd A)$)
s prefix t	Séquence de relation préfixée ($s \frown v = t$)
s suffix t	Séquence de relation suffixée ($u \frown s = t$)
s in t	Séquence de relation intervalle ($u \frown s \frown v = t$)
disjoint A	Disjonction d'une famille indexée d'ensembles
A partition B	Partition d'une famille indexée d'ensembles

Sacs

bag A	Sacs ou multi-ensemble ($A \pfun \mathbb{N}_1$)
$[\![]\!]$	Sac vide
$[\![x, y, \ldots]\!]$	Sac $\{x \mapsto 1, y \mapsto 1, \ldots\}$
count C x	Multiplicité d'un élément dans un sac
$C \sharp x$	idem à *count* C x
$n \otimes C$	mise à l'échelle de sacs
x in C	Appartenance à un sac
$C \sqsubseteq D$	Relation de sous-sacs
$C \uplus D$	Union de sacs
$C \setminus D$	Différence entre sacs
items s	Sac des éléments d'une séquence

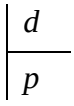
Notations de Schémas

Schéma Vertical



Les nouvelles lignes représentent ' ;' et '∧'. Le nom du schéma et le prédicat sont optionnels. Le schéma peut être référencé dans un texte par son nom.

Définition Axiomatique



La définition peut ne pas être unique. Le prédicat est optionnel. La définition s'applique globalement au document.

Définition Générale

$$[a, \ldots] == \begin{array}{|l} d \\ \hline p \end{array}$$

Les paramètres génériques sont optionnels. La définition doit exister. La définition s'applique globalement au document unique.

$S \mathrel{\widehat=} [X]$	Schéma horizontal
$[T ; \ldots \mid \ldots]$	Inclusion de schémas
$z.a$	Sélection de composant (given $z : S$)
θS	Formation de liaison
$\neg\, S$	Négation de schéma
$\mathrm{pre}\, S$	Précondition de schéma
$S \wedge T$	Conjonction de schémas
$S \vee T$	Disjonction de schémas
$S \Rightarrow T$	Implication de schémas
$S - T$	Équivalence de schémas
$S \setminus (a, \ldots)$	Masquage de composant(s)
$S \upharpoonright T$	Projection de composant
$S \fatsemi T$	Composition de schémas (S then T)
$S \gg T$	"Tubage" de schémas (S outputs to T inputs)
$S[a/b, \ldots]$	Renommage de composant (b devient a, etc.)
$\forall X \bullet S$	Quantificateur universal de schéma
$\exists X \bullet S$	Quantificateur existentiel de schéma
$\exists_1 X \bullet S$	Quantificateur existentiel unique de schéma

Conventions

$a?$	Entrée d'une opération
$a!$	Sortie d'une opération
a	État avant une opération
a'	État après une opération
S	Schéma d'état avant une opération
S'	Schéma d'état après une opération
ΔS	Changement d'état (normalement $S \wedge S'$)
ΞS	Pas de changement d'état (normalement $[S \wedge S' \mid \theta S = \theta S']$)

F.2 Z Adapté aux Objets

Les concepts principaux introduits par les travaux de A. Hall [Hal90b, Hal94] sont présentés ci-dessous à travers l'exemple introduit dans le deuxième article : la modélisation d'une école d'équitation.

Un objet est représenté par un schéma qui décrit son état ainsi que par une collection de schémas qui définissent les opérations de l'objet. Dans le schéma qui définit l'état, une variable (*self*) dont le type est un ensemble d'identités du type et dont la valeur est l'identité d'une instance particulière, définit l'identité d'un objet. La spécification Z de l'école d'équitation doit commencer par la définition des types de base du système qui sont :

$$[\mathit{RIDER}, \mathit{NAME}, \mathit{WEIGHT}, \mathit{MAN}, \mathit{WOMAN}, \mathit{LESSON}, \mathit{TIME}, \mathit{HORSE}]$$

Le schéma ci-dessous présente la modélisation d'un objet "cavalier" pour ce système.

Rider

$$\begin{array}{l} \mathit{self} : \mathit{RIDER} \\ \mathit{name} : \mathit{NAME} \\ \mathit{weight} : \mathit{WEIGHT} \\ \mathit{skill} : \mathbb{N}1 \end{array}$$

Afin de présenter les concepts de *Type Schéma, Liaison* et *Formation de Liaisons* que nous utilisons ensuite, nous introduisons les schémas Level_1 et Level_2 qui décrivent deux types de cavaliers avec des niveaux ("skill") différents en équitation.

Level_1

$$\begin{array}{l} \mathit{self} : \mathit{RIDER} \\ \mathit{name} : \mathit{NAME} \\ \mathit{weight} : \mathit{WEIGHT} \\ \mathit{skill} : \mathbb{N}1 \\ \hline 0 < \mathit{skill} < 2 \end{array}$$

Level_2

$$\begin{array}{l} \mathit{self} : \mathit{RIDER} \\ \mathit{name} : \mathit{NAME} \\ \mathit{weight} : \mathit{WEIGHT} \\ \mathit{skill} : \mathbb{N}1 \\ \hline 1 < \mathit{skill} < 3 \end{array}$$

Bien que ces deux schémas soient différents, ils définissent le même *Type Schéma*, qui représente l'objet *Liaison*, et qui est noté :

$$\langle\!|\ \mathit{self} : \mathit{RIDER}, \mathit{name} : \mathit{NAME}, \mathit{weight} : \mathit{WEIGHT}, \mathit{skill} : \mathbb{N}1\ |\!\rangle$$

Pour une définition formelle de Liaison et Type Schéma, nous reprenons le texte original de M. Spivey [Spi89] :

'Si $p_1, \ldots, p_n$ sont des identificateurs distincts et $x_1, \ldots, x_n$ sont des objets de types respectifs $t_1, \ldots, t_n$, alors il y a une **liaison** z avec les composants $z.p_i = x_i$, pour tout i, $1 \leqslant i \leqslant n$. Cette liaison est un objet représenté par le **type schéma** $\langle\!| p_1 : t_1, \ldots, p_n : t_n |\!\rangle$."

Ainsi, en utilisant le type schéma, nous pouvons réecrire les schémas $Level_1$ et $Level_2$ comme suit :

$Level_1$

$rider_1 : Rider$

$0 < skill < 2$

$Level_2$

$rider_2 : Rider$

$1 < skill < 3$

La *Formation de Liaison*, notée θS, où S est un schéma quelconque, est utilisée dans la simplification de prédicats de schémas. Pour le schéma *Rider*, $\theta Rider$ est un terme de type *Rider*, dont les composanst ont les valeurs suivants :

$\theta Rider.self = self$
$\theta Rider.name = name$
$\theta Rider.weight = weight$
$\theta Rider.skill = skill$

Ainsi pour déclarer, par exemple, que l'état après exécution d'une opération est identique à l'état avant, il suffit d'écrire : $\theta Rider' = \theta Rider$.

La collection de tous les cavaliers nommés uniquement par leurs identités est donnée par le schéma $\mathbb{S}Rider$ présenté ci-dessous. Ce schéma introduit l'ensemble de toutes les instances *Rider* à travers la variable *riders*, la fonction qui assure l'identité de l'objet cavalier ("*idRider*") ainsi que la variable *riderIds* qui est utilisée dans d'autres spécifications.

$\mathbb{S}Rider$

$riders : \mathbb{P}\, RIDER$
$idRider : RIDER \pfun Rider$
$ridersIds : \mathbb{P}\, RIDER$

$idRider = \{r : riders \bullet r.self \mapsto r\}$
$riderIds = \mathrm{dom}\, idRider$

La définition d'une opération pour une classe doit toujours inclure un schéma comme celui présenté ci-dessous qui indique que l'identité d'un objet ne change pas avec l'application d'une opération bien que son état puisse normalement changer.

$$
\begin{array}{|l}
\textit{RiderOp} \\
\hline
\Delta \textit{Rider} \\
\hline
\textit{self}' = \textit{self} \\
\hline
\end{array}
$$

Pour définir l'effet d'une opération sur la totalité du système, un schéma appelé $\mathbb{R}Op$ (Op == nom de l'opération) est défini. Ce schéma introduit une fonction entre des paramètres d'entrée et une relation définie entre les états avant et après la réalisation de l'opération, comme une manière de représenter cet effet. Ci-dessous, pour une opération particulière *ChangeSkillRider* qui change le niveau d'un cavalier (le schéma à gauche), on présente $\mathbb{R}$*ChangeSkillRider* qui définit la fonction entre l'entrée (*skill*?) et les états avant et après (le schéma à droite).

$$
\begin{array}{|l}
\textit{ChangeSkillRider} \\
\hline
\textit{RiderOP} \\
\textit{skill}? : \textit{SKILL} \\
\hline
\cdots \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\mathbb{R}\textit{ChangeSkillRider} : \textit{SKILL} \rightarrow \textit{Rider} \leftrightarrow \textit{Rider} \\
\hline
\mathbb{R}\textit{ChangeSkillRider} = \\
\quad \{ s : \textit{SKILL} \bullet s \mapsto \\
\qquad \{ \textit{ChangeSkillRider} \mid \textit{skill}? = s \\
\qquad\quad \bullet \theta \textit{Rider} \mapsto \theta \textit{Rider}' \\
\qquad \} \\
\quad \} \\
\end{array}
$$

La notion de classe est présentée simplement par l'introduction du type de base *CLASS* et de la définition des éléments de ce type, comme par exemple la classe *RiderClass*.

$$
\begin{array}{|l}
[\textit{CLASS}] \\
\hline
\textit{RiderClass} : \textit{CLASS} \\
\hline
\end{array}
$$

L'héritage est introduit dans le modèle à travers une relation appelée *subSuper* entre classes . Cette relation est présentée ci-dessous à travers une définition axiomatique (la définition de gauche) qui définit la sous-classe immédiate d'une sur-classe. Un exemple d'utilisation dans le modèle de l'école est l'introduction des professeurs d'équitation (la classe `TeacherClass`) qui sont représentés par un schéma qui suit la définition axiomatique (la définition de droite).

$$
\begin{array}{|l}
\textit{subSuper} : \textit{CLASS} \leftrightarrow \textit{CLASS} \\
\hline
\textit{subSuper}^{+} \cap \mathrm{id}\, \textit{Class} = \varnothing \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\textit{TeacherClass} : \textit{CLASS} \\
\hline
(\textit{RiderClass}, \textit{TeacherClass}) \in \textit{subSuper} \\
\hline
\end{array}
$$

Pour la représentation des structures d'héritage dans une spécification, Hall précise qu'on doit toujours faire l'inclusion du schéma donné ci-dessous, qui introduit implicitement la définition de la relation *subSuper*, dans la spécification du système.

ClassSystem

$$class : CLASS$$
$$Classe : \mathbb{P}\, CLASS$$
$$subSuper : CLASS \leftrightarrow CLASS$$

$$subSuper^{+} \cap \mathrm{id}\, class = \varnothing$$
$$\mathrm{dom}\, subSuper \subseteq Classe$$
$$\mathrm{ran}\, subSuper \subseteq Classe$$

La définition d'une classe en extension est faite à travers une relation entre types. Pour cela, un type de base appelé *OBJECT* est introduit ; c'est sur ce type, que portent les définitions des identificateurs des objets. Ci-dessous, on introduit ce type de base ainsi que la définition des identificateurs pour cavalier et maître.

$$[OBJECT]$$

$$RIDER, TEACHER : \mathbb{P}\, OBJET$$

La relation entre une classe et ses instances est donnée par une relation appelée *extension*, qui est présentée ci-dessous. Dans la présentation de la relation *extension*, nous utilisons le concept d'*Image Relationnelle*. Si X et Y sont des ensembles, on peut définir l'image relationnelle de la manière suivante :

$$\forall R : X \leftrightarrow Y ; S : \mathbb{P}\, X$$
$$R(\!| S |\!) = \{x : X ; y : Y \mid x \in S \land x \mapsto y \in R \bullet y\}$$

Cela veut dire que $R(\!| S |\!)$ sert à determiner l'ensemble de valeurs de Y qui sont liées à un ensemble de valeurs de X dans la relation R.

La définition des classes *RiderClass* et *TeacherClass* est introduite en utilisant la relation *extension* donnée alors par :

$$extension : CLASS \leftrightarrow OBJECT$$

$$extension(\!| \{RiderClass\} |\!) = RIDER$$

$$extension(\!| \{TeacherClass\} |\!) = TEACHER$$

En utilisant la relation *extension* et la relation *subSuper* le modèle formalise le fait que toutes les instances d'une sous-classe doivent aussi être des instances de sa sur-classe.

$$\forall C_{SUB}, C_{SUPER} : CLASS \bullet \\ C_{SUB} \mapsto C_{SUPER} \in subSuper \Rightarrow \\ extension(\!| \{C_{SUB}\} |\!) \subseteq extension(\!| \{C_{SUPER}\} |\!)$$

Le modèle proposé par Hall restreint les objets à être des instances directes d'une classe, et la relation *direct* présentée ci-dessous est utilisée pour présenter cette restriction, qui n'oblige cependant pas que les sous-classes soient disjointes (bien que cela puisse être représenté par disjoint$\langle Class_1, Class_2 \rangle$).

$$direct : CLASS \leftrightarrow OBJET$$

$$direct = extension\ (subSuper^{\sim}\ \fatsemi\ extension)$$

L'exemple ci-dessous utilise les définitions données pour représenter des hommes et des femmes cavaliers dans le système. La première définition introduit les classes et l'héritage et la deuxième les instances. L'héritage multiple est facilement représenté à travers l'utilisation de la définition de *subSuper*.

$$ManClass, WomanClass : CLASS$$

$$\{(ManClass, RiderClass), (WomanClass, RiderClass)\} \subseteq subSuper$$

$$MAN, WOMAN : \mathbb{P}\, RIDER$$

$$extension(\!| \{ManClass\} |\!) = MAN \\ extension(\!| \{WomanClass\} |\!) = WOMAN$$

La définition d'une classe en intention est faite à travers la description de son comportement, ce qui, en Z, est donné par le schéma d'état et par les schémas des opérations. Il est impossible cependant d'utiliser Z standard pour encapsuler l'état et les opérations. Hall établit par convention une relation entre l'extension et l'intention d'une classe pour résoudre ce problème : le nom du schéma d'état est le même que le nom de la classe et les valeurs de la variable *self* sont contraintes à être dans l'extension de la classe. De plus, les opérations des sur-classes et sous-classes doivent respecter certaines règles.

Une signification de l'intention des sous-classes est aussi introduite à travers une analogie avec les relations de raffinements Z. On ne présente pas ce point ici.

Pour la définition d'une sous-classe, Hall propose l'utilisation de l'inclusion de schémas. Ainsi, si on définit une sur-classe leçon (le schéma de gauche), la sous-classe "leçon à

l'intérieur" est donnée par un schéma qui fait l'inclusion du schéma de leçon (le schéma de droite).

Lesson

$self : LESSON$
$time : TIME$
$riders : \mathbb{F}\, RIDER$
$horses : \mathbb{F}\, HORSE$
$maxRiders : \mathbb{N}$

$\#riders \leq maxRiders$

Indoor

$Lesson$
$indoorLimit : \mathbb{N}$

$self \in INDOOR$
$maxRiders \leq indoorLimit$

La définition formelle de la relation hiérarchique entre *Lesson* et *Indoor* peut être vérifiée seulement si on vérifie les définitions des opérations : il faut que *Indoor* présente le même comportement que sa sur-classe. Pour une utilisation à titre de modélisation, cette vérification peut ne pas être mise en valeur. Pour une analyse plus approfondie de l'héritage concernant les opérations, consulter l'article [Hal94].

F.3 Z Orienté Objets

Les concepts principaux du langage Object-Z sont présentés ci-dessous à travers l'exemple introduit dans l'article [DKRS91] qui présente la modélisation d'une hiérarchie de formes. Cet exemple introduit la représentation des classes ainsi que les concepts d'héritage et d'instanciation présents dans Object-Z. La hiérarchie de formes est présentée à la figure F.1 en utilisant la notation OMT.

L'exemple introduit un type de base appelé *Vecteur* pour lequel les opérations suivantes sont définies :

$|\_| : Vecteur \rightarrow Real$
$pivoter : Vecteur \times \mathbb{R} \rightarrow Vecteur$
$\_ \perp \_ : Vecteur \leftrightarrow Vecteur$
$\_ + \_ : Vecteur \times Vecteur \rightarrow Vecteur$

La signification de ces opérations est la suivante : $| v |$ indique la magnitude d'un vecteur, $pivoter(v, \theta)$ indique que le résultat de l'opération est un vecteur sur lequel a été appliquée une rotation de θ degrés dans le sens inverse des aiguilles d'une montre ; pour deux vecteurs

v et w, $v \perp w$ indique que les deux vecteurs sont perpendiculaires et $v + w$ indique l'addition vectorielle des deux vecteurs. $\mathbf{O} \in \mathit{Vecteur}$ indique le vecteur zéro de magnitude zéro.

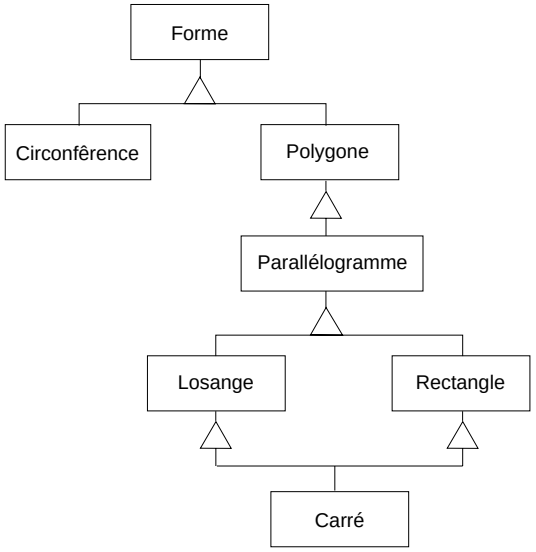


Figure F.1 - Hiérarchie de Formes

La classe de la hiérarchie dont toutes les autres classes héritent est la classe *Forme*.

Forme

$$\mathit{position} : \mathit{Vecteur}$$
$$\mathit{perim} : \mathbb{R}$$
$$\mathit{perim} > 0$$

Déplacer

$$\Delta(\mathit{position})$$
$$v? : \mathit{Vecteur}$$
$$\mathit{position}' = \mathit{position} + v?$$

Pivoter

$$\Delta(\mathit{position})$$
$$\theta? : \mathbb{R}$$
$$\mathit{position}' = \mathit{pivoter}(\mathit{position}, \theta?)$$

La classe *Forme* a deux variables *position* et *perim* ; *position* indique la position d'un point d'intérêt de la forme par rapport à l'origine et *perim* indique le périmètre de la forme. L'invariant établit que *perim* doit être positif et plus grand que zéro. Comme il n'y a pas de schémas d'état initial, aucune valeur initiale n'est donnée ni à *position* ni à *perim*. L'opération *Déplacer* change la position du point de référence de la forme d'un vecteur $v?$; comme la Δ-liste ne spécifie que *position*, la variable *perim* ne change pas avec l'opération. L'opération *Pivoter* fait tourner la forme sur son origine d'un angle de $\theta?$ degrés ; *Pivoter* ne change pas *perim*.

La classe *Circonférence* est définie comme une sous-classe de *Forme* par le schéma donné ci-dessous. Dans *Circonférence*, la première déclaration change le nom de la variable *position* héritée de *Forme* en *centre* ainsi que *perim* en *circon*. Le schéma d'état introduit une nouvelle variable *rayon* utilisée dans l'invariant pour la définition de *circon*. Les deux opérations *Déplacer* et *Pivoter* sont héritées avec la variable *position* changée en *centre* dans leurs définitions ; tout le reste est hérité sans aucun changement.

Circonférence

$$Forme[centre/position, circon/perim]$$

$$rayon : \mathbb{R}$$

$$circon = 2\pi\ rayon$$

La classe *Polygone* présentée ci-dessous, hérite de *Forme* avec la variable *position* changée en *début*. Le schéma d'état introduit une nouvelle variable *côtés* qui a comme type une séquence de *Vecteur*. L'invariant établit que le nombre de côtés d'un polygone doit être égal ou plus grand que trois, que le périmètre est plus grand que zéro, que l'addition des vecteurs qui définissent le polygone est égale au vecteur zéro et que le périmètre du polygone est donné par l'addition des magnitudes des *côtés*. L'opération *Déplacer* est héritée sans changement et pour l'opération *Pivoter* les variables *côtés* changent : l'invariant établit que chaque côté tourne sur son origine d'un angle $\theta?$. L'opération *Pivoter* est donnée par l'ensemble des déclarations de *Pivoter* de la classe *Forme* et de la classe *Polygone*.

Polygone

Forme[*début*/*position*]

$$
\begin{array}{l}
côtés : \text{seq } Vecteur \\
\hline
\#côtés \geq 3 \\
\mathbf{O} \notin \text{ran } côtés \\
(\sum i : \text{dom } côtés \bullet côtés(i)) = \mathbf{O} \\
perim = \sum i : \text{dom } côtés \bullet | \, côtés(i) \, |
\end{array}
$$

Pivoter

$$
\begin{array}{l}
\Delta(côtés) \\
\hline
\text{dom } côtés' = \text{dom } côtés \\
\forall i : \text{dom } côtés \bullet côtés'(i) = pivoter(côtés(i), \theta?)
\end{array}
$$

La classe *Parallélogramme* présentée ci-dessous hérite de *Polygone*. Son schéma d'état établit qu'elle doit avoir quatre côtés et que l'addition vectorielle du premier et du troisième côté est égale au vecteur zéro.

Parallélogramme

Polygone

$$
\begin{array}{l}
\#côtés = 4 \\
côtés(1) + côtés(3) = \mathbf{O}
\end{array}
$$

Les deux classes *Losange* et *Rectangle* présentées ci-dessous héritent toutes les deux de la classe *Parallélogramme*. *Losange* établit que le premier et le deuxième côté doivent avoir la même magnitude et *Rectangle* établit que le premier et le deuxième côté doivent être perpendiculaires.

Losange

Parallélogramme

$$| \, côtés(1) \, | = | \, côtés(2) \, |$$

Rectangle

Parallélogramme

$$côtés(1) \perp côtés(2)$$

La classe *Carré* présentée ci-dessous montre un exemple d'utilisation de l'héritage multiple : un carré est à la fois un losange et un rectangle. Les définitions de *Losange* et

Rectangle sont mélangées dans la définition de *Carré*.

Carré

Losange
Rectangle

Une des possibilités d'utilisation des définitions de classes a été donnée avec l'héritage. L'autre possibilité se donne avec l'instanciation. Ainsi, des instances des classes *Circonférence* et *Rectangle* sont combinées pour produire une classe *Sucette* (présentée ci-dessous), dont la figure F.2 présente la forme avec des restrictions entre les dimensions.

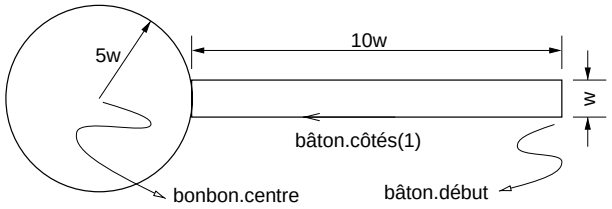


Figure F.2 - Exemple d'Instanciation

Pour la classe *Sucette*, la variable *w* est une constante qui spécifie la largeur du bâton et implicitement la taille de la sucette car toutes les dimensions de celle-ci sont définies par rapport à *w*. Le schéma d'état, à travers les invariants, introduit les dimensions de la sucette par rapport a *w* (cf. figure F.2). La définition des opérations *Déplacer* et *Pivoter* de la sucette sont données par la combinaison des opérations des classes *Circonférence* et *Rectangle*.

Sucette

$w : \mathbb{R}$

bâton : *Rectangle*
bonbon : *Circonférence*

$| \textit{bâton.côtés}(1) | = 10w$
$| \textit{bâton.côtés}(2) | = w$
$\textit{bonbon.rayon} = 5w$
$\textit{bonbon.centre} = \textit{bâton.début} + 1.5\ \textit{bâton.côtés}(1) + 0.5\ \textit{bâton.côtés}(2)$

Déplacer $\widehat{=}$ *bâton.Déplacer* $\wedge$ *bonbon.Déplacer*
Pivoter $\widehat{=}$ *bâton.Pivoter* $\wedge$ *bonbon.Pivoter*

R. Duke, P. King, G. Rose et G. Smith proposent d'autres exemples [DKRS91] qui présentent l'utilisation du polymorphisme et des invariants historiques. L'a structure hiérarchique de formes présentée ci-dessus introduit la notation en donnant des exemples d'utilisation. Pour une analyse plus approfondie on peut consulter le document original.

Bibliographie

[Abb83] Abbot (R.). – Program design by informal english description. *Communications of the ACM*, vol. 26, n11, november 1983.

[Abr78] Abrial (J. R.). – *Manuel du Langage Z*. – EDF - Paris, 1978.

[Abr96] Abrial (J. R.). – *The B-Book : Assigning Programs to Meanings*. – Cambridge University Press, 1996.

[Adi95] Adiba (M.). – Storm : Structural and Temporal Object-oRiented Multimedia database system. *Dans : IEEE International Workshop on Multimedia DBMS*, pp. 10–15. – Minnowbrook Conference Center - Blue Mountain Lake, NY, USA, august 1995.

[Adi96] Adiba (M.). – *STORM : an Object-Oriented Multimedia DBMS*, chap. 3 of Multimedia Database Systems : design and implementation strategies. – Nwosu, K. and Thuraisingham, B. and Berra, B., may 1996, Kluwer Academic Publishers édition.

[AFN94] AFNOR. – *Norme Française - NF Z 67-100-3*, octobre 1994.

[AG91] Alencar (A. J.) et Goguen (J. A.). – OOZE : An Object-Oriented Z Environment. *Dans : ECOOP'91*, éd. par America (Pierre). pp. 180–199. – Geneve - Switzerland, 1991.

[All83] Allen (J. F.). – Maintaining knowledge about temporal intervals. *Communications of the ACM*, vol. 26, n11, november 1983.

[AM97] Adiba (M.) et Mocellin (F.). – STORM : une approche à objets pour les Bases de Données Multimédias. *à paraître dans Technique et Science Informatiques*, 1997.

[And95] André (P.). – *Méthodes formelles et à objet pour le développement du logiciel : Études et propositions*. – Thèse de Doctorat, Université de Rennes I, juillet 1995.

[AT93] Atzeni (P.) et Torlone (R.). – A metamodel approach for the management of multiple models and the translation of schemes. *Information Systems*, vol. 18, n6, 1993, pp. 349–362.

[Bar92] Bari (M.). – *Une méthode d'analyse et de conception orienté objet de systèmes d'information actifs*. – Thèse de Doctorat, Université Paris 6, février 1992.

[BC94] B-Core(UK) (Limited). – B, Z and VDM. – october 1994. ftp ://ftp.tees.ac.uk/pub/bresource/docs.

[Ber93] Berard (E.V.). – *Essays on Object-Oriented Software Engineering*. – New Jersey, Prentice Hall - Englewood Clifs, 1993.

[BGW78] Balzer (R.), Goldman (N.) et Wile (D.). – Informality in program specifications. *IEEE Transactions on Software Engineering*, vol. SE-4, n2, march 1978.

[BH95] Bowen (J. P.) et Hinchey (M. G.). – Seven more myths of formal methods. *IEEE Software*, july 1995, pp. 34–41.

[Bid89] Bidoit (M.). – PLUSS, un langage pour le développement de spécifications algébriques modulaires. – Thèse d'état, may 1989. Université de Paris Sud, Orsay.

[BJR96] Booch (G.), Jacobson (I.) et Rumbaugh (J.). – The unified modeling language - version 0.91 addendum. – 1996. http ://www.rational.com/ot/uml91.pdf.

[Bla93] Blaha (M.). – Agregation of parts of parts. *Journal of Object Oriented Programming*, september 1993, pp. 14–20.

[Boe76] Boehm (B. W.). – Software engineering. *IEEE Transactions on Software Engineering*, vol. C-25, n12, 1976, pp. 1226–1241.

[Boe88] Boehm (B. W.). – A spiral model of software development and enhencement. *IEEE Computer*, may 1988.

[Boo81] Booch (G.). – Describing software design in Ada. *SIGPLAN Notices*, vol. 16, n9, 1981.

[Boo82] Booch (G.). – Object oriented design. *Ada Letters*, vol. 1, n3, march/april 1982.

[Boo83] Booch (G.). – *Software Enginnering with ADA*. – Benjamin/Cumings, 1983.

[Boo91] Booch (G.). – *Object-Oriented Design with Applications*. – Benjamin/Cumings, 1991.

[Boo94] Booch (G.). – *Object-Oriented Analysis and Design with Applications*. – Benjamin/Cumings, 1994.

[Bow96a] Bowen (J.). – Formal Methods/Individual notations, methods and tools. – http ://www.comlab.ox.ac.uk/archive/formal-methods.html, 1996.

[Bow96b] Bowen (J.). – The Z notation. – http ://www.comlab.ox.ac.uk/archive/z.html, 1996.

[Bow96c] Bowen (J. P.). – Glossary of Z notation. – ftp ://ftp.comlab.ox.ac.uk/pub/Zforum/zglossary.ps.Z, 1996.

[BP83] Bodart (F.) et Pigneur (Y.). – *Conception Assistée des Applications Informatiques : Étude d'Opportunité et Analyse Conceptuelle*. – Masson, Paris, 1983.

[BR93] Bicarregui (J.) et Ritchie (B.). – Invariants, frames and postconditions : A comparison of the VDM and B notations. *Dans : Proceedings of Formal Methods Europe 1993*. Industrial Strenght Formal Methods, pp. 162–182. – LNCS 670.

[Bru93] Brunet (J.). – *Analyse Conceptuelle Orientée Objet*. – Thèse de Doctorat, Université PARIS VI, 1993.

[BSE92a] Berard Software Engineering (Inc.). – *A Comparison of Object-Oriented Development Methodologies*. – Rapport technique, Berard Software Engineering Inc., 1992.

[BSE92b] Berard Software Engineering (Inc.). – *A Project Management Handbook for Object-Oriented Software Development*. – Rapport technique, Berard Software Engineering Inc, 1992.

[Bus45] Bush (V.). – As we may think. *Atlantic Monthly*, vol. 176, n1, 1945.

[CAB+93] Coleman (D.), Arnold (P.), Bodoff (S.), Dollin (C.), Gilchrist (H.) et Hayes (F.). – *Object-Oriented Development : the Fusion Method*. – Prentice-Hall, 1993.

[CD94a] Cook (S.) et Daniels (J.). – *Designing Object Systems - Object-Oriented Modelling with Syntropy*. – Prentice-Hall, 1994.

[CD94b] Cook (S.) et Daniels (J.). – Let's get formal. *Journal of Object Oriented Programming*, july-august 1994, pp. 22–24 and 64–66.

[CF92] Champeaux (D.) et Faure (P.). – A comparative study of object-oriented analysis methods. *Journal of Object Oriented Programming*, march-april 1992, pp. 21–33.

[Che76] Chen (P. P.). – The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems*, vol. 1, n1, march 1976, pp. 9–36.

[CJD93] Coleman (D.), Jeremaes (P.) et Dollin (C.). – *Fusion : A Systematic Method for Object-Oriented Development*. – Rapport technique, Information Modelling Department - HP Laboratories Bristol UK, july 1993.

[Col89] Colbert (E.). – The object-oriented software development method : A pratical approach to object oriented development. *Dans : Proceeding of TRI-Ada 89 - Ada Technology in Context : Application, Development and Deployment*. pp. 400–415. – New York, october 1989.

[Cor74] Corporation (IBM). – *HIPO : Design Aid and Documentation Technique*. – Rapport technique, Data Processing Division, 1974.

[CS96] Chmura (A.) et Sharon (D.). – Tools fair : Web with web and client/server development tools. *IEEE Software*, september 1996, pp. 29–38.

[CY91a] Coad (P.) et Yourdon (E.). – *Object-Oriented Analysis - 2nd Edition*. – Prentice Hall, 1991.

[CY91b] Coad (P.) et Yourdon (E.). – *Object-Oriented Design*. – Prentice Hall, 1991.

[DDG85] Dardailler (P.), Delobel (P.) et Giraudin (J. P.). – *Modélisation Progressive d'une Base de Données*. – Rapport technique n 493, Laboratoire de Génie Informatique - LGI/IMAG, janvier 1985.

[De 78] De Marco (T.). – *Structured Analysis and System Specification*. – Yourdon Inc, New-York, 1978.

[Dil94] Diller (A.). – *Z - An Introduction to Formal Methods*. – John Wiley et Sons, 1994.

[DKRS91] Duke (R.), King (P.), Rose (G.) et Smith (G.). – *The Object-Z Specification Language : Version 1*. – Rapport technique n91-1, Department of Computer Science, The University of Queensland, 1991.

[Dür94] Dürr (E.). – *VDM++ Language Reference Manual*. – Rapport technique nafro/cg/lrm/v9, Cap Gemini - Ultrecht University, 1994. Esprit III AFRODITE Project.

[D'S93] D'Souza (D.). – Working with OMT. *Journal of Object Oriented Programming*, october 1993, pp. 63–65 – 68.

[Dup97] Dupuy (S.). – *Intégration d'annotations formelles à OMT*. – Memoire de DEA, Université Joseph Fourier, 1997.

[EG93] Eckert (G.) et Golder (P.). – *Improving Object-Oriented Analysis*. – Rapport technique, Swiss Federal Institute of Technology, 1993.

[EKW92] Embley (R.G.), Kurtz (B.D.) et Woodfield (S.N.). – *Object Oriented Systems Analysis - A Model Driven Approach*. – New Jersey, Yourdon Press/Prentice Hall - Englewood Cliffs, 1992.

[Eme94] Emeraude (GIE). – Emeraude PCTE - environment guide. – 1994. http ://gille.loria.fr :7000/Emeraude/emeraude.html.

[Fer96] Ferguson (I.). – Metacase tools. – http ://osiris.sunderland.ac.uk/rif/metacase/metacase.tools.html/, 1996.

[FF95] Freire (J. C.) et Front (A.). – *Les Méthodes d'Analyse et Conception Orientées Objets - Approche Qualitative*. – Rapport technique, LSR-IMAG-Grenoble, 1995.

[Fin94] Findeisen (P.). – The Metaview System. – http ://ewb.cs.ualberta.ca/∼softeng/Metaview/doc/system.ps, 1994.

[FJ92] Feijs (L. M. G.) et Jonkers (F. B. M.). – *Formal Specification and Design*, chap. 35. – Cambridge University Press, 1992.

[FJ95] Freire Jr. (J. C.). – Analyse de la puissance d'expression des modèles orientés objet. *In : Actes du Congrès INFORSID*, éd. par INFORSID, pp. 523–538. – Grenoble, juin 1995.

[FJ96] Freire Jr. (J. C.). – Pouvoir d'expression de modéles orientés objet. *Ingénierie des systèmes d'information*, vol. 4, n2, 1996, pp. 219–237.

[FJCPG95] Freire Jr. (J. C.), Chabre-Peccoud (M.) et Giraudin (J. P.). – On the expressiveness of OO models and techniques, a methodology for its analysis applied to OMT-Rumbaugh. *In : International Conference on Information System Concepts (ISCO3) - Towards a Consolidation of Views - IFIP WG 8.1*, pp. 523–538. – Marburg - Allemagne, mars 1995.

[FJGF97] Freire Jr. (J. C.), Giraudin (J. P.) et Front (A.). – Atelier Modsi : un outil de méta-modélisation et de multi-modélisatio. *In : AFADL Approches Formelles dans l'Assistance au Développement de Logiciels*, pp. 35–46. – Toulouse, mai 1997.

[FJLM97] Freire Jr. (J. C.), Lozano (R.) et Mocellin (F.). – Vers un atelier de structuration et construction de présentations multimédias. *In : Actes du Congrès INFORSID*, éd. par INFORSID, pp. 267–286. – Toulouse, juin 1997.

[FK92] Fichman (R.G.) et Kemerer (C.F.). – Object-oriented and conventional analysis and design methodologies. *IEEE Computer*, October 1992, pp. 22–39.

[FKV91] Fraser (M. D.), Kumar (K.) et Vaishnavi (V. K.). – Informal and formal requirements specification laguages : Bridging the gap. *IEEE Transactions on Software Engineering*, vol. 17, n5, may 1991, pp. 454–465.

[FKV94] Fraser (M. D.), Kumar (K.) et Vaishnavi (V. K.). – Strategies for incorporating formal specifications in software development. *Communications of the ACM*, vol. 37, n10, october 1994, pp. 74–84.

[FL95] Facon (P.) et Laleau (R.). – Des spécification informelles aux spécification formelles : compilation ou interprétation ? *Dans : Actes du 13ème Congres INFORSID*, éd. par INFORSID, pp. 47–62. – Grenoble, juin 1995.

[Flu95] Flumet (J.). – *Un Environnement de Développement de Spécifications pour Systèmes Concurrents*. – Thèse de Doctorat, Université de Genève, juillet 1995.

[FTAF94] Fayad (M. E.), Tsai (W.), Anthony (R. L.) et Fulghum (M. L.). – Object Modeling Technique (OMT) : Experience report. *Journal of Object Oriented Programming*, november-december 1994.

[GH83] Guttag (J. V.) et Horning (J. J.). – An introduction to the Larch shared language. *Dans : Proceedings of the 9th IFIP Word Computer Congress*. pp. 809–814. – Amsterdam, 1983.

[Gir95] Giraudin (J. P.). – Évolution de la modélisation des systèmes d'informations. *Dans : Le Génie Logiciel et ses Application - Huitièmes Journées Internationales*. EC2 et Cie, pp. 103–113. – Paris-La Défense, 1995.

[GLO91] Gallouzi (S.), Logrippo (L.) et Obaid (A.). – Le LOTOS : théorie, outils, applications. *Dans : Proceedings of CFIP'91*, pp. 385–404.

[GMSB96] Gaudel (M. C.), Marre (B.), Schlienger (F.) et Bernot (G.). – *Précis de génie logiciel*. – Masson, 1996.

[Gra92] Gray (P. M. D.). – *Object-Oriented Databases : A Semantic Data Model Approach*. – Prentice-Hall-Englewood Cliffs, 1992.

[GS79] Gane (C.) et Sarson (T.). – *Structured Systems Analysis*. – Prentice-Hall, 1979.

[Hab88] Habrias (H.). – *Le modèle relationnel binaire - Méthode I.A. (NIAM)*. – Éditions Eyrolles, 1988.

[Hab94] Habrias (H.). – Les spécifications formelles pour les systèmes d'information, quoi ?, pourquoi ?, comment ? *Dans : XII INFORSID 17-20 mai 1994*. INFORSID, pp. 1–31.

[Hab97] Habrias (H.). – *Dictionnaire Encyclopédique du Génie Logiciel*. – Masson, 1997.

[Hal88] Halasz (F. G.). – Reflexions on notecards : Seven issues for the next generation of hypermedia systems. *Communications of the ACM*, vol. 31, 1988, pp. 836–852.

[Hal90a] Hall (A.). – Seven myths of formal methods. *IEEE Software*, september 1990.

[Hal90b] Hall (A.). – Using Z as a specification calculus for object-oriented systems. *Dans : VDM and Z - Formal Methods in Software Development*, éd. par Bjorner (D.), Hoare (C.A.R) et Langmaack (H.). pp. 290–318. – Springer-Verlag.

[Hal94] Hall (A.). – Specifying and interpreting class hierarchies in Z. *Dans : Proceedings of the Eighth Z User Meeting, Cambridge 29-30 june/1994*, éd. par Bowen (J.P.) et Hall (J.A.). Z User Group, pp. 120–138. – Springer-Verlag.

[Ham93] Hamon (Y.). – *Intégration de Techniques d'Intelligence Artificielle dans les Ateliers de Génie Logiciel*. – Thèse de Doctorat, Université Claude Bernard - Lyon 1, octobre 1993.

[Ham94] Hammond (J.). – Producing Z specifications from object-oriented analysis. *Dans : Proceedings of the Eighth Z User Meeting, Cambridge 29-30 june/1994*, éd. par Bowen (J.P.) et Hall (J.A.). Z User Group, pp. 316–336. – Springer-Verlag.

[Har87] Harel (D.). – Statecharts : A visual formalism for complex systems. *Science of Computer Programming*, no8, 1987, pp. 213–274.

[Har88] Harel (D.). – On visual formalisms. *Communications of the ACM*, vol. 31, n 5, May 1988, pp. 514–531.

[HJN93] Hayes (I. J.), Jones (C. B.) et Nicholls (J. E.). – *Understanding the Differences between VDM and Z*. – Rapport technique nUMCS-93-8-1, Computer Science/University of Manchester, 1993.

[HKR+92] Haan (B. J.), Kahn (P.), Riley (V. A.), Coombs (J. H.) et Meyrowitz (N. K.). – IRIS Hypermedia services. *Communications of the ACM*, vol. 41, n1, january 1992, pp. 36–51.

[Hoa85] Hoare (C. A. R.). – *Communicating Sequential Processes*. – Prentice-Hall, 1985.

[HPP93] Haxthausen (A.E.), Pedersen (J.S.) et Prehn (S.). – RAISE : a product supporting industrial use of formal methods. *Technique et Science Informatiques*, vol. 12, n3, march 1993, pp. 319–346.

[HSE90] Henderson-Sellers (B.) et Edwards (J. M.). – The object oriented life cycles. *Communications of the ACM*, september 1990, pp. 142–159.

[HvdGB93] Hong (S.), van der Goor (G.) et Brinkkemper (S.). – A formal approach to the comparison of objet-oriented analysis and design methodologies. *Dans : Proceedings of the Twenty-Sixty Annual Hawaii International Conference on System Sciences*, pp. 689–698.

[IFA96] IFAD. – The IFAD VDM-SL Toolbox : a pragmatic approach to formal methods. – http ://www.ifad.dk/products/toolbox.html, 1996. The Institute of Applied Computer Science.

[Iiv96] Iivari (J.). – Why are CASE Tools not used ? *Communications of the ACM*, vol. 39, n10, october 1996.

[Jac83] Jackson (M.A.). – *System Development*. – Prentice-Hall, 1983.

[JCJO92] Jacobson (I.), Christerson (M.), Jonsson (P.) et Overgaard (G.). – *Object-Oriented Software Engineering - A Use Case Driven Approach*. – Addison-Wesley, 1992.

[Jia95] Jia (X.). – *ZTC : A Type Checker for Z Notation - User's Guide*. – DePaul University, Chicago - Illinois - USA, 1995.

[Jon90] Jones (C. B.). – *Systematic Software Development using VDM*. – Prentice Hall International, 1990, 2nd édition.

[Kin90] King (S.). – *Z and the refinement calculus*. – Rapport technique nPRG-79, Programming Research Groupe - Oxford University Computing Laboratory, 1990.

[KKM87] Kirchner (C.), Kirchner (H.) et Meseguer (J.). – *Operational Semantic of OBJ3*. – Rapport technique n87-R-87, CRIN, 1987.

[Lan95] Lano (K.). – *Formal Object-Oriented Development*. – Springer-Verlag, 1995.

[Lan96] Lano (K.). – *The B Language and Method - A Guide to Pratical Formal Development*. – Springer-Verlag, 1996.

[Lar96] Larsen (P. G.). – Information on VDM. – http ://www.ifad.dk/vdm/vdm.html, 1996.

[Led94] Ledru (Y.). – An introduction to Z and formal methods. – 1994. Laboratoire Logiciel Systèmes et Réseaux - Grenoble - France.

[LH94] Lano (K.) et Haughton (H.). – *The Z++ Manual*, 1994. ftp ://theory.doc.ic.ac.uk/theory/papers/Lano/z++.ps.

[Lin93] Lindsay (P.). – *Reasoning about Z Specifications : A VDM Perspective*. – Rapport technique nTechnical Report 93-20, University of Queensland - Computer Science Departement, 1993.

[Loz96] Lozano (R.). – *Système de Gestion de Bases de Données Multimédias et Vidéos*. – Rapport technique, Grenoble, DEA d'Informatique, Université Joseph Fourier., Juin 1996.

[LT77] Ledgard (H. F.) et Taylor (R. W.). – Two views of data abstraction. *Communications of the ACM*, june 1977, pp. 382–384.

[Mal71] Mallet (R.). – *La Méthode Informatique*. – Hermann, 1971.

[MC92] Meira (S. L.) et Cavalcanti (A. L. C.). – *The MooZ Specification Language - version 0.4*. – Rapport technique nES/1.92, Universidade Federal de Pernambuco, 1992.

[Mey86] Meyrowitz (N.). – Intermedia : The architecture and construction of an object-oriented happened system and applications framework. *Dans : OOPSLA'86 Proceedings*. – Portland, OR, september 1986.

[Mey88] Meyer (B.). – *Object-Oriented Software Construction*. – Prentice-Hall, 1988.

[Min68] Minsky (M. L.). – Matter, mind and models. *Dans : Semantic Information Processing*. – MIT Press, 1968.

[MJ82] McCracken (D. D.) et Jackson (M. A.). – Life cycles concept considered harmful. *ACM Software Engineering Notes*, vol. 7, n2, april 1982.

[MMA96] Mocellin (F.), Martin (H.) et Adiba (M.). – STORM : a Structural and Temporal Object-oRiented Multimedia database system. *Dans : Demonstration and poster, Conference EDBT96*. – Avignon, march 1996.

[MO95] Martin (J.) et Odell (J.J.). – *Object Oriented Methods : A Foundation*. – New Jersey, Prentice Hall - Englewood Cliffs, 1995.

[MO96] Martin (J.) et Odell (J.J.). – *Object Oriented Methods : Pragmatic Considerations*. – New Jersey, Prentice Hall - Englewood Cliffs, 1996.

[MP92] Monarchi (D.E.) et Puhr (G.I.). – A research topology for object-oriented analysis ans design. *Communications of the ACM*, vol. 35, n9, september 1992, pp. 35–47.

[MSP84] McMenamim (S. M.), Stephem (M.) et Palmer (J. F.). – *Essential Systems Analysis*. – Yourdon Press - Englewood Cliffs, 1984.

[Mul97] Muller (P. A). – *Modélisation Objet avec UML*. – Eyrolles, 1997.

[Ner92] Nerson (J.M.). – Applying object-oriented analysis and design. *Communications of the ACM*, vol. 35, n9, september 1992, pp. 63–74.

[Nic96] Nicholls (J.). – Z standardization. – http ://www.comlab.ox.ac.uk/oucl/groups/zstandards/index.html, 1996.

[Nie90] Nielsen (J.). – *Hypertext and Hypermedia*. – London : Academic Press, 1990.

[Nor92] Noran (R. J.). – Working together to integrate case. *IEEE Software*, vol. 12, march 1992, pp. 12–16.

[NR69] Naur (P.) et Randel (B.). – *Software Engineering : A Report on a Conference Sponsored by NATO Science Commitee*. – Rapport technique, NATO, 1969.

[O293] O2 (Technology). – *The O_2 User Manual*. – Versailles, France, 1993.

[Obj92] Object Management Group - OMG. – Object oriented analysis and design, reference model. – 1992. Draft 7.0, unofficial position from OOAD SIG.

[Ode93] Odell (J. J). – Specifying structural constraints. *Journal of Object Oriented Programming*, 1993, pp. 12–16.

[Ode94] Odell (J. J.). – Six different kinds of composition. *Journal of Object Oriented Programming*, january 1994, pp. 10–15.

[Oma90] Oman (P. W.). – Case analysis and design tools. *IEEE Software*, may 1990.

[OMG93] OMG (The Object Management Group). – Common object request broker architecture (CORBA) 1.2 specifications. – 1993. OMG Document 93-12-43 / http ://www.mitre.org/research/domis/omg/orb/corba12-93-12-43.ps.

[Opé96] Opéra (Projet). – Thot - un éditeur de documents structurés. – http ://opera.inrialpes.fr/OPERA/Thot.fr.html, 1996.

[Ous97] Oussalah (M.). – *Ingénierie des Objets : Concepts, Techniques et Méthodes*. – Masson, 1997. Ouvrage collectif.

[Par93a] Parallax (Software Technologies). – *GraphTalk 2.5 - Interface de Programmation*. – 1993.

[Par93b] Parallax (Software Technologies). – *GraphTalk 2.5 - Méta-modélisation - Manuel de Référence*. – 1993.

[Par94] Parallax (Software Technologies). – *LEdit - Interface de Programmation*. – 1994.

[PLA96a] PLATINUM Technology Inc. – Enterprise component modeling. – http ://www.platinum.com/clearlake/paradigm30/ecm/ecm_wp.html, 1996.

[PLA96b] PLATINUM Technology Inc. – Fusion method overview. – 1996. http ://www.platinum.com/clearlake/paradigm30/oo_methods/ fusion.html.

[PLA96c] PLATINUM Technology Inc. – Paradigm plus - information. – http ://www.platinum.com/clearlake/paradigm30/paradigm30.html, 1996.

[Pre94] Pressman (R. S.). – *Software Engineering - A Practitioner Approach*. – McGraw-Hill Book Company Europe, 1994, third édition.

[QRRV95] Quint (V.), Richy (H.), Roisin (C.) et Vatton (I.). – *Grif - Manuel utilisateur*. – 1995.

[QV92] Quint (V.) et Vatton (I.). – Combining hypertext and structured documents in Grif. *Dans : Proceeding of the ACM Conference on Hypertext*, éd. par Lucarrella (D.), Nanard (J.), Nanard (M.) et Paolini (P.). – Milano, Italy, november/december 1992.

[Rag96] Raggett (D.). – Html 3.2 reference specification. – http ://www.w3.org/pub/WWW/TR/PR-html32-961105.html, 1996. World Wide Web Consortium - W3C.

[Rat96a] Rational Software Corporation. – The unified modeling language faq. – 1996. http ://www.rational.com/ot/uml/faq.html.

[Rat96b] Rational Software Corporation. – Unified modeling language for real-time systems design. – 1996. http ://www.rational.com/pst/tech_papers/uml_rt.html.

[Rat97] Rational (Software Corporation). – Unified Modeling Language - UML Summary. – http ://www.rational.com/uml, 1997.

[RBP+91] Rumbaugh (J.), Blaha (M.), Premerlani (W.), Eddy (F.) et Lorensen (W.). – *Object Oriented Modeling and Design*. – Prentice Hall, 1991.

[RFB88] Rolland (C.), Foucault (O.) et Benci (G.). – *Conception des Systèmes d'Information - La méthode REMORA*. – Eyrolles, Paris, 1988.

[RG92] Rubim (K. S.) et Golberg (A.). – Object behavior analysis. *Communications of the ACM*, vol. 35, n09, september 1992.

[Rhi94] Rhiemeier (J.). – VDM-SL Parser version 2. – ftp ://ftp.ips.cs.tu-bs.de/ftp/pub/local/sw/vdm-parser.2.readme, 1994.

[Ril95] Riley (G). – CLIPS - a tool for building expert systems. – http ://www.jsc.nasa.gov/ clips/CLIPS.html, 1995.

[RLB92] Rolland (C.), Lee (S.P.) et Brunet (J.). – Abstraction in the O* object-oriented method. *Dans : Indo-French Workshop on Object-oriented Systems*. – Goa, Inde, 1992.

[Rol92] Rolland (C.). – Outils case : État de l'art et perspectives. *Dans : Les Systèmes d'Information - De l'idée à la concétisation*. Université de Tunis II - L'École Nationale des Sciences de l'Informatique, pp. 1–15. – Tunis, mai 1992.

[Rol94] Rolland (C.). – Reformuler les démarches de conception des systèmes d'information. *Ingénierie des systèmes d'information*, vol. 2, n6, 1994, pp. 719–741.

[Roy70] Royce (W. W.). – Managing the developement of large software systems : concepts and techniques. *Dans : Proceedings, Wescon*, pp. 1–9.

[RS77] Ross (D. T.) et Schoman (K. E.). – Structured analysis for requirement definition. *IEEE Transactions on Software Engineering*, vol. 3, n1, january 1977.

[Rum95a] Rumbaugh (J.). – OMT : The dynamic model. *Journal of Object Oriented Programming*, february 1995, pp. 6–12.

[Rum95b] Rumbaugh (J.). – OMT : The functional model. *Journal of Object Oriented Programming*, march-april 1995, pp. 10–14.

[Rum95c] Rumbaugh (J.). – OMT : The object model. *Journal of Object Oriented Programming*, january 1995, pp. 21–27.

[Rum95d] Rumbaugh (J.). – What is a method ? *Journal of Object Oriented Programming*, 1995, pp. 10–16 and 26.

[SK89] Shneiderman (B.) et Kearsley (G.). – *Hypertext Hands-On !* – Addison-Wesley Publishing Company, 1989.

[SM88] Shlaer (S.) et Mellor (S. J.). – *Object Oriented System Analysis - Modeling the World in Data*. – Prentice Hall, 1988.

[SM92] Shlaer (S.) et Mellor (S. J.). – *Objects Life Cycles - Modeling the Word in States*. – Prentice Hall, 1992.

[Sma96] Smart (J.). – Hardy - a tool for building diagramming applications. – http ://www.aiai.ed.ac.uk/ hardy/hardy/hardy.html, 1996.

[SO90] Smith (D. B.) et Oman (P. W.). – Software tools in context. *IEEE Software*, may 1990, pp. 15–19.

[Som92] Sommerville (I). – *Software Enginnering*. – Addison Wesley, 1992, 4th édition.

[Spi89] Spivey (J.M.). – *The Z Notation : A reference Manual*. – Prentice Hall, 1989.

[Spi94] Spivey (J.M.). – *La Notation A*. – Masson-Prentice Hall, 1994.

[Sto96] Stobart (S.). – Tools - computer aided software engineering. – http ://osiris.sunderland.ac.uk/sst/casehome.html, 1996.

[tHPvdW93] ter Hofstede (A.H.M.), Proper (H.A.) et van der Weide (T.P.). – Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, vol. 18, november 1993, pp. 489–523.

[TRR83] Tardieu (H.), Rochfeld (A.) et Rolland (C.). – *La méthode Merise : principes et outils*. – Editions d'Organisation, Paris, 1983.

[Ver95] Verhoef (M.). – VDM Frequently-Asked Questions - FAQ. – file ://ftp.ifad.dk/pub/vdm/VDM-FAQ-ENG, 1995.

[Was89] Wasserman (A. I.). – *The Architecture of CASE Environments*. – CASE Outlook, 1989.

[WBWW90] Wirfs-Brock (R.), Wilkerson (B.) et Wiener (L.). – *Designing Object-Oriented Software*. – Prentice Hall - Englewood Cliffs, 1990.

[Win90] Wing (J. M.). – A specifier's introduction to formal methods. *IEEE Computer*, september 1990, pp. 8–24.

[Win93] Winskel (G.). – *The Formal semantics of programming languages : an introduction*. – The MIT Press, 1993, *Foundations of Computing Series*.

[Wor92] Wordsworth (J. B.). – *Software Development with Z.* – Addison-Wesley, 1992, *International Computer Science-Series*.

[YC79] Yourdon (E.) et Constantine (L. L.). – *Structured Design*. – Prentice-Hall, 1979.

[You94] Yourdon (E.). – *Object-Oriented systems Design - An Integrated Approach*. – Prentice-Hall, 1994.

Index

- T -

- U -

- V -

- Z -

Résumé : Ce travail de recherche est centré sur une proposition pour l'intégration de différents formalismes dans la modélisation soit de modèles de méthodes orientées objets, soit de modèles de systèmes d'information. Pour remplir cet objectif nous avons établi et expérimenté un méta-modèle. Une originalité de notre méta-modèle est son organisation en vues pour décrire des modèles selon des aspects statiques, dynamiques mais aussi semi-formels, formels et informels. Nos propositions sont précédées d'une étude de différents langages et formalismes qui caractérisent les approches actuelles de modélisation. Ce cadre de méta-modélisation est étendu pour favoriser une multi-modélisation de systèmes. D'une manière classique, la multi-modélisation est limitée à l'utilisation de diagrammes complémentaires pour décrire un système. Nous complétons cette forme de multi-modélisation par une étude de relations nécessaires entre différents fragments de modèles décrits avec des formalismes variés : semi-formels, formels ou informels.

Pour concrétiser notre proposition nous avons réalisé un premier prototype d'atelier pour illustrer et expérimenter les services que nous attendons d'un atelier de modélisation adapté à des modélisations plus rigoureuses de systèmes d'information. Ce prototype est en réalité un générateur d'ateliers de modélisation basé sur l'utilisation de trois logiciels différents de type générateur d'éditeurs graphiques, générateur d'éditeurs syntaxiques et éditeur paramétré de documents structurés. Notre prototype a été utilisé dans plusieurs situations : méta-modélisation de sous-ensembles des méthodes OMT et OOA, proposition d'un noyau de référentiel de comparaison de modèles de méthodes, multi-modélisation d'un système d'information spécifique (modèle STORM d'applications multimédias) et guidage d'une multi-modélisation au travers d'un canevas standard de dossier de spécifications de systèmes.

Mots clés : Méta-Modélisation, Multi-Modélisation, Atelier de Modélisation, Spécifications Hétérogènes.

Abstract :This research focuses on a proposal which integrates different formalisms during the modeling process that concerns either models of object oriented techniques or models of information systems. To reach this goal, we established and experimented a meta-model. The originality of this meta-model is its structure based on views in order to describe models from static and dynamic aspects but also using semi-formal, formal and informal approaches. Before introducing our proposals, we present a study about different languages and formalisms which are characteristic of various modeling approaches. This framework of meta-modeling is extended in order to improve the system multi-modeling. Usually, multi-modeling is essentially concerned by the use of additional diagrams describing the system. We complete this form of modeling with a study of the relationships between different pieces of models which are described using various formalisms : semi-formal, formal or informal.

A prototype has been developed in order to show and to experiment capabilities that we expect from a modeling tool providing a more rigorous modeling of information systems. Our prototype, a modeling tool generator, is based on the use of three different softwares components, a graphical editor generator, a syntactic editor generator and a parameterized editor for structured documents. The prototype has been used in several situations : the meta-modeling of a subset of OMT and OOA methods, the proposal of a reference kernel for the comparison of varius methods models, the multi-modeling of a specific information system (STORM systems for multimedia applications) and to help to multi-modeling using a customized framework of systems specifications documents.

Keywords : Meta-Modeling, Multi-Modeling, Modeling Tool, Heterogeneous Specifications.